

30. Spring事务模板及afterCommit存在的坑

_V20240225

- 1. 为什么不使用@Transaction注解
- 2. afterCommit存在的问题及解法
- 3. 扩展：长事务
- 4. 结束语

大家好，我是墨哥（隐墨星辰）。今天的内容来源于两个线上问题，主要和大家聊聊为什么支付系统中基本只使用事务模板方法，而不使用声明式事务@Transaction注解，以及使用afterCommit()出现连接未按预期释放导致的性能问题。

1. 为什么不使用@Transaction注解

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，基本上没有使用@Transaction，全部使用事务模板来做。主要有两个考虑：

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。

```
1  @Transactional
2  public PayOrder process(PayRequest request) {
3      validate(request);
4      PayOrder payOrder = buildOrder(request);
5      save(payOrder);
6      // 其它处理
7      otherProcess(payOrder);
8  }
```

比如上面的校验，构建订单，其它处理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。

以前就在生产上碰到过类似的问题，因为在父方法使用了@Transactional注解，子函数抛出异常，去数据库找问题单据，竟然没有记录，翻代码一行行看，才发现问题。

2. afterCommit存在的问题及解法

有一次参与线上压测，在流量上去后，应用持续报获取数据库连接超时，排查很久才找到原因，问题非常经典，值得和大家聊聊。

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：**在一个事务中既保存多个数据库表，又要外发请求，且这个外发请求耗时很长。**

比如：方法A保存数据库表A，方法B保存数据库表B，并且要外发给其它系统且耗时长，方法C要保存数据库表C。这三个方法需要在一个事务里面。

我见过三种方案：

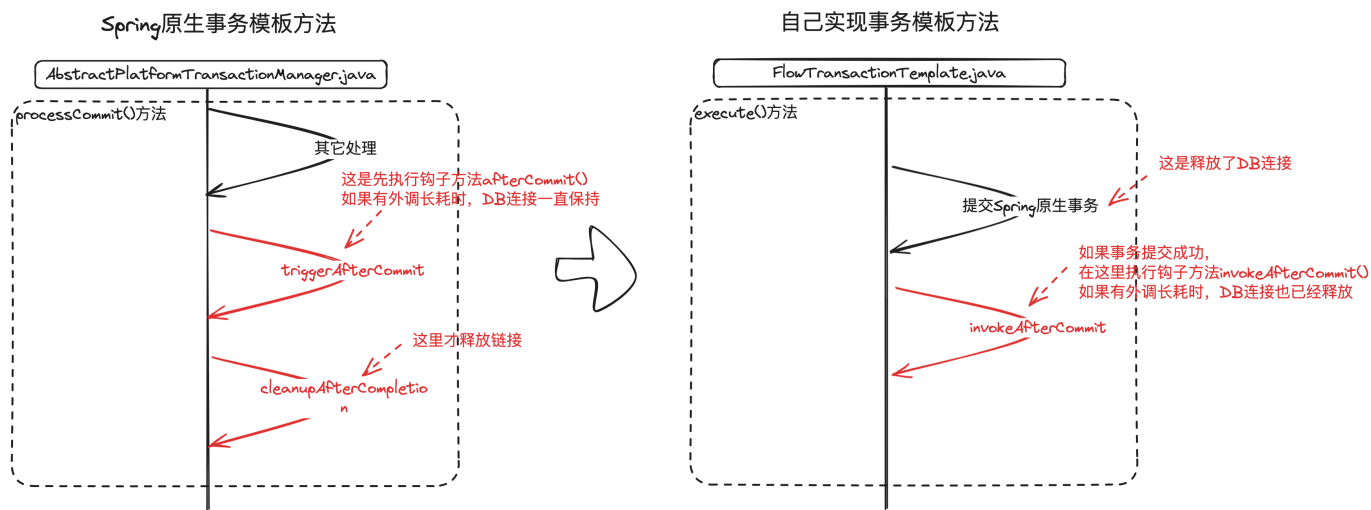
方案一：不管三七二十一，就直接放在一个事务中。请求量不大时，看不出长事务的影响。

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。

第一种没什么好说的，下面介绍方案二和方案三，两者区别如下图所示：



在支付系统中，经常需要做一些流程编排，这些流程操作需要放在一个事务中，比如先保存主单据，再保存流水单据，然后外发银行请求扣款，有同学写的代码类似这样：

主方法伪代码（流程引擎入口）：

```

1  public void process(FlowContext context) {
2      // 获取流程处理链
3      List<FlowProcess> flows = fetchFlow(context);
4      for (FlowProcess flow : flows) {
5          // 使用事务模板
6          dataSourceManager.getTransactionTemplate().execute(status -> {
7              // 执行子流程
8              flow.execute(context);
9
10             // 更新主单信息
11             context.getPayOrder().putJournal(context.getJournal());
12             context.getPayOrder().transToNextStatus(context.getJournal().getTargetStatus());
13             save(context.getPayOrder());
14             return true;
15         });
16     }
17 }
18

```

其中一个外发银行子流程伪代码：

```

1  public void execute(FlowContext context) {
2      Journal journal = buildJournal(context);
3      // 子函数里面保存了3张表的数据
4      save(journal);
5
6      TransactionSynchronizationManager.registerSynchronization(new TransactionSynchronizationAdapter() {
7          @Override
8          public void afterCommit() {
9              // 事务提交后，再发送给外部银行
10             gatewayService.sendToChannel(journal);
11         }
12     });
13 }

```

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，加上线上服务器的连接数只分配了30个给每台应用。这就意味着最大并发也小于30。

通过查看AbstractPlatformTransactionManager.java，发现是先调用：
triggerAfterCommit(status)，然后才清理并释放连接：cleanupAfterCompletion(status)。

```
Java |  
1 private void processCommit(DefaultTransactionStatus status) throws TransactionException {  
2     try {  
3         // 其它代码省略  
4         ...  
5  
6         // Trigger afterCommit callbacks, with an exception thrown the  
7         re // propagated to callers but the transaction still considered  
8         as committed.  
9         try {  
10             triggerAfterCommit(status);  
11         }  
12         // 其它代码省略  
13         ...  
14     }  
15     finally {  
16         cleanupAfterCompletion(status);  
17     }  
18 }  
19 }
```

解决办法：自己创建一个事务模板，实现afterCommit()。

```
1 public class FlowTransactionTemplate {
2
3     public static <R> R execute(FlowContext context, Supplier<R> callback)
4     {
5         TransactionTemplate template = context.getTransactionTemplate();
6         Assert.notNull(template, "transactionTemplate cannot be null");
7
8         PlatformTransactionManager transactionManager = template.getTransactionManager();
9         Assert.notNull(transactionManager, "transactionManager cannot be null");
10
11         boolean commit = false;
12         try {
13             TransactionStatus status = transactionManager.getTransaction(new DefaultTransactionDefinition()); // Corrected "TranscationStatus" to "TransactionStatus"
14             R result = null;
15             try {
16                 result = callback.get();
17             } catch (Exception e) {
18                 transactionManager.rollback(status);
19                 throw e;
20             }
21             transactionManager.commit(status);
22             commit = true;
23             return result;
24         } finally {
25             if (commit) {
26                 invokeAfterCommit(context);
27             }
28         }
29
30     private static void invokeAfterCommit(FlowContext context) {
31         try {
32             context.invokeAfterCommit();
33         } catch (Exception e) {
34             // 打印日志
35             ... ..
36         }
37     }
38 }
39
```

FlowContext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：

```
1 public class FlowContext {
2     // 其它代码不变
3     ...
4
5     private List<AfterCommitHook> afterCommitHooks = new ArrayList<>();
6
7     public void registerAfterCommitHook(AfterCommitHook hook) {
8         afterCommitHooks.add(hook);
9     }
10
11     public void invokeAfterCommit() {
12         try {
13             for(AfterCommitHook hook : afterCommitHooks) {
14                 hook.afterCommit();
15             }
16         } catch (Exception e) {
17             // 异常处理
18             ...
19         } finally {
20             // 钩子已执行完，清理掉
21             afterCommitHooks.clear();
22         }
23     }
24
25     public static abstract class AfterCommitHook {
26         public abstract void afterCommit();
27     }
28 }
```

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

```

1  public void process(FlowContext context) {
2      context.setTransactionTemplate(dataSourceManager.getTransactionTemplate());
3
4      List<FlowProcess> flows = fetchFlow(context);
5
6      for (FlowProcess flow : flows) {
7          // 把Spring模板方法修改自己的模板方法，其它不变
8          FlowTransactionTemplate.execute(context, () -> {
9              flow.execute(context);
10
11              context.getPayOrder().putJournal(context.getJournal());
12              context.getPayOrder().transToNextStatus(context.getJournal().getTargetStatus());
13              save(context.getPayOrder());
14
15              return true;
16          });
17      }
18  }
19

```

子流程修改为把afterCommit要做的事注册到流程上下文中：

```

1  public void execute(FlowContext context) {
2      Journal journal = buildJournal(context);
3      // 子函数里面保存了3张表的数据
4      save(journal);
5
6      // 把外发动作注册到流程上下文中的钩子方法中，
7      // 而不是直接使用Spring原生的TransactionSynchronizationAdapter.afterCommit()
8      // 其它保持不变
9      context.registerAfterCommitHook(() -> {
10          // 事务提交后发给银行
11          gatewayService.sendToChannel(journal);
12      });
13  }

```


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。

3. 扩展：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。

4. 结束语

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@Transactional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《百图解码支付系统设计与实现》专栏系列文章中的第（30）篇。和墨哥（隐墨星辰）一起深入解码支付系统的方方面面。

欢迎转载。

Github（PDF文档全集，不定时更新）：<https://github.com/yinmo-sc/Decoding-Payment-System-Book>

公众号：隐墨星辰。



微信搜一搜



隐墨星辰

有个小群不定时解答一些问题或知识点，有兴趣的同学可先加微信（yinmo_sc）后进入，加微信请备注：加支付系统设计与实现讨论群。

