12.奇葩的支付故障：我叒被数据库事务坑了

大家好，我是隐墨星辰，深耕境内/跨境支付架构设计十余年。

今天的内容来源于奇葩的线上故障，主要和大家聊聊为什么支付交易系统中**只使用事务模板方法**，而不使用声明式事务@Transactional注解，以及**使用afterCommit()出现连接未按预期释放导致的性能问题**。

以前写管理平台的代码时，经常使用@Transaction注解，也就是所谓的声明式事务，简单而实用，但是在做支付后，碰到了麻烦。

第一个故障：**上游有数据，下游没数据**。

当前系统做得比较早，直接使用了@Transactional注解来负责事务管理。有一天上游过来找我，说有笔单支付失败，问我原因是什么。我拿着单号去数据库里捞数据，神了，竟然没有这笔数据。

但是上游一口咬定已经发下来了，于是只好去日志系统里捞日志。果然，上游有发下来，也进行了保存数据库操作，但是很不幸，之后的一个操作抛了异常，导致事务回滚，数据没了。

伪代码大致如下：

```java
@Transactional
public PayOrder process(PayRequest request) {
    validate(request);
    PayOrder payOrder = buildOrder(request);
    save(payOrder);
    // 其它处理
    otherProcess(payOrder);
}
```

在otherProcess()方法中抛出了异常。

第二个故障：**银行支付成功，我方数据库没有请求流水记录**。

有一天财务小妹过来找我，说对账出了银行长款，银行对账文件里面有一条支付成功的数据，但是我们系统没有记录。

我当时有点蒙圈：“银行不至于这么神经大条吧，还主动给我们送钱不成？”。去查了数据库，的确是没有这条数据。

于是让财务小妹找银行确认，银行反馈明确是我们发送的请求。没有办法，去翻日志，结果发现还真是我们的发出去的请求。

代码基本和上面一致：

```java
@Transactional
public PayOrder process(PayRequest request) {
    validate(request);
    PayOrder payOrder = buildOrder(request);
    save(payOrder);
    // 发送处理
    send(payOrder);
}
```

结果在send()方法里面，因为请求银行超时抛出了异常，导致事务回滚，数据没了，但是银行扣款成功。

为什么之前没有发现？因为当时经验不足，每对接一个银行渠道，都是创建独立的数据库流水表，也都是写独立的对接代码，在代码里面进行包括校验、保存数据库，组装报文，外发请求等。且当时还没有到大促，日常银行返回比较快，也没有出现性能瓶颈。

真是万幸没有拖到大促时才发现。

所以从那以后，**就在支付交易系统中把@Transactional注解全部干掉，换成了事务模板方法来做**。

@Transactional在管理平台可能的确好用，但在支付交易系统中，存在2个致命问题。

1）**事务的粒度控制不够灵活，容易出现长事务。**

@Transactional注解通常应用于方法级别，这意味着被注解的方法将作为一个整体运行在事务上下文中。在复杂的支付流程中，需要做各种运算处理，很多前置处理是不需要放在事务里面的。

而使用事务模板的话，就可以更精细的控制事务的开始和结束，以及更细粒度的错误处理逻辑。

比如上面第一个故障的示例代码中，校验，构建订单，其它处理等都不需要放在事务中。

如果把@Transactional从process()中拿走，放到save()方法，也会面临另外的问题：otherProcess()依赖数据库保存成功后才能执行，如果保存失败，不能执行otherProcess()处理。全部考虑进来后，使用注解处理起来就很麻烦。

2）**事务传播行为的复杂性。**

@Transactional注解支持不同的事务传播行为，虽然这提供了灵活性，但在实际应用中，错误的事务传播配置可能导致难以追踪的问题，如意外的事务提交或回滚。

而且经常有多层子函数调用，很容易子函数有一个耗时操作（比如RPC调用或请求外部应用），一方面可能出事长事务，另一方面还可能因为外调抛异步，导致事务回滚，数据库中都没有记录保存。

以前上面第二个故障的示例代码中，因为在父方法使用了@Transactional注解，子函数抛出异常，事务回滚，在数据库就找为到问题单据，配合日志和翻代码一行行看，才发现问题。

第三个故障：**大流量情况下，应用持续报获取数据库连接超时**。这就涉及到事务spring自带模板方法中afterCommit()挖的坑。

换了事务模板方法后，有一天流量比较大，监控一直在报“获取数据库连接超时”，以为是连接池配置太小，去找DBA要扩大，但是DBA说按当时的流量，从理论上是足够的，死活不愿意扩，让我们先分析清楚原因。

不得已，回来继续找原因。

还真找到了。

无论在支付系统，还是电商系统，还是其它各种业务系统，都存在这样的需求：**在一个操作中既要保存数据到多张表中，又要外发请求，且这个外发请求耗时很长。**

比如：如先保存主单据，再保存流水单据，然后外发银行请求扣款。这三个方法需要在一个事务里面。

一共有三个方案：

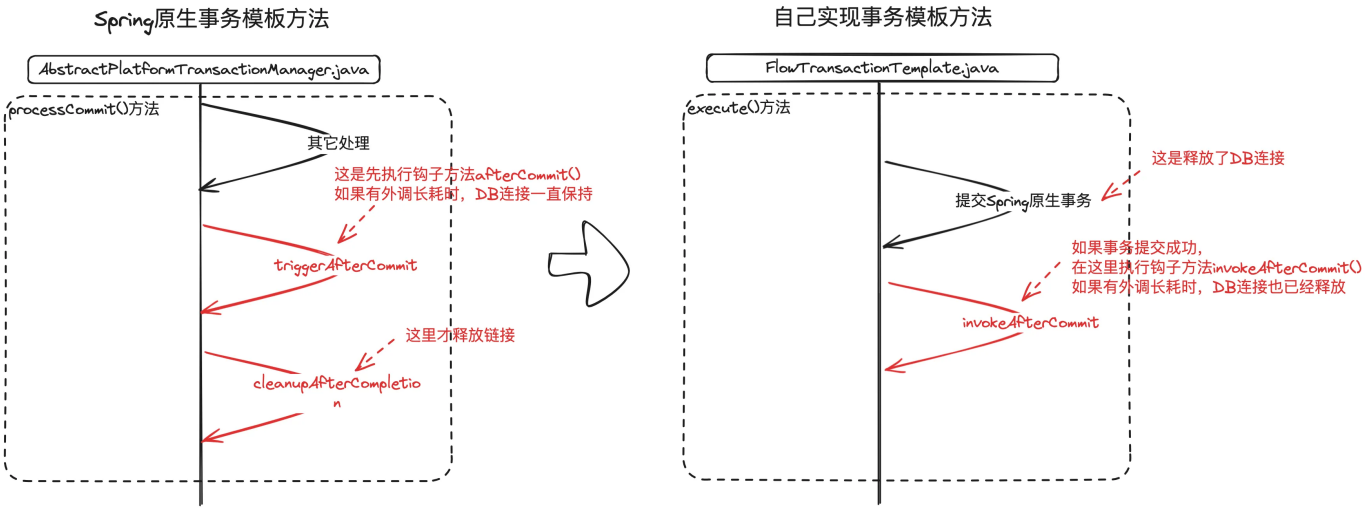
方案一：不管三七二十一，就直接放在一个事务中。请求量不大时，看不出长事务的影响。

方案二：知道使用Spring提供的模板方法：TransactionSynchronizationAdapter.afterCommit()。外发请求耗时长过长时，在大并发下仍然有连接未能及时释放的问题。

方案三：自己实现事务模板方法，在Spring提交事务并释放连接后，再执行耗时长的外发。

方案一完全不可用，直接忽略。我们当时用了方案二，出了问题。最后换了方案三。

方案二和方案三区别如下图所示：



当时是这样的：先保存主单据，再保存流水单据，然后外发银行请求扣款。写的代码类似这样：

主方法伪代码：

```java
public void process(OderContext context) {
    // 获取流程处理链
    List<OrderProcess> processes = fetchProcess(context);
    for (OrderProcess process : processes) {
        // 使用事务模板
        dataSourceManager.getTransactionTemplate().execute(status -> {
            // 执行子流程
            process.execute(context);

            // 更新主单信息
            context.getPayOrder().putJournal(context.getJournal());
            context.getPayOrder().transToNextStatus(context.getJournal().getTargetStatus());
            save(context.getPayOrder());
            return true;
        });
    }
}

```

其中一个外发银行子流程伪代码：

```java
public void execute(OderContext context) {
    Journal journal = buildJournal(context);
    // 子函数里面保存了3张表的数据
    save(journal);

    TransactionSynchronizationManager.registerSynchronization(new TransactionSynchronizationAdapter() {
        @Override
        public void afterCommit() {
            // 事务提交后，再发送给外部银行
            gatewayService.sendToChannel(journal);
        }
    });
}
```

预期是事务提交后再调用发给银行。

但是实际情况却是，**Spring提交事务后，调用了afterCommit()，但是并没有释放连接，导致在外发银行的长达1000多毫秒的时间内，数据库连接一直在保持**，而不是提交事务后马上归还了连接，加上只分配了x个连接数给每台应用。这就意味着最大并发也小于x。

继续翻代码，通过查看AbstractPlatformTransactionManager.java，**发现是先调用：riggerAfterCommit(status)，然后才清理并释放连接：cleanupAfterCompletion(status)**。如下：

```java
private void processCommit(DefaultTransactionStatus status) throws TransactionException {
    try {
        // 其它代码省略
        ... ...

        // Trigger afterCommit callbacks, with an exception thrown there
        // propagated to callers but the transaction still considered as committed.
        try {
            triggerAfterCommit(status);
        }

        // 其它代码省略
        ... ...

    } finally {
        cleanupAfterCompletion(status);
    }
}
```

解决办法：**自己创建一个事务模板，实现afterCommit()**。

```java
public class OrderTransactionTemplate {

    public static <R> R execute(OderContext context, Supplier<R> callback) {
        TransactionTemplate template = context.getTransactionTemplate();
        Assert.notNull(template, "transactionTemplate cannot be null");

        PlatformTransactionManager transactionManager = template.getTransactionManager();
        Assert.notNull(transactionManager, "transactionManager cannot be null");

        boolean commit = false;
        try {
            TransactionStatus status = transactionManager.getTransaction(new DefaultTransactionDefinition());
            R result = null;
            try {
                result = callback.get();
            } catch (Exception e) {
                transactionManager.rollback(status);
                throw e;
            }
            // 这里是关键：直接提交事务，连接也被马上释放
            transactionManager.commit(status);
            commit = true;
            return result;
        } finally {
            if (commit) {
                // 事务提交成功，才执行后续操作
                invokeAfterCommit(context);
            }
        }
    }

    private static void invokeAfterCommit(OrderContext context) {
        try {
            context.invokeAfterCommit();
        } catch (Exception e) {
            // 打印日志
            ... ...
        }
    }
}

```

OrderContext加上事务提交后的执行的钩子方法，在钩子方法中实现一些长耗时工作：

```java
public class OderContext {
    // 其它代码不变
    ... ...

    private List<AfterCommitHook> afterCommitHooks = new ArrayList<>();

    public void registerAfterCommitHook(AfterCommitHook hook) {
        afterCommitHooks.add(hook);
    }

    public void invokeAfterCommit() {
        try {
            for(AfterCommitHook hook : afterCommitHooks) {
                hook.afterCommit();
            }
        } catch (Exception e) {
            // 异常处理
            ... ...
        } finally {
            // 钩子已执行完，清理掉
            afterCommitHooks.clear();
        }
    }

    public static abstract class AfterCommitHook {
        public abstract void afterCommit();
    }
}
```

主流程修改为直接调用：OrderTranscationTemplate.execute。

```java
public void process(OderContext context) {
    context.setTransactionTemplate(dataSourceManager.getTransactionTemplate());

    List<OderProcess> processes = fetchProcess(context);

    for (OrderProcess process : processes) {
        // 把Spring模板方法修改自己的模板方法，其它不变
        OrderTransactionTemplate.execute(context, () -> {
            process.execute(context);

            context.getPayOrder().putJournal(context.getJournal());
            context.getPayOrder().transToNextStatus(context.getJournal().getTargetStatus());
            save(context.getPayOrder());

            return true;
        });
    }
}

```

子流程修改为把afterCommit要做的事注册到流程上下文中：

```java
public void execute(OderContext context) {
    Journal journal = buildJournal(context);
    // 子函数里面保存了3张表的数据
    save(journal);

    // 把外发动作注册到流程上下文中的钩子方法中，
    // 而不是直接使用Spring原生的TransactionSynchronizationAdapter.afterCommit()
    // 其它保持不变
    context.registerAfterCommitHook(() -> {
        // 事务提交后发给银行
        gatewayService.sendToChannel(journal);
    });
}
```

这样处理的优点有几个：

1. 清晰的事务边界管理：通过显式控制事务的提交和回调执行，增加了代码的可控性。
2. 资源使用优化：确保数据库连接在不需要时能够及时释放，提升了资源的使用效率。
3. 灵活的后续操作扩展：允许注册多个回调，方便地添加事务提交后需要执行的操作，增强了代码的扩展性和复用性。

有个注意的点，就是确保invokeAfterCommit的稳健性，代码里是通过捕获异常打印日志，避免对其它操作有影响。

提点小拓展知识：长事务

长事务指的是在数据库管理和应用开发中，持续时间较长的事务处理过程。一般来说，在分布式应用中，每个服务器分配的连接数是有限的，比如每个服务器20个连接，这就要求我们必要尽量减少长事务，以便处理更多请求。

典型的方案有：

1）非事务类操作，就放在事务外面。比如前置处理，先请求下游获取资源，做各种校验，全部通过后，再启动事务。还有就是使用hook的方式，等事务提交后，再请求外部耗时的服务。

2）事务拆分。把一个长事务拆分为多个短事务。

3）异步处理。有点类似hook的方案。

Spring事务管理提供了强大而灵活的机制来处理复杂的业务逻辑，但是每个特性和工具的使用都需要对其行为有深入的理解，而不能想当然。比如文中的afterCommit就是这样一个典型例子。

自定义事务模板的实践向我们展示了，虽然@Transcation注解很方便，但在一些特殊场景下，需要我们深入了解框架的工作原理并结合实际业务需求，既高效地利用Spring提供的工具，同时也规避潜在的坑点。

希望本文能够帮助读者更好地理解和应用Spring事务管理中的afterCommit钩子，以及如何在对资源或性能要求很严格的情况下，比如支付场景，如何定义自己的事务模板，帮助我们构建更健壮、更高效的应用。

这是《图解支付系统设计与实现》专栏系列文章中的第（30）篇。

深耕境内/跨境支付架构设计十余年，欢迎关注并星标公众号“隐墨星辰”，和我一起深入解码支付系统的方方面面。

专栏系列文章PDF合集不定时更新，欢迎关注我的公众号“隐墨星辰”，留言“PDF”获取。

隐墨星辰 公众号

10年顶尖境内/跨境支付公司架构经验

著有《图解支付系统设计与实现》
和我一起解码支付系统方方面面

有个支付系统设计与实现讨论群，添加个人微信（yinmon_sc）备注666进入。

隐墨星辰 个人微信

10年顶尖境内/跨境支付公司架构经验

著有《图解支付系统设计与实现》
备注666进支付讨论群