# 1分布式消息中间件

## 1.1 MQ的主要作用

主要解决对多个系统进行解耦、异步执行提高业务本身性能、最终一致性，异步RPC、错峰流控。

不关注业务处理返回结果

## 1.2 设计MQ考虑点

1、RPC通信协议

2、消息存储

文件系统>分布式KV（持久化）>分布式文件系统>数据库

3、消费关系

ZK等配置服务

4、防重复

5、防丢失

6、事务

7、批量发送/异步发送

9、性能、安全性、容灾、集群、负载均衡

9、根据业务特性场景定制化

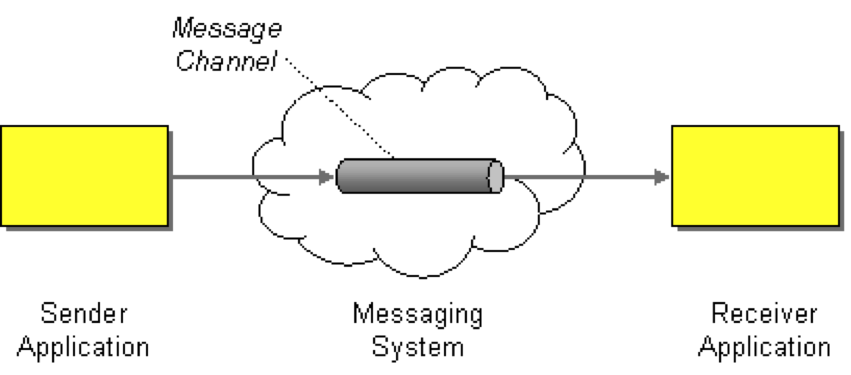
## 1.3 消息模型

拉模型：消费者主动获取消息，按需消费。缺点就是消费者无法感知到消息不知道该何时进行消息拉取。如果第一次拉取没有消息就进行等待，下次再拉取，但是这个等待时间多久无法准确评估。若不等待势必会有很多请求到broker上。

推模型：生产者主动推送消息，缺点消费能力跟不上生产能力则造成消息在Broker堆积，生产者推送一批消息而消费者无法消费拒绝或者报错，势必造成生产者重复发送，来回踢皮球。

### 1.3.1 P2P模型

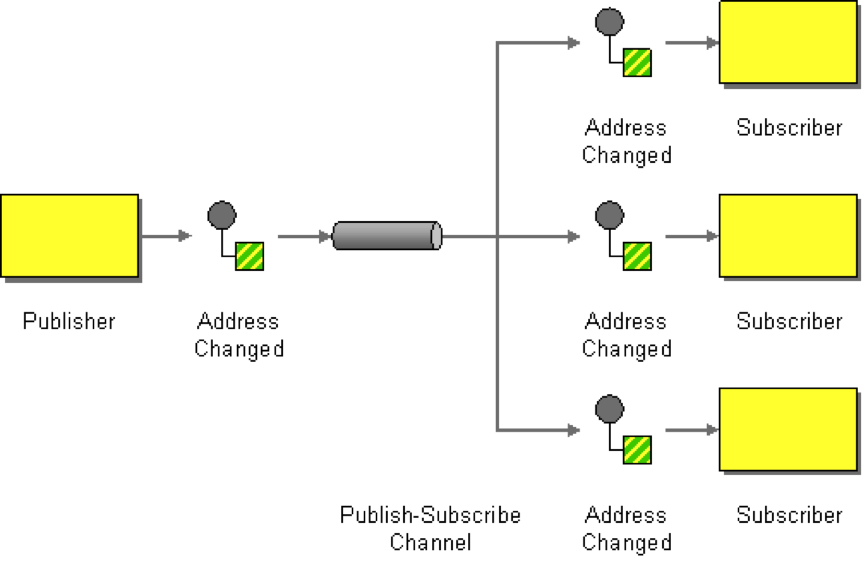
点对点：队列实现该模型，消息只被一个消费者消费, 抢占式机制.



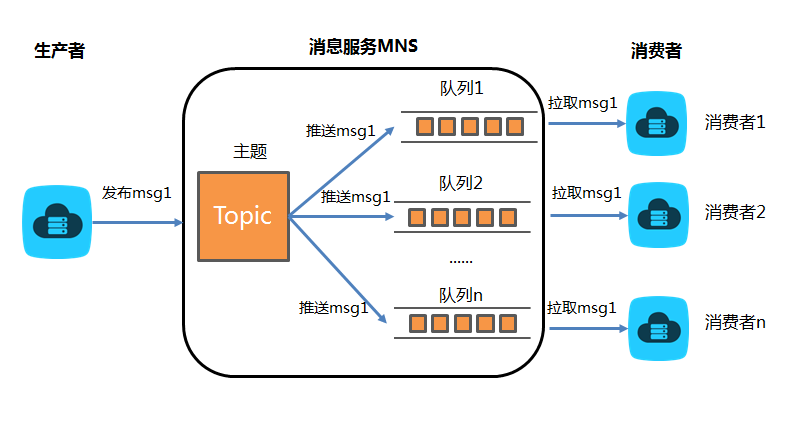
### 1.3.2 发布订阅

发布订阅：多个消费者可以同时消费该消息, 广播消息机制。

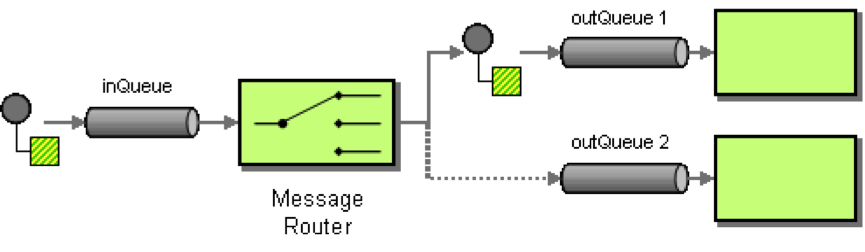
消息发送者将消息发送到broker，再由broker去发现消费者，然后主动调用消费者。两次RPC过程。消费者若需要确认消息则是三次RPC.



在该模式下消息broker需要知道消费者服务地址信息，若需要隐藏暴露服务地址信息，则可以采用如下模式



### 1.3.3 消息路由



## 1.4 消息如何保证可靠

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

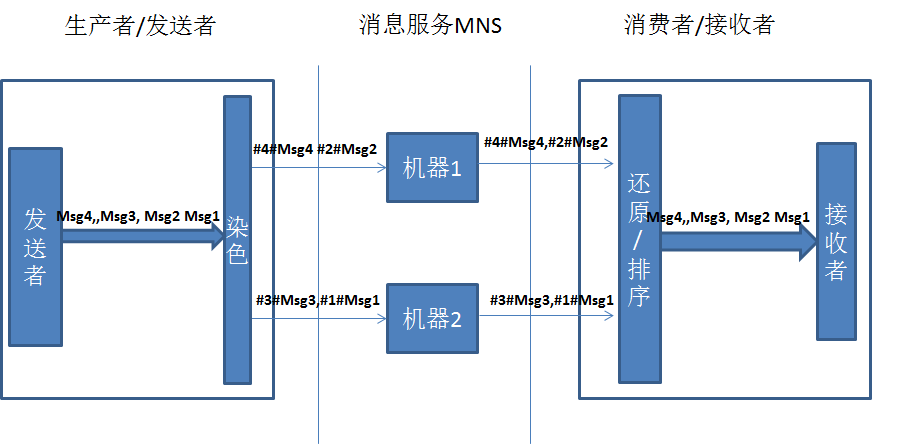
1. producer往broker发送消息之前，需要做一次落地。
2. 请求到server后，server确保数据落地后再告诉客户端发送成功。
3. 支持广播的消息队列需要对每个待发送的endpoint，持久化一个发送状态，直到所有endpoint状态都OK才可删除消息。

消息确认：消息送达和消息处理需要分开，允许消费端主动响应反馈机制。对于拒绝业务往往无感知

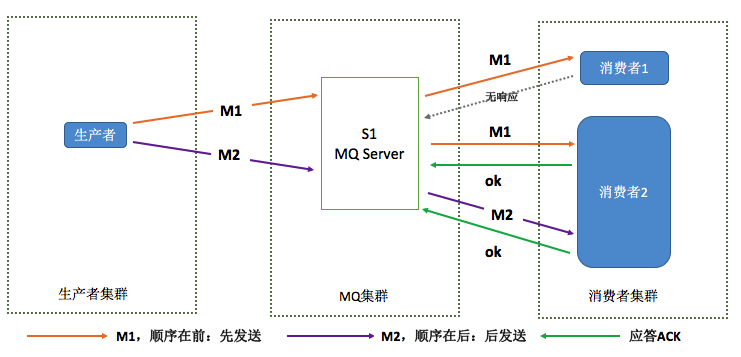
## 1.5 严格消息顺序如何保证

http://www.infoq.com/cn/articles/high-availability-broker-design

阿里消息服务解决方案

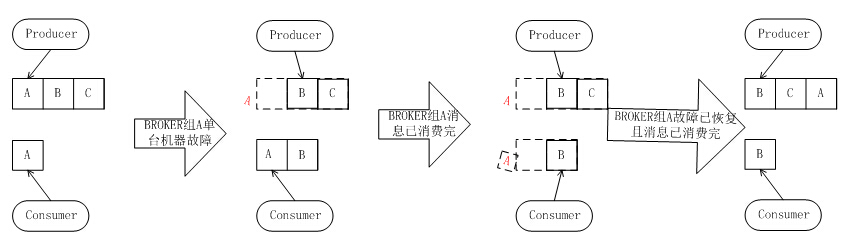


其他方案



JD方案

<http://www.infoq.com/cn/articles/high-availability-broker-design>



保证生产者 - broker - 消费者是一对一对一的关系

在要求严格顺序消息的场景下，消息的发送者，BROKER端（BROKER端和消息存储放在一起），消息的消费者都要求按照顺序进行，三者任何一个环节的乱序都会导致消息最终的消费顺序被打乱。

消息有序会带来消息重复问题，主要由于M1无响应或者消费失败导致消息需要重复发送

## 消息重复

主要从以下几方面考虑：如何鉴别消息重复，并幂等的处理重复消息。一个消息队列如何尽量减少重复消息的投递。

幂等解决方案：

1、版本号

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。

缺陷：1）消息带版本号 2）下游必须存储版本号

2、状态机

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。

举例子说明，假设产品本身状态是下线，1是上线消息，2是下线消息，3是上线消息，正常情况下，消息应该的到来顺序是123，但实际情况下收到的消息状态变成了3123。那么下游收到3消息的时候，判断状态机流转是下线->上线，可以接收消息。然后收到消息1，发现是上线->上线，拒绝接收，要求重发。然后收到消息2，状态是上线->下线，于是接收这个消息。此时无论重发的消息1或者3到来，还是可以接收。另外的重发，在一定次数拒绝后停止重发，业务正确。

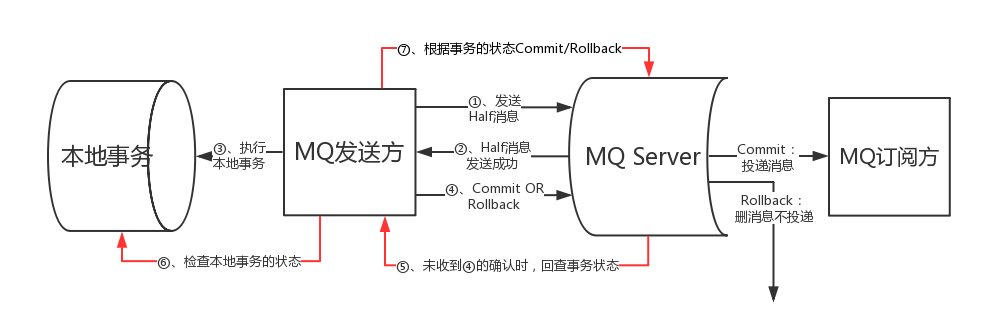
3、每个消息都生成一个消息ID（时间戳），并记录这个消息ID，消息来到进行消息重复鉴定。

减少重复解决方案：

## 1.7 消息事务

1. 两阶段提交，分布式事务。
2. 本地事务，本地落地，补偿发送。

### 1.7.1 2pc



其中：

1. 发送方向 MQ 服务端发送消息；
2. MQ Server 将消息持久化成功之后，向发送方 ACK 确认消息已经发送成功，此时消息为半消息。
3. 发送方开始执行本地事务逻辑。
4. 发送方根据本地事务执行结果向 MQ Server 提交二次确认（Commit 或是 Rollback），MQ Server 收到 Commit 状态则将半消息标记为可投递，订阅方最终将收到该消息；MQ Server 收到 Rollback 状态则删除半消息，订阅方将不会接受该消息。
5. 在断网或者是应用重启的特殊情况下，上述步骤4提交的二次确认最终未到达 MQ Server，经过固定时间后 MQ Server 将对该消息发起消息回查。
6. 发送方收到消息回查后，需要检查对应消息的本地事务执行的最终结果。
7. 发送方根据检查得到的本地事务的最终状态再次提交二次确认，MQ Server 仍按照步骤4对半消息进行操作。

## 1.8 异步发送消息

## 1.9 消息回放

## 1.10 消息如何过滤

## 1.11 消息重试

# 2 分布式数据库中间件

## 2.1 分库分表原因

## 2.2 常用分库分表方案

## 2.3 中间件解决方案

### 2.3.1 数据切分策略

### 2.3.2 动态扩容

### 2.3.3 SQL语法解析

### 2.3.4 分布式事务

#### 2.3.4.1 Best effort 1PC

一次性最大努力提交，98%的情况都是正常情况，避免分布式事务，性能达到最佳，但是一旦出现网络中断、数据库宕机，那么数据会造成不一致，对数据一致性要求比较高那么该方案不适用。

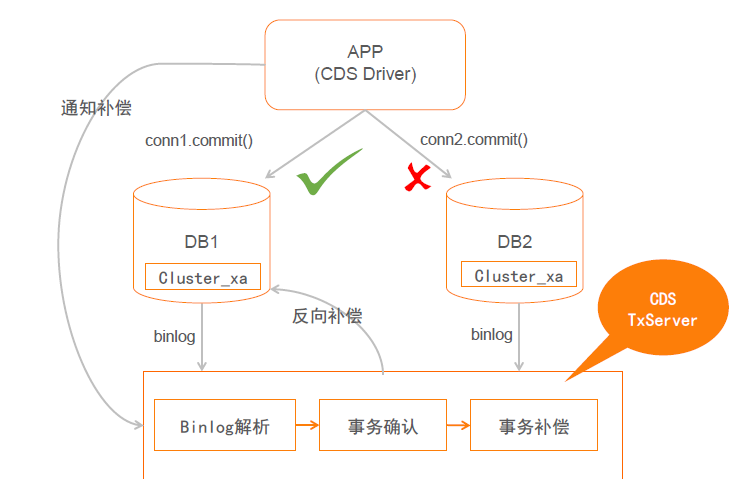
#### 2.3.4.2 2PC

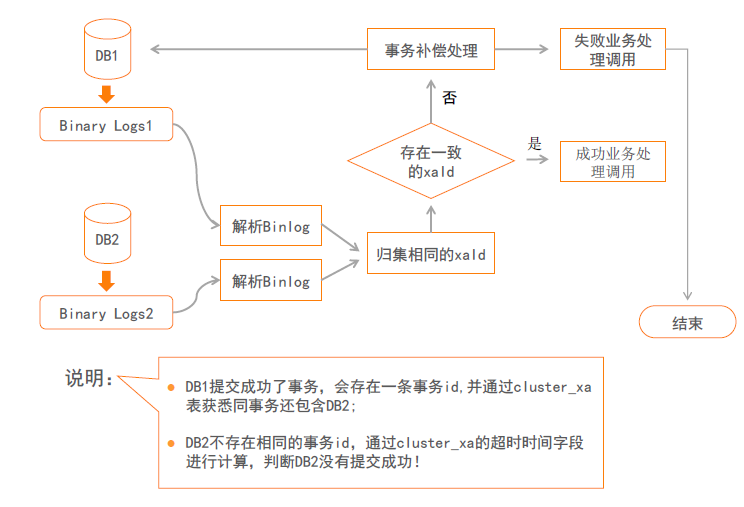
两阶段提交



#### 2.3.4.3 3PC

#### 2.3.4.4 补偿事务





#### 2.3.4.5 Sagas长事务

#### 2.3.4.6 TCC

#### 2.3.4.7可靠事件模式

#### 2.3.4.8 一致性决策

### 2.3.5 跨库跨表JOIN

### 2.3.6 数据运维成本

# 3 分布式缓存

# 4 服务框架

# 5 服务治理监控