基本概念

2020/6/28 09:17

1、消息模型

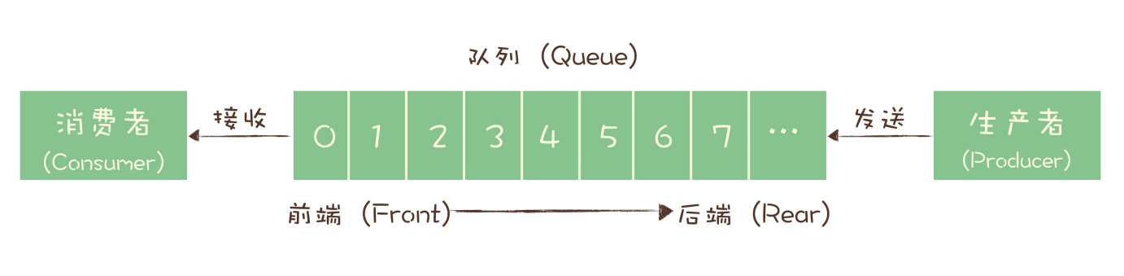
1.1、种类

主要是队列模型和发布订阅模型。

队列模型就是基于数据结构中的队列：

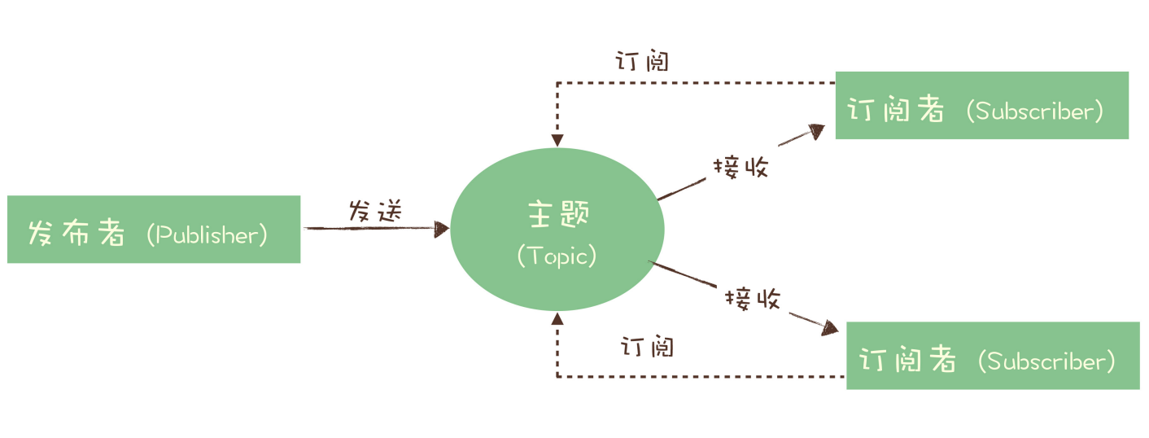
队列是先进先出（FIFO, First-In-First-Out）的线性表（Linear List）。在具体应用中通常用链表或者数组来实现。队列只允许在后端（称为 rear）进行插入操作，在前端（称为 front）进行删除操作。

队列中的元素是严格有序的，且没有只读操作，它的读操作就是出队：



有序性是消息队列需要满足的最基本的特性，这里的很多架构演进过程都需要保证消息的有序性。

这样当有多个消费者需要同一份消息的时候，就无法做到了，大家都只能消费到对列中的部分数据。所以又有了发布订阅模型，把存储消息的容器改为了“主题”



此时，消息的发送方被称为发布者，消息的接收方称为订阅者。发布者将消息发送到主题中，订阅者在接收消息之前需要先“订阅主题”。

“订阅”在这里既是一个动作，同时还可以认为是主题在消费时的一个逻辑副本，每份订阅中，订阅者都可以接收到主题的所有消息。

订阅作为主题的逻辑副本，就像一个逻辑上的队列，这样每个订阅者就可以有一个自己专属的逻辑上的队列了。

1.2、异同

在消息领域的历史上很长的一段时间，队列模式和发布 - 订阅模式是并存的，有些消息队列同时支持这两种消息模型，比如 ActiveMQ。

不同点：这两种模型，生产者就是发布者，消费者就是订阅者，队列就是主题，并没有本质的区别。它们最大的区别其实就是，一份消息数据能不能被消费多次的问题。

共通点：在这种发布 - 订阅模型中，如果只有一个订阅者，那它和队列模型就基本是一样的了。也就是说，发布 - 订阅模型在功能层面上是可以兼容队列模型的。

1.3、要点

假设有一个主题 MyTopic，我们为主题创建 5 个队列，分布到 2 个 Broker 中。

Broker	主题	队列
Broker0	MyTopic	Q0，Q1
Broker1	MyTopic	Q2，Q3，Q4

假设我们有 3 个生产者实例：Produer0，Produer1 和 Producer2。

- 生产者、BROKER、队列之间没有对应关系。每个生产者可以在 5 个队列中轮询发送，也可以随机选一个队列发送，或者只往某个队列发送。
- 每个消费组就是一份订阅，它要消费主题 MyTopic 下，所有队列的全部消息。注意，队列里的消息并不是消费掉就没有了，这里的“消费”，只是去队列里面读了消息，并没有删除，消费完这条消息还是在队列里面。
- 多个消费组在消费同一个主题时，消费组之间是互不影响的。
- 一个消费组中可以包含多个消费者的实例。比如说消费组 G1，包含了 2 个消费者 C0 和 C1，那这 2 个消费者又是怎么和主题 MyTopic 的 5 个队列对应的呢？

  首先，由于消费确认机制的限制，在同一个消费组里面，（应该是在同一个时刻——因为保证有序性带来的并发控制的原因）**每个队列只能被一个消费者实例占用**。所以，**必须做到保证每个队列分配一个消费者就行了**，即多对一的关系。

- 每个消费组内部维护自己的一组消费位置，每个队列对应一个消费位置。消费位置在服务端保存，并且，消费位置和消费者是没有关系的。每个消费位置一般就是一个整数。
- 如何保证消息的严格顺序？主题层面是无法保证严格顺序的，只有在队列上才能保证消息的严格顺序。如果业务必须要求**全局严格顺序**，就只能把消息队列数配置成 1，生产者和消费者也只能是一个实例，这样才能保证全局严格顺序。如果需要保证**局部严格顺序**，可以这样来实现。在发送端，我们使用账户 ID 作为 Key，采用一致性哈希算法计算出队列编号，指定队列来发送消息。

2、具体实现时需要考虑的非功能性需求

消息模型解决了业务需求，但是具体产品还要满足可水平扩展这一非功能需求。具体来说，是在高可靠前提下的可水平扩展，进一步说，也就是在高可靠前提下的高可用需求。

2.1、如何保证可靠性

几乎所有的消息队列产品都使用一种非常朴素的“请求 - 确认”机制，确保消息不会在传递过程中由于网络或服务器故障丢失。

在**生产端**，生产者先将消息发送给服务端，也就是 Broker，服务端在收到消息并将消息写入主题或者队列中后，会给生产者发送确认的响应。如果生产者没有收到服务端的确认或者收到失败的响应，则会重新发送消息。

在**消费端**，消费者在收到消息并完成自己的消费业务逻辑（比如，将数据保存到数据库中）后，也会给服务端发送消费成功的确认，服务端只有收到消费确认后，才认为一条消息被成功消费，否则它会给消费者重新发送这条消息，直到收到对应的消费成功确认。

2.2、可靠性带来的新问题

引入这个机制在消费端带来了一个不小的问题。什么问题呢？为了确保消息的有序性，在某一条消息被成功消费之前，下一条消息是不能被消费的，否则就会出现消息空洞，违背了**有序性**这个原则。

也就是说，每个主题在任意时刻，每个下游系统至多只能有一个消费者实例在进行消费，那就没法通过**水平扩展**下游系统消费者的数量来提升消费端总体的消费性能。

上面为了满足多个消费者消费同一份数据的需要，将主题分成了多个逻辑队列——订阅。为了解决水平扩展问题，又将订阅“水平扩展”为多个队列（RocketMQ）或分区（Kafka）。

即将有序性问题转移给了队列或分区，也即在队列或分区上，消息是有序的（因为上一个消息被确认后，才能消费下一个消息），从而可以在主题上取消有序性的限制，即主题层面不再保证消息的严格有序性。

每个主题（觉得这里用“订阅”应该更合适）包含多个队列，通过多个队列来实现多实例并行生产和消费。需要注意的是，只在队列或分区上保证消息的有序性，主题层面是无法保证消息的严格顺序的。

最后，要注意到，队列或者分区是为了实现水平扩展而来，**从水平扩展或者消费性能的角度来说**，一个队列或分区对应一个消费者，消费者的数量和分区的数量是对应的，在规划容量时，要考虑好数量，避免消费能力不足导致消息积压。

为了容易检测到丢消息，Consumer 实例的数量最好和分区数量一致，做到 Consumer 和分区一一对应。参见第4节。

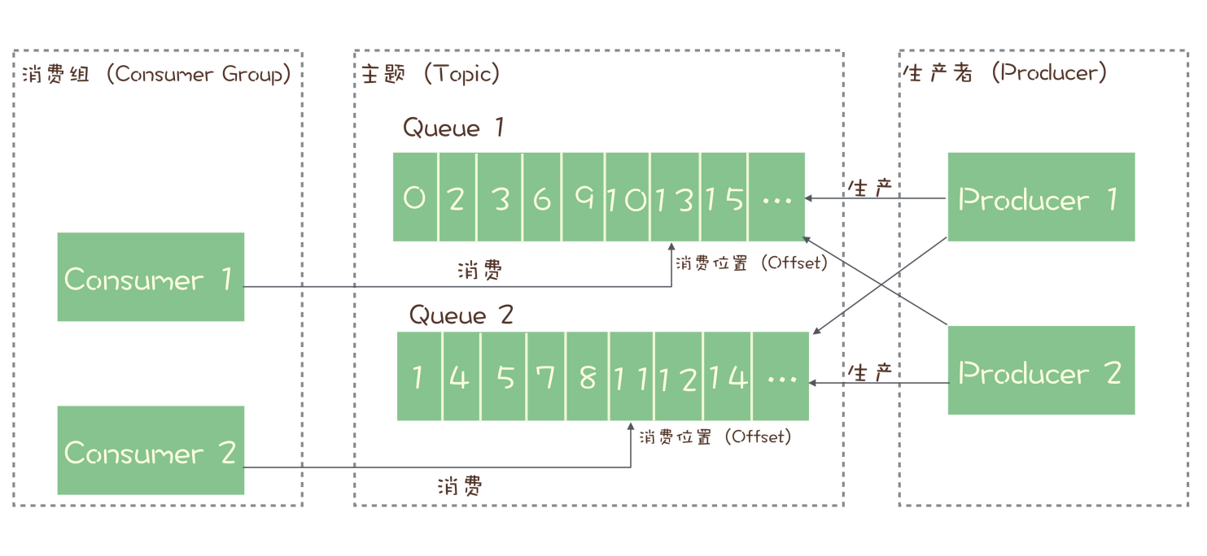
2.2.1、RocketMQ

受中文的限制，订阅者和发布者应该都是复数形式，所以应该写成“订阅者's”和“发布者's”。

RocketMQ 中，订阅者的概念是通过消费组（Consumer Group）来体现的。每个消费组都消费主题中一份完整的消息，不同消费组之间消费进度彼此不受影响，也就是说，一条消息被 Consumer Group1 消费过，也会再给 Consumer Group2 消费。

消费组中包含多个消费者，同一个组内的消费者是竞争消费的关系，每个消费者负责消费组内的一部分消息。如果一条消息被消费者 Consumer1 消费了，那同组的其他消费者就不会再收到这条消息。

在 Topic 的消费过程中，由于消息需要被不同的组进行多次消费，所以消费完的消息并不会立即被删除，这就需要 RocketMQ 为每个消费组在每个队列上维护一个消费位置（Consumer Offset），这个位置之前的消息都被消费过，之后的消息都没有被消费过，每成功消费一条消息，消费位置就加一。这个消费位置是非常重要的概念，我们在使用消息队列的时候，丢消息的原因大多是由于消费位置处理不当导致的。



2.2.2、KAFKA

Kafka 的消息模型和 RocketMQ 是完全一样的。所有 RocketMQ 中对应的概念，和生产消费过程中的确认机制，都完全适用于 Kafka。唯一的区别是，**在 Kafka 中，队列这个概念的名称不一样，Kafka 中对应的名称是“分区（Partition）”，含义和功能是没有任何区别的。**

分区还必须保证高可用。

实现高可用的另一个手段就是备份机制（Replication）。备份的思想很简单，就是把相同的数据拷贝到多台机器上，而这些相同的数据拷贝在 Kafka 中被称**为副本（Replica）**。

副本的数量是可以配置的，这些副本保存着相同的数据，但却有不同的角色和作用。Kafka 定义了两类副本：**领导者副本（Leader Replica）**和**追随者副本（Follower Replica）**。前者对外提供服务，这里的对外指的是与客户端程序进行交互；而后者只是被动地追随领导者副本而已，不能与外界进行交互。

副本的工作机制也很简单：**生产者总是向领导者副本写消息；而消费者总是从领导者副本读消息。**至于追**随者副本，它只做一件事：向领导者副本发送请求，请求领导者把最新生产的消息发给它**，这样它能保持与领导者的同步。即追随者副本完全不对外提供服务。

副本如何与这里的分区联系在一起呢？实际上，**副本是在分区这个层级定义的。**每个分区下可以配置若干个副本，其中只能有 1 个领导者副本和 N-1 个追随者副本。生产者向分区写入消息，每条消息在分区中的位置信息由一个叫位移（Offset）的数据来表征。分区位移总是从 0 开始，假设一个生产者向一个空分区写入了 10 条消息，那么这 10 条消息的位移依次是 0、1、2、…、9。

Kafka 的三层消息架构：

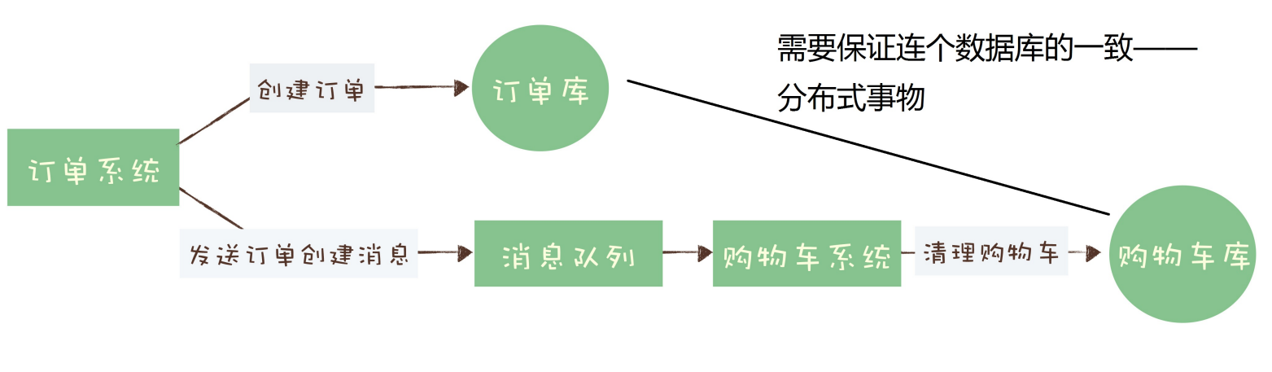
- 第一层是主题层，每个主题可以配置 M 个分区，而每个分区又可以配置 N 个副本。
- 第二层是分区层，每个分区的 N 个副本中只能有一个充当领导者角色，对外提供服务；其他 N-1 个副本是追随者副本，只是提供数据冗余之用。
- 第三层是消息层，分区中包含若干条消息，每条消息的位移从 0 开始，依次递增。
- 最后，客户端程序只能与分区的领导者副本进行交互。

3、事务消息与分布式事物

其实很多场景下，我们“发消息”这个过程，目的往往是通知另外一个系统或者模块去更新数据，消息队列中的“事务”，主要解决的是消息生产者和消息消费者的数据一致性问题。

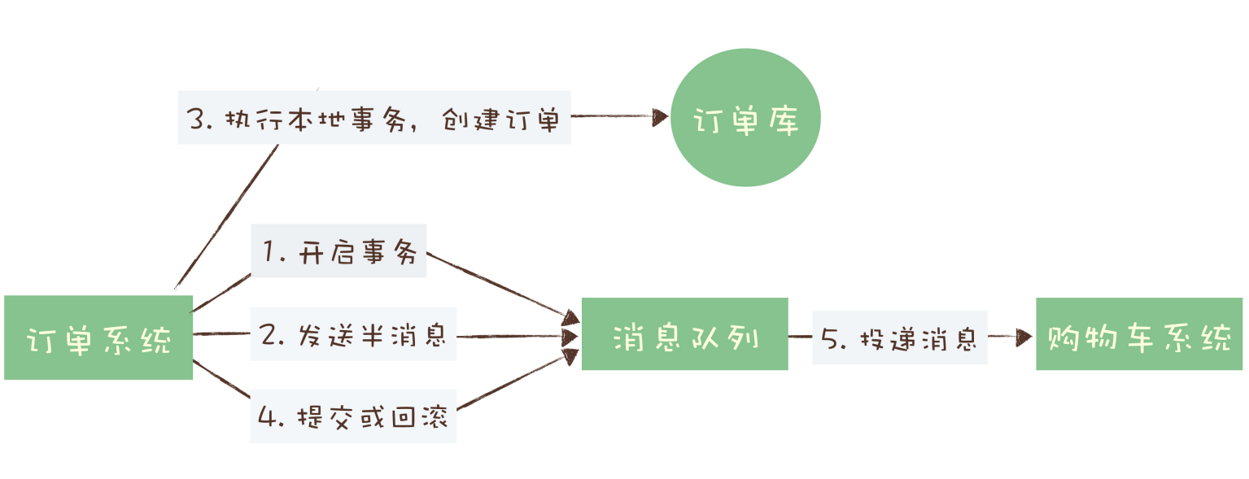
ACID参见《分布式系统——分布式协调》

拿我们熟悉的电商来举个例子。一般来说，用户在电商 APP 上购物时，先把商品加到购物车里，然后几件商品一起下单，最后支付，完成购物流程，就可以愉快地等待收货了。



需要保证连个数据库的一致——分布式事物

事务消息需要消息队列提供相应的功能才能实现，Kafka 和 RocketMQ 都提供了事务相关功能。

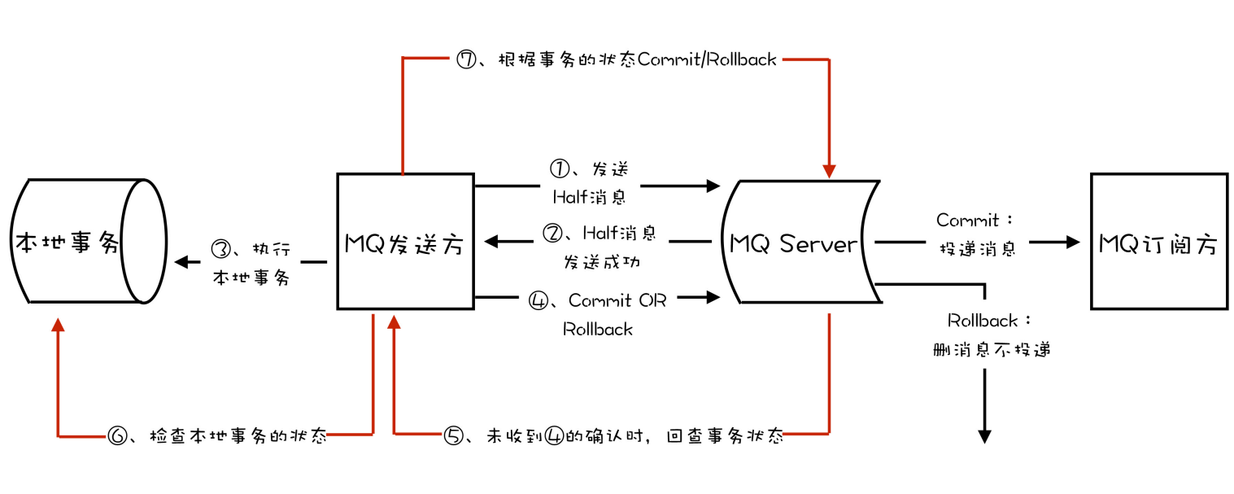


上图中第2步订单系统给消息服务器发送一个“**半消息**”，这个半消息不是说消息内容不完整，它包含的内容就是完整的消息内容，半消息和普通消息的唯一区别是，在事务提交之前，对于消费者来说，这个消息是不可见的。

上图中第4步提交或者回滚是指提交或者回滚事务消息。这一步也有可能失败，对于这个问题，Kafka 和 RocketMQ 给出了 2 种不同的解决方案。

（1）Kafka 的解决方案比较简单粗暴，直接抛出异常，让用户自行处理。我们可以在业务代码中反复重试提交，直到提交成功，或者删除之前创建的订单进行补偿。

（2）在 RocketMQ 中的事务实现中，增加了事务反查的机制来解决事务消息提交失败的问题。如果 Producer 也就是订单系统，在提交或者回滚事务消息时发生网络异常，RocketMQ 的 Broker 没有收到提交或者回滚的请求，Broker 会定期去 Producer 上反查这个事务对应的本地事务的状态，然后根据反查结果决定提交或者回滚这个事务。为了支撑这个事务反查机制，我们的业务代码需要实现一个反查本地事务状态的接口，告知 RocketMQ 本地事务是成功还是失败。



4、确保消息不丢失

现在主流的消息队列产品都提供了非常完善的消息可靠性保证机制，完全可以做到在消息传递过程中，即使发生网络中断或者硬件故障，也能确保消息的可靠传递，不丢消息。

绝大部分丢消息的原因都是由于开发者不熟悉消息队列，没有正确使用和配置消息队列导致的。虽然不同的消息队列提供的 API 不一样，相关的配置项也不同，但是在保证消息可靠传递这块儿，它们的实现原理是一样的。

我们可以利用消息队列的有序性来验证是否有消息丢失。原理非常简单，在 Producer 端，我们给每个发出的消息附加一个连续递增的序号，然后在 Consumer 端来检查这个序号的连续性。

如果没有消息丢失，Consumer 收到消息的序号必然是连续递增的，或者说收到的消息，其中的序号必然是上一条消息的序号 +1。如果检测到序号不连续，那就是丢消息了。还可以通过缺失的序号来确定丢失的是哪条消息，方便进一步排查原因。

大多数消息队列的客户端都支持拦截器机制，你可以利用这个拦截器机制，在 Producer 发送消息之前的拦截器中将序号注入到消息中，在 Consumer 收到消息的拦截器中检测序号的连续性，这样实现的好处是消息检测的代码不会侵入到你的业务代码中，待你的系统稳定后，也方便将这部分检测的逻辑关闭或者删除。

像 Kafka 和 RocketMQ 这样的消息队列，它是不保证在 Topic 上的严格顺序的，只能保证分区上的消息是有序的（参见2.2节），**所以我们在发消息的时候必须要指定分区，并且，在每个分区单独检测消息序号的连续性。**

Consumer 实例的数量最好和分区数量一致，做到 Consumer 和分区一一对应，这样会比较方便地在 Consumer 内检测消息序号的连续性。

即将有序性问题转移给了队列或分区，也即在队列或分区上，消息是有序的（因为上一个消息被确认后，才能消费下一个消息），从而可以在主题上取消有序性的限制，即主题层面不再保证消息的严格有序性。

不丢消息主要是要利用好消息队列的请求确认机制。

4.1、消息发送

可以同步发送，也可以异步发送。同步发送注意捕获异常，异步发送注意在回调中进行检查。

4.2、存储消息

对于单节点的BROKER，可以设置同步刷盘。

对于集群，需要将 Broker 集群配置成：至少将消息发送到 2 个以上的节点，再给客户端回复发送确认响应。

4.3、消费消息

在编写消费代码时需要注意的是，不要在收到消息后就立即发送消费确认，而是应该在执行完所有消费业务逻辑之后，再发送消费确认。

5、处理重复消息

在 MQTT 协议中，给出了三种传递消息时能够提供的服务质量标准（对所有的消息队列都是适用的），这三种服务质量从低到高依次是：

At most once: 至多一次。消息在传递时，最多会被送达一次。换一个说法就是，没什么消息可靠性保证，**允许丢消息**。一般都是一些对消息可靠性要求不太高的监控场景使用，比如每分钟上报一次机房温度数据，可以接受数据少量丢失。**大部分消息队列都选择只提供 At least once 的服务质量。**
At least once: 至少一次。消息在传递时，至少会被送达一次。也就是说，**不允许丢消息，但是允许有少量重复消息出现。**
Exactly once：恰好一次。消息在传递时，只会被送达一次，不允许丢失也不允许重复，这个是最高的等级。

5.1、用幂等性解决重复消息问题

At least once + 幂等消费 = Exactly once

那么如何实现幂等操作呢？最好的方式就是，**从业务逻辑设计上入手，将消费的业务逻辑设计成具备幂等性的操作。**

- 利用数据库的唯一约束实现幂等。

  比如利用数据库的唯一性约束来实现去重，可以增加一个流水表，以业务标示作为主键。

  不光是可以使用关系型数据库，只要是支持类似“INSERT IF NOT EXIST”语义的存储类系统都可以用于实现幂等，比如，你可以用 Redis 的 SETNX 命令来替代数据库中的唯一约束，来实现幂等消费。

- 为更新的数据设置前置条件。

  类似于乐观锁。

  比较通用的方法是，给你的数据增加一个版本号属性，每次更数据前，比较当前数据的版本号是否和消息中的版本号一致，如果不一致就拒绝更新数据，更新数据的同时将版本号 +1，一样可以实现幂等更新。

- 记录并检查操作。

  通用性最强，适用范围最广的实现幂等性方法：记录并检查操作，也称为“Token 机制或者 GUID（全局唯一 ID）机制”，实现的思路特别简单：在执行数据更新操作之前，先检查一下是否执行过这个更新操作。

  具体的实现方法是，在发送消息时，给每条消息指定一个全局唯一的 ID，消费时，先根据这个 ID 检查这条消息是否有被消费过，如果没有消费过，才更新数据，然后将消费状态置为已消费。

  对于这个问题，当然我们可以用事务来实现，也可以用锁来实现，但是在分布式系统中，无论是分布式事务还是分布式锁都是比较难解决问题。**即实现起来且同时满足简单、高可用和高性能不容易。**

6、处理消息积压

消息积压的直接原因，一定是系统中的某个部分出现了性能问题，来不及处理上游发送的消息，才会导致消息积压。

6.1、优化性能来避免消息积压

注意，发送端都是先执行自己的业务逻辑，最后再发送消息。

6.1.1、发送端性能优化

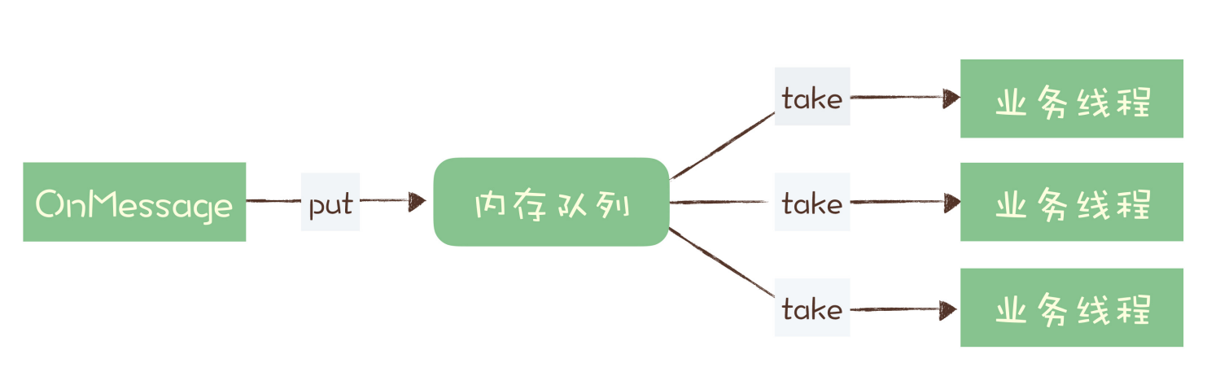
对于发送消息的业务逻辑，只需要注意设置合适的并发或者批量大小，就可以达到很好的发送性能。

6.1.2、消费端性能优化

在设计系统的时候，一定要保证消费端的消费性能要高于生产端的发送性能，这样的系统才能健康的持续运行。否则，要么，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故障。

消费端的性能优化除了优化消费业务逻辑以外，也可以通过水平扩容，增加消费端的并发数来提升总体的消费性能。特别需要注意的一点是，**在扩容 Consumer 的实例数量的同时，必须同步扩容主题中的分区（也叫队列）数量，确保 Consumer 的实例数和分区数量是相等的。**如果 Consumer 的实例数量超过分区数量，这样的扩容实际上是没有效果的，因为要保证队列或分区上消息的有效性，参见2.2节。

一个常见错误做法。



为了避免消息积压，在收到消息的 OnMessage 方法中，不处理任何业务逻辑，把这个消息放到一个内存队列里面就返回了。然后它可以启动很多的业务线程，这些业务线程里面是真正处理消息的业务逻辑，这些线程从内存队列里取消息处理。

但这是一个非常常见的错误方法！ 为什么错误？因为会丢消息。如果收消息的节点发生宕机，在内存队列中还没来及处理的这些消息就会丢失。

6.2、解决消息积压

日常系统正常运转的时候，没有积压或者只有少量积压很快就消费掉了，但是某一个时刻，突然就开始积压消息并且积压持续上涨。这种情况下需要在短时间内找到消息积压的原因，迅速解决问题才不至于影响业务。

能导致积压突然增加，最粗粒度的原因，只有两种：要么是发送变快了，要么是消费变慢了。

- 如果是单位时间发送的消息增多，唯一的方法是通过扩容消费端的实例数来提升总体的消费能力。
- 如果短时间内没有足够的服务器资源进行扩容，没办法的办法是，将系统降级，通过关闭一些不重要的业务，减少发送方发送的数据量，最低限度让系统还能正常运转，服务一些重要业务。
- 还有一种不太常见的情况，你通过监控发现，无论是发送消息的速度还是消费消息的速度和原来都没什么变化，这时候你需要检查一下你的消费端，是不是消费失败导致的一条消息反复消费这种情况比较多，这种情况也会拖慢整个系统的消费速度。比如发生了死锁。