链滴

人生不相见，动如参与商。 今夕复何夕，共此灯烛光。

作者：VoidCup

原文链接：https://ld246.com/article/1565248857710

来源网站：链滴

分布式事物

事物介绍

数据库事物(Transaction)是指数据库执行过程中的一个逻辑单元，由一个有限的数据库操作序列构成。

事物的四个特性（ACID）

● 原子性（Atomicity）：整个事物中的所有操作，要么成功要么失败；不存在一个中间的连续过程，像电子跃迁.

● 一致性（Consistency）：一个事物可以封装状态改变（除非他是一个只读的）。事物必须始终保系统处于一致的状态，不管在任何给定的时间并发事物有多少。就像能量守恒定理，在一个封闭系统能量可以从一种形式转为另一种，但总能量保持不变；

● 隔离性（Isolation）：多个事物并发执行时，一个事物的执行不应该影响其他事物的执行，如同只这一个操作在被数据库执行一样；真实情况中存在隔离级别，不同的隔离级别对其他事物有不同程度影响；

● 持久性（Durability）：**已被提交的事物**对数据库的修改应该永久保存在数据库中；在事物结束，此操作不可逆转；

本地事物

倘若将一个单一的服务操作作为事物，那么整个服务操作只能涉及一个单一的数据库资源，这类基于个服务单一数据库资源访问的事物，被称为本地事物（Local Transaction）；

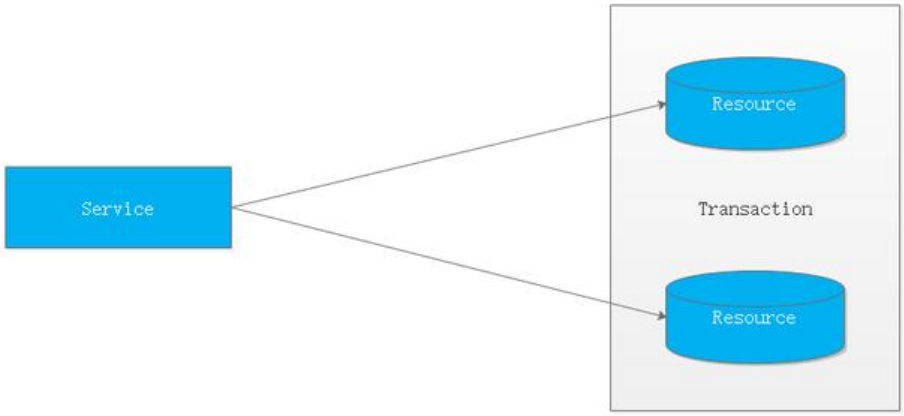
分布式事物

分布式事务指事务的参与者、支持事务的服务器、资源服务器以及事务管理器分别位于不同的分布式统的不同节点之上,且属于不同的应用，分布式事务需要保证这些操作要么全部成功，要么全部失败。

质上来说，分布式事务就是为了保证不同数据库的数据一致性 "

单一分布式事物

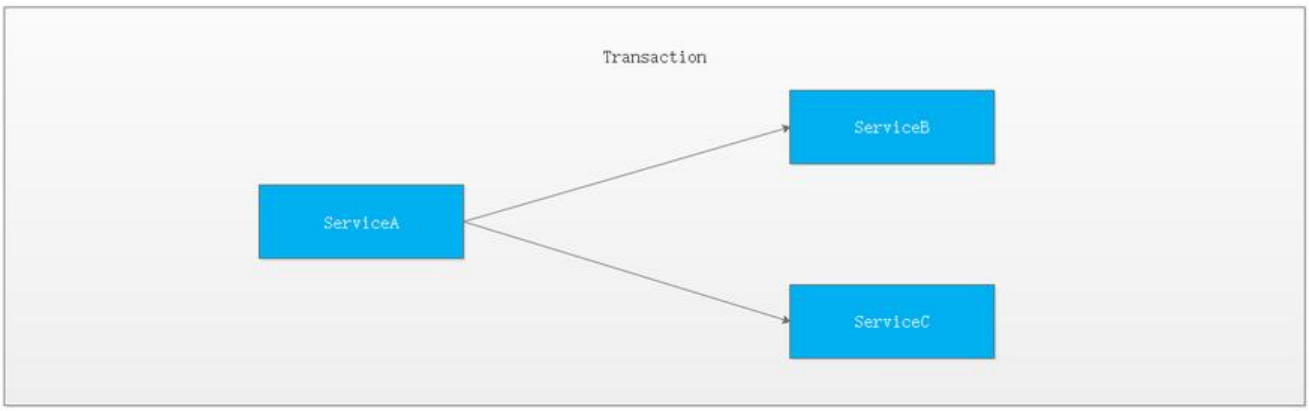
最早的分布式事务应用架构很简单，不涉及服务间的访问调用，仅仅是服务内操作涉及到对多个数据资源的访问。



多服务分布式事物

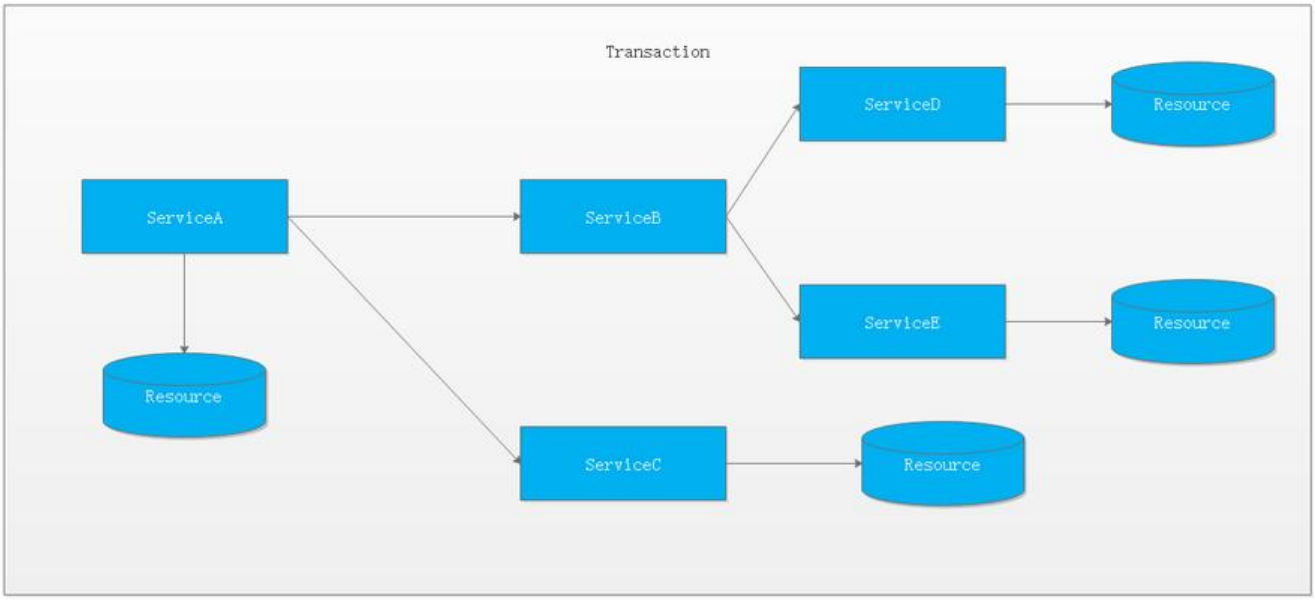
当一个服务操作访问不同的数据库资源，又希望对它们的访问具有事务特性时，就需要采用分布式事来协调所有的事务参与者。

对于上面介绍的分布式事务应用架构，尽管一个服务操作会访问多个数据库资源，但是毕竟整个事务是控制在单一服务的内部。如果一个服务操作需要调用另外一个服务，这时的事务就需要跨越多个服了。在这种情况下，起始于某个服务的事务在调用另外一个服务的时候，需要以某种机制流转到另外个服务，从而使被调用的服务访问的资源也自动加入到该事务当中来。下图反映了这样一个跨越多个务的分布式事务：



多服务多数据源分布式事物

如果将上面这两种场景(一个服务可以调用多个数据库资源，也可以调用其他服务)结合在一起，对此行延伸，整个分布式事务的参与者将会组成如下图所示的树形拓扑结构。在一个跨服务的分布式事务，事务的发起者和提交均系同一个，它可以是整个调用的客户端，也可以是客户端最先调用的那个服。



较之基于单一数据库资源访问的本地事务，分布式事务的应用架构更为复杂。在不同的分布式应用架下，实现一个分布式事务要考虑的问题并不完全一样，比如对多资源的协调、事务的跨服务传播等，现机制也是复杂多变。

CAP定理

1998年，加州大学的计算机科学家 Eric Brewer(布鲁尔) 提出，分布式系统有三个指标。

核心思想是：在分布式系统中一致性(C)、 可用性(A)、分区容错性(P)，三者不可兼得。

Consistency。（一致性）：在分布式系统中的所有数据备份，在同一时刻是否同样的值（等同于所节点访问同一份最新的数据副本）。

Availability（可用性）：在集群中一部分节点故障后，集群整体是否还能满足响应客户端的读写请。（对数据更新具备高可用性）。

Partition tolerance（分区容错性）：以实际效果而言，分区相当于对通信的时限要求。系统如果能在时限内达成数据一致性，就意味着发生了分区的情况，必须就当前操作在C和A之间做出选择；

CAP理论就是说在分布式存储系统中，最多只能实现上面的两点。**而由于当前的网络硬件肯定会出现迟丢包等问题，所以分区容错性是我们必须需要实现的。所以我们只能在**
致性和**可用性**之间**进行权衡。**

BASE理论

CAP不可能同时满足，而分区容错是对于分布式系统而言，是必须的。Base 理论是对 CAP 中一致性可用性权衡的结果，其来源于对大型互联网分布式实践的总结，是基于 CAP 定理逐步演化而来的，由 ebay 的架构师提出。

全称为：

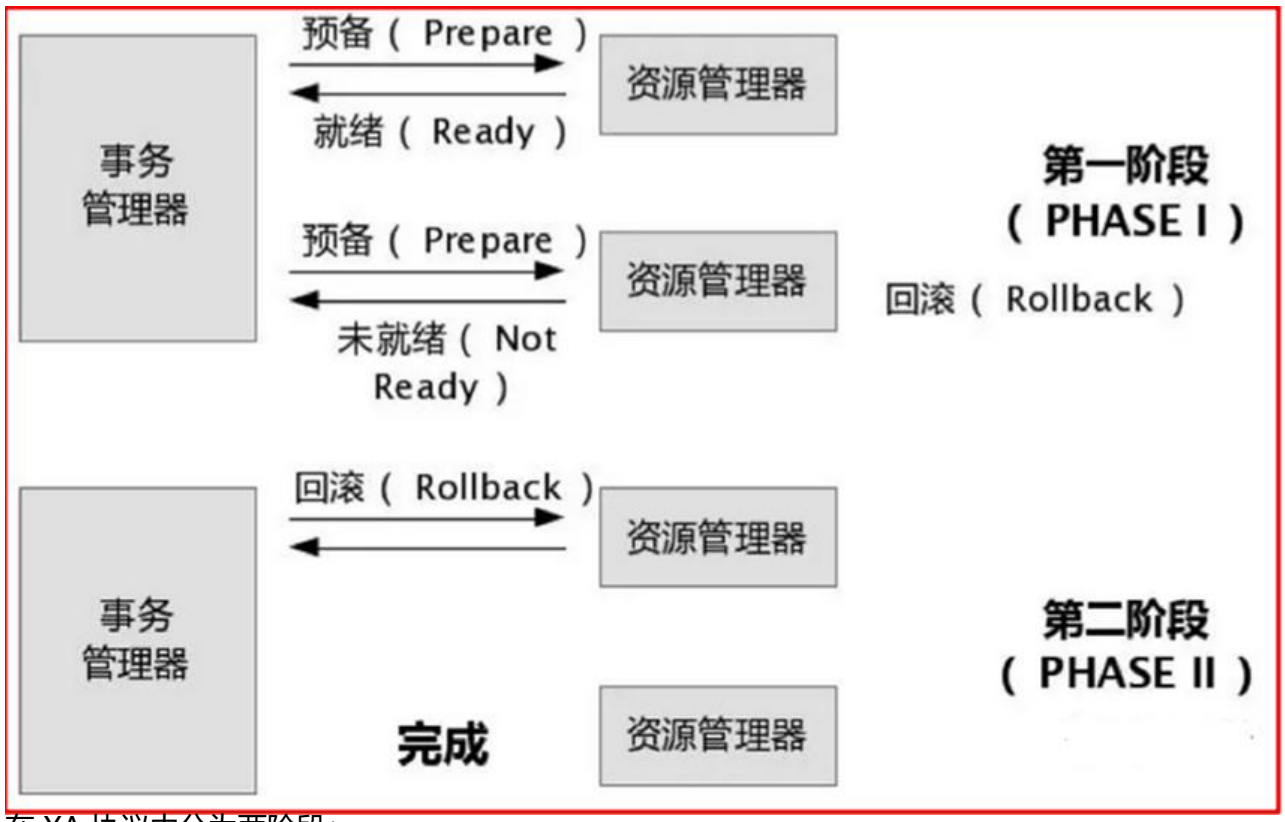
1. Basically Available（基本可用）
2. Soft state（软状态）
3. Eventually consistent（最终一致性）

BASE理论是对CAP中的一致性和可用性进行一个权衡的结果，理论的核心思想就是：即使无法做到一致性（Strong consistency），但每个应用都可以根据自身的业务特点，采用适当的方式来使系统到最终一致性（Eventual consistency）。

分布式事物解决方案

基于XA协议的两段提交

XA是一个分布式事务协议，由Tuxedo(分布式操作扩展之后的Unix事务系统)提出。XA中大致分为两分：事务管理器和本地资源管理器。其中本地资源管理器往往由数据库实现，比如Oracle、DB2这些业数据库都实现了XA接口，而事务管理器作为全局的调度者，负责各个本地资源的提交和回滚。XA现分布式事务的原理如下：



在 XA 协议中分为两阶段：

事务管理器要求每个涉及到事务的数据库预提交(precommit)此操作，并反映是否可以提交。

事务管理器要求每个数据库提交数据，或者回滚数据。

优点： 尽量保证了数据的强一致，实现成本较低，在各大主流数据库都有自己实现，对于 MySQL 是 5.5 开始支持。

缺点：

单点问题：事务管理器在整个流程中扮演的角色很关键，如果其宕机，比如在第一阶段已经完成，在二阶段正准备提交的时候事务管理器宕机，资源管理器就会一直阻塞，导致数据库无法使用。

同步阻塞：在准备就绪之后，资源管理器中的资源一直处于阻塞，直到提交完成，释放资源。

数据不一致：两阶段提交协议虽然为分布式数据强一致性所设计，但仍然存在数据不一致性的可能。如在第二阶段中，假设管理者发出了事务 Commit 的通知，但是因为网络问题该通知仅被一部分参者所收到并执行了 Commit 操作，其余的参与者则因为没有收到通知一直处于阻塞状态，这时候就生了数据的不一致性。

总的来说，XA 协议比较简单，成本较低，但是其单点问题，以及不能支持高并发(由于同步阻塞)依然其最大的弱点。

TCC（补偿事务）

关于 TCC（Try-Confirm-Cancel）的概念，最早是由 Pat Helland 于 2007 年发表的一篇名为《Life eyond Distributed Transactions：an Apostate’ s Opinion》的论文提出。

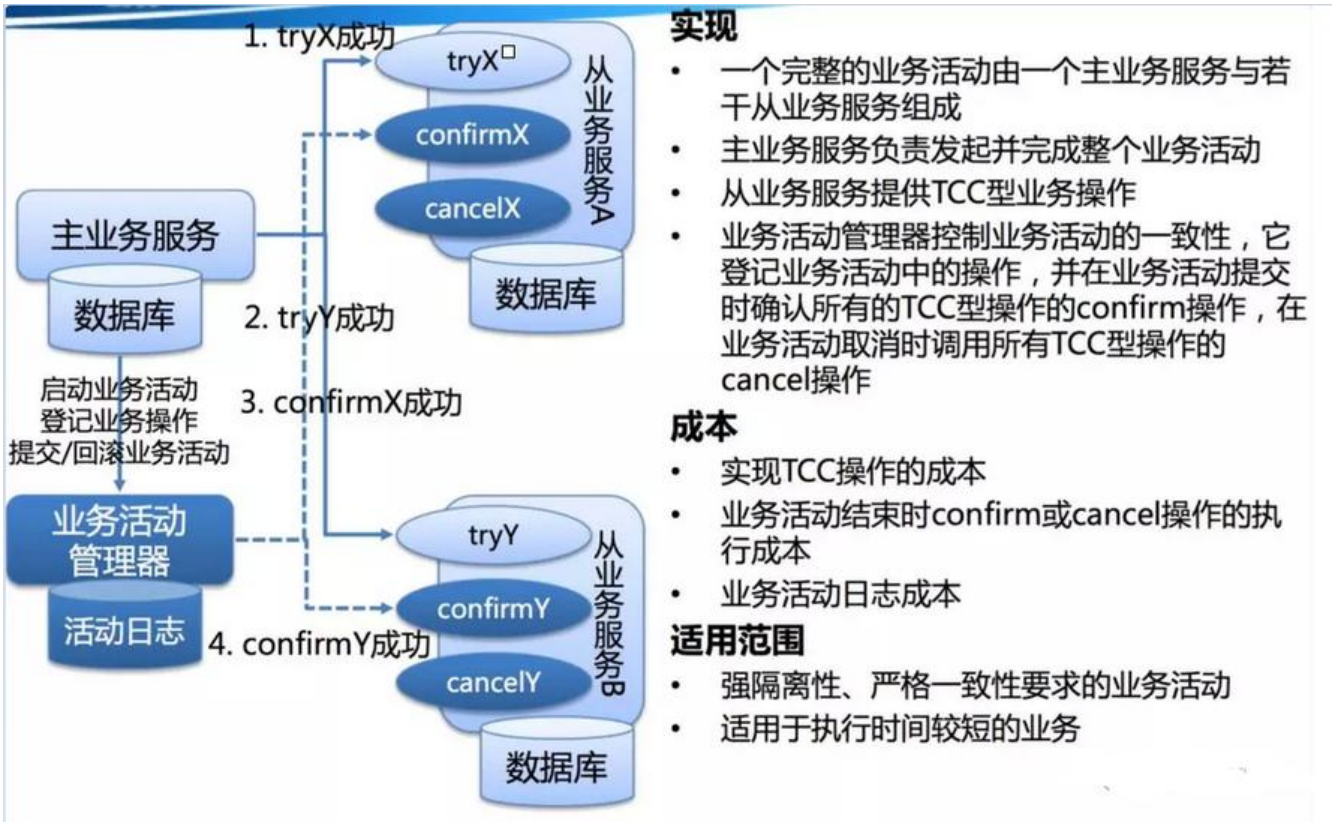
TCC就是提供了一个编程框架，将整个业务逻辑分为三块：Try、Confirm和Cancel三个操作。其核思想是针对每一个操作，都要注册一个与其对应的确认和补偿（撤销）操作。总之，TCC就是通过代人为实现了两阶段提交，不同的业务场景所写的代码都不一样，复杂度也不一样。

TCC 事务机制相比于上面介绍的 XA，解决了如下几个缺点：

解决了协调者单点，由主业务方发起并完成这个业务活动。业务活动管理器也变成多点，引入集群。

同步阻塞：引入超时，超时后进行补偿，并且不会锁定整个资源，将资源转换为业务逻辑形式，粒度小。

数据一致性，有了补偿机制之后，由业务活动管理器控制一致性。



对于 TCC 的解释：

Try 阶段：尝试执行，完成所有业务检查（一致性），预留必需业务资源（准隔离性）。

Confirm 阶段：确认真正执行业务，不作任何业务检查，只使用 Try 阶段预留的业务资源，Confirm

作满足幂等性。要求具备幂等设计，Confirm 失败后需要进行重试。

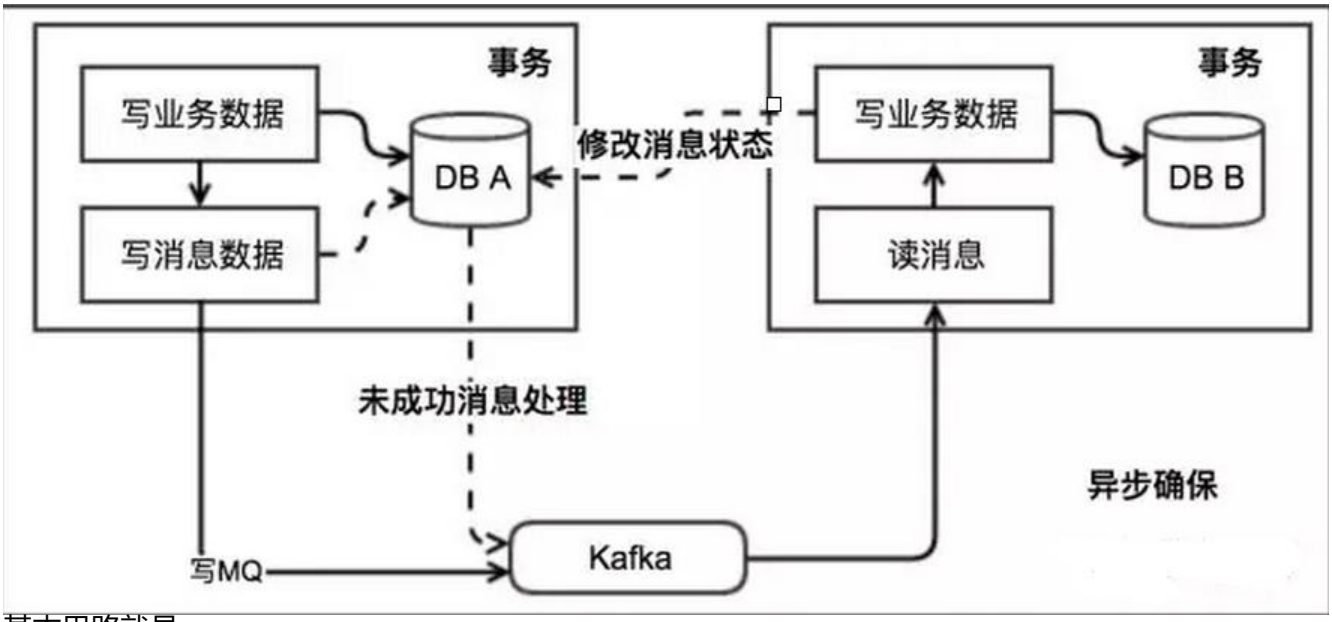
Cancel 阶段：取消执行，释放 Try 阶段预留的业务资源，Cancel 操作满足幂等性。Cancel 阶段的常和 Confirm 阶段异常处理方案基本上一致。

本地消息表（异步确保）

本地消息表这个方案最初是 eBay 提出的，eBay 的完整方案 https://queue.acm.org/detail.cfm?id=394128。

此方案的核心是将需要分布式处理的任务通过消息日志的方式来异步执行。消息日志可以存储到本地本、数据库或消息队列，再通过业务规则自动或人工发起重试。

人工重试更多的是应用于支付场景，通过对账系统对事后问题的处理。



基本思路就是：

消息生产方，需要额外建一个消息表，并记录消息发送状态。消息表和业务数据要在一个事务里提交也就是说他们要在一个数据库里面。然后消息会经过MQ发送到消息的消费方。如果消息发送失败，进行重试发送。

消息消费方，需要处理这个消息，并完成自己的业务逻辑。此时如果本地事务处理成功，表明已经处成功了，如果处理失败，那么就会重试执行。如果是业务上面的失败，可以给生产方发送一个业务补消息，通知生产方进行回滚等操作。

生产方和消费方定时扫描本地消息表，把还没处理完成的消息或者失败的消息再发送一遍。如果有靠的自动对账补账逻辑，这种方案还是非常实用的。

MQ事物消息

有一些第三方的MQ是支持事务消息的，比如RocketMQ，他们支持事务消息的方式也是类似于采用二阶段提交，但是市面上一些主流的MQ都是不支持事务消息的，比如 RabbitMQ 和 Kafka 都不支。

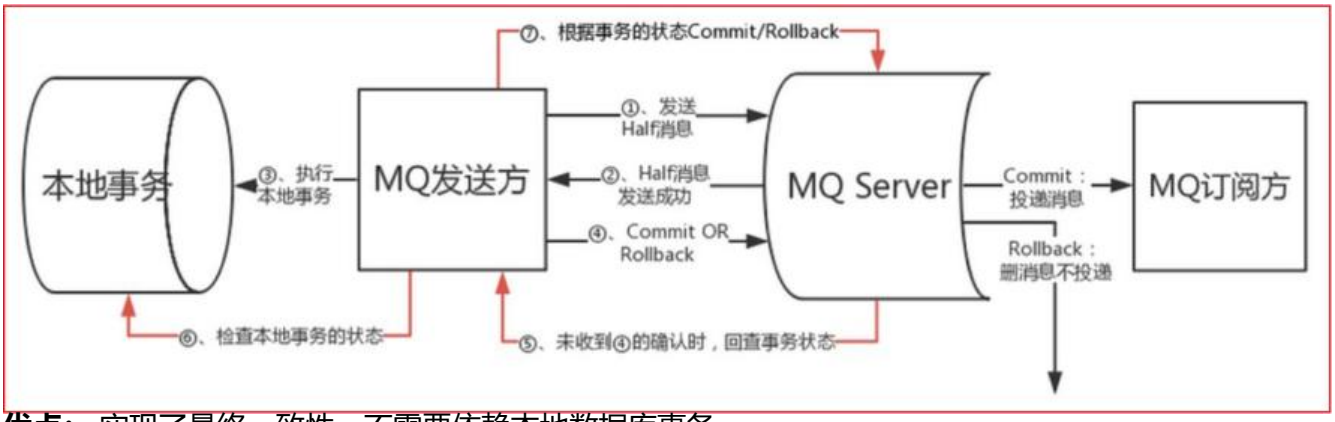
在 RocketMQ 中实现了分布式事务，实际上是对本地消息表的一个封装，将本地消息表移动到了 MQ 内部

以阿里的 RocketMQ 中间件为例，其思路大致为：

第一阶段Prepared消息，会拿到消息的地址。

第二阶段执行本地事务，第三阶段通过第一阶段拿到的地址去访问消息，并修改状态。

也就是说在业务方法内要想消息队列提交两次请求，一次发送消息和一次确认消息。如果确认消息发失败了RocketMQ会定期扫描消息集群中的事务消息，这时候发现了Prepared消息，它会向消息发者确认，所以生产方需要实现一个check接口，RocketMQ会根据发送端设置的策略来决定是回滚还继续发送确认消息。这样就保证了消息发送与本地事务同时成功或同时失败。



优点： 实现了最终一致性，不需要依赖本地数据库事务。

缺点： 目前主流MQ中只有RocketMQ支持事务消息。

参阅：

分布式开放消息系统（RocketMQ）的原理与实践 --CHEN川

聊聊分布式，再说说解决方案 --Savorboard