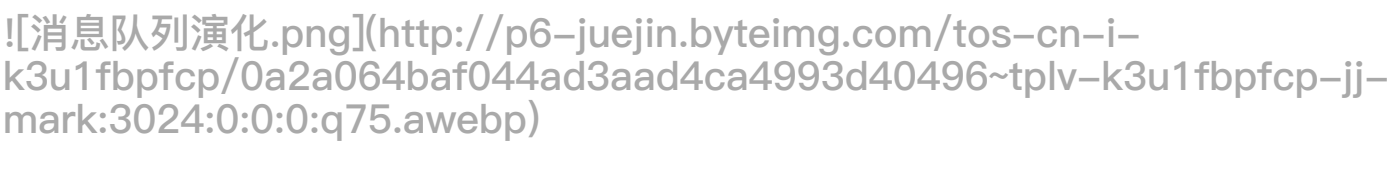
浅谈RocketMQ、Kafka、Pulsar的事务消息

导语

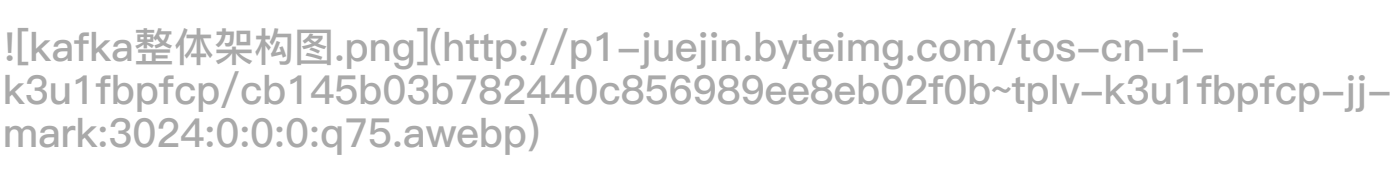
事务是一个程序执行单元，里面的所有操作要么全部执行成功，要么全部执行失败。RocketMQ、Kafka和Pulsar都是当今业界应用十分广泛的开源消息队列（MQ）组件，笔者在工作中遇到关于MQ选型相关的内容，了解到关于“事务消息”这个概念在不同的MQ组件里有不同内涵。故借此文，试着浅析一番这三种消息队列（MQ）的事务消息有何异同，目的是形成关于消息队列事务消息的全景视图，给有类似业务需求的同学提供一些参考和借鉴。

一、消息队列演化

`消息队列`（Message Queue，简称MQ），是指在消息的传输中保存消息的容器或服务，是一种异步的服务间通信方式，适用于无服务器和微服务架构，是分布式系统实现高性能、高可用、可伸缩等高级特效的重要组件。 常见的主流消息队列有ActiveMQ、RabbitMQ、ZeroMQ、Kafka、MetaMQ、RocketMQ、Pulsar等。

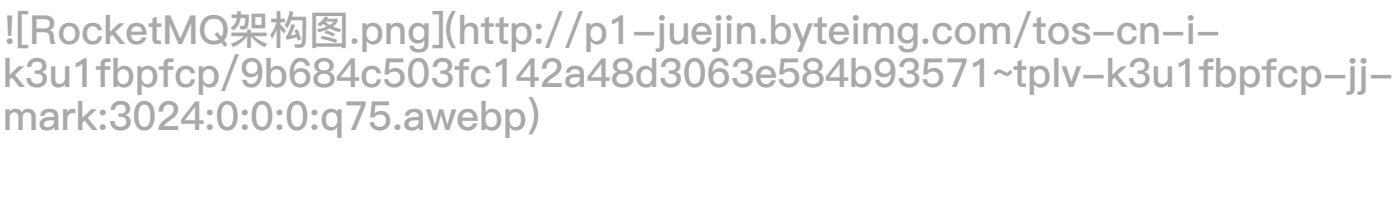


Kafka： Apache Kafka是由Apache软件基金会开发的一个开源消息系统项目，由Scala写成。Kafka最初是由LinkedIn开发，并于2011年初开源。2012年10月从Apache Incubator毕业。该项目的目标是为处理实时数据提供一个统一、高通量、低等待的平台。
Kafka是一个分布式的、分区的、多复本的日志提交服务。它通过一种独一无二的设计提供了一个消息系统的功能，其整体架构图如下所示。

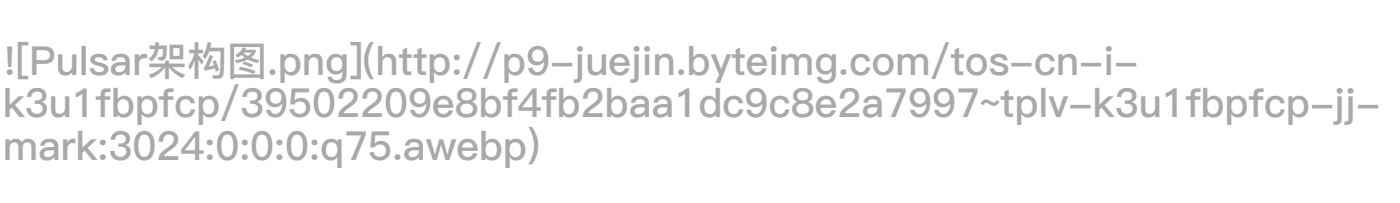


RocketMQ：Apache RocketMQ是一个分布式消息和流媒体平台，具有

低延迟、强一致、高性能和可靠性、万亿级容量和灵活的可扩展性。它有借鉴Kafka的设计思想，但不是kafka的拷贝，其整体架构图如下所示。



Pulsar：Apache Pulsar 是 Apache 软件基金会顶级项目，是下一代云原生分布式消息流平台，集消息、存储、轻量化函数式计算为一体，采用计算与存储分离架构设计，支持多租户、持久化存储、多机房跨区域数据复制，具有强一致性、高吞吐、低延时及高可扩展性等流数据存储特性，被看作是云原生时代实时消息流传输、存储和计算最佳解决方案，其整体架构图如下所示。



二、背景知识

2.1 什么是事务?

2.1.1 事务（Trasaction）

事务是一个程序执行单元，里面的所有操作要么全部执行成功，要么全部执行失败。

一个事务有四个基本特性，也就是我们常说的（`ACID`）。

`Atomicity`（原子性）：事务是一个不可分割的整体，事务内所有操作要么全做成功，要么全失败。

`Consistency`（一致性）：事务执行前后，数据从一个状态到另一个状态必须是一致的（A向B转账，不能出现A扣了钱，B却没收到）。

`Isolation`（隔离性）： 多个并发事务之间相互隔离，不能互相干扰。

`Durablity`（持久性） ： 事务完成后，对数据的更改是永久保存的，不能回滚。

2.1.2 分布式事务

分布式事务是指事务的参与者、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系统的不同节点之上。分布式事务通常用于在分布式系统中保证不同节点之间的数据一致性。

分布式事务的解决方案一般有以下几种：

XA（2PC/3PC）

最具有代表性的是由Oracle Tuxedo系统提出的XA分布式事务协议。XA中大致分为两部分：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这些商业数据库都实现了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协议通常包含**两阶段提交（2PC）**和**三阶段提交（3PC）**两种实现。两阶段提交顾名思义就是要进行两个阶段的提交：第一阶段，准备阶段(投票阶段)； 第二阶段，提交阶段（执行阶段）。实现过程如下所示：

二阶段提交看似能够提供原子性的操作，但它存在着一些缺陷，三段提交（3PC）是对两段提交（2PC）的一种升级优化，有兴趣的可以深入了解一下，这里不再赘述。

TCC

TCC（Try-Confirm-Cancel）是Try、Commit、Cancel三种指令的缩写，又被称`补偿事务`，其逻辑模式类似于XA两阶段提交，事务处理流程也很相似，但2PC是应用于在DB层面，TCC则可以理解为在应用层面的2PC，是需要我们编写业务逻辑来实现。

TCC它的核心思想是：”针对每个操作都要注册一个与其对应的确认（Try）和补偿（Cancel）”。

消息事务

所谓的消息事务就是基于消息队列的两阶段提交，本质上是对消息队列的一种特殊利用，它是将本地事务和发消息放在了一个分布式事务里，保证要么本地操作成功成功并且对外发消息成功，要么两者都失败。

基于消息队列的两阶段提交往往用在高并发场景下，将一个分布式事务拆成一个消息事务（A系统的本地操作+发消息）+B系统的本地操作，其中B系统的操作由消息驱动，只要消息事务成功，那么A操作一定成功，消息也一定发出来了，这时候B会收到消息去执行本地操作，如果本地操作失败，消息会重投，直到B操作成功，这样就变相地实现了A与B的分布式事务。原理如下：

虽然上面的方案能够完成A和B的操作，但是A和B并不是强一致的，而是**最终一致（Eventually consistent）**的。而这也是满足BASE理论的要求的。这里引申一下，BASE 是 Basically Available（基本可用）、Soft state（软状态）和 Eventually consistent （最终一致性）三个短语的缩写。BASE 理论是对 CAP 中 AP （CAP 已经被证实一个分布式系统最多只能同时满足CAP三项中的两项）的一个扩展，通过牺牲强一致性来获得可用性，当出现故障允许部分不可用但要保证核心功能可用，允许数据在一段时间内是不一致的，但最终达到一致状态。满足BASE理论的事务，我们称之为**“柔性事务”**。

2.2 什么是 Exactly-once (精确一次)语义？

在分布式系统中，任何节点都有可能出现异常甚至宕机。在 消息队列中也一样，当 Producer 在生产消息时，可能会发生 Broker 宕机不可用，或者网络突然中断等异常情况。根据在发生异常时 Producer 处理消息的方式，系统可以具备以下三种消息语义。

2.2.1 At-least-once (至少一次)语义

Producer 通过接收 Broker 的 ACK （消息确认）通知来确保消息成功写入 Topic。然而，当 Producer 接收 ACK 通知超时，或者收到 Broker 出错信息时，会尝试重新发送消息。如果 Broker 正好在成功把消息写入到 Topic，但还没有给 Producer 发送 ACK 时宕机，Producer 重新发送的消息会被再次写入到 Topic，最终导致消息被重复分发至 Consumer。即：**消息不会丢失，但有可能被重复发送。**

2.2.2 At-most-once (最多一次)语义

当 Producer 在接收 ACK 超时，或者收到 Broker 出错信息时不重发消息，那就有可能导致这条消息丢失，没有写入到 Topic 中，也不会被 Consumer 消费到。在某些场景下，为了避免发生重复消费，我们可以容许消息丢失的发生。即：**消息可能会丢失，但绝不会被重复发送。**

2.2.3 Exactly-once (精确一次)语义

Exactly-once 语义保证了即使 Producer 多次发送同一条消息到服务端，服务端也仅仅会记录一次。Exactly-once 语义是最可靠的，同时也是最难理解的。Exactly-once 语义需要消息队列服务端，消息生产端和消费端应用三者的协同才能实现。比如，当消费端应用成功消费并且 ACK 了一条消息之后，又把消费位点回滚到之前的一个消息 ID，那么从那个消息 ID 往后的所有消息都会被消费端应用重新消费到。即：**消息不会丢失，也不会被重复发送。**

三、RocketMQ、Kafka、Pulsar事务消息

3.1 RocketMQ的事务消息

RocketMQ在4.3.0版中已经支持**分布式事务消息**，这里RocketMQ采用了2PC的思想来实现了提交事务消息，同时增加一个补偿逻辑来处理二阶段超时或者失败的消息，流程如下图所示：


其具体工作流程分为正常事务消息的发送及提交和不正常情况下事务消息的补偿流程：

1.在消息队列上开启一个事务主题。
2.事务中第一个执行的服务发送一条“半消息”（半消息和普通消息的唯一区别是，在事务提交之前，对于消费者来说，这个消息是不可见的）给消息队列。
3.半消息发送成功后，发送半消息的服务就会开始执行本地事务，根据本地事务执行结果来决定事务消息提交或者回滚。

补偿流程：RocketMQ提供**事务反查**来解决异常情况，如果

RocketMQ没有收到提交或者回滚的请求，Broker会定时到生产者上去反查本地事务的状态，然后根据生产者本地事务的状态来处理这个“半消息”是提交还是回滚。值得注意的是我们需要根据自己的业务逻辑来实现反查逻辑接口，然后根据返回值Broker决定是提交还是回滚。而且这个反查接口需要是无状态的，请求到任意一个生产者节点都会返回正确的数据。
4.本地事务成功后会让这个“半消息”变成正常消息，供分布式事务后面的步骤执行自己的本地事务。（这里的事务消息，Producer 不会因为Consumer消费失败而做回滚，采用事务消息的应用，其所追求的是**高可用和最终一致性**，消息消费失败的话，RocketMQ自己会负责重推消息，直到消费成功。）

其中，补偿流程用于解决消息Commit或者Rollback发生超时或者失败的情况。在RocketMQ事务消息的主要流程中，一阶段的消息如何对用户不可见。其中，事务消息相对普通消息最大的特点就是一阶段发送的消息对用户是不可见的。那么，如何做到写入消息但是对用户不可见呢？RocketMQ事务消息的做法是：如果消息是“半消息”，将备份原消息的主题与消息消费队列，然后改变主题为RMQ_SYS_TRANS_HALF_TOPIC。由于消费组未订阅该主题，故消费端无法消费“半消息”的消息，然后RocketMQ会开启一个定时任务，从Topic为RMQ_SYS_TRANS_HALF_TOPIC中拉取消息进行消费，根据生产者组获取一个服务提供者发送回查事务状态请求，根据事务状态来决定是提交或回滚消息。

讲到这里大家就明白了，这里说的就是2.1.2节里提到分布式事务中的**消息事务**，目的是在分布式事务中实现系统的最终一致性。

3.2 Kafka的事务消息

与RocketMQ的事务消息用途不同，**Kafka 的事务基本上是配合其幂等机制来实现 Exactly-once （见2.2.3节）语义的。**

开发此功能的原因可以总结如下。

流处理的需求
随着流处理的兴起，对具有更强处理保证的流处理应用的需求也在增长。 例如，在金融行业，金融机构使用流处理引擎为用户处理借款和信贷。 这种类型的用例要求每条消息都只处理一次，无一例外。

换句话说，如果流处理应用程序消费消息 A 并将结果作为消息B (B = f(A))，那么恰好一次处理保证意味着当且仅当 B 被成功生产后 A 才能被标记为消费，反之亦然。

k3u1fbpfcp/3e553ce73327459d89167304744d10bc~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp)
事务 API 使流处理应用程序能够在一个原子操作中使用、处理和生成消息。这意味着，事务中的一批消息可以从许多主题分区接收、生成和确认。一个事务涉及的所有操作都作为整体成功或失败。

目前，Kafka 默认提供的交付可靠性保障是At-least-once。如果消息成功“提交”，但 Broker 的应答没有成功发送回 Producer 端（比如网络出现瞬时抖动），那么 Producer 就无法确定消息是否真的提交成功了。因此，它只能选择重试，这就是 Kafka 默认提供At-least-once保障的原因，不过这会导致消息重复发送。大部分用户还是希望消息只会被交付一次，这样的话，消息既不会丢失，也不会被重复处理。或者说，即使 Producer 端重复发送了相同的消息，Broker 端也能做到自动去重。在下游 Consumer 看来，消息依然只有一条。那么问题来了，Kafka 是怎么做到精确一次的呢？简单来说，这是通过两种机制：幂等性（Idempotence）和事务（Transaction）。

3.2.1 幂等性Producer

“幂等”这个词原是数学领域中的概念，指的是某些操作或函数能够被执行多次，但每次得到的结果都是不变的。幂等性有很多好处，其最大的优势在于我们可以安全地重试任何幂等性操作，反正它们也不会破坏我们的系统状态。如果是非幂等性操作，我们还需要担心某些操作执行多次对状态的影响，但对于幂等性操作而言，我们根本无需担心此事。

在 Kafka 中，Producer 默认不是幂等性的，但我们可以创建幂等性 Producer。它其实是 0.11.0.0 版本引入的新功能。enable.idempotence 被设置成 true 后，Producer 自动升级成幂等性 Producer，其他所有的代码逻辑都不需要改变。Kafka 自动帮你做消息的重复去重。Kafka为了实现幂等性，它在底层设计架构中引入了**ProducerID**和**SequenceNumber**。
ProducerID：在每个新的Producer初始化时，会被分配一个唯一的ProducerID，用来标识本次会话。

SequenceNumber：对于每个ProducerID，Producer发送数据的每个Topic和Partition都对应一个从0开始单调递增的SequenceNumber值。Broker在内存维护(pid,seq)映射，收到消息后检查seq。Producer在收到明确的的消息丢失ack，或者超时后未收到ack，要进行重试。

`new_seq = old_seq+1: 正常消息； new_seq <= old_seq : 重复消息；new_seq > old_seq+1: 消息丢失；`

另外我们需要了解幂等性Producer的作用范围。首先，它只能保证单分区上的

幂等性，即一个幂等性 Producer 能够保证某个主题的一个分区上不出现重复消息，它无法实现多个分区的幂等性。其次，它只能实现单会话上的幂等性，不能实现跨会话的幂等性。这里的会话，你可以理解为 Producer 进程的一次运行。当你重启了 Producer 进程之后，这种幂等性保证就丧失了。如果想实现多分区以及多会话上的消息无重复，应该怎么做呢？答案就是事务（transaction）或者依赖事务型 Producer。这也是幂等性 Producer 和事务型 Producer 的最大区别。

3.2.2 事务型Producer

事务型 Producer 能够保证将消息原子性地写入到多个分区中。这批消息要么全部写入成功，要么全部失败。另外，事务型 Producer 也不受进程的重启影响。Producer 重启后，Kafka 依然保证它们发送消息的Exactly-once处理。和普通 Producer 代码相比，事务型 Producer 的显著特点是调用了一些事务 API，如 initTransaction、beginTransaction、commitTransaction 和 abortTransaction，它们分别对应事务的初始化、事务开始、事务提交以及事务终止。

Kafka事务消息是由Producer、事务协调器、Broker、组协调器、Consumer等共同参与实现的。

1）Producer

为Producer指定固定的TransactionalId(事务id)，可以穿越Producer的多次会话(Producer重启/断线重连)中，持续标识Producer的身份。

每个生产者增加一个epoch。用于标识同一个TransactionalId在一次事务中的epoch，每次初始化事务时会递增，从而让服务端可以知道生产者请求是否旧的请求。使用epoch标识Producer的每一次”重生”，可以防止同一Producer存在多个会话。

Producer遵从幂等消息的行为，并在发送的BatchRecord中增加事务id和epoch。

2）事务协调器(Transaction Coordinator)

引入事务协调器，类似于消费组负载均衡的协调者，每一个实现事务的生产端都被分配到一个事务协调者。以两阶段提交的方式，实现消息的事务提交。

事务协调器使用一个特殊的Topic：即事务Topic，事务Topic本身也是持久化的，日志信息记录事务状态信息，由事务协调者写入。

事务协调器通过RPC调用，协调 Broker 和 Consumer实现事务的两阶段提交。

每一个Broker都会启动一个事务协调器，使用hash(TransactionalId)确定Producer对应的事务协调器，使得整个集群的负载均衡。

3）Broker

引入控制消息(Control Messages)：这些消息是客户端产生的并写入到主题的特殊消息，但对于使用者来说不可见。它们是用来让Broker告知消费者之前拉取的消息是否被原子性提交。

Broker处理事务协调器的commit/abort控制消息，把控制消息向正常消息一样写入Topic(图中标c的消息，和正常消息交织在一起，用来确认事务提交的日志偏移)，并向前推进消息提交偏移hw。

4）组协调器

如果在事务过程中，提交了消费偏移，组协调器在offset log中写入事务消费偏移。当事务提交时，在offset log中写入事务offset确认消息。

5）Consumer

Consumer过滤未提交消息和事务控制消息，使这些消息对用户不可见。

有两种实现方式，

- Consumer缓存方式

设置isolation.level=read_uncommitted，此时topic的所有消息对Consumer都可见。Consumer缓存这些消息，直到收到事务控制消息。若事务commit，则对外发布这些消息；若事务abort，则丢弃这些消息。

– Broker过滤方式

设置isolation.level=read_committed，此时topic中未提交的消息对Consumer不可见，只有在事务结束后，消息才对Consumer可见。Broker给Consumer的BatchRecord消息中，会包含以列表，指明哪些是”abort”事务，Consumer丢弃abort事务的消息即可。

因为事务机制会影响消费者所能看到的消息的范围，它不只是简单依赖高水位来判断。它依靠一个名为 LSO（Log Stable Offset）的位移值来判断事务型消费者的可见性。

3.3 Pulsar的事务消息

Apache Pulsar 在 2.8.0 正式支持了事务相关的功能，Pulsar 这里提供的事务区别于 RocketMQ 中 2PC 那种事务的实现方式，没有本地事务回查的机制，更类似于 Kafka 的事务实现机制。Apache Pulsar 中的事务主要用来保证类似 Pulsar Functions 这种流计算场景中 Exactly-once 语义的实现，这也符合 Apache Pulsar 本身 Event Streaming 的定位，即保证端到端（End-to-End）的事务实现的语义。

在Pulsar中，对于事务语义是这样定义的：允许事件流应用将消费、处理、生产消息整个过程定义为一个原子操作，即生产者或消费者能够处理跨多个主题和分区的消息，并确保这些消息作为一个单元被处理。

Pulsar事务具有以下语义：

* 事务中的所有操作都作为一个单元提交。要么提交所有消息，要么都不提交。
* 每条消息只写入或处理一次，不会丢失数据或重复（即使发生故障）。
* 如果事务中止，则此事务中的所有写入和确认都将回滚。

事务中的批量消息可以被以多分区接收、生产和确认。

* 消费者只能读取已提交（确认）的消息。 换句话说，Broker不传递属于打开

事务的事务消息或属于中止事务的消息。
* 跨多个分区的消息写入是原子性的。
* 跨多个订阅的消息确认是原子性的。 订阅下的消费者在确认带有事务ID的消息时，只会成功确认一次消息。

Pulsar事务消息由以下几个关键点构成：

1）事务ID

事务ID（TxnID）标识Pulsar中的唯一事务。 事务 ID 长度是 128-bit。 最高 16 位保留给事务协调器的 ID，其余位用于每个事务协调器中单调递增的数字。

2）事务协调器(Transaction Coordinator)

事务协调器(TC)是运行在 Pulsar Broker 中的一个模块。

* 它维护事务的整个生命周期，并防止事务进入错误状态。
* 它处理事务超时，并确保事务在事务超时后中止。

3）事务日志

所有事务元数据都保存在事务日志中。 事务日志由 Pulsar 主题记录。 如果事务协调器崩溃，它可以从事务日志恢复事务元数据。

事务日志存储事务状态，而不是事务中的实际消息（实际消息存储在实际的主题分区中）。

4）事务缓存

向事务内的主题分区生成的消息存储在该主题分区的事务缓冲区（TB）中。 在提交事务之前，事务缓冲区中的消息对消费者不可见。 当事务中止时，事务缓冲区中的消息将被丢弃。

事务缓冲区将所有正在进行和中止的事务存储在内存中。 所有消息都发送到实际的分区 Pulsar 主题。 提交事务后，事务缓冲区中的消息对消费者具体化（可见）。 事务中止时，事务缓冲区中的消息将被丢弃。

5）待确认状态

挂起确认状态在事务完成之前维护事务中的消息确认。 如果消息处于挂起确认状态，则在该消息从挂起确认状态中移除之前，其他事务无法确认该消息。

挂起的确认状态被保留到挂起的确认日志中(cursor ledger)。 新启动的broker可以从挂起的确认日志中恢复状态，以确保状态确认不会丢失。

处理流程一般分为以下几个步骤：

1. 开启事务。
2. 使用事务发布消息。
3. 使用事务确认消息。
4. 结束事务。

Pulsar 的事务处理流程与 Kafka 的事务处理思路大致上保持一致，大家都有一个 TC 以及对应的一个用于持久化 TC 所有操作的 Topic 来记录所有事务状态变更的请求。同样的在事务开始阶段也都有一个专门的 Topic 来去 查询 TC 对应的 Owner Broker 的位置在哪里。不同的是，第一：Kafka 中对于未确认的消息是维护在 Broker 端的，但是 Pulsar 的是维护在 Client 端的，通过 Transaction Timeout 来决定这个事务是否执行成功，所以有了 Transaction Timeout 的存在之后，就可以确保 Client 和 Broker 侧事务处理的一致性。第二：由于 Kafka 本身没有单条消息的 Ack，所以 Kafka 的事务处理只能是顺序执行的，当一个事务请求被阻塞之后，会阻塞后续所有的事务请求，但是 Pulsar 是可以对消息进行单条 Ack 的，所以在这里每一个事务的 Ack 动作是独立的，不会出现事务阻塞的情况。

四、结论
====

RocketMQ和Kafka/Pulsar的事务消息实用的场景是不一样的。

RocketMQ中的事务，它解决的问题是，确保执行本地事务和发消息这两个操作，要么都成功，要么都失败。并且RocketMQ增加了一个事务反查的机制，来尽量提高事务执行的成功率和数据一致性。

Kafka 中的事务，它解决的问题是，确保在一个事务中发送的多条消息，要么

都成功，要么都失败。（这里面的多条消息不一定要在同一个主题和分区中，可以是发往多个主题和分区的消息）当然也可以在kafka事务执行过程中开启本地事务来实现类似RocketMQ事务消息的效果，但是Kafka是没有事务消息反查机制的，它是直接抛出异常的，用户可以根据异常来实现自己的重试等方法保证事务正常运行。

它们的共同点就是：都是通过两阶段提交来实现事务的，事务消息都保存在单独的主题上。不同的地方就是RocketMQ是通过“半消息”来实现的，kafka是直接将消息发送给对应的topic，通过客户端来过滤实现的。而且它们两个使用的场景区别是非常之大的，RockteMQ主要解决的是基于本地事务和消息的数据一致性，而Kafka的事务则是用于实现它的Exactly-once机制，应用于实时流计算的场景中。

Pulsar的事务消息和Kafka应用场景和语义类似，只是由于底层实现机制有差别，在一些细节上有区别。

相信看到这里就非常清楚了，对于事务消息如何选型和应用，首先要明白你的业务需求是什么。是要实现分布式事务的最终一致性，还是要实现Exactly-once (精确一次)语义？明白之后需求，选择什么组件就十分明确了。

原文链接: https://juejin.cn/post/7383006856664694803