1、TCC分布式事务：https://blog.csdn.net/kobejayandy/article/details/54783212

2、集中解决分布式事务的方案：https://segmentfault.com/a/1190000012762869

3、LCN: <https://github.com/1991wangliang/springcloud-lcn-demo>

4、LCN说明：<https://github.com/1991wangliang/tx-lcn/wiki>

5、

二、LCN具体说明

1、LCN：事务协调器。

2、通过TxManager协调控制与事务补偿机制确保数据一致性。

3、TxManager启动说明:

<https://github.com/codingapi/tx-lcn/wiki/TxManager%E5%90%AF%E5%8A%A8%E8%AF%B4%E6%98%8E>

4、SpringCloud下配置LCN、Demo

<https://github.com/codingapi/springcloud-lcn-demo/wiki>

5、SpringCloud Demo：

三：多应用的几种解决方案

1、基于消息队列最终一致性：https://github.com/yu199195/myth

2、TCC型解决方案：https://github.com/yu199195/happylifeplat-tcc

3、二阶段提交强一致性：https://github.com/yu199195/happylifeplat-transaction

4、LCN:https://github.com/codingapi/tx-lcn

四、LCN方案

1、基本原理

2、工程依赖的服务包括

（1）TxManager

（2）Mysql

（3）Redis

（4）Eureka

3、核心步骤

（1）创建事务组

是指在事务发起方开始执行业务代码之前先调用TxManager创建事务组对象，然后拿到事务标示GroupId的过程。

（2）添加事务组

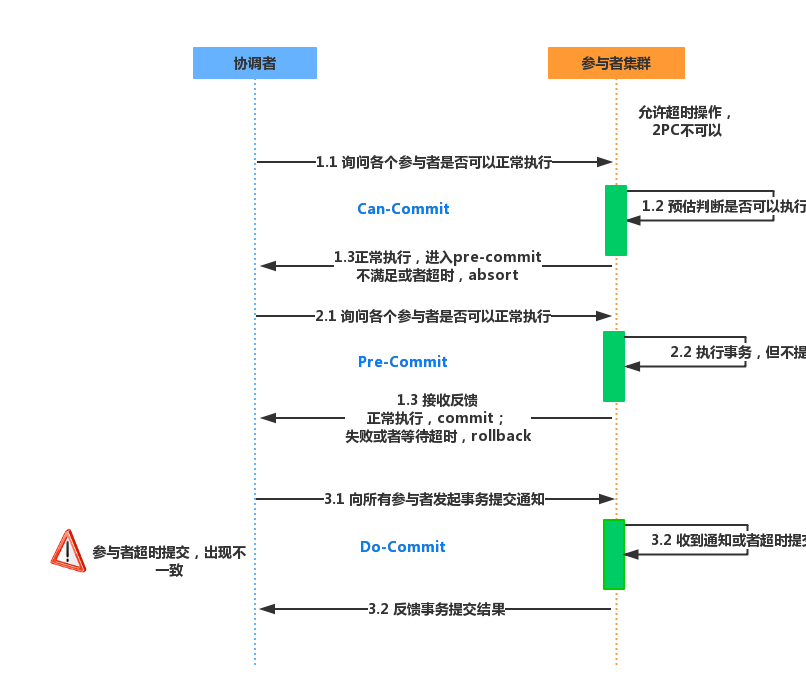
添加事务组是指参与方在执行完业务方法以后，将该模块的事务信息添加通知给TxManager的操作。

（3）关闭事务组

是指在发起方执行完业务代码以后，将发起方执行结果状态通知给TxManager的动作。当执行完关闭事务组的方法以后，TxManager将根据事务组信息来通知相应的参与模块提交或回滚事务。

4、LCN采用三段式提交的方式。两段式提交和三段式提交的比较：

<https://www.cnblogs.com/binyue/p/3678390.html>



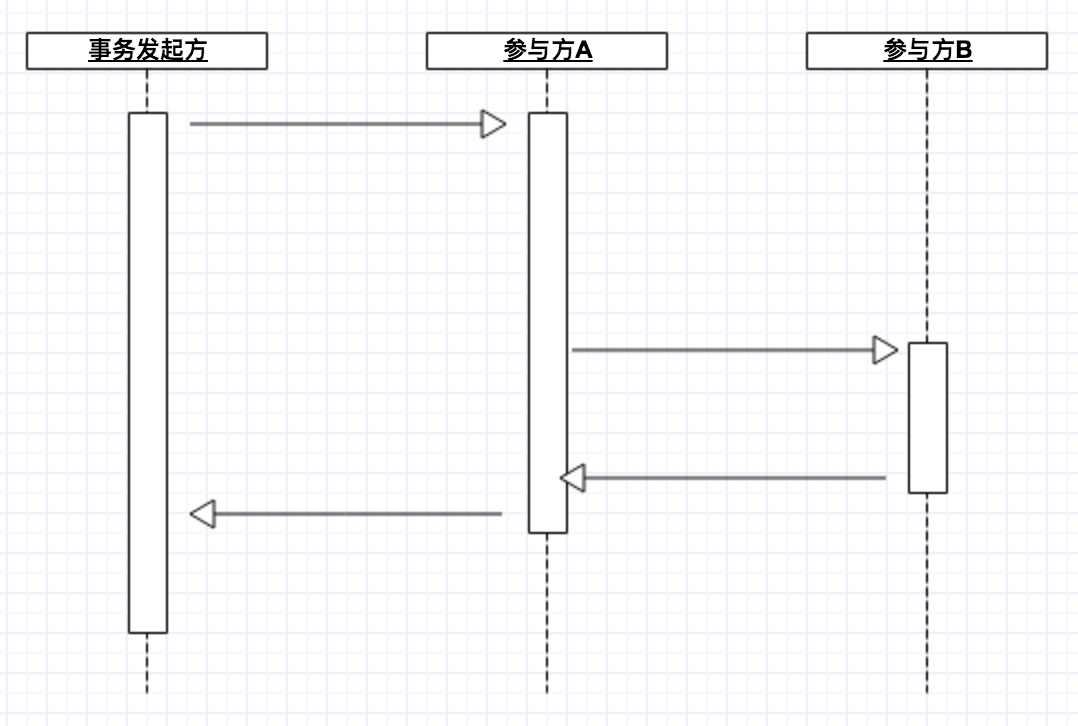
5、通讯协议

（1）

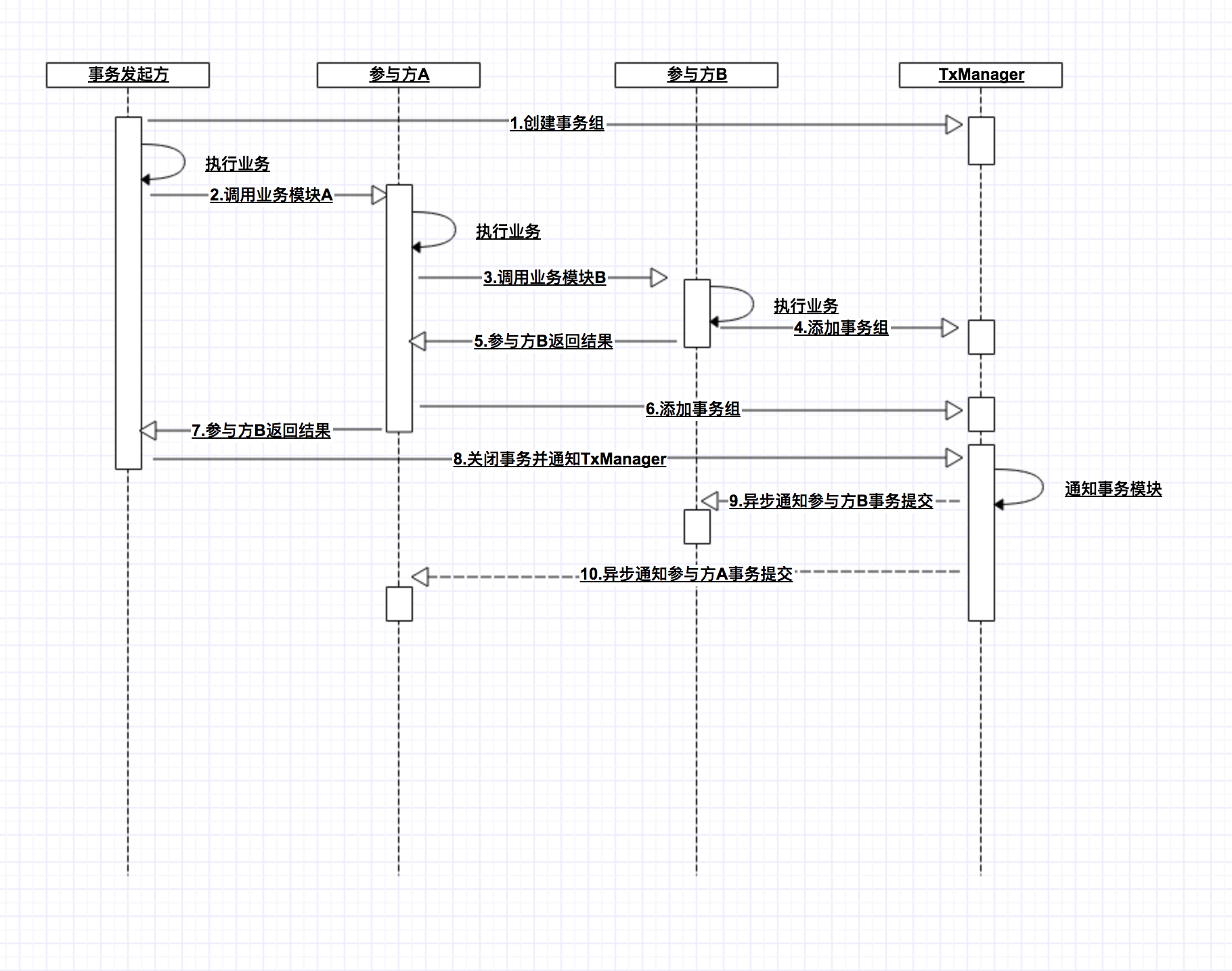
<https://github.com/codingapi/tx-lcn/wiki/%E9%80%9A%E8%AE%AF%E5%8D%8F%E8%AE%AE>

6、模拟场景

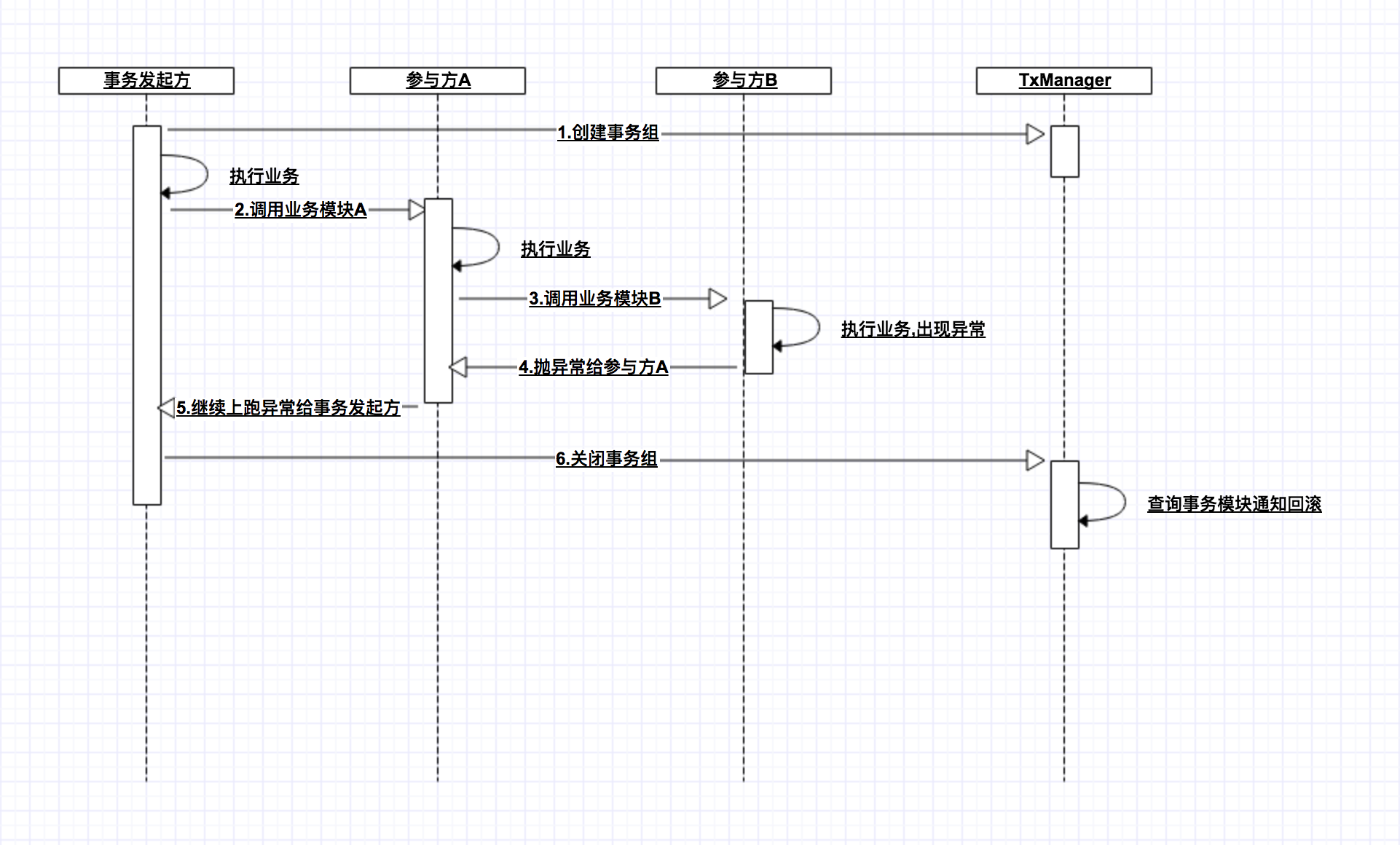
（1）若存在事务发起方、参与方A、参与方B。调用关系图如下



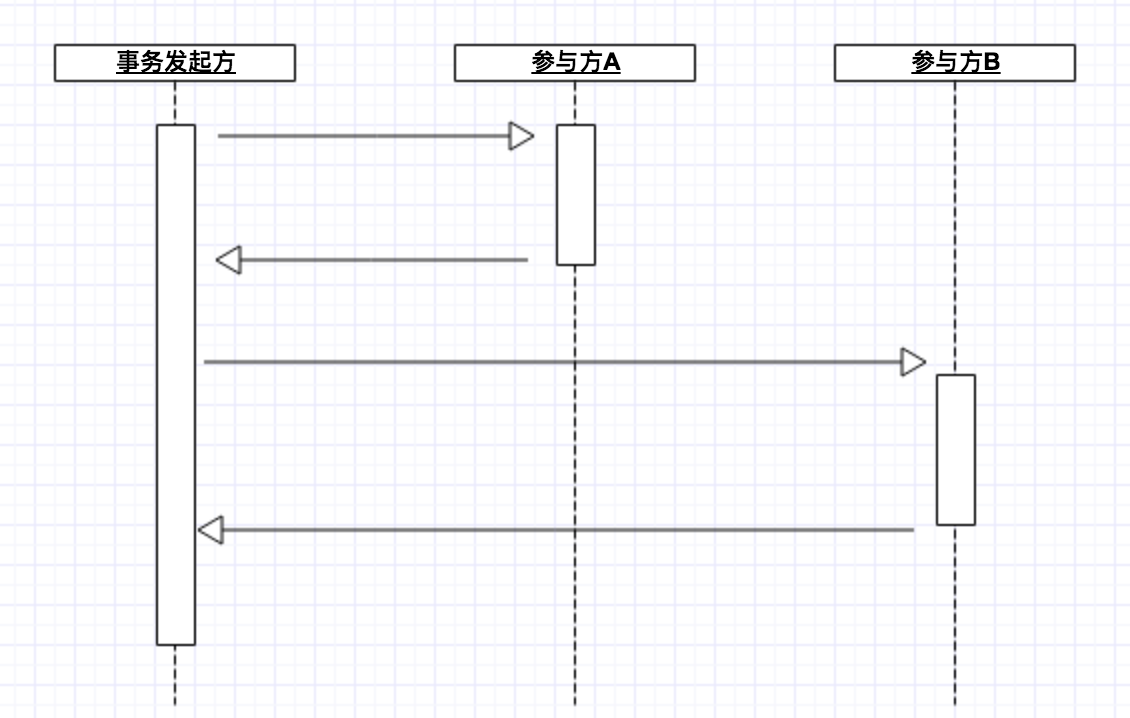
（2）那么他们正常执行业务的时序图为：



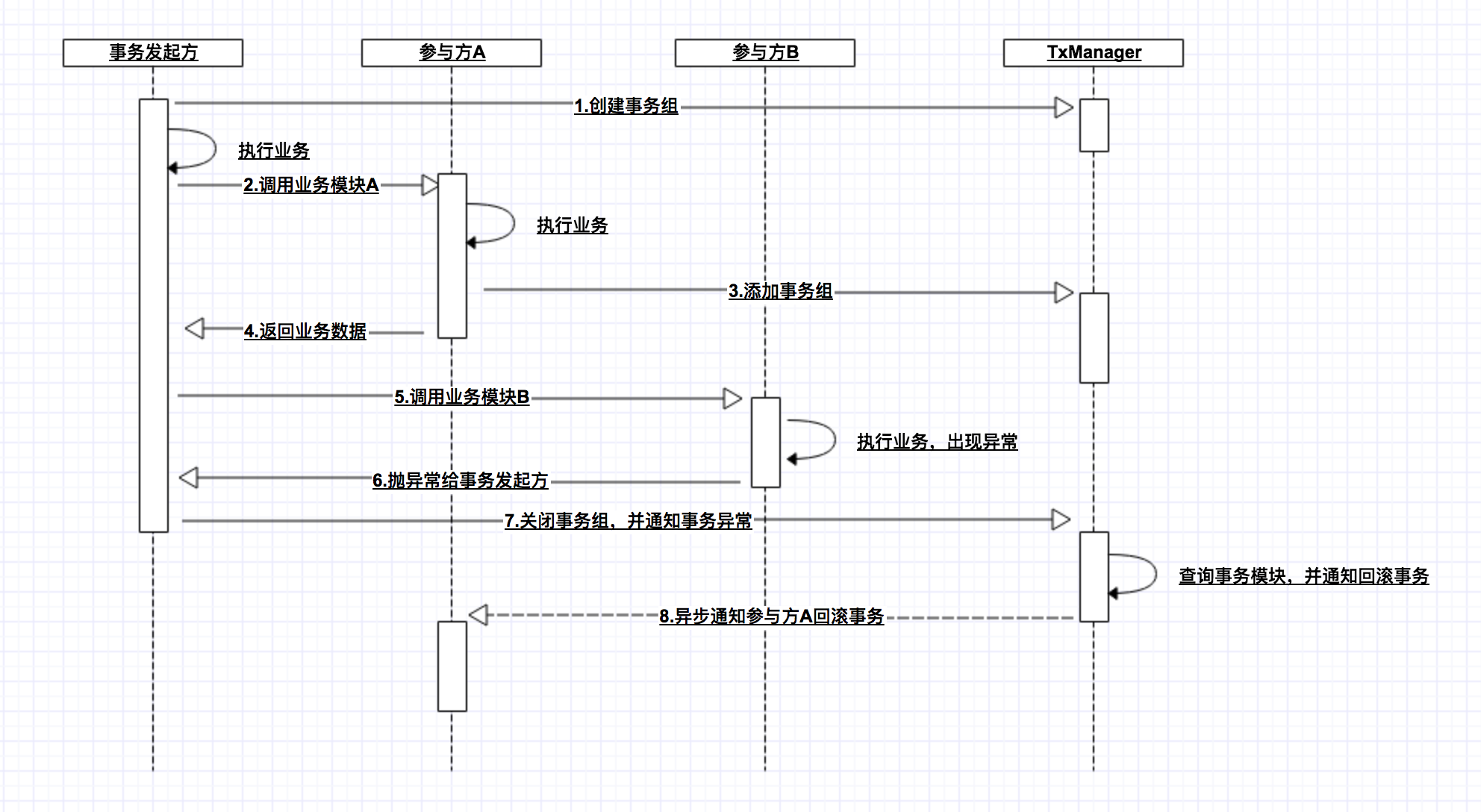
（3）若参与方B出现异常，那么他们的业务时序图为：



（4）若他们的调用关系是这样的情况



（5）此时发生参与方B出现异常时他们的时序图为：



7、LCN方案几点需要注意的问题：

（1）事务协调器可以进行集群部署。

（2）

五、两段式提交方案

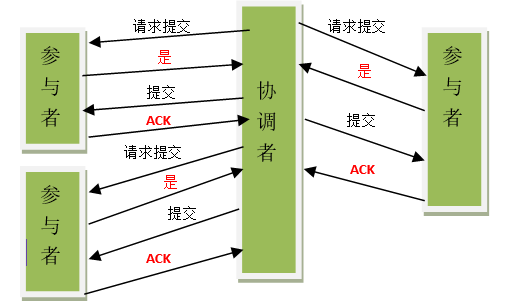
1、总体概括：协调者询问参与者是否准备好了提交，并根据所有参与者的反馈情况决定向所有参与者发送commit或者rollback指令。

2、两阶段具体步骤：

**准备阶段**：又称投票阶段。在这一阶段，协调者询问所有参与者是否准备好提交，参与者如果已经准备好提交则回复**Prepared**，否则回复**Non-Prepared**。

**提交阶段**：又称执行阶段。协调者如果在上一阶段收到所有参与者回复的**Prepared**，则在此阶段向所有参与者发送**commit**指令，所有参与者立即执行**commit**操作；否则协调者向所有参与者发送**rollback**指令，参与者立即执行**rollback**操作。

（1）提交阶段第一种情况



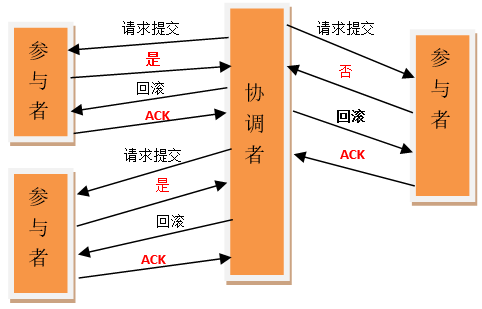
a、协调者节点向所有参与者节点发出“正式提交(commit)”的请求。

b、参与者节点正式完成操作，并释放在整个事务期间内占用的资源。

c、参与者节点向协调者节点发送“完成”消息。

d、协调者节点受到所有参与者节点反馈的“完成”消息后，完成事务。

（2）如果任一参与者节点在第一阶段返回的响应消息为“中止”，或者 协调者节点在第一阶段的询问超时之前无法获取所有参与者节点的响应消息时。



a、协调者节点向所有参与者节点发出“回滚操作(rollback)”的请求。

b、参与者节点利用之前写入的Undo信息执行回滚，并释放在整个事务期间内占用的资源。

c、参与者节点向协调者节点发送“回滚完成”消息。

d、协调者节点受到所有参与者节点反馈的“回滚完成”消息后，取消事务。

3、两段式提交存在的问题：

（1）同步阻塞问题。执行过程中，所有参与节点都是事务阻塞型的。当参与者占有公共资源时，其他第三方节点访问公共资源不得不处于阻塞状态。

（2）单点故障。由于协调者的重要性，一旦协调者发生故障。参与者会一直阻塞下去。尤其在第二阶段，协调者发生故障，那么所有的参与者还都处于锁定事务资源的状态中，而无法继续完成事务操作。（如果是协调者挂掉，可以重新选举一个协调者，但是无法解决因为协调者宕机导致的参与者处于阻塞状态的问题）

（3）数据不一致。在二阶段提交的阶段二中，当协调者向参与者发送commit请求之后，发生了局部网络异常或者在发送commit请求过程中协调者发生了故障，这回导致只有一部分参与者接受到了commit请求。而在这部分参与者接到commit请求之后就会执行commit操作。但是其他部分未接到commit请求的机器则无法执行事务提交。于是整个分布式系统便出现了数据部一致性的现象。

（4）二阶段无法解决的问题：协调者再发出commit消息之后宕机，而唯一接收到这条消息的参与者同时也宕机了。那么即使协调者通过选举协议产生了新的协调者，这条事务的状态也是不确定的，没人知道事务是否被已经提交。

**六：TCC**（原理、优缺点、最终一致性）

1. TCC原理图+流程图：<https://github.com/yu199195/hmily/wiki/Theory>
2. <https://github.com/liuyangming/ByteTCC/wiki/Developer-Guide>
3. TCC机制

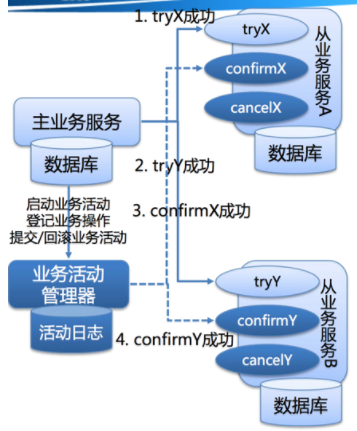
（1）Try：预留业务资源。（DML）

（2）Confirm：确认执行业务操作。（Commit）

（3）Cancel：取消执行业务操作。（Rollback）

在一个跨应用的业务操作中，Try操作是先把多个应用中的业务资源预留和锁定住，为后续的确认打下基础，类似的，DML操作要锁定数据库记录行，持有数据库资源；Confirm操作是在Try操作中涉及的所有应用均成功之后进行确认，使用预留的业务资源，和Commit类似；而Cancel则是当Try操作中涉及的所有应用没有全部成功，需要将已成功的应用进行取消(即Rollback回滚)。其中Confirm和Cancel操作是一对反向业务操作。

4、TCC原理图



（1）主业务服务：主业务服务为整个业务活动的发起方。从业务服务：从业务服务负责提供TCC业务操作，是整个业务活动的操作方。从业务服务必须实现Try、Confirm和Cancel三个接口，供主业务服务调用。由于Confirm和Cancel操作可能被重复调用，故要求Confirm和Cancel两个接口必须是幂等的。

（2）业务活动管理器：业务活动管理器管理控制整个业务活动，包括记录维护TCC全局事务的事务状态和每个从业务服务的子事务状态，并在业务活动提交时确认所有的TCC型操作的confirm操作，在业务活动取消时调用所有TCC型操作的cancel操作。

5、TCC优点和缺点

优点：TCC事务的优点如下：解决了跨应用业务操作的原子性问题，在诸如组合支付、账务拆分场景非常实用。TCC实际上把数据库层的二阶段提交上提到了应用层来实现，对于数据库来说是一阶段提交，规避了数据库层的2PC性能低下问题。

缺点：TCC的Try、Confirm和Cancel操作功能需业务提供，开发成本高。

6、TCC案例：

账务拆分的业务场景如下，分别位于三个不同分库的帐户A、B、C，A和B一起向C转帐共80元：

Try：尝试执行业务。完成所有业务检查(一致性)：检查A、B、C的帐户状态是否正常，帐户A的余额是否不少于30元，帐户B的余额是否不少于50元。预留必须业务资源(准隔离性)：帐户A的冻结金额增加30元，帐户B的冻结金额增加50元，这样就保证不会出现其他并发进程扣减了这两个帐户的余额而导致在后续的真正转帐操作过程中，帐户A和B的可用余额不够的情况。Confirm：确认执行业务。真正执行业务：如果Try阶段帐户A、B、C状态正常，且帐户A、B余额够用，则执行帐户A给账户C转账30元、帐户B给账户C转账50元的转帐操作。不做任何业务检查：这时已经不需要做业务检查，Try阶段已经完成了业务检查。只使用Try阶段预留的业务资源：只需要使用Try阶段帐户A和帐户B冻结的金额即可。Cancel：取消执行业务释放Try阶段预留的业务资源：如果Try阶段部分成功，比如帐户A的余额够用，且冻结相应金额成功，帐户B的余额不够而冻结失败，则需要对帐户A做Cancel操作，将帐户A被冻结的金额解冻掉。

7、要不要使用TCC:

（1）是否真正有保证跨应用业务操作的原子性需求。

（2）能否投入资源开发相对应的TCC接口。

（3）需要开发一个稳定的、高可用的、扩展性强的TCC事务管理器。

8、Try阶段成功；Confirm阶段出现问题；此时回滚事务，同时通过事务补偿的方式进行事务回滚或提交。

9、解决方案：以订单、库存、金额为例，库存减30、金额减50。

（1）订单表中增加订单状态的字段，分为四种状态：未支付、支付中、支付成功和支付失败。

（2）在订单形成的过程中，检查订单、库存、金额三个服务的状态是否正常，库存是否不少于30，金额是否不少于50。

a、库存表中有总数量字段和冻结数量字段。

b、资金表中有总金额和冻结金额字段。

c、订单表中保存新形成的订单信息。包括的字段有：订单号、订单状态、消费的库存数量、消费的金额数量。

（3）

**七、事务强一致性**

1、Atomikos优缺点：

（1）优点：XA协议简单，使用分布式事务成本低。

（2）缺点：与本地事务相比性能不理想，不能满足高并发的场景，许多nosql数据库不支持XA协议。

2、java平台提供的开源事务管理器。

3、Atomikos主要功能

（1）兼容标准的JTA API。

（2）支持嵌套事务。

（3）为XA和非XA提供内置的JDBC适配器。

4、能够保证数据的强一致性。

5、XA流程，通过日志来记录操作。从上到下，哪一部发生错误都会回滚。

