**消息队列设计精要**

消息队列已经逐渐成为企业IT系统内部通信的核心手段。它具有低耦合、可靠投递、广播、流量控制、最终一致性等一系列功能，成为异步RPC的主要手段之一。

## 何时需要消息队列

可以使用mq的场景有很多，最常用的几种，是做业务解耦/最终一致性/广播/错峰流控等。反之，如果需要强一致性，关注业务逻辑的处理结果，则RPC显得更为合适。

### 解耦

解耦是消息队列要解决的最本质问题。所谓解耦，简单点讲就是一个事务，只关心核心的流程。而需要依赖其他系统但不那么重要的事情，有通知即可，无需等待结果。换句话说，基于消息的模型，关心的是“通知”，而非“处理”。

比如在美团旅游，我们有一个产品中心，产品中心上游对接的是主站、移动后台、旅游供应链等各个数据源；下游对接的是筛选系统、API系统等展示系统。当上游的数据发生变更的时候，如果不使用消息系统，势必要调用我们的接口来更新数据，就特别依赖产品中心接口的稳定性和处理能力。但其实，作为旅游的产品中心，也许只有对于旅游自建供应链，产品中心更新成功才是他们关心的事情。而对于团购等外部系统，产品中心更新成功也好、失败也罢，并不是他们的职责所在。他们只需要保证在信息变更的时候通知到我们就好了。  
而我们的下游，可能有更新索引、刷新缓存等一系列需求。对于产品中心来说，这也不是我们的职责所在。说白了，如果他们定时来拉取数据，也能保证数据的更新，只是实时性没有那么强。但使用接口方式去更新他们的数据，显然对于产品中心来说太过于“重量级”了，只需要发布一个产品ID变更的通知，由下游系统来处理，可能更为合理。

举个例子，对于我们的订单系统，订单最终支付成功之后可能需要给用户发送短信积分什么的，但其实这已经不是我们系统的核心流程了。如果外部系统速度偏慢（比如短信网关速度不好），那么主流程的时间会加长很多，用户肯定不希望点击支付过好几分钟才看到结果。那么我们只需要通知短信系统“我们支付成功了”，不一定非要等待它处理完成。

更新数据时如果使用接口就显得太过重量级了。

### 最终一致性

最终一致性指的是两个系统的状态保持一致，要么都成功，要么都失败。当然有个时间限制，理论上越快越好，但实际上在各种异常的情况下，可能会有一定延迟达到最终一致状态，但最后两个系统的状态是一样的。

以一个银行的转账过程来理解最终一致性，转账的需求很简单，如果A系统扣钱成功，则B系统加钱一定成功。反之则一起回滚，像什么都没发生一样。

然而，这个过程中存在很多可能的意外：

1. A扣钱成功，调用B加钱接口失败。
2. A扣钱成功，调用B加钱接口虽然成功，但获取最终结果时网络异常引起超时。
3. A扣钱成功，B加钱失败，A想回滚扣的钱，但A机器down机。

可见，想把这件看似简单的事真正做成，真的不那么容易。所有跨VM的一致性问题，从技术的角度讲通用的解决方案是：

1. 强一致性，分布式事务，但落地太难且成本太高，后文会具体提到。
2. 最终一致性，主要是用“记录”和“补偿”的方式。在做所有的不确定的事情之前，先把事情记录下来，然后去做不确定的事情，结果可能是：成功、失败或是不确定，“不确定”（例如超时等）可以等价为失败。成功就可以把记录的东西清理掉了，对于失败和不确定，可以依靠定时任务等方式把所有失败的事情重新搞一遍，直到成功为止。  
    回到刚才的例子，系统在A扣钱成功的情况下，把要给B“通知”这件事记录在库里（为了保证最高的可靠性可以把通知B系统加钱和扣钱成功这两件事维护在一个本地事务里），通知成功则删除这条记录，通知失败或不确定则依靠定时任务补偿性地通知我们，直到我们把状态更新成正确的为止。

具体来说，本地事务维护业务变化和通知消息，一起落地（失败则一起回滚），然后RPC到达broker，在broker成功落地后，RPC返回成功，本地消息可以删除。否则本地消息一直靠定时任务轮询不断重发，这样就保证了消息可靠落地broker。broker往consumer发送消息的过程类似，一直发送消息，直到consumer发送消费成功确认。  
我们先不理会重复消息的问题，通过两次消息落地加补偿，下游是一定可以收到消息的。然后依赖状态机版本号等方式做判重，更新自己的业务，就实现了最终一致性。

最终一致性不是消息队列的必备特性，但确实可以依靠消息队列来做最终一致性的事情。另外，所有不保证100%不丢消息的消息队列，理论上无法实现最终一致性。哪怕只丢千分之一的消息，业务也必须用其他的手段来保证结果正确。

### 广播

消息队列的基本功能之一是进行广播。如果没有消息队列，每当一个新的业务方接入，我们都要联调一次新接口。有了消息队列，我们只需要关心消息是否送达了队列，至于谁希望订阅，是下游的事情，无疑极大地减少了开发和联调的工作量。

### 错峰与流控

试想上下游对于事情的处理能力是不同的。比如，Web前端每秒承受上千万的请求，并不是什么神奇的事情，只需要加多一点机器，再搭建一些LVS负载均衡设备和Nginx等即可。但数据库的处理能力却十分有限，即使使用SSD加分库分表，单机的处理能力仍然在万级。由于成本的考虑，我们不能奢求数据库的机器数量追上前端。  
 这种问题同样存在于系统和系统之间，如短信系统可能由于短板效应，速度卡在网关上（每秒几百次请求），跟前端的并发量不是一个数量级。但用户晚上个半分钟左右收到短信，一般是不会有太大问题的。如果没有消息队列，两个系统之间通过协商、滑动窗口等复杂的方案也不是说不能实现。但系统复杂性指数级增长，势必在上游或者下游做存储，并且要处理定时、拥塞等一系列问题。而且每当有处理能力有差距的时候，都需要单独开发一套逻辑来维护这套逻辑。所以，利用中间系统转储两个系统的通信内容，并在下游系统有能力处理这些消息的时候，再处理这些消息，是一套相对较通用的方式。

总而言之，消息队列不是万能的。对于需要强事务保证而且延迟敏感的，RPC是优于消息队列的。  
对于一些无关痛痒，或者对于别人非常重要但是对于自己不是那么关心的事情，可以利用消息队列去做。  
支持最终一致性的消息队列，能够用来处理延迟不那么敏感的“分布式事务”场景，而且相对于笨重的分布式事务，可能是更优的处理方式。  
 当上下游系统处理能力存在差距的时候，利用消息队列做一个通用的“漏斗”。在下游有能力处理的时候，再进行分发。

## 如何设计一个消息队列

我们之所以要设计一个消息队列，并且配备broker，无外乎要做两件事情：

1. 消息的转储，在更合适的时间点投递，或者通过一系列手段辅助消息最终能送达消费机。
2. 规范一种范式和通用的模式，以满足解耦、最终一致性、错峰等需求。  
   掰开了揉碎了看，最简单的消息队列可以做成一个消息转发器，把一次RPC做成两次RPC。发送者把消息投递到服务端（以下简称broker），服务端再将消息转发一手到接收端，就是这么简单。

一般来讲，设计消息队列的整体思路是先build一个整体的数据流,例如producer发送给broker,broker发送给consumer,consumer回复消费确认，broker删除/备份消息等。  
利用RPC将数据流串起来。然后考虑RPC的高可用性，尽量做到无状态，方便水平扩展。  
之后考虑如何承载消息堆积，然后在合适的时机投递消息，而处理堆积的最佳方式，就是存储，存储的选型需要综合考虑性能/可靠性和开发维护成本等诸多因素。  
为了实现广播功能，我们必须要维护消费关系，可以利用zk/config server等保存消费关系。  
 在完成了上述几个功能后，消息队列基本就实现了。然后我们可以考虑一些高级特性，如可靠投递，事务特性，性能优化等。

### 实现队列基本功能

#### RPC通信协议

所谓消息队列，无外乎两次RPC加一次转储，当然需要消费端最终做消费确认的情况是三次RPC。既然是RPC，就必然牵扯出一系列话题，负载均衡、服务发现、通信协议、序列化协议等。简单来讲，服务端提供两个RPC服务，一个用来接收消息，一个用来确认消息收到。并且做到不管哪个server收到消息和确认消息，结果一致即可。

#### 高可用

其实所有的高可用，是依赖于RPC和存储的高可用来做的。先来看RPC的高可用，美团的基于MTThrift的RPC框架，阿里的Dubbo等，其本身就具有服务自动发现，负载均衡等功能。而消息队列的高可用，只要保证broker接受消息和确认消息的接口是幂等的，并且consumer的几台机器处理消息是幂等的，这样就把消息队列的可用性，转交给RPC框架来处理了。

那么怎么保证幂等呢？最简单的方式莫过于共享存储。broker多机器共享一个DB或者一个分布式文件/kv系统，则处理消息自然是幂等的。就算有单点故障，其他节点可以立刻顶上。另外failover可以依赖定时任务的补偿，这是消息队列本身天然就可以支持的功能。存储系统本身的可用性我们不需要操太多心，放心大胆的交给DBA们吧！  
对于不共享存储的队列，如Kafka使用分区加主备模式，就略微麻烦一些。需要保证每一个分区内的高可用性，也就是每一个分区至少要有一个主备且需要做数据的同步，关于这块HA的细节，可以参考下篇pull模型消息系统设计。

#### 服务端承载消息堆积的能力

消息到达服务端如果不经过任何处理就到接收者了，broker就失去了它的意义。为了满足我们错峰/流控/最终可达等一系列需求，把消息存储下来，然后选择时机投递就显得是顺理成章的了。只是这个存储可以做成很多方式。比如存储在内存里，存储在分布式KV里，存储在磁盘里，存储在数据库里等等。但归结起来，主要有持久化和非持久化两种。  
持久化的形式能更大程度地保证消息的可靠性（如断电等不可抗外力），并且理论上能承载更大限度的消息堆积（外存的空间远大于内存）。但并不是每种消息都需要持久化存储。很多消息对于投递性能的要求大于可靠性的要求，且数量极大（如日志）。这时候，消息不落地直接暂存内存，尝试几次failover，最终投递出去也未尝不可。市面上的消息队列普遍两种形式都支持。当然具体的场景还要具体结合公司的业务来看。

#### 消费关系解析

市面上的消息队列定义了一堆让人晕头转向的名词，如JMS 规范中的Topic/Queue，Kafka里面的Topic/Partition/ConsumerGroup，RabbitMQ里面的Exchange等等。抛开现象看本质，无外乎是单播与广播的区别。所谓单播，就是点到点；而广播，是一点对多点。当然，对于互联网的大部分应用来说，组间广播、组内单播是最常见的情形。消息需要通知到多个业务集群，而一个业务集群内有很多台机器，只要一台机器消费这个消息就可以了。  
 当然这不是绝对的，很多时候组内的广播也是有适用场景的，如本地缓存的更新等等。另外，消费关系除了组内组间，可能会有多级树状关系。这种情况太过于复杂，一般不列入考虑范围。所以，一般比较通用的设计是支持组间广播，不同的组注册不同的订阅。组内的不同机器，如果注册一个相同的ID，则单播；如果注册不同的ID(如IP地址+端口)，则广播。至于广播关系的维护，一般由于消息队列本身都是集群，所以都维护在公共存储上，如config server、zookeeper等。维护广播关系所要做的事情基本是一致的:

1. 发送关系的维护。
2. 发送关系变更时的通知。

### 队列高级特性设计

#### 可靠投递（最终一致性）

这是个激动人心的话题，完全不丢消息，究竟可不可能？答案是，完全可能，前提是消息可能会重复，并且，在异常情况下，要接受消息的延迟。方案说简单也简单，就是每当要发生不可靠的事情（RPC等）之前，先将消息落地，然后发送。当失败或者不知道成功失败（比如超时）时，消息状态是待发送，定时任务不停轮询所有待发送消息，最终一定可以送达。

1. producer往broker发送消息之前，需要做一次落地。
2. 请求到server后，server确保数据落地后再告诉客户端发送成功。
3. 支持广播的消息队列需要对每个待发送的endpoint，持久化一个发送状态，直到所有endpoint状态都OK才可删除消息。

对于各种不确定（超时、down机、消息没有送达、送达后数据没落地、数据落地了回复没收到），其实对于发送方来说，都是一件事情，就是消息没有送达。重推消息所面临的问题就是消息重复。重复和丢失就像两个噩梦，你必须要面对一个。好在消息重复还有处理的机会，消息丢失再想找回就难了。Anyway，作为一个成熟的消息队列，应该尽量在各个环节减少重复投递的可能性，不能因为重复有解决方案就放纵的乱投递。

#### 消费确认

当broker把消息投递给消费者后，消费者可以立即响应我收到了这个消息。但收到了这个消息只是第一步，我能不能处理这个消息却不一定。或许因为消费能力的问题，系统的负荷已经不能处理这个消息；或者是刚才状态机里面提到的消息不是我想要接收的消息，主动要求重发。把消息的送达和消息的处理分开，这样才真正的实现了消息队列的本质-解耦。

允许消费者主动进行消费确认是必要的。当然，对于没有特殊逻辑的消息，默认Auto Ack也是可以的，对于正确消费ack的，没什么特殊的。但是对于reject和error，需要特别说明。reject这件事情，往往业务方是无法感知到的。消费能力不匹配的时候，直接拒绝，过一段时间重发，减少业务的负担。但业务出错这件事情是只有业务方自己知道的，就像上文提到的状态机等等。这时应该允许业务方主动ack error，并可以与broker约定下次投递的时间。

#### 重复消息和顺序消息

上文谈到重复消息是不可能100%避免的，除非可以允许丢失，那么，顺序消息能否100%满足呢? 答案是可以，但条件更为苛刻：

1. 允许消息丢失。
2. 从发送方到服务方到接受者都是单点单线程。

所以绝对的顺序消息基本上是不能实现的，当然在METAQ/Kafka等pull模型的消息队列中，单线程生产/消费，排除消息丢失，也是一种顺序消息的解决方案。一般来讲，一个主流消息队列的设计范式里，应该是不丢消息的前提下，尽量减少重复消息，不保证消息的投递顺序。

谈到重复消息，主要是两个话题：

1. 如何鉴别消息重复，并幂等的处理重复消息。
2. 一个消息队列如何尽量减少重复消息的投递。

先来看看第一个话题，每一个消息应该有它的唯一身份。不管是业务方自定义的，还是根据IP/PID/时间戳生成的MessageId，如果有地方记录这个MessageId，消息到来是能够进行比对就能完成重复的鉴定。

幂等的处理消息是一门艺术，因为种种原因重复消息或者错乱的消息还是来到了。两种通用的解决方案

##### 版本号

举个简单的例子，一个产品的状态有上线/下线状态。如果消息1是下线，消息2是上线。不巧消息1判重失败，被投递了两次，且第二次发生在2之后，如果不做重复性判断，显然最终状态是错误的。  
 但是，如果每个消息自带一个版本号。上游发送的时候，标记消息1版本号是1，消息2版本号是2。如果再发送下线消息，则版本号标记为3。下游对于每次消息的处理，同时维护一个版本号。每次只接受比当前版本号大的消息。初始版本为0，当消息1到达时，将版本号更新为1。消息2到来时，因为版本号>1.可以接收，同时更新版本号为2.当另一条下线消息到来时，如果版本号是3.则是真实的下线消息。如果是1，则是重复投递的消息。

如果业务方只关心消息重复不重复，那么问题就已经解决了。但很多时候另一个头疼的问题来了，就是消息顺序如果和想象的顺序不一致。比如应该的顺序是12，到来的顺序是21。则最后会发生状态错误。参考TCP/IP协议，如果想让乱序的消息最后能够正确的被组织，那么就应该只接收比当前版本号大一的消息。并且在一个session周期内要一直保存各个消息的版本号。（缓冲区）如果到来的顺序是21，则先把2存起来，待1到来后，先处理1，再处理2，这样重复性和顺序性要求就都达到了。

##### 状态机

基于版本号来处理重复和顺序消息听起来是个不错的主意，但凡事总有瑕疵。使用版本号的最大问题是：

1. 对发送方必须要求消息带业务版本号。
2. 下游必须存储消息的版本号，对于要严格保证顺序的。

还不能只存储最新的版本号的消息，要把乱序到来的消息都存储起来。而且必须要对此做出处理。试想一个永不过期的"session"，比如一个物品的状态，会不停流转于上下线。那么中间环节的所有存储就必须保留，直到在某个版本号之前的版本一个不丢的到来，成本太高。（有可能有多个上下游）。

业务方只需要自己维护一个状态机，定义各种状态的流转关系。例如，"下线"状态只允许接收"上线"消息，“上线”状态只能接收“下线消息”，如果上线收到上线消息，或者下线收到下线消息，在消息不丢失和上游业务正确的前提下。要么是消息发重了，要么是顺序到达反了。这时消费者只需要把“我不能处理这个消息”告诉投递者，要求投递者过一段时间重发即可。而且重发一定要有次数限制，比如5次，避免死循环，就解决了。

##### 中间件对于重复消息的处理

回归到消息队列的话题来讲。上述通用的版本号/状态机/ID判重解决方案里，哪些是消息队列该做的、哪些是消息队列不该做业务方处理的呢？其实这里没有一个完全严格的定义，但回到我们的出发点，我们保证不丢失消息的情况下尽量少重复消息，消费顺序不保证。那么重复消息下和乱序消息下业务的正确，应该是由消费方保证的，我们要做的是减少消息发送的重复。

我们无法定义业务方的业务版本号/状态机，如果API里强制需要指定版本号，则显得过于绑架客户了。况且，在消费方维护这么多状态，就涉及到一个消费方的消息落地/多机间的同步消费状态问题，复杂度指数级上升，而且只能解决部分问题。

减少重复消息的关键步骤：

1. broker记录MessageId，直到投递成功后清除，重复的ID到来不做处理，这样只要发送者在清除周期内能够感知到消息投递成功，就基本不会在server端产生重复消息。
2. 对于server投递到consumer的消息，由于不确定对端是在处理过程中还是消息发送丢失的情况下，有必要记录下consumer的IP地址。决定重发之前询问这个IP，消息处理成功了吗？如果询问无果，再重发。

### 性能相关

#### 异步/同步

首先澄清一个概念，异步，同步和oneway是三件事。异步，归根结底你还是需要关心结果的，但可能不是当时的时间点关心，可以用轮询或者回调等方式处理结果；同步是需要当时关心的结果的；而oneway是发出去就不管死活的方式，这种对于某些完全对可靠性没有要求的场景还是适用的。回归来看，任何的RPC都是存在客户端异步与服务端异步的，而且是可以任意组合的。

那么，服务端使用异步最大的好处是什么呢？说到底，是解放了线程和I/O。试想服务端有一堆I/O等待处理，如果每个请求都需要同步响应，每条消息都需要结果立刻返回，那么就几乎没法做I/O合并。（当然接口可以设计成batch的，但可能batch发过来的仍然数量较少）。而如果用异步的方式返回给客户端future，就可以有机会进行I/O的合并，把几个批次发过来的消息一起落地（这种合并对于MySQL等允许batch insert的数据库效果尤其明显），并且彻底释放了线程。不至于说来多少请求开多少线程，能够支持的并发量直线提高。

回到消息队列的议题上，我们当然不希望消息的发送阻塞主流程（前面提到了，server端如果使用异步模型，则可能因消息合并带来一定程度上的消息延迟），所以可以先使用线程池提交一个发送请求，主流程继续往下走。但是线程池中的请求关心结果吗？Of course，必须等待服务端消息成功落地，才算是消息发送成功。所以这里的模型，准确地说事客户端半同步半异步（使用线程池不阻塞主流程，但线程池中的任务需要等待server端的返回），server端是纯异步。客户端的线程池wait在server端吐回的future上，直到server端处理完毕，才解除阻塞继续进行。

#### 批量

谈到批量就不得不提生产者消费者模型。但生产者消费者模型中最大的痛点是：消费者到底应该何时进行消费。大处着眼来看，消费动作都是事件驱动的。主要事件包括：

1. 攒够了一定数量。
2. 到达了一定时间。
3. 队列里有新的数据到来。

对于及时性要求高的数据，可用采用方式3来完成，比如客户端向服务端投递数据。只要队列有数据，就把队列中的所有数据刷出，否则将自己挂起，等待新数据的到来。

相反对于可以用适量的延迟来换取高性能的场景来说，用定时/定量二选一的方式可能会更为理想，既到达一定数量才发送，但如果数量一直达不到，也不能干等，有一个时间上限。具体说来，在上文的submit之前，多判断一个时间和数量，并且Runnable内部维护一个定时器，避免没有新任务到来时旧的任务永远没有机会触发发送条件。对于server端的数据落地，使用这种方式就非常方便。

曾经有人问我，为什么网络请求小包合并成大包会提高性能？主要原因有两个：

1. 减少无谓的请求头，如果你每个请求只有几字节，而头却有几十字节，无疑效率非常低下。
2. 减少回复的ack包个数。把请求合并后，ack包数量必然减少，确认和重发的成本就会降低。

## push还是pull

我们简要分析下push和pull模型各自存在的利弊。

### 慢消费

慢消费无疑是push模型最大的致命伤，穿成流水线来看，如果消费者的速度比发送者的速度慢很多，势必造成消息在broker的堆积。假设这些消息都是有用的无法丢弃的，消息就要一直在broker端保存。当然这还不是最致命的，最致命的是broker给consumer推送一堆consumer无法处理的消息，consumer不是reject就是error，然后来回踢皮球。

反观pull模式，consumer可以按需消费，不用担心自己处理不了的消息来骚扰自己，而broker堆积消息也会相对简单，无需记录每一个要发送消息的状态，只需要维护所有消息的队列和偏移量就可以了（kafka）。所以对于建立索引等慢消费，消息量有限且到来的速度不均匀的情况，pull模式比较合适。

### 消息延迟与忙等

这是pull模式最大的短板。由于主动权在消费方，消费方无法准确地决定何时去拉取最新的消息。如果一次pull取到消息了还可以继续去pull，如果没有pull取到则需要等待一段时间重新pull。但等待多久就很难判定了。你可能会说，我可以有xx动态pull取时间调整算法，但问题的本质在于，有没有消息到来这件事情决定权不在消费方。也许1分钟内连续来了1000条消息，然后半个小时没有新消息产生，可能你的算法算出下次最有可能到来的时间点是31分钟之后，或者60分钟之后，结果下条消息10分钟后到了，是不是很让人沮丧？

当然也不是说延迟就没有解决方案了，业界较成熟的做法是从短时间开始（不会对broker有太大负担），然后指数级增长等待。比如开始等5ms，然后10ms，然后20ms，然后40ms……直到有消息到来，然后再回到5ms。即使这样，依然存在延迟问题：假设40ms到80ms之间的50ms消息到来，消息就延迟了30ms，而且对于半个小时来一次的消息，这些开销就是白白浪费的。在阿里的RocketMq里，有一种优化的做法-长轮询，来平衡推拉模型各自的缺点。基本思路是:消费者如果尝试拉取失败，不是直接return,而是把连接挂在那里wait,服务端如果有新的消息到来，把连接notify起来，这也是不错的思路。但海量的长连接block对系统的开销还是不容小觑的，还是要合理的评估时间间隔，给wait加一个时间上限比较好。

### 顺序消息

如果push模式的消息队列，支持分区，单分区只支持一个消费者消费，并且消费者只有确认一个消息消费后才能push送另外一个消息，还要发送者保证全局顺序唯一，听起来也能做顺序消息，但成本太高了，尤其是必须每个消息消费确认后才能发下一条消息，这对于本身堆积能力和慢消费就是瓶颈的push模式的消息队列，简直是一场灾难。

反观pull模式，如果想做到全局顺序消息，就相对容易很多：

1. producer对应partition，并且单线程。
2. consumer对应partition，消费确认（或批量确认），继续消费即可。