[**https://blog.csdn.net/zhou280648461/article/details/78918747**](https://blog.csdn.net/zhou280648461/article/details/78918747)

**LCN方案测试(LCN4.0)**

|  |  |
| --- | --- |
| 测试服务器 | 192.168.1.104，192.168.1.101 |
| 启动服务 | springcloudEureka注册中心，TXManager分布式事务调度中心，mybatisDemo1，mybatisDemo2。 |
| 部署情况 | 192.168.1.104(springcloudEureka注册中心，TXManager分布式事务调度中心，mybatisDemo1)；192.168.1.101(mybatisDemo2)。 |
| 测试内容 | 在分布式调用中：事务提交，程序报错事务回滚，网络中断事务回滚，服务停止事务回滚。 |
| 调用模式 | mybatisDemo1在自行存储后远程调用mybatisDemo2的存储业务。 |
| 其他要求 | Redis用于存储事务组。 |
| 基本原理 | socket通知，任务分组式的事务提交，基于aop实现 |
| 测试结果 | 分布式事务提交和回滚均可以实现 |

**1.LCN**

LCN分布式事务框架其本身并不创建事务，而是基于对本地事务的协调从而达到事务一致性的效果。

<https://github.com/codingapi/tx-lcn>

<https://github.com/codingapi/tx-lcn/wiki/LCN%E5%8E%9F%E7%90%86>

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。

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。

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。

事务控制原理

LCN事务控制原理是由事务模块TxClient下的代理连接池与TxManager的协调配合完成的事务协调控制。

TxClient的代理连接池实现了javax.sql.DataSource接口，并重写了close方法，事务模块在提交关闭以后TxClient连接池将执行"假关闭"操作，等待TxManager协调完成事务以后在关闭连接。

事务补偿机制：

事务补偿是指在执行某个业务方法时，本应该执行成功的操作却因为服务器挂机或者网络抖动等问题导致事务没有正常提交，此种场景就需要通过补偿来完成事务，从而达到事务的一致性。

补偿机制的触发条件：

当执行关闭事务组步骤时，若发起方接受到失败的状态后将会把该次事务识别为待补偿事务，然后发起方将该次事务数据异步通知给TxManager。TxManager接受到补偿事务以后先通知补偿回调地址，然后再根据是否开启自动补偿事务状态来补偿或保存该次切面事务数据。

补偿事务机制

LCN的补偿事务原理是模拟上次失败事务的请求，然后传递给TxClient模块然后再次执行该次请求事务。

**2.TCC（Try-Confirm-Cancel）**

Try：预留业务资源

Confirm：确认执行业务操作

Cancel：取消执行业务操作

是基于补偿型事务的AP系统的一种实现, 具有最终一致性.

关系型数据库事务的三种操作：

DML、Commit和Rollback，会发现和TCC有异曲同工之妙。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。

简而言之，TCC是应用层的2PC(2 Phase Commit, 两阶段提交)，如果你将应用看做资源管理器的话。

详细来说，TCC每项操作需要做的事情如下：

1、Try：尝试执行业务。完成所有业务检查(一致性)预留必须业务资源(准隔离性)

2、Confirm：确认执行业务。真正执行业务不做任何业务检查只使用Try阶段预留的业务资源

3、Cancel：取消执行业务。释放Try阶段预留的业务资源

一个完整的TCC事务参与方包括三部分：

主业务服务：

主业务服务为整个业务活动的发起方，如前面提到的组合支付场景，支付系统即是主业务服务。

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。前面的组合支付场景中的余额系统和红包系统即为从业务服务。

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。

可见整个TCC事务对于主业务服务来说是透明的，其中业务活动管理器和从业务服务各自干了一部分工作。

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。

TCC事务的缺点：

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

案例分析：

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：

1、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。

2、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。

3、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。