分布式事务

44- 50：50-01：05：06

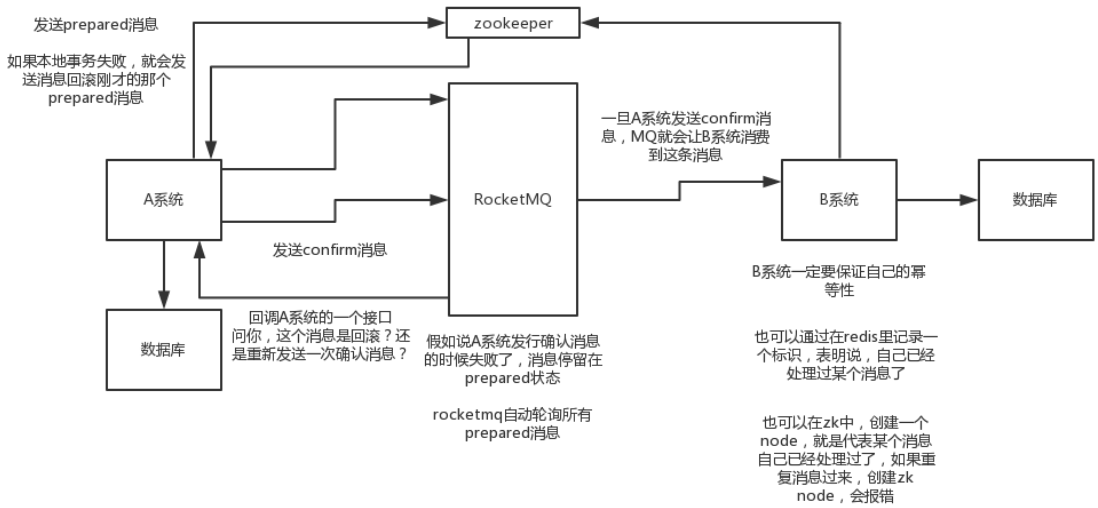
##

可靠消息最终一致性方案

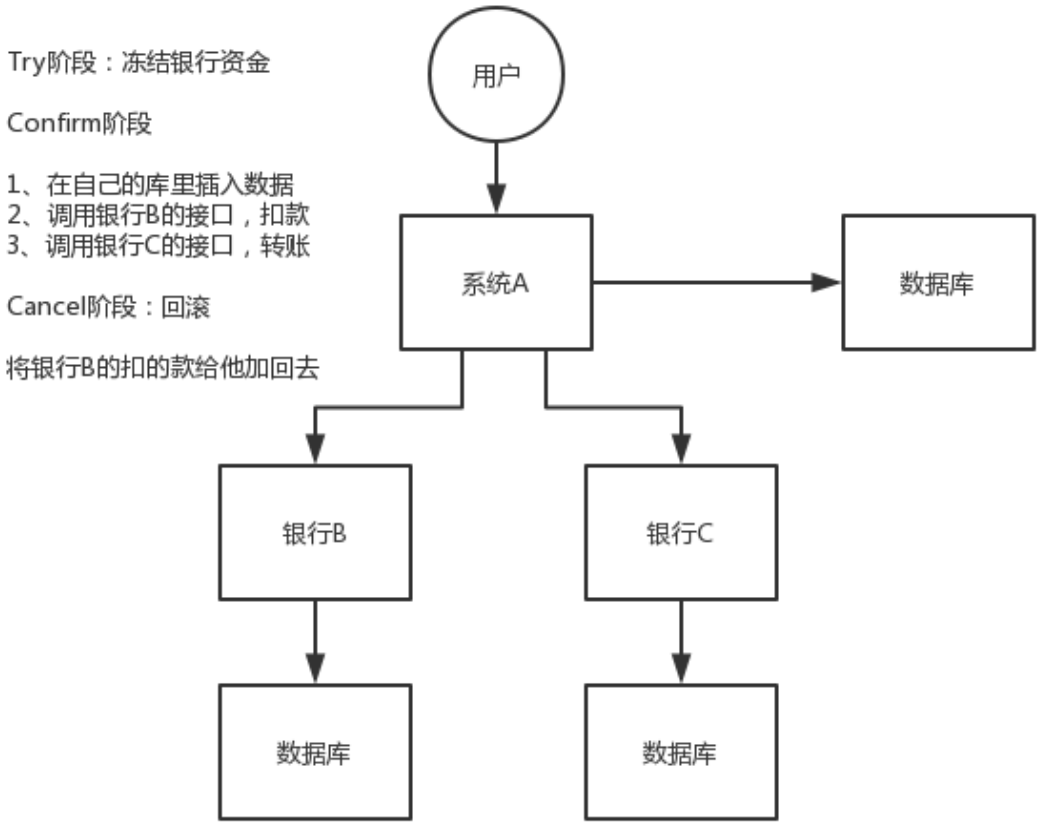
这个的意思，就是干脆不要用本地的消息表了，直接基于MQ来实现事务。比如阿里的RocketMQ就支持消息事务。

大概的意思就是：
1）A系统先发送一个prepared消息到mq，如果这个prepared消息发送失败那么就直接取消操作别执行了
2）如果这个消息发送成功过了，那么接着执行本地事务，如果成功就告诉mq发送确认消息，如果失败就告诉mq回滚消息
3）如果发送了确认消息，那么此时B系统会接收到确认消息，然后执行本地的事务
4）mq会自动定时轮询所有prepared消息回调你的接口，问你，这个消息是不是本地事务处理失败了，所有没发送确认消息？那是继续重试还是回滚？一般来说这里你就可以查下数据库看之前本地事务是否执行，如果回滚了，那么这里也回滚吧。这个就是避免可能本地事务执行成功了，别确认消息发送失败了。
5）这个方案里，要是系统B的事务失败了咋办？重试咯，自动不断重试直到成功，如果实在是不行，要么就是针对重要的资金类业务进行回滚，比如B系统本地回滚后，想办法通知系统A也回滚；或者是发送报警由人工来手工回滚和补偿

这个还是比较合适的，目前国内互联网公司大都是这么玩儿的，要不你举用RocketMQ支持的，要不你就自己基于类似ActiveMQ？RabbitMQ？自己封装一套类似的逻辑出来，总之思路就是这样子的



TCC



（2）TCC方案

TCC的全程是：Try、Confirm、Cancel。

这个其实是用到了补偿的概念，分为了三个阶段：

1）Try阶段：这个阶段说的是对各个服务的资源做检测以及对资源进行锁定或者预留
2）Confirm阶段：这个阶段说的是在各个服务中执行实际的操作
3）Cancel阶段：如果任何一个服务的业务方法执行出错，那么这里就需要进行补偿，就是执行已经执行成功的业务逻辑的回滚操作

给大家举个例子吧，比如说跨银行转账的时候，要涉及到两个银行的分布式事务，如果用TCC方案来实现，思路是这样的：

1）Try阶段：先把两个银行账户中的资金给它冻结住就不让操作了
2）Confirm阶段：执行实际的转账操作，A银行账户的资金扣减，B银行账户的资金增加
3）Cancel阶段：如果任何一个银行的操作执行失败，那么就需要回滚进行补偿，就是比如A银行账户如果已经扣减了，但是B银行账户资金增加失败了，那么就得把A银行账户资金给加回去

这种方案说实话几乎很少用人使用，我们用的也比较少，但是也有使用的场景。因为这个事务回滚实际上是严重依赖于你自己写代码来回滚和补偿了，会造成补偿代码巨大，非常之恶心。

比如说我们，一般来说跟钱相关的，跟钱打交道的，支付、交易相关的场景，我们会用TCC，严格严格保证分布式事务要么全部成功，要么全部自动回滚，严格保证资金的正确性，在资金上出现问题

比较适合的场景：这个就是除非你是真的一致性要求太高，是你系统中核心之核心的场景，比如常见的就是资金类的场景，那你可以用TCC方案了，自己编写大量的业务逻辑，自己判断一个事务中的各个环节是否ok，不ok就执行补偿/回滚代码。