**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”。

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