RocketMQ 4.3正式发布，支持分布式事务

原创：冯嘉 聊聊架构 昨天

近日，Apache RocketMQ 4.3 版本宣布发布，此次发布不仅包括提升性能，减少内存使用等原有特性增强，还修复了部分社区提出的若干问题，更重要的是该版本开源了社区最为关心的分布式事务消息，而且实现了对外部组件的零依赖。接下来，本文将详细探秘 RocketMQ 事务消息的设计原理以及实现机制。

需求缘起

在微服务架构中，随着服务的逐步拆分，数据库私有已经成为共识，这也导致所面临的分布式事务问题成为微服务落地过程中一个非常难以逾越的障碍，但是目前尚没有一个完整通用的解决方案。

其实不仅仅是在微服务架构中，随着用户访问量的逐渐上涨，数据库甚至是服务的分片、分区、水平拆分、垂直拆分已经逐渐成为较为常用的提升瓶颈的解决方案，因此越来越多的原

子操作变成了跨库甚至是跨服务的事务操作。最终结果是在对高性能、高扩展性，高可用性的追求的道路上，我们开始逐渐放松对一致性的追求，但是在很多场景下，尤其是账务，电商等业务中，不可避免的存在着一致性问题，使得我们不得不去探寻一种机制，用以在分布式环境中保证事务的一致性。

理论基石

微服务使得单体架构扩展为分布式架构，在扩展的过程中，逐渐丧失了单体架构中数据源单一，可以直接依赖于数据库进行事务操作的能力，而关系型数据库中，提供了强大的事务处理能力，可以满足 ACID（Atomicity，Consistency，Isolation，Durability）的特性，这种特性保证了数据操作的强一致性，这也是分布式环境中弱一致性以及最终一致性能够得以实现的基础。

数据一致性分为三个种类型：强一致性，弱一致性以及最终一致性，正如上文所述，数据库实现的就是强一致性，能够保证在写入一份新的数据库，立即使其可见。最终一致性是弱一致性的强化版，系统保证在没有后续更新的前提下，系统最终返回上一次更新操作的值。在没有故障发生的前提下，不一致窗口的时间主要受通信延迟，系统负载和复制副本的个数影响。

然而，微服务作为分布式系统，同样受 CAP[1] 原理的制约，在 CAP 理论中，C：Consistency、A：Availability、P：Partition tolerance 三者不可同时满足，而服务化中，更多的是提升 A 以及 P，在这个过程中不可避免的会降低对 C 的要求，因此，BASE 理论随之而来。

BASE[2] 理论来源于 ebay 在 2008 年 ACM 中发表的论文，BASE 理论的基本原则有三个：Basically Available，Soft state，Eventually consistent，主要目的是为了提升分布式系统的可伸缩性，论文同样阐述了如何对业务进行调整以及折中的手段，BASE 理论的提出为分布式事务的发展指出了一个方向。

在最终一致性的实现过程中，最基本的操作就是保证事务参与者的幂等性，所谓的幂等性，就是业务方能够使用相关的手段，保证单个事务多次提交依然能够保证达到同样的目的。

当前解决方案

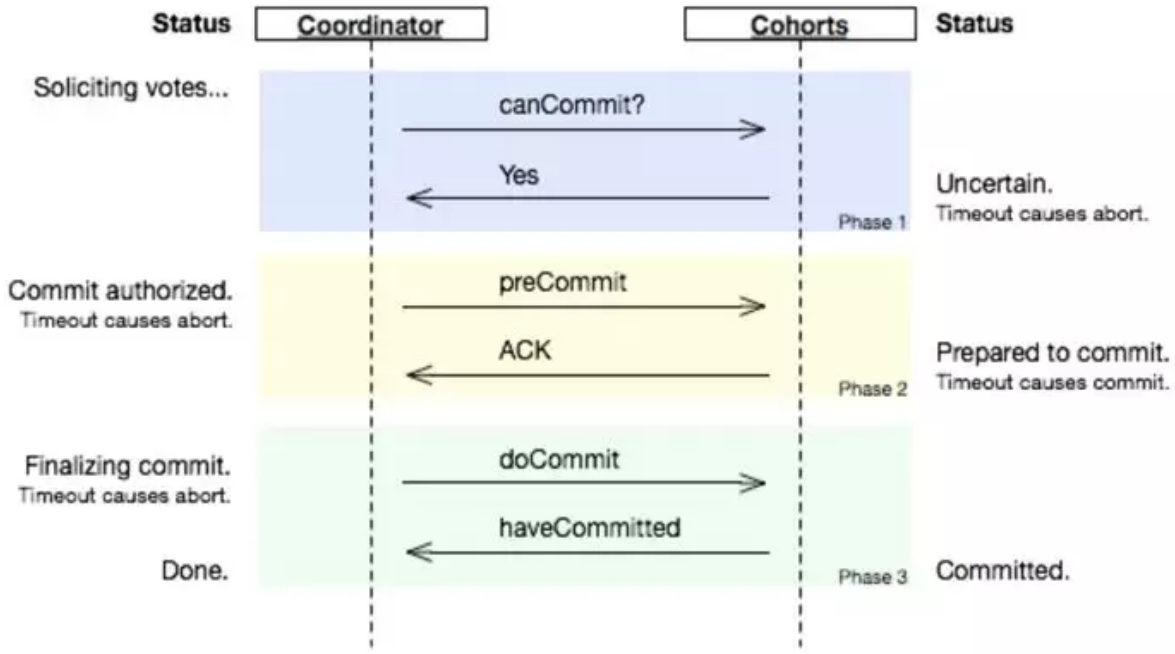
2PC/3PC

谈到分布式事务，首先要说的就是 2PC（two phase commit）方案，如下图所示 [3]：

```
Coordinator                                          Cohort
                        QUERY TO COMMIT
                    ---------------------------------->
                        VOTE YES/NO                  prepare*/abort*
                    <----------------------------------
commit*/abort*          COMMIT/ROLLBACK
                    ---------------------------------->
                        ACKNOWLEDGMENT               commit*/abort*
                    <----------------------------------
end
```

2PC 把事务的执行分为两个阶段，第一个阶段即 prepare 阶段，这个阶段实际上就是投票阶段，协调者向参与者确认是否可以共同提交，再得到全部参与者的所有回答后，协调者向所有的参与者发布共同提交或者共同回滚的指令，用以保证事务达到一致性。

但是分布式系统中的所有通信均存在着三种状态：成功，失败，超时。其中，超时状态的存在是我们在设计分布式系统时所面对的永远的痛，2PC 同样存在问题，尤其是在发送完可以提交的指令后，参与者在没有收到提交或者回滚的指令时，面对已经上锁的资源，面对已经写出去的 undo 或者 redo 日志，参与者会一时陷入手足无措的状态，为了解决这个问题，3PC 应运而生，如下图所示 [4]：



3PC 在 commit 之前增加了 preCommit 的过程，使得在参与者在收不到确认时，依然可以从容 commit 或者 rollback，避免资源锁定太久导致浪费。但是 3PC 同样存在着很多问题。实现起来非常复杂，因为很难通过多次询问来解决系统间分歧问题，尤其是存在超时状态互不信任的分布式网络中，这也就是著名的拜占庭将军问题 [5]。

总结一下，2PC 是几乎所有分布式事务算法的基础，后续的分布式事务算法几乎都由此改进而来，其优缺点非常明显：

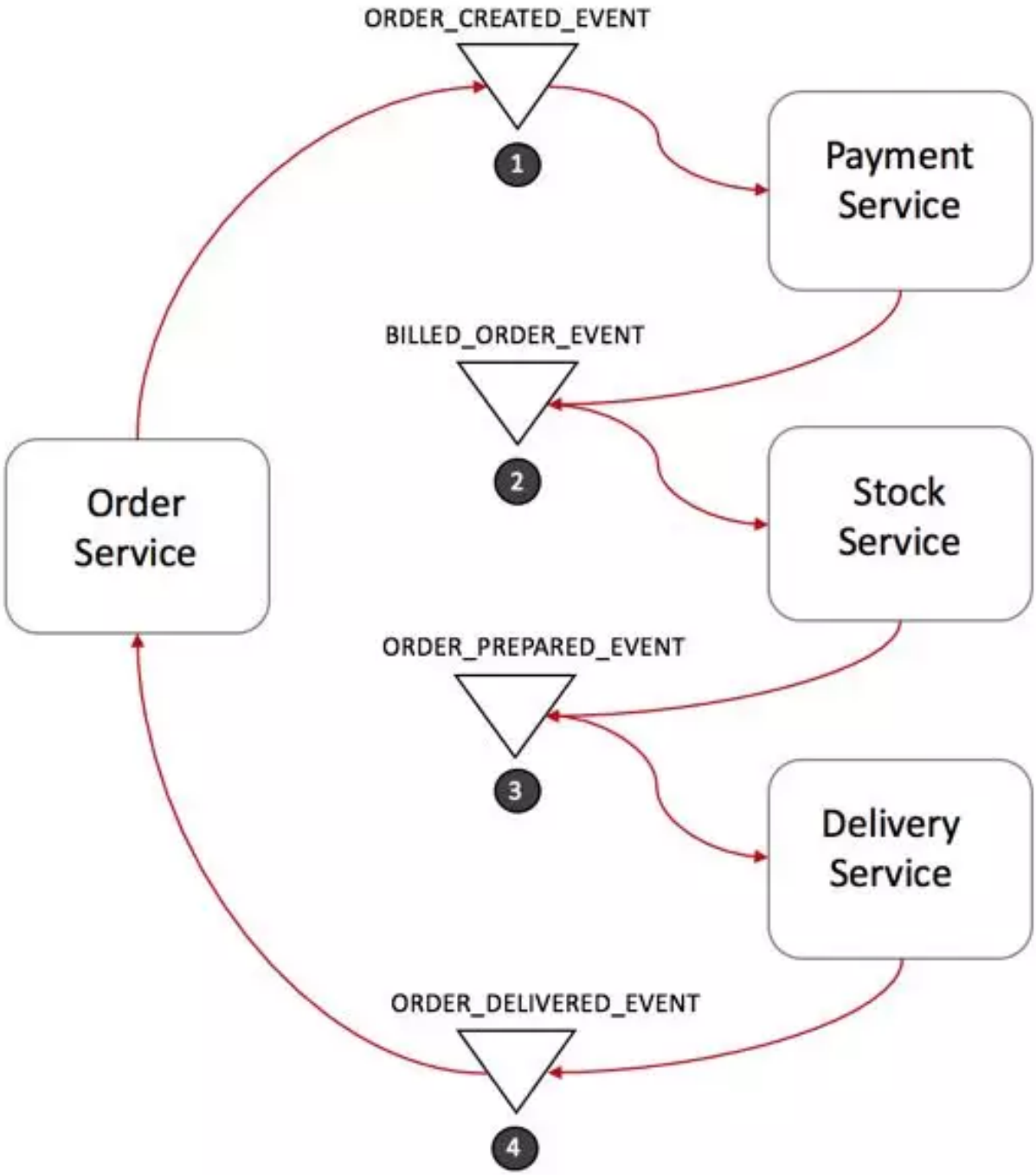
优点：在于已经有较为成熟的实现方案，比如 XA。

缺点：XA 是一个阻塞协议。服务在投票后需要等待协调器的决定，此时服务会阻塞并锁定资源。由于其阻塞机制和最差时间复杂度高， 因此，这种设计不能适应随着事务涉及的服务数量增加而扩展的需要，很难用于并发较高以及子事务声明周期较长 (long-running transactions) 的分布式服务中。

SAGA

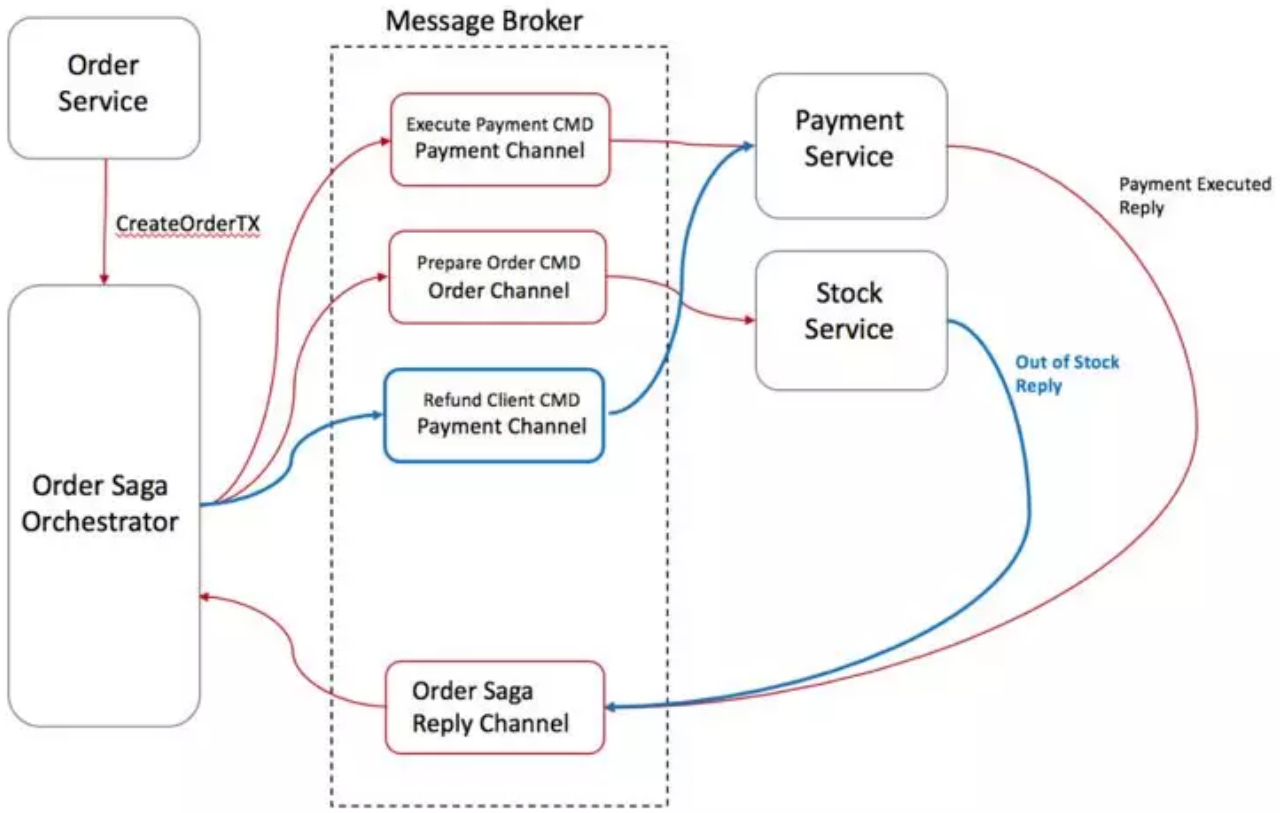
SAGA 算法 [6] 于 1987 年提出，是一种异步的分布式事务解决方案，其理论基础在于，其假设所有事件按照顺序推进，总能达到系统的最终一致性，因此 saga 需要服务分别定义提交接口以及补偿接口，当某个事务分支失败时，调用其他的分支的补偿接口来进行回滚，saga 的具体实现分为两种：Choreography 以及 Orchestration，

（1）Choreography：如下图所示：



这种模式下不存在协调器的概念，每个节点均对自己的上下游负责，在监听处理上游节点事件的同时，对下游节点发布事件。

（2）Orchestration：存在中心节点的模式，如下图所示：



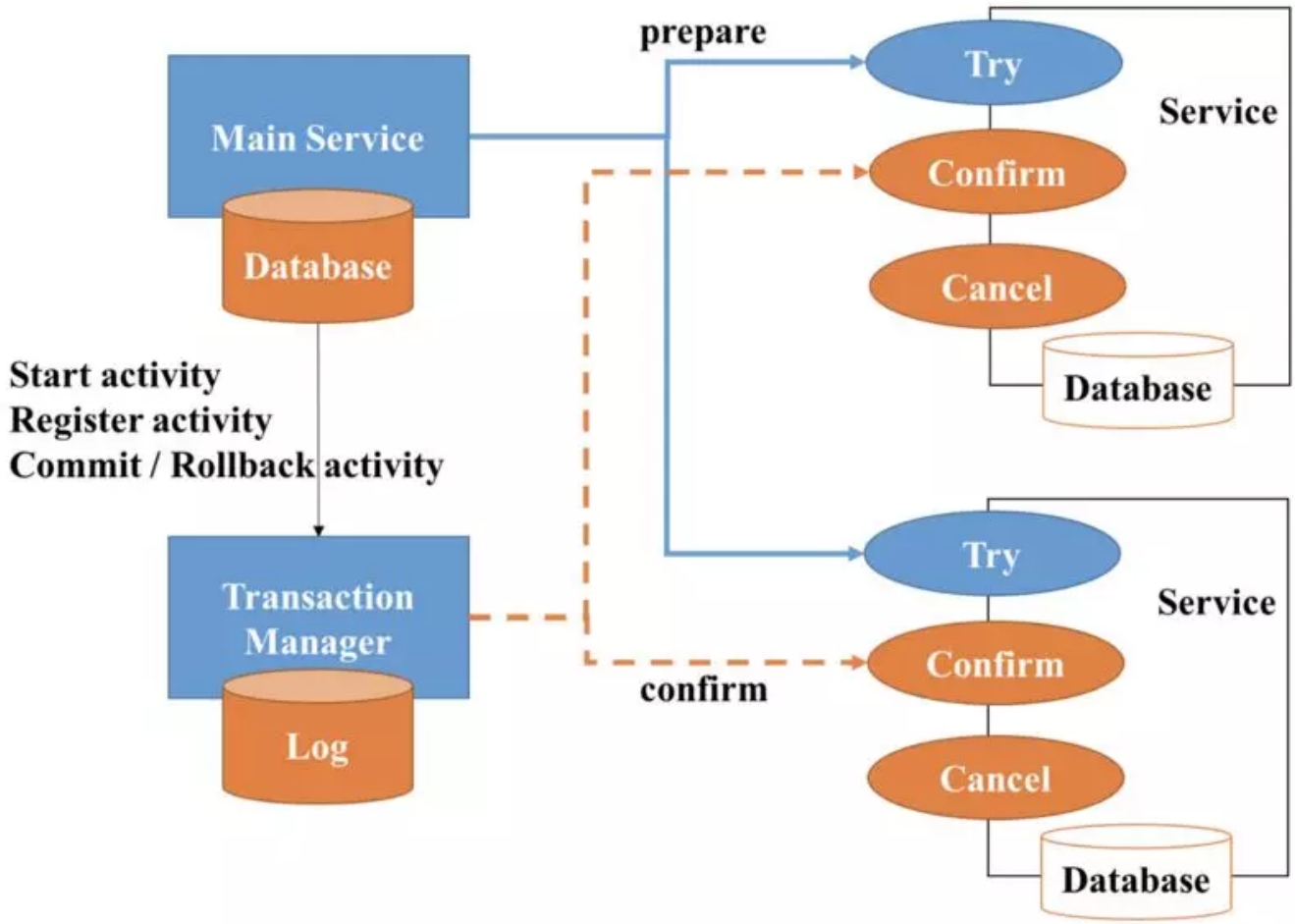
该中心节点，即协调器知道整个事务的分布状态，相比于无中心节点方式，该方式有着许多优点：

1. 能够避免事务之间的循环依赖关系。
2. 参与者只需要执行命令 / 回复 (其实回复消息也是一种事件消息)，降低参与者的复杂性。
3. 开发测试门槛低。
4. 在添加新步骤时，事务复杂性保持线性，回滚更容易管理。因此大多数 saga 模型实现均采用了这种思路。

总结一下：SAGA 模型的优点在于其降低了事务粒度，使得事务扩展更加容易，同时采用了异步化方式提升性能。但是其缺点在于很多时候很难定义补偿接口，回滚代价高，而且由于 SAGA 在执行过程中采用了先提交后补偿的思路进行操作，所以单个子事务在并发提交时的隔离性很难保证。

TCC

TCC(Try-Confirm-Concel) 模型 [7] 同样是一种补偿性事务，主要分为 Try：检查、保留资源，Confirm：执行事务，Concel：释放资源三个阶段，如下图所示：



其中，活动管理器记录了全局事务的推进状态以及各子事务的执行状态，负责推进各个子事务共同进行提交或者回滚。同时负责在子事务处理超时后不停重试，重试不成功后转手工处理，用以保证事务的最终一致性。

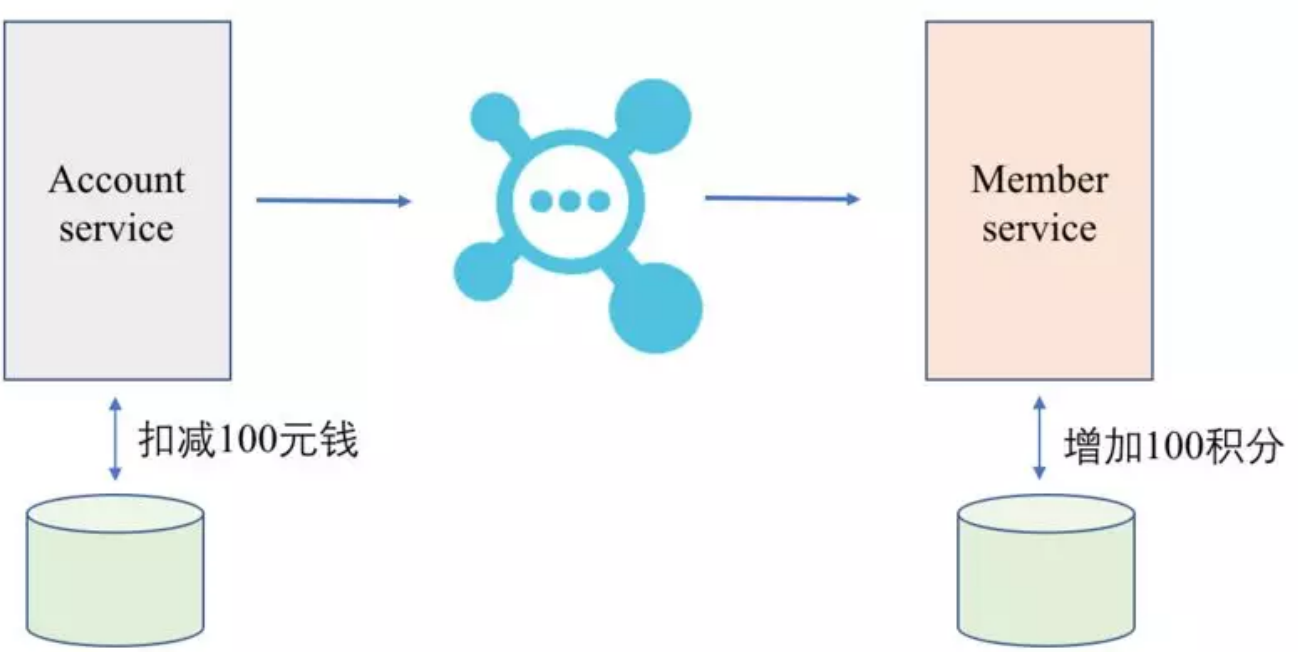
总结一下，相比于 SAGA 模型，其优点在于尝试阶段仅仅只是对业务系统做检测，并保留业务资源，并没有真正提交，所以后续 SAGA 需要针对提交的事务做补偿，而 TCC 则仅仅需要释放保留资源，降低了补偿成本；并且，由于在 Try 阶段对资源进行了保留锁定，所以相比于 SAGA 模式，TCC 模式拥有更高的隔离性。

缺点：相比于 SAGA 模式，TCC 模式多增加了一个状态，导致在业务开发过程中，复杂度上升，而且协调器与子事务的通信过程增加，状态轮转处理也更为复杂。

事务消息

以购物场景为例，张三购买物品，账户扣款 100 元的同时，需要保证在下游的会员服务中给该账户增加 100 积分。由于数据库私有，所以导致在实际的操作过程中会出现很多问题，比如先发送消息，可能会因为扣款失败导致账户积分无故增加，如果先执行扣款，则有可能因

服务宕机，导致积分不能增加，无论是先发消息还是先执行本地事务，都有可能导致出现数据不一致的结果。



事务消息的本质就是为了解决此类问题，解决本地事务执行与消息发送的原子性问题。目前，事务消息在多种分布式消息中间件种均有实现，但是其实现方式思路却各有不同。

1、传统事务消息实现

传统事务消息实现，一种思路是依赖于 AMQP 协议用来确保消息发送成功，AMQP 模式下需要在发送在发送事务消息时进行两阶段提交，首先进行 tx_select 开启事务，然后再进行消息发送，最后进行消息的 commit 或者是 rollback。这个过程可以保证在消息发送成功的同时本地事务也一定成功执行，但事务粒度不好控制，而且会导致性能急剧下降，同时依然无法解决本地事务执行与消息发送的原子性问题。

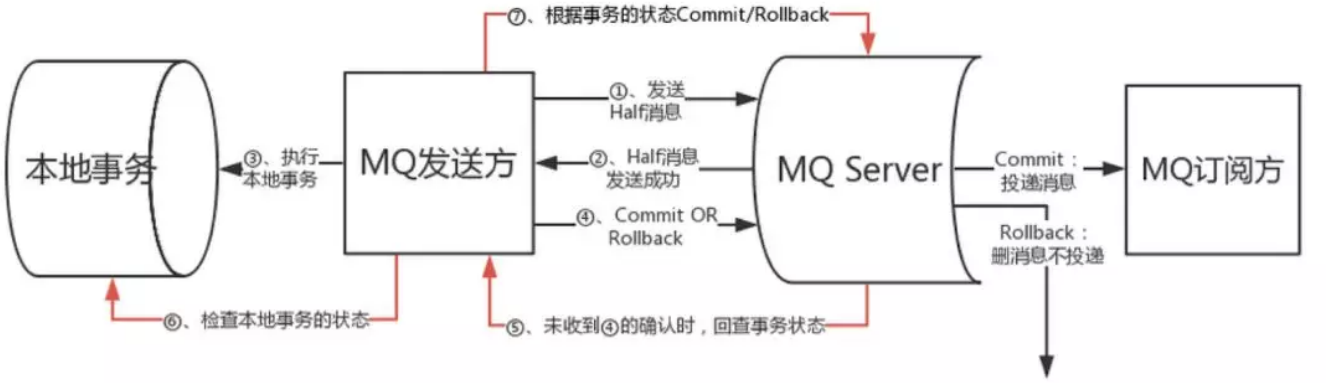
还有另外一种思路，就是通过保证多条消息的同时可见性来保证事务一致性。但是此类消息事务实现机制更多的是用到 consume-transform-produce 场景中，其本质还是用来保证消息自身事务，并没有把外部事务包含进来。

2、RocketMQ 事务消息

RocketMQ 事务消息设计则主要是为了解决 Producer 端的消息发送与本地事务执行的原子性问题，RocketMQ 的设计中 broker 与 producer 端的双向通信能力，使得 broker 天生可以作为一个事务协调者存在；而 RocketMQ 本身提供的存储机制，则为事务消息提供了持久化能力；RocketMQ 的高可用机制以及可靠消息设计，则为事务消息在系统在发生异常时，依然能够保证事务的最终一致性达成。

2.1 RocketMQ 事务消息设计

事务消息作为一种异步确保型事务， 将两个事务分支通过 MQ 进行异步解耦，RocketMQ 事务消息的设计流程同样借鉴了两阶段提交理论，整体交互流程如下图所示：

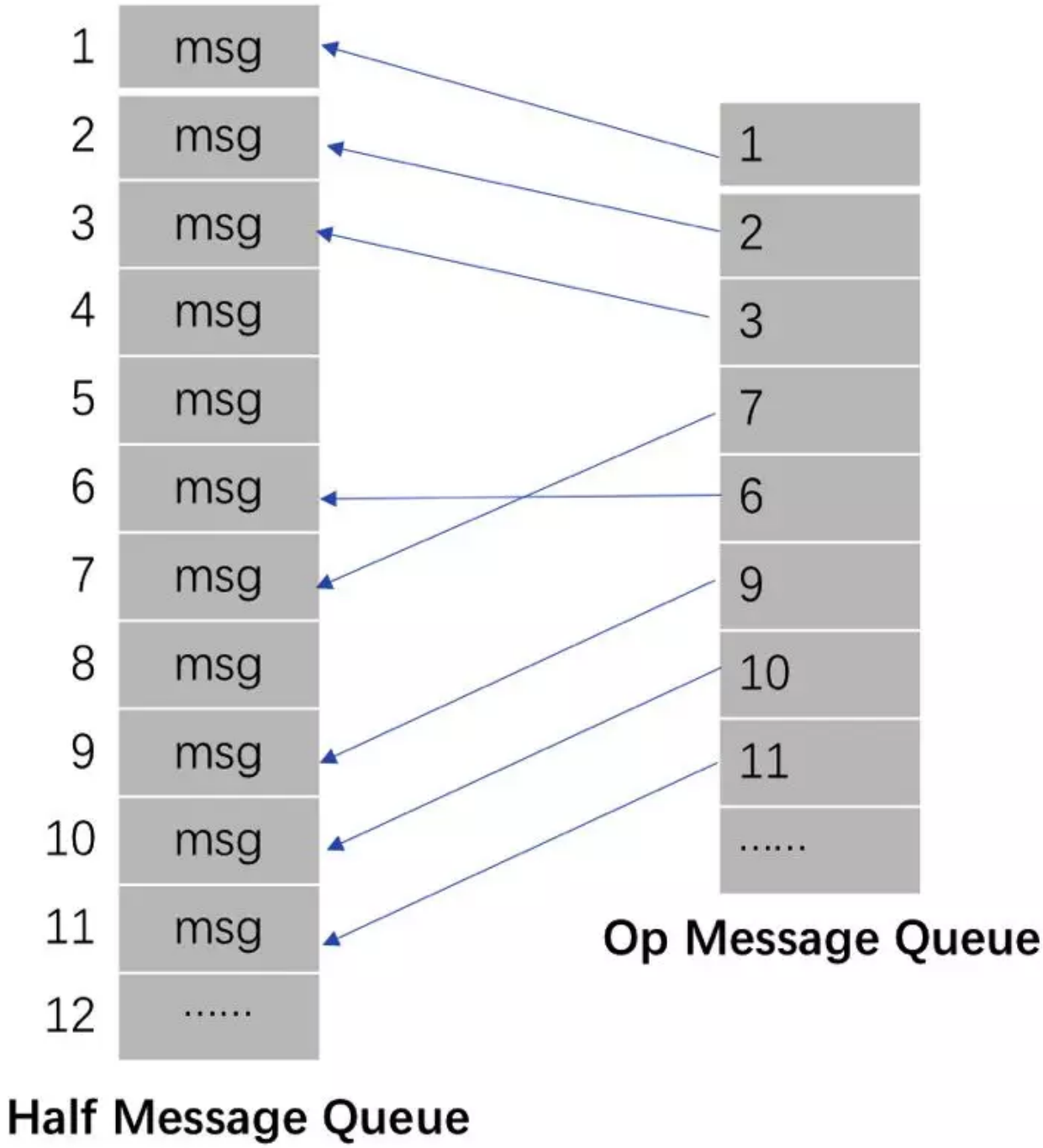


1. 事务发起方首先发送 prepare 消息到 MQ。
2. 在发送 prepare 消息成功后执行本地事务。
3. 根据本地事务执行结果返回 commit 或者是 rollback。
4. 如果消息是 rollback，MQ 将删除该 prepare 消息不进行下发，如果是 commit 消息，MQ 将会把这个消息发送给 consumer 端。
5. 如果执行本地事务过程中，执行端挂掉，或者超时，MQ 将会不停的询问其同组的其他 producer 来获取状态。
6. Consumer 端的消费成功机制有 MQ 保证。

2.2 RocketMQ 事务消息实现

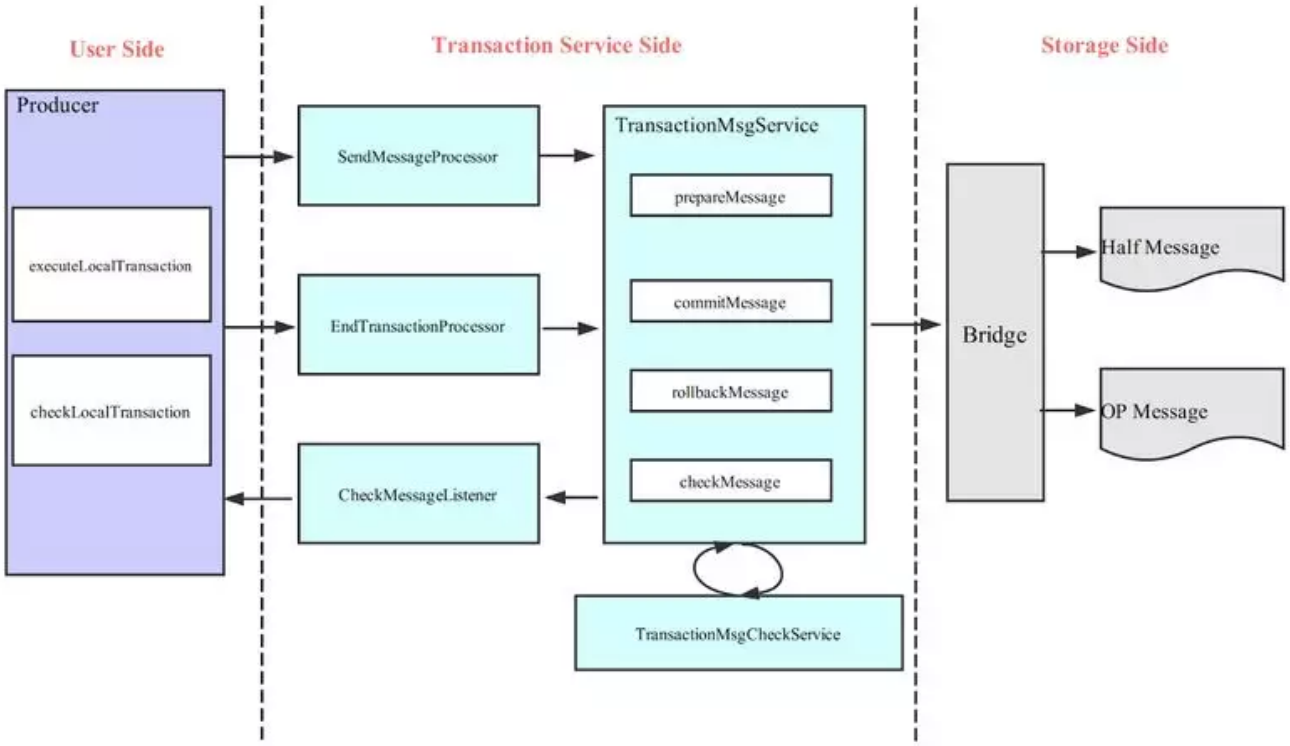
RocketMQ 事务消息在实现上充分利用了 RocketMQ 本身机制，在实现零依赖的基础上，同样实现了高性能、可扩展、全异步等一系列特性。

在具体实现上，RocketMQ 通过使用 Half Topic 以及 Operation Topic 两个内部队列来存储事务消息推进状态，如下图所示：



其中，Half Topic 对应队列中存放着 prepare 消息，Operation Topic 对应的队列则存放了 prepare message 对应的 commit/rollback 消息，消息体中则是 prepare message 对应的 offset，服务端通过比对两个队列的差值来找到尚未提交的超时事务，进行回查。

在具体实现上，事务消息作为普通消息的一个应用场景，在实现过程中进行了分层抽象，从而避免了对 RocketMQ 原有存储机制的修改，如下图所示：



从用户侧来说，用户需要分别实现本地事务执行以及本地事务回查方法，因此只需关注本地事务的执行状态即可；而在 service 层，则对事务消息的两阶段提交进行了抽象，同时针对超时事务实现了回查逻辑，通过不断扫描当前事务推进状态，来不断反向请求 Producer 端获取超时事务的执行状态，在避免事务挂起的同时，也避免了 Producer 端的单点故障。而在存储层，RocketMQ 通过 Bridge 封装了与底层队列存储的相关操作，用以操作两个对应的内部队列，用户也可以依赖其他存储介质实现自己的 service，RocketMQ 会通过 ServiceProvider 加载进来。

从上述事务消息设计中可以看到，RocketMQ 事务消息较好的解决了事务的最终一致性问题，事务发起方仅需要关注本地事务执行以及实现回查接口给出事务状态判定等实现，而且在上游事务峰值高时，可以通过消息队列，避免对下游服务产生过大压力。

事务消息不仅适用于上游事务对下游事务无依赖的场景，还可以与一些传统分布式事务架构相结合，而 MQ 的服务端作为天生的具有高可用能力的协调者，使得我们未来可以基于 RocketMQ 提供一站式轻量级分布式事务解决方案，用以满足各种场景下的分布式事务需求。

作者介绍

冯嘉，Apache RocketMQ 联合创始人，LInux OpenMessaging 标准创始人。阿里巴巴高级技术专家，带领团队、社区打造了中国分布式云计算领域第一个 Apache 顶级开源中间件项目，创立分布式消息领域的国际标准 OpenMessaging。 冯嘉作为阿里巴巴 RocketMQ 技术负责人，具有丰富的分布式软件架构、高并发网站设计、性能调优经验，拥有国内外多项分布式、推荐领域的专利授权。目前专注于大规模分布式系统、分布式消息引擎、流计算领域，关注 Hbase/Hadoop/Spark/Flink 等大数据技术栈。

杜恒，阿里巴巴技术专家，Apache RocketMQ 内核控，拥有多年分布式系统研发经验，对 Microservice, Messaging，Storage 等领域有深刻理解，拥有多年金融领域开发设计经验。目前专注 RocketMQ 内核优化以及 Messaging 生态建设。

参考文献

[1] https://en.wikipedia.org/wiki/Cap

[2] https://dl.acm.org/citation.cfm?id=1394128

[3] https://en.wikipedia.org/wiki/Two-phase_commit_protocol

[4] https://en.wikipedia.org/wiki/Three-phase_commit_protocol

[5] https://people.eecs.berkeley.edu/~luca/cs174/byzantine.pdf

[6] https://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf

[7] https://www.cloud.alipay.com/docs/2/69656

业务的增长会带来大量数据，很多公司都搭建了自己的大数据处理平台，或者向新的数据平台 / 框架迁移。大数据平台技术选型、搭建、系统迁移和优化方面有哪些需要注意的地方？

QCon 上海 2018 邀请到 Dell EMC 工程总监、ConfluentKafka Streams 系统架构师和技术负责人等技术大牛来分享他们的经验与收获。大会8 折报名中，立减 1360 元。有任何问题欢迎咨询票务经理 Hanna，电话：010-84782011，微信：qcon-0410。


Geekbang
InfoQ
《大数据系统架构》
QCon 全球软件开发大会
【2018】上海站 2018年10月18-20日
8折 预售中，现在报名
立减1360元
团购享更多优惠，截至2018年8月19日

阅读原文