70_Spring 事务框架源码初探（二）：如果业务逻辑没报错则提交事务否则事务回滚

1、开篇

上节课主要围绕 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。本节课通过源码分析的方式分析事务提交和回滚的部分。

今天课程的内容包括以下几个部分：

- 分析事务提交和回滚源码
- 总结提交回滚流程

2、分析事务提交和回滚源码

接上节课的内容，在我们启动了 Spring 的事务以后，如果事务执行顺利就会提交事务，否则会对事务进行回滚。还记得上节课中介绍过的 invokeWithinTransaction 方法吗？里面不仅定义了拦截器的执行还定义了事务的提交和回滚。如图 1 所示，在 invokeWithinTransaction 方法中 try catch 语句中定义了执行拦截器，和事务回滚的代码，其中事务回滚会调用 completeTransactionAfterthrowing。如果事务执行顺利就会调用 commitTransactionAfterReturning 方法，下面我们就顺着这两个方法展开说明事务的回滚和提交操作。

```
@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, targetCla
    //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 PlatformTransac
        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 1

如图 2 所示，在 completeTransactionAfterThrowing 方法内部主要处理异常情况下是否需要回滚，注意方法内部会向上递归查找父类是否和 Exception 相同，如果相同进行回滚。其中用红色框体框起来的部分调用了 rollback 方法，该方法是从事务信息对象中的 getTransactionManager 中获取的，传入了事务状态对象（getTransactionStatus 获取状态）。

```
protected void completeTransactionAfterThrowing(@Nullable TransactionInfo txInfo, Throwable ex) {
    if (txInfo != null && txInfo.getTransactionStatus() != null) {
        if (logger.isTraceEnabled()) {
            logger.trace("Completing transaction for [" + txInfo.getJoinpointIdentification() +
                    "] after exception: " + ex);
        }
        //使用RuleBasedTransactionAttribute判断当前异常是否需要回滚-如果注解上定义了@Transactional(rollbackFor = Exception.class)
        //这里会使用ex向上递归查找父类是否和Exception是否相同，如果相同则进行回滚
        if (txInfo.transactionAttribute != null && txInfo.transactionAttribute.rollbackOn(ex)) {
            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 | Error ex2) {
                logger.error("Application exception overridden by rollback exception", ex);
                throw ex2;
            }
        }
        //否则执行提交
        else {
            //commit() 会判断通过手动设置回滚的操作，然后也会进行回滚
            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 | Error ex2) {
                logger.error("Application exception overridden by commit exception", ex);
                throw ex2;
            }
        }
    }
}
```

图 2

由于此处使用了 TransactionManager，这里顺带提一下，PlatformTransactionManager 继承自 TransactionManager 接口，AbstractPlatformTransactionManager 对 PlatformTransactionManager 进行了抽象，对方法进行了具体的实现；下面我们来看看源代码是什么样的吧。如图 3 所示，PlatformTransactionManager 继承了 TransactionManager 接口。其中也继承了创建事务并开启事务的方法 getTransaction、提交事务的方法 commit 以及回滚事务的方法 rollback。

```
public interface PlatformTransactionManager extends TransactionManager {
    //1.创建一个事务并开启事务
    TransactionStatus getTransaction(@Nullable TransactionDefinition definition)
            throws TransactionException;
   //2.提交事务
    void commit(TransactionStatus status) throws TransactionException;
   //3.回滚事务
    void rollback(TransactionStatus status) throws TransactionException;
}
```

图 3

看完了 Rollback 相关的方法以后再来看看 commitTransactionAfterReturning，如图 4 所示，在方法体中通过 getTransactionManager 调用 commit 方法传入事务状态。

```
protected void commitTransactionAfterReturning(@Nullable TransactionInfo txInfo) {
    if (txInfo != null && txInfo.getTransactionStatus() != null) {
        if (logger.isTraceEnabled()) {
            logger.trace("Completing transaction for [" + txInfo.getJoinpointIdentification() + "]");
        }
        txInfo.getTransactionManager().commit(txInfo.getTransactionStatus());
    }
}
```

图 4

如图 5 所示，需要注意的是在 commit 方法中会针对 isLocalRollbackOnly 进行判断，如果为 true 会执行 processRollback 方法进行手动回滚。同时还会通过 shouldCommitOnGlobalRollbackOnly 和 isGlobalRollbackOnly 判断进行 processRollback 的手动回滚。

```
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
    }
    DefaultTransactionStatus defStatus = (DefaultTransactionStatus) status;
    //如果执行了TransactionInterceptor.currentTransactionStatus().setRollbackOnly();
    //这里会判断为true并且进行回滚操作
    if (defStatus.isLocalRollbackOnly()) {
        if (defStatus.isDebug()) {
            logger.debug("Transactional code has requested rollback");
        }
        processRollback(defStatus, false);
        return;
    }

    //通过ConnectHolder setRollbackOnly() 设置的回滚
    if (!shouldCommitOnGlobalRollbackOnly() && defStatus.isGlobalRollbackOnly()) {
        if (defStatus.isDebug()) {
            logger.debug("Global transaction is marked as rollback-only but transactional co
        }
        processRollback(defStatus, true);
        return;
    }
    //执行提交
    processCommit(defStatus);
}
```

图 5

3、总结提交回滚流程

上面把主要的源代码进行了讲解，下面还是回到 Spring 事务的大图，将提交和回滚的流程进行梳理。流程图已久从 TransactionInterceptor 开始，调用父类的 invokeWithinTransaction 方法业务逻辑如果报错走左边的逻辑调用 TransactionAspectSupport 中的 completeTransactionAfterThrowing，继续通过 AbstractPlatformTransactionManager 的 rollback 方法执行回滚操作。

如果没有报错调用 commitTransactionAfterReturning 方法，使用 AbstractPlatformTransactionManager 的 commit 方法进行提交操作。Rollback 和 commit 的后续操作涉及到 JpaTransactionManager 类，在这里不展开描述。

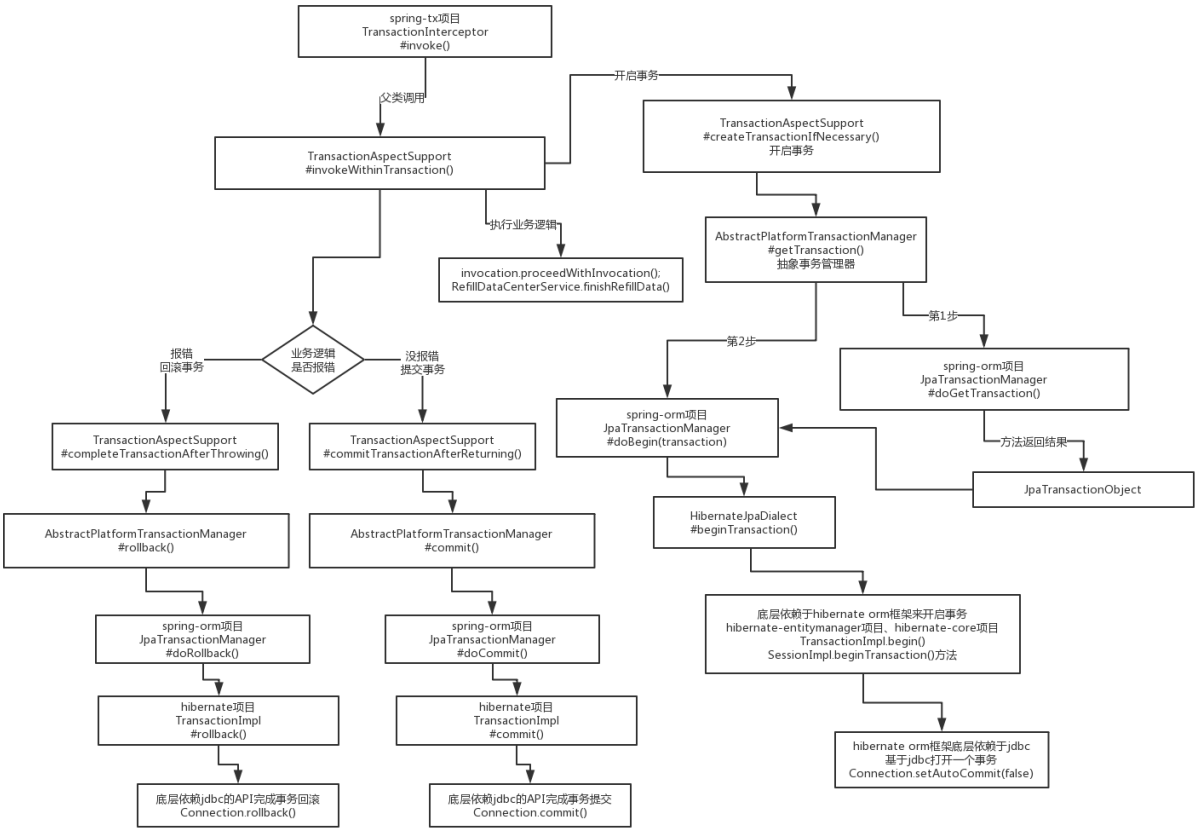

图 6

4、总结

上节课针对源码分析了 Spring 事务的几个方法：

TransactionAspectSupport#completeTransactionAfterThrowing 和 commitTransactionAfterReturning、以及 AbstractPlatformTransactionManager@rollback 和 commit。接着对整个提交和回滚的流程图进行了梳理。

下节课会基于 Spring 事务保存订单数据，解决一致性问题。下期见，拜拜。