MQ核心基础篇

2021-03-02 | 技术干货

这是《吃透XXX》技术系列的开篇，这个系列的思路是：先找到每个技术栈最本质的东西，然后以此为出发点，逐渐延伸出其他核心知识。所以，整个系列侧重于思考力的训练，不仅仅是讲清楚 What，而是更关注 Why 和 How，以帮助大家构建出牢固的知识体系。

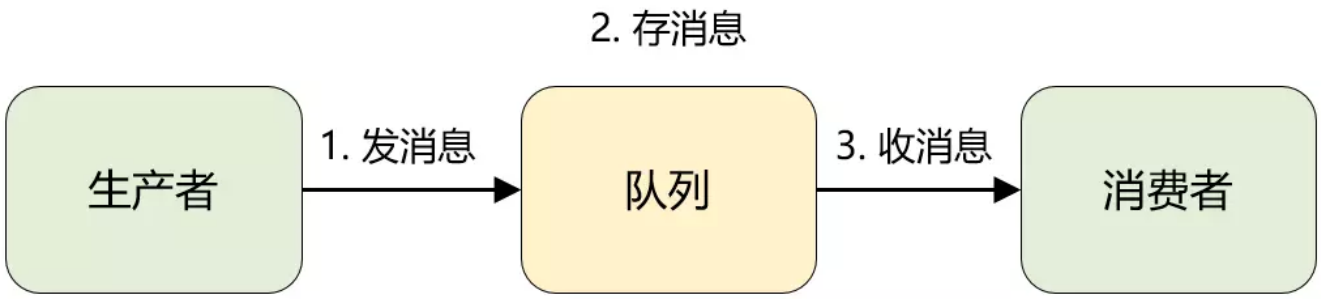
回到正文，这是《吃透 MQ》的第一篇文章。本文主要讲解 MQ 的通用知识，让大家先弄明白：**如果让你来设计一个 MQ，该如何下手？需要考虑哪些问题？又有哪些技术挑战？**

有了这个基础后，我相信后面几篇文章再讲 Kafka 和 RocketMQ 这两种具体的消息中间件时，大家能很快地抓住主脉络，同时分辨出它们各自的特点。

对于 MQ 来说，不管是 RocketMQ、Kafka 还是其他消息队列，**它们的本质都是：一发一存一消费。**下面我们以这个本质作为根，一起由浅入深地聊聊 MQ。

01 从 MQ 的本质说起

将 MQ 掰开了揉碎了来看，都是「一发一存一消费」，再直白点就是一个「转发器」。生产者先将消息投递一个叫做「队列」的容器中，然后再从这个容器中取出消息，最后再转发给消费者，仅此而已。



上面这个图便是消息队列最原始的模型，它包含了两个关键词：消息和队列。

> 1、消息：就是要传输的数据，可以是最简单的文本字符串，也可以是自定义的复杂格式（只要能按预定格式解析出来即可）。
>
> 2、队列：大家应该再熟悉不过了，是一种先进先出数据结构。它是存放消息的容器，消息从队尾入队，从队头出队，入队即发消息的过程，出队即收消息的过程。

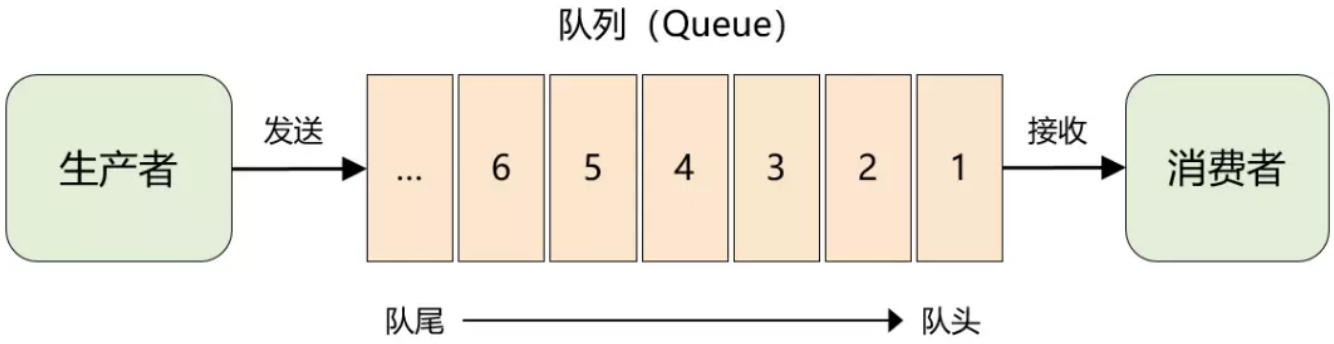
02 原始模型的进化

再看今天我们最常用的消息队列产品（RocketMQ、Kafka 等等），你会发现：它们都在最原始的消息模型上做了扩展，同时提出了一些新名词，比如：主题（topic）、分区（partition）、队列（queue）等等。

要彻底理解这些五花八门的新概念，我们化繁为简，先从消息模型的演进说起（**道理好比：架构从来不是设计出来的，而是演进而来的**）

2.1 队列模型

最初的消息队列就是上一节讲的原始模型，它是一个严格意义上的队列（Queue）。消息按照什么顺序写进去，就按照什么顺序读出来。不过，队列没有 “读” 这个操作，读就是出队，从队头中 “删除” 这个消息。



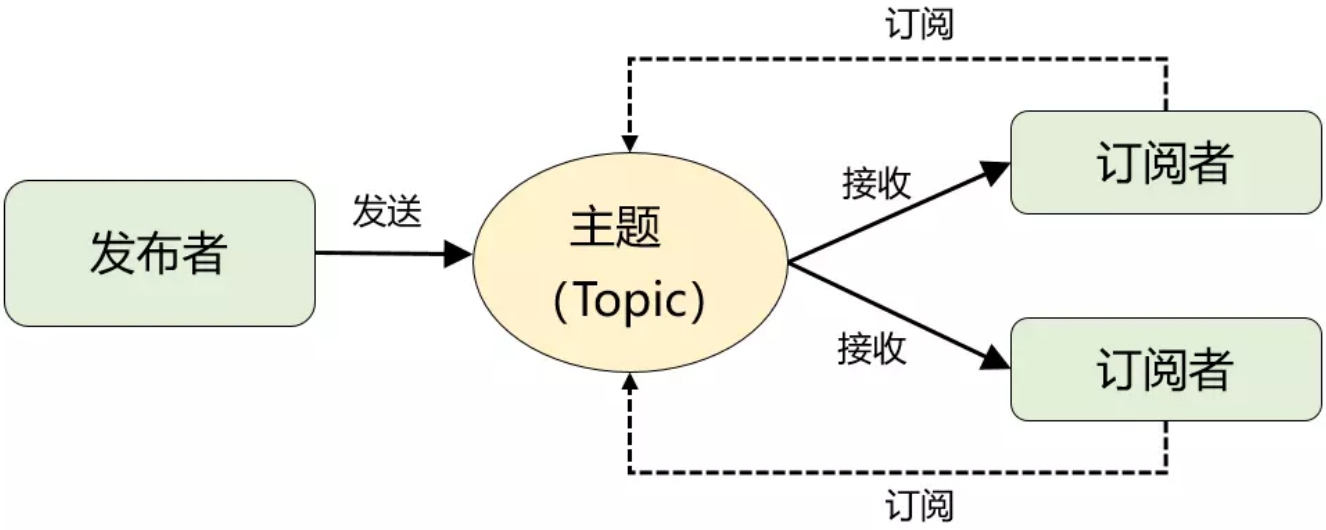
这便是队列模型：它允许多个生产者往同一个队列发送消息。但是，如果有多个消费者，实际上是竞争的关系，也就是一条消息只能被其中一个消费者接收到，读完即被删除。

2.2 发布-订阅模型

如果需要将一份消息数据分发给多个消费者，并且每个消费者都要求收到全量的消息。很显然，”' 列模型无法满足这个需求。

一个可行的方案是：为每个消费者创建一个单独的队列，让生产者发送多份。这种做法比较笨，而且同一份数据会被复制多份，也很浪费空间。

为了解决这个问题，就演化出了另外一种消息模型：发布-订阅模型。



在发布-订阅模型中，存放消息的容器变成了 “主题”，订阅者在接收消息之前需要先 “订阅主题”。最终，每个订阅者都可以收到同一个主题的全量消息。

仔细对比下它和 “队列模式” 的异同：生产者就是发布者，队列就是主题，消费者就是订阅者，无本质区别。唯一的不同点在于：一份消息数据是否可以被多次消费。

2.3 小结

最后做个小结，上面两种模型说白了就是：单播和广播的区别。而且，当发布-订阅模型中只有 1 个订阅者时，它和队列模型就一样了，因此在功能上是完全兼容队列模型的。

这也解释了为什么现代主流的 RocketMQ、Kafka 都是直接基于发布-订阅模型实现的？此外，RabbitMQ 中之所以有一个 Exchange 模块？其实也是为了解决消息的投递问题，可以变相实现发布-订阅模型。

包括大家接触到的 “消费组”、“集群消费”、“广播消费” 这些概念，都和上面这两种模型相关，以及在应用层面大家最常见的情形：组间广播、组内单播，也属于此范畴。

所以，先掌握一些共性的理论，对于大家再去学习各个消息中间件的具体实现原理时，其实能更好地抓住本质，分清概念。

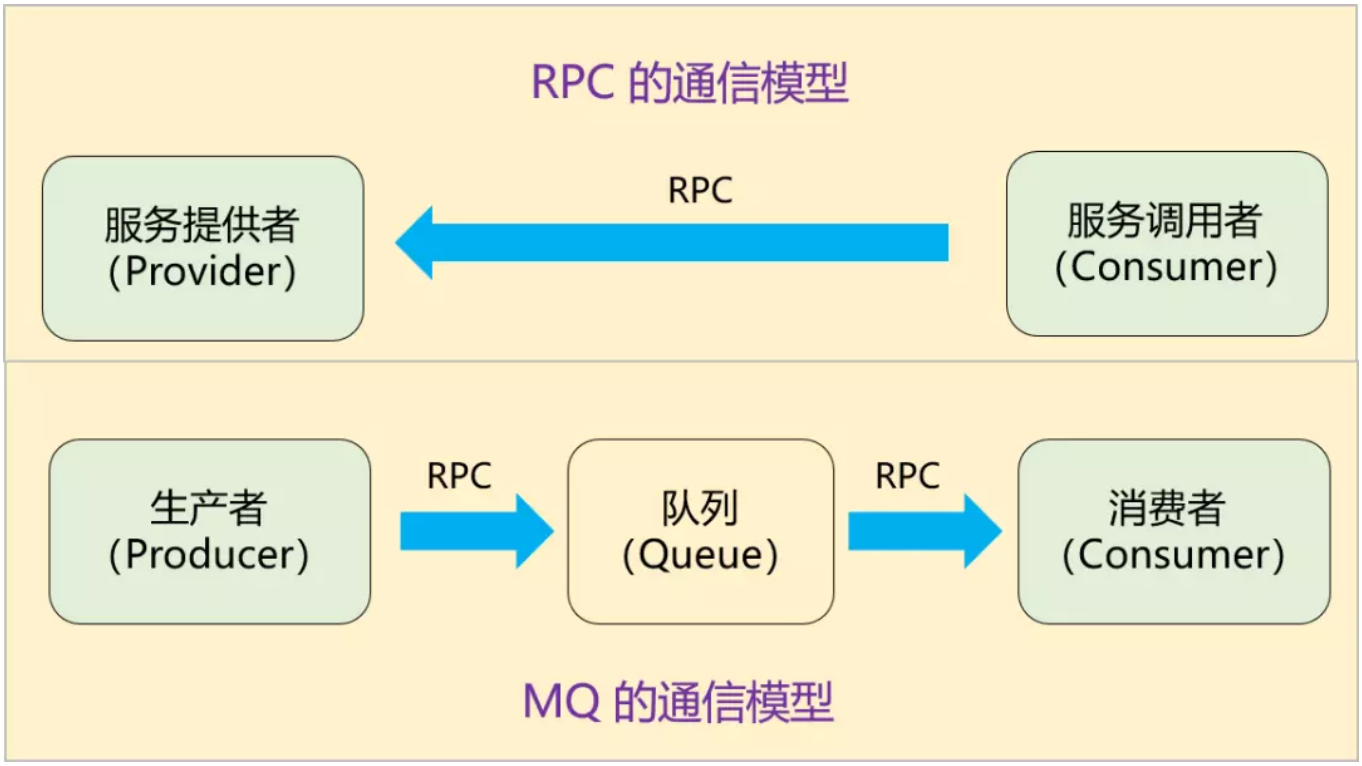
03 透过模型看 MQ 的应用场景

目前，MQ 的应用场景非常多，大家能倒背如流的是：系统解耦、异步通信和流量削峰。除此之外，还有延迟通知、最终一致性保证、顺序消息、流式处理等等。

那到底是先有消息模型，还是先有应用场景呢？答案肯定是：先有应用场景（也就是先有问题），再有消息模型，因为消息模型只是解决方案的抽象而已。

MQ 经过 30 多年的发展，能从最原始的队列模型发展到今天百花齐放的各种消息中间件（平台级的解决方案），我觉得万变不离其宗，还是得益于：消息模型的适配性很广。

我们试着重新理解下消息队列的模型。它其实解决的是：生产者和消费者的通信问题。那它对比 RPC 有什么联系和区别呢？

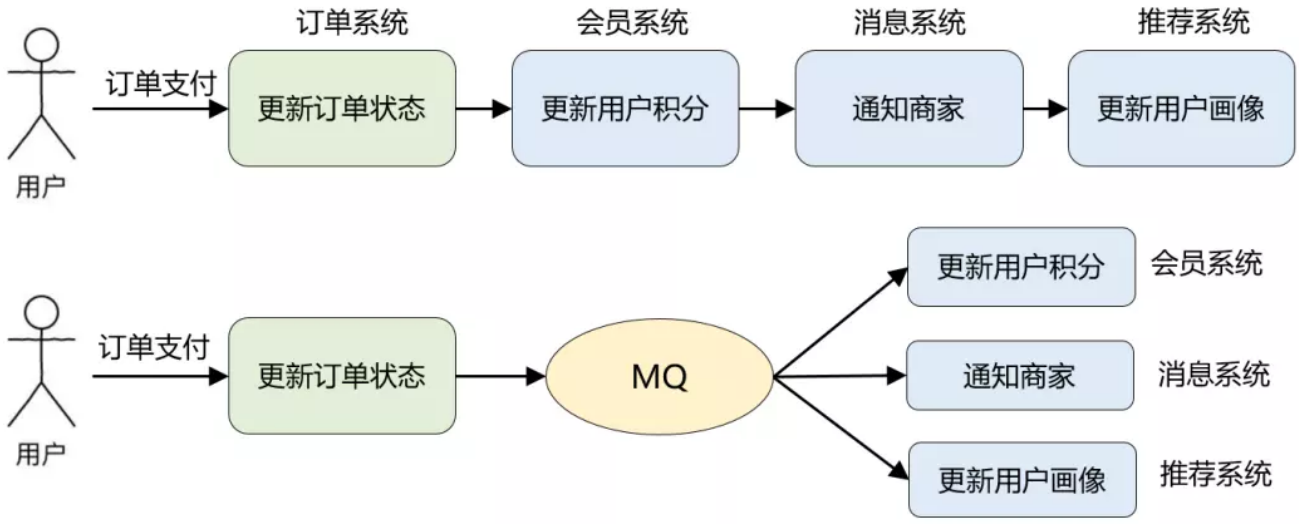


通过对比，能很明显地看出两点差异：

> 1、引入 MQ 后，由之前的一次 RPC 变成了现在的两次 RPC，而且生产者只跟队列耦合，它根本无需知道消费者的存在。
>
> 2、多了一个中间节点「队列」进行消息转储，相当于将同步变成了异步。

再返过来思考 MQ 的所有应用场景，就不难理解 MQ 为什么适用了？因为这些应用场景无外乎都利用了上面两个特性。

举一个实际例子，比如说电商业务中最常见的「订单支付」场景：在订单支付成功后，需要更新单状态、更新用户积分、通知商家有新订单、更新推荐系统中的用户画像等等。



引入 MQ 后，订单支付现在只需要关注它最重要的流程：更新订单状态即可。其他不重要的事情全部交给 MQ 来通知。这便是 MQ 解决的最核心的问题：系统解耦。

改造前订单系统依赖 3 个外部系统，改造后仅仅依赖 MQ，而且后续业务再扩展（比如：营销系统打算针对支付用户奖励优惠券），也不涉及订单系统的修改，从而保证了核心流程的稳定性，降低了维护成本。

这个改造还带来了另外一个好处：因为 MQ 的引入，更新用户积分、通知商家、更新用户画像这些步骤全部变成了异步执行，能减少订单支付的整体耗时，提升订单系统的吞吐量。这便是 MQ 的另一个典型应用场景：异步通信。

除此以外，由于队列能转储消息，对于超出系统承载能力的场景，可以用 MQ 作为 “漏斗” 进行限流保护，即所谓的流量削峰。我们还可以利用队列本身的顺序性，来满足消息必须按顺序投递的场景；利用队列 + 定时任务来实现消息的延时消费 ……

MQ 其他的应用场景基本类似，都能回归到消息模型的特性上，找到它适用的原因，这里就不一一分析了。总之，就是建议大家多从复杂多变的实践场景再回归到理论层面进行思考和抽象，这样能吃得更透。

04 如何设计一个 MQ?

了解了上面这些理论知识以及应用场景后，下面我们再一起看下：到底如何设计一个 MQ?

4.1 MQ 的雏形

我们还是先从简单版的 MQ 入手，如果只是实现一个很粗糙的 MQ，完全不考虑生产环境的要 ，该如何设计呢?

文章开头说过，任何 MQ 无外乎：一发一存一消费，这是 MQ 最核心的功能需求。另外，从技术维度来看 MQ 的通信模型，可以理解成：两次 RPC + 消息转储。

有了这些理解，我相信只要有一定的编程基础，不用 1 个小时就能写出一个 MQ 雏形：

> 1、直接利用成熟的 RPC 框架（Dubbo 或者 Thrift），实现两个接口：发消息和读消息。
>
> 2、消息放在本地内存中即可，数据结构可以用 JDK 自带的 ArrayBlockingQueue 。

4.2 写一个适用于生产环境的 MQ

当然，我们的目标绝不止于一个 MQ 雏形，而是希望实现一个可用于生产环境的消息中间件，那难度肯定就不是一个量级了，具体我们该如何下手呢?

1、先把握这个问题的关键点

假如我们还是只考虑最基础的功能：发消息、存消息、消费消息（支持发布-订阅模式）。

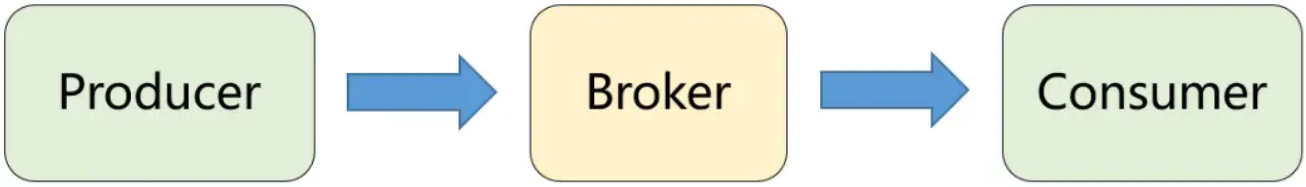
那在生产环境中，这些基础功能将面临哪些挑战呢？我们能很快想到下面这些：

> 1、高并发场景下，如何保证收发消息的性能?
>
> 2、如何保证消息服务的高可用和高可靠?
>
> 3、如何保证服务是可以水平任意扩展的?
>
> 4、如何保证消息存储也是水平可扩展的?
>
> 5、各种元数据（比如集群中的各个节点、主题、消费关系等）如何管理，需不需要考虑数据的一致性?

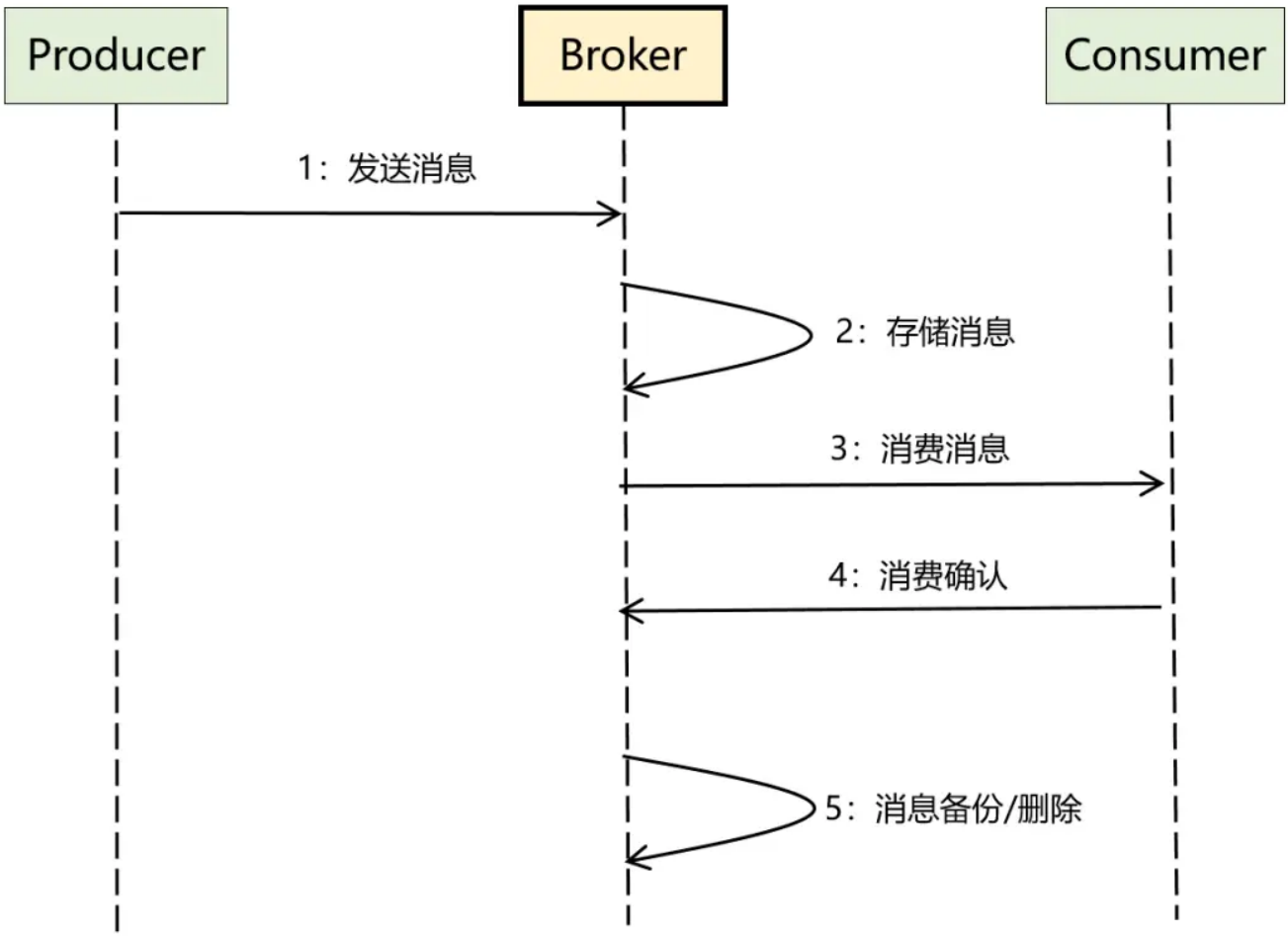
可见，高并发场景下的三高问题在你设计一个 MQ 时都会遇到，「如何满足高性能、高可靠等非功能性需求」才是这个问题的关键所在。

2、整体设计思路

先来看下整体架构，会涉及三类角色：



另外，将「一发一存一消费」这个核心流程进一步细化后，比较完整的数据流如下：



基于上面两个图，我们可以很快明确出 3 类角色的作用，分别如下：

> 1、Broker（服务端）：MQ 中最核心的部分，是 MQ 的服务端，核心逻辑几乎全在这里，它为生产者和消费者提供 RPC 接口，负责消息的存储、备份和删除，以及消费关系的维护等。
>
> 2、Producer（生产者）：MQ 的客户端之一，调用 Broker 提供的 RPC 接口发送消息。
>
> 3、Consumer（消费者）：MQ 的另外一个客户端，调用 Broker 提供的 RPC 接口接收消息，同时完成消费确认。

3、详细设计

下面，再展开讨论下一些具体的技术难点和可行的解决方案。

难点1：RPC 通信

解决的是 Broker 与 Producer 以及 Consumer 之间的通信问题。如果不重复造轮子，直接利用成熟的 RPC 框架 Dubbo 或者 Thrift 实现即可，这样不需要考虑服务注册与发现、负载均衡、通信协议、序列化方式等一系列问题了。

当然，你也可以基于 Netty 来做底层通信，用 Zookeeper、Euraka 等来做注册中心，然后自定义一套新的通信协议（类似 Kafka），也可以基于 AMQP 这种标准化的 MQ 协议来做实现（类似 RabbitMQ）。对比直接用 RPC 框架，这种方案的定制化能力和优化空间更大。

难点2：高可用设计

高可用主要涉及两方面：Broker 服务的高可用、存储方案的高可用。可以拆开讨论。

Broker 服务的高可用，只需要保证 Broker 可水平扩展进行集群部署即可，进一步通过服务自动注册与发现、负载均衡、超时重试机制、发送和消费消息时的 ack 机制来保证。

存储方案的高可用有两个思路：1）参考 Kafka 的分区 + 多副本模式，但是需要考虑分布式场景下数据复制和一致性方案（类似 Zab、Raft等协议），并实现自动故障转移；2）还可以用主流的 DB、分布式文件系统、带持久化能力的 KV 系统，它们都有自己的高可用方案。

难点3：存储设计

消息的存储方案是 MQ 的核心部分，可靠性保证已经在高可用设计中谈过了，可靠性要求不高的话直接用内存或者分布式缓存也可以。这里重点说一下存储的高性能如何保证？这个问题的决定因素在于存储结构的设计。

目前主流的方案是：追加写日志文件（数据部分） + 索引文件的方式（很多主流的开源 MQ 都是这种方式），索引设计上可以考虑稠密索引或者稀疏索引，查找消息可以利用跳转表、二份查找等，还可以通过操作系统的页缓存、零拷贝等技术来提升磁盘文件的读写性能。

如果不追求很高的性能，也可以考虑现成的分布式文件系统、KV 存储或者数据库方案。

难点4：消费关系管理

为了支持发布-订阅的广播模式，Broker 需要知道每个主题都有哪些 Consumer 订阅了，基于这个关系进行消息投递。由于 Broker 是集群部署的，所以消费关系通常维护在公共存储上，可以基于 Zookeeper、Apollo 等配置中心来管理以及进行变更通知。

难点5：高性能设计

存储的高性能前面已经谈过了，当然还可以从其他方面进一步优化性能。比如 Reactor 网络 IO 模型、业务线程池的设计、生产端的批量发送、Broker 端的异步刷盘、消费端的批量拉取等等。

4.3 小结

再总结下，要回答好：如何设计一个 MQ?

1、需要从功能性需求（收发消息）和非功能性需求（高性能、高可用、高扩展等）两方面入手。

2、功能性需求不是重点，能覆盖 MQ 最基础的功能即可，至于延时消息、事务消息、重试队列等高级特性只是锦上添花的东西。

3、最核心的是：能结合功能性需求，理清楚整体的数据流，然后顺着这个思路去考虑非功能性的诉求如何满足，这才是技术难点所在。

05 写在最后

这篇文章从 MQ 一发一存一消费这个本质出发，讲解了消息模型的演进过程，这是 MQ 最核心的理论基础。基于此，大家也能更容易理解 MQ 的各种新名词以及应用场景。

最后通过回答：如何设计一个 MQ？目的是让大家对 MQ 的核心组件和技术难点有一个清晰的认识。另外，带着这个问题的答案再去学习 Kafka、RocketMQ 等具体的消息中间件时，也会更有侧重点。

希望大家有所收获，如果有任何意见和建议，欢迎评论区留言反馈！《吃透 MQ 系列》的下一篇是 Kafka，我们下期见！

作者简介：985硕士，前亚马逊工程师，现58转转技术总监

欢迎扫描下方的二维码，关注我的个人公众号：武哥漫谈IT，分享硬核技术和职场成长

消息队列 吃透系列

一脚迈进大厂，聊几点大家关心的 《吃透XXX》技术系列开篇