分布式事务解决方案

2PC

3PC

TCC 补偿事务

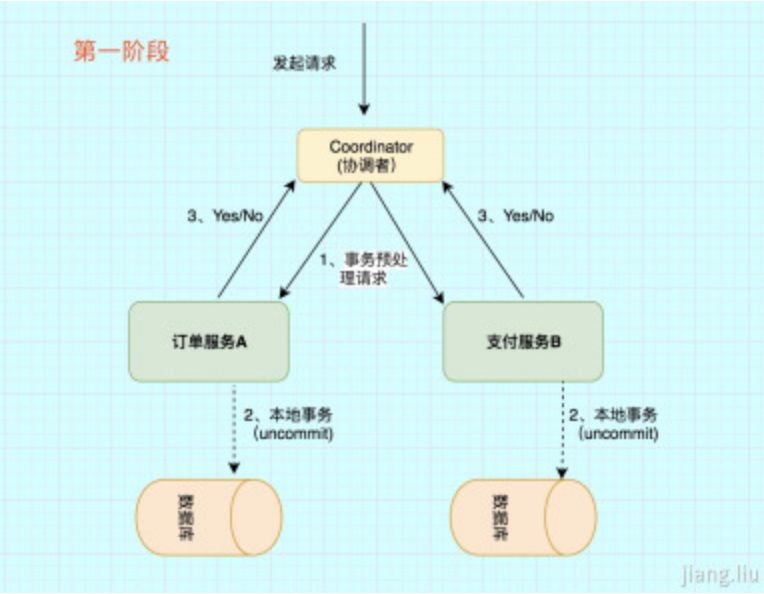
本地消息表

可靠消息最终一致性

最大努力通知方案

2PC

两个阶段：第一阶段：**准备**阶段　第二阶段：**提交/执行**阶段。



第一阶段：**准备阶段**

1）协调者 向所有的 **参与者** 发送事务预处理请求

2）各个 **参与者** 执行本地事务，执行完成后并不会真正提交本地事务，而是先向 **协调者** 报告是否可以成功执行事务。

3）如果 **参与者** 成功执行了事务操作,那么就反馈给协调者 **Yes** 响应, 表示事务可以执行,如果没有 **参与者** 成功执行事务,那么就反馈给协调者 **No** 响应,表示事务不可以执行。

第二阶段: **提交/执行阶段**

1）如果所有参与者都返回Yes，那 **协调者** 发送commit请求，**参与者** 收到commit请求后提交事务。

2）任何一个 **参与者** 向 **协调者** 反馈了 **No** 响应，或者协调者等待超时。**协调者** 向所有参与者发出 **RoollBack** 回滚请求。参与者回滚事务。

优点：强一致性。
缺点：同步阻塞（每一个节点内部是事务阻塞的），单点故障会导致程序阻塞。数据无法100%一致。（发送commit之后一部分参与者宕机，导致无法全部commit）

解决方案： **基于数据库的XA协议来实现、Seata实现2PC事务**

3PC

3PC的一个场景：

> 组织者：小A，我们想在晚上8点五黑，你有时间嘛？有时间你就说YES，没有你就说NO，然后我还会再去问其他人，这段时间你可先去干你自己的事儿，不用一直等着我。
>
> 小A：好的，我有时间。
>
> 组织者：小B、小C、小D，我们想定在晚上8点王者荣耀五黑……不用一直等我。
>
> 组织者收集完情况了，一看大家都有时间，那么就再次通知大家。（协调者接收到所有YES指令）
>
> 组织者：小A，我跟其他人确认过了大家都可以，你要把段时间空出来，你不能再安排其他的事儿了。
>
> 小A顺手设置了晚上8点闹钟，然后跟组织者说，我可以去。
>
> 组织者：小B，我们决定了晚上8点王者荣耀五黑……你就8点上号就行了。
>
> 组织者通知完一圈之后。所有朋友都跟他说：”我已经把8点这个时间段空出来了”。于是，他在8点的时候邀请A、B、C、D加入游戏。

三阶段提交协议（3PC）主要是为了解决两阶段提交协议的阻塞问题。

- 同时在协调者和参与者中都引入超时机制。这个优化点，主要是避免了 **参与者** 在长时间无法与 协调者 通讯（协调者挂掉了）的情况下，无法释放资源的问题，因为参与者自身拥有超时机制会在超时后，自动进行本地commit从而进行释放资源。
- 在第一阶段和第二阶段中插入一个准备阶段。保证了在最后提交阶段之前各参与节点的状态是一致的。

3PC其实是把2PC的准备阶段再次一分为二。

这样三阶段提交就有CanCommit（**事务询问**）、PreCommit（**事务执行**）、DoCommit（**事务提交**）三个阶段。

在第一阶段，只是询问所有参与者是否可以执行事务操作，并不在本阶段执行事务操作。当协调者收到所有的参与者都返回YES时，在第二阶段才执行事务操作，然后在第三阶段在执行commit或者rollback。

优点：**降低同步阻塞**（第一阶段没有事务操作，不会阻塞其他动作）、**提升了数据一致性**（在3PC中，参与者没有收到协调者的消息时，超时之后会自动执行事务）

缺点：无法解决数据一致性问题（参与者超时之后自动提交，万一其他节点回滚了呢）。

TCC 补偿事务

TCC（Try-Confirm-Cancel）又称补偿事务。其核心思想是："针对每个操作都要注册一个与其对应的确认和补偿（撤销操作）"。它分为三个操作：

- Try阶段：主要是对业务系统做检测及资源预留。
- Confirm阶段：确认执行业务操作。
- Cancel阶段：取消执行业务操作。

TCC处理流程与2PC类似，不过2PC通常都是在**跨数据库操作层面**，而TCC是在业务层面，需要通过业务逻辑来实现。

TCC优势在于，可以让**应用自己定义数据库操作的粒度，使得降低锁冲突、提高吞吐量成为可能**。

不足之处则在于**对应用的侵入性非常强**，业务逻辑的每个分支都需要实现try、confirm、cancel三个操作。

> TCC 分布式事务框架，比如国内开源的 ByteTCC、Himly、TCC-transaction。

本地消息表

1）A系统在自己本地事务里操作的同时，插入一条数据到消息表。

2）接着A系统将这个消息发送到MQ中去。

3）B系统接收到消息之后，在一个事务里，往自己本地消息表里插入一条数据，同时执行其他的业务操作，如果这个消息已经被处理过了，那么此时这个事务会回滚，这样保证不会重复处理消息。
4）B系统执行成功之后，就会更新自己本地消息表的状态以及A系统消息表的状态。
5）如果B系统处理失败了，那么就不会更新消息表状态，那么此时A系统会定时扫描自己的消息表，如果有没处理的消息，会再次发送到MQ中去，让B再次处理。
6）这个方案保证了最终一致性，哪怕B事务失败了，但是A会不断重发消息，直到B那边成功为止。

可靠消息最终一致性

基于MQ来实现事务：

RocketMQ 就很好的支持了消息事务，让我们来看一下如何通过消息实现事务。

第一步先给 Broker 发送事务消息即半消息，**半消息不是说一半消息，而是这个消息对消费者来说不可见**，然后**发送成功后发送方再执行本地事务。**

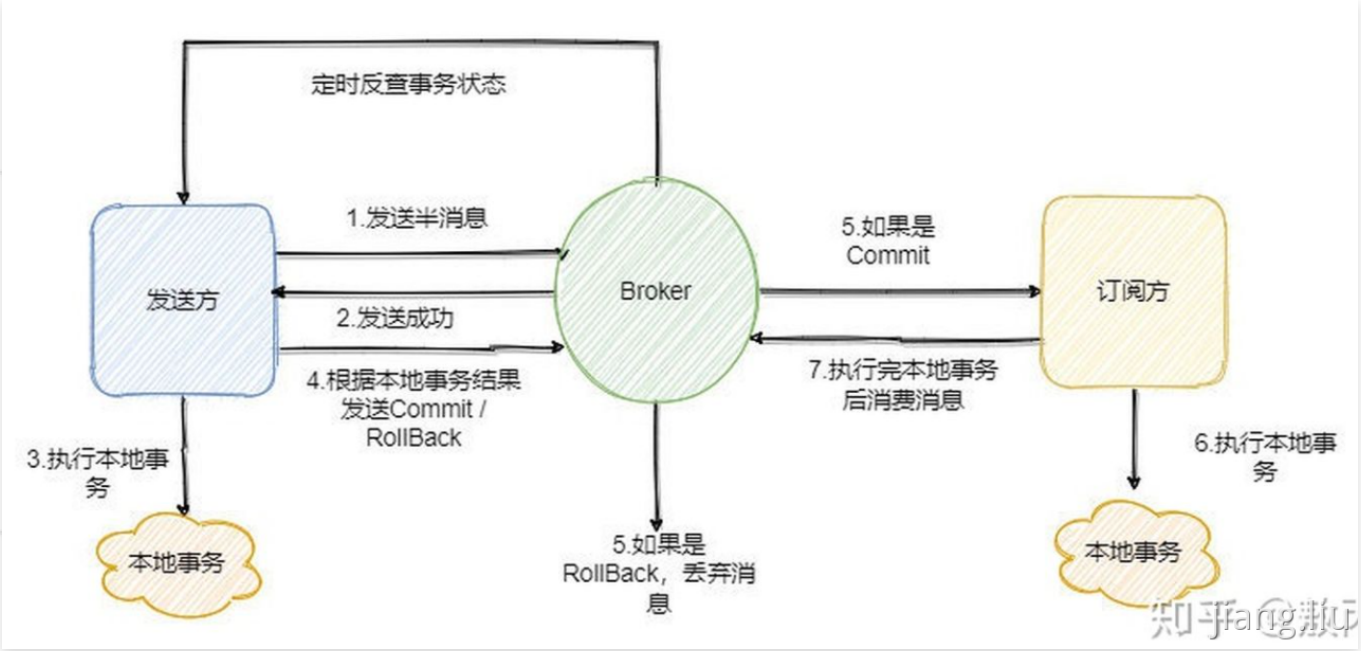
再根据**本地事务的结果向 Broker 发送 Commit 或者 RollBack 命令。**

并且 RocketMQ 的发送方会提供一个**反查事务状态接口**，如果一段时间内半消息没有收到任何操作请求，那么 Broker 会通过反查接口得知发送方事务是否执行成功，然后执行 Commit 或者 RollBack 命令。

如果是 Commit 那么订阅方就能收到这条消息，然后再做对应的操作，做完了之后再消费这条消息即可。

如果是 RollBack 那么订阅方收不到这条消息，等于事务就没执行过。

可以看到通过 RocketMQ 还是比较容易实现的，RocketMQ 提供了事务消息的功能，我们只需要定义好事务反查接口即可。



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，要不你就用RocketMQ支持的事务，要不你就自己基于该原理在 ActiveMQ 或者 RabbitMQ 自己封装一套类似的逻辑出来，总之思路就是这样子的。

3.2.6之前的版本有上面的所有机制，之后的版本砍掉了好多东西。

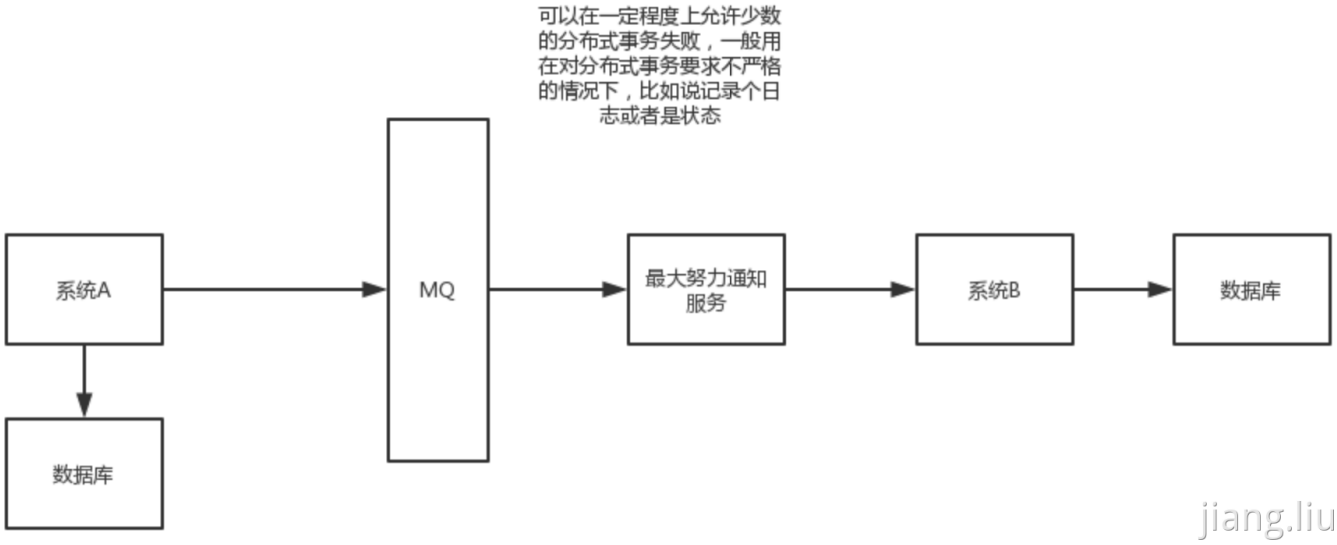
最大努力通知方案

这个方案的大致意思就是：

1）系统A本地事务执行完之后，发送个消息到MQ。

2）这里会有个专门消费MQ的最大努力通知服务，这个服务会消费MQ然后写入数据库中记录下来，或者是放入个内存队列也可以，接着调用系统B的接口。

3）要是系统B执行成功就ok了；要是系统B执行失败了，那么最大努力通知服务就定时尝试重新调用系统B，反复N次，最后还是不行就放弃。


可以在一定程度上允许少数的分布式事务失败，一般用在对分布式事务要求不严格的情况下，比如说记录个日志或者是状态
系统A
MQ
最大努力通知服务
系统B
数据库
数据库
jiang.liu

该方案用的比较少。