链滴

消息队列笔记 02

作者：lucianolixin

原文链接：https://ld246.com/article/1568815978806

来源网站：链滴

<img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfile.com/bing/20181007.jpg?imageView2/1/w/768/h/432/interlace/1/q/100">
<h2 id="如何利用事务消息实现分布式事务">如何利用事务消息实现分布式事务</h2>
<h3 id="什么是事务">什么是事务</h3>
消息队列中的“事务”，主要解决的事消息生产者和消息消费者的数据一致性问题。我们需要对干数据进行更新操作，多个操作要么都成功要么都失败。一般数据都具有四大特性 ACID ——原子性一致性、隔离性、持久性。在分布式式系统中实现严格的事务代价太大。所以衍生出了 BASE，还比顺序一致性和最终一致性。
<h3 id="分布式事务">分布式事务</h3>
比较常见的分布式事务实现由 2PC(Two-phase Commit）、TTC（Try-Commit-Cannel) 和消事务。事务消息适用于那些异步更新数据，并且对数据实时性要求不太高的场景。
<h3 id="消息队列是如何实现分布式事务的">消息队列是如何实现分布式事务的</h3>
<img src="https://ld246.com/images/img-loading.svg" alt="27ebf12e0dc79e00e1e42c8f0f4e2e6.jpg" data-src="https://b3logfile.com/file/2019/09/27ebf12e0dc79e00e1e42c8ff0f4e26-012c08b7.jpg?imageView2/2/interlace/1/format/jpg">
首先，订单系统在消息队列上开启一个事务。然后订单系统给消息服务器发送一个半消息，这个半消不是说消息内容不完整，它和普通消息的区别是，在事务提交之前，对消费者消息时不可见的。之后订单系统执行本地事务，根据本地事务的执行结果决定是提交还是回滚本地和消息事务。
这里存在一个问题：如果在提交事务消息时失败了怎么办？对于这个问题 Kafka 和 RocketMQ 出了不同解决方案。

Kafka 直接抛异常，让用户自行处理。比如重试，或者补偿。
RocketMQ 提供了事务反查机制。

<img src="https://ld246.com/images/img-loading.svg" alt="11ea249b164b893fb9c36e8ae32577a.jpg" data-src="https://b3logfile.com/file/2019/09/11ea249b164b893fb9c36e86ae3577a-f44f3362.jpg?imageView2/2/interlace/1/format/jpg">
RocketMQ 的 Broker 会定期去检查没有提交的事务，然后 Producer 本地事务的成功与否决定息事务是提交还是回滚。为了实现反查机制，需要业务端提供一个反查本地状态的接口。这种反查机是一种最终一致性，如果是需要比较强一致性的业务，要慎重考虑。
<h2 id="如何保证消息不丢失">如何保证消息不丢失</h2>
<h3 id="利用消息的有序性来验证是否有消息丢失">利用消息的有序性来验证是否有消息丢失</h3>

利用消息拦截器将序号注入消息中，在 Consumer 端监测消息的连续性
每个分区单独监测连续性，而不是 Topic
如果是多个 Producer，也需要各自生成小徐序号

<h3 id="确保消息可靠性传递">确保消息可靠性传递</h3>

生产阶段

处理返回值和捕获异常
异步发送注意回调检查

存储阶段

单节点设置参数同步刷盘
集群需要配置至少将消息发送到两个以上的节点，再给客服端回复确认相应

消费阶段

处理完业务逻辑后再发 ACK

<img src="https://ld246.com/images/img-loading.svg" alt="81a01f5218614efea2838b080879205.jpg" data-src="https://b3logfile.com/file/2019/09/81a01f5218614efea2838b080870920525bc5323.jpg?imageView2/2/interlace/1/format/jpg">

<h2 id="如何处理消费过程中的重复消息">如何处理消费过程中的重复消息</h2>
<h3 id="消息重复必然存在">消息重复必然存在</h3>
在 MQTT 协议中给出传递消息时能够提供的服务质量标准，从低到高此次是

At most once： 之多一次。消息在传递时，最多会送达一次。
At least once： 至少一次。消息在传递时，至少会送达一次。
Exactly once： 恰好一次。消息在传递时，恰好传递一次。

现在主流的消息队列支持的都是 At least once ， RabbitMQ、RocketMQ、Kafka 都是这样，就是意味着不能保证消息不重复。
<h3 id="用幂等性解决重复消息问题">用幂等性解决重复消息问题</h3>
一般决绝重复消息的办法是，在消费端，让我们消费消息的操作具有幂等性。
幂等（Idempotence）本来是一个数学概念，它是这样定义的：
<blockquote>
如果函数 f(x)满足：f(f(x)) = f(x) ,则函数 f(x)满足幂等性。 ‘
</blockquote>
这个概念被拓展到计算机领域，被用作描述一个操作、方法或者服务。一个幂等操作的特点是，任意多次执行所产生的影响，均与一次产生的影响相同。
从系统结果来说：At least once + 幂等消费 = Exactly once
最好的实现幂等操作是，从业务逻辑设计上入手，将消费的业务逻辑设计成具有幂等性的操作。
<h4 id="1-利用数据的唯一约束实现幂等">1.利用数据的唯一约束实现幂等</h4>
通过数据的流水表中的唯一约束，实现写入多条重复数据会失败，这样实现了幂等操作。基于这思路，不光是可以使用关系型数据库，只要支持类似“insert if not exist”语义的存储类系统都可以现幂等。比如，redis 的 setnx 命令来代替唯一约束，来实现幂等。
<h4 id="2-为更新的数据设置前置条件">2.为更新的数据设置前置条件</h4>
另外一个实现幂等思路的事，给拘束变更设置一个前置条件，如果满足条件就更新数据，否则拒更新数据，在更新数据的时候，同时变更前置条件中需要判断的数据。这样，重复操作的时候，由于一次更新数据的时候已经变更了前置条件中巫妖判断的数据，不满足前置条件，则不会更新数据。</>
这种情况更通用的方法是，给你的数据增加一个版本号属性，每次更新数据之前，比较当前数据版本号是否和消息中的版本号一致，如果不一致则拒绝更新数据，更新数据的同事版本号 +1，一样以实现幂等性。
<h4 id="3-记录并检查操作">3.记录并检查操作</h4>
这种通用性最强，适用范围最广的实现幂等方法：记录并检查操作。也称作为“token 机制或者 UID（全局唯一 ID）机制” ，实现的思路是：在执行数据更新之前，先检查是否执行过这个更新操。
具体方法是，在发送消息时，给每个消息指定一个全局唯一的 ID，消费时，先根据这个 ID 检查条消息是否被消费过，如果没有消费过，才更新数据，然后将消费状态置为已消费。
由生产者将不同业务的不同唯一约束（如 A 业务是 a+b 字段须唯一，B 业务是 a+c 字段须唯一，统一处理成对消费者友好的全局唯一 ID，如 A 业务是 md5(a+b)，B 业务是 md5(a+c)。生成全唯一 ID，可以是上面举例的本地 md5 计算，也可以是包装成服务接口，但其本身也必须是幂等的，样一来，消费者不管处理什么业务消息，都只需要针对"全局唯一 ID"来保证幂等即可
<h3 id="消息积压了该如何处理">消息积压了该如何处理</h3>

发送端性能优化

如果说，你的代码发哦少年宫消息的性能上不去，你需要有限检查一下，是不是发消息之前的业

逻辑耗时太多导致的。
对于发送消息的业务逻辑，只需要注意设计合适的并发和批量大小，就可以达到很好的发送性能
Producter 发送消息给 Broker，Broker 收到消息后返回确认相应，这是一次完整的交互。

发送端准备数据、序列化消息、构造请求等逻辑的时间，也就是发送端在发送网络请求之前的耗
发送消息和返回消息的网络传输的耗时
Broker 处理消息的时延

如果是单线程发送，每次发送 1 条消息，那么每秒只能发送 1000ms/1ms*1 条/ms = 1000 条 消息，这种情况下并不能发挥出消息队列的全部实力。
提升性能的方式

批量发送
增加并发

根据业务的性质来选择实现方式。比如在线业务多用增加并发来解决。李先分析系统不关心时延更关注系统的吞吐量。发送端的数据都是来自数据库，这种情况就更适合批量发送，可以批量从数据读取数据，然后批量来发送消息。

消费端性能优化

使用消息队列的时候，大部分的性能主要出现在消费端，如果消费的速度跟不上发送端生产消息速度，就会造成消息积压。如果消费速度一直比生产消息慢，时间长了，整个系统就会出现问题，要，消息队列的存储被填满无法提供服务，要么消息丢失，这对于整个系统来说都是严重的故障。
所以，一定要保证消费端的消费性能高于生产端的发送性能，这样才能持续健康的运行。

优化消费业务逻辑
水平扩容

水平扩容 consumer 的实力数量的同时，必须同步扩容主题中的分区（也叫队列）数量，确保 consumer 的实力数和分区数量是相等的。如果 consumer 的实例数量超过分区的数量，这样的扩容是有效果的。因为对于每个消费者来说，每个分区上时机上只能支持单线程消费。
很多消费程序，他们是这样解决消费慢的问题的：

在收到消息的 onMessage 方法中，不处理任何逻辑，把这个消息放到一个内存队列中就返回了后面启动很多业务线程，去处理消息的业务逻辑。这里存在一个严重的问题，就是当节点发生宕机的候，在内存队列中还没来得及处理的消息会丢失。
<h4 id="消息积压了该如何处理-">消息积压了该如何处理</h4>
导致消息积压的原因只有两种:要么发送快了，要么消费变慢了。

扩容消费端实例
降级不重要的系统，减少生产消息的速度
如果，消费和生产速度正常需要检查是否有消费失败导致一条消息重复消费这种情况笔记哦啊多

不管怎样，系统监控在排查问题中是十分重要的。