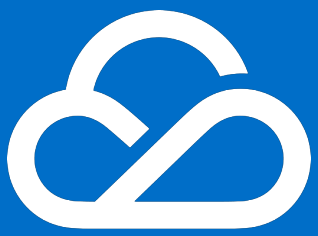
腾讯云消息队列 CMQ

最佳实践

产品文档



腾讯云

【版权声明】

【商标声明】

【服务声明】

本文档意在向客户介绍腾讯云全部或部分产品、服务的当时的整体概况，部分产品、服务的内容可能有所调整。您所购买的腾讯云产品、服务的种类、服务标准等应由您与腾讯云之间的商业合同约定，除非双方另有约定，否则，腾讯云对本文档内容不做任何明示或模式的承诺或保证。

文档目录

最佳实践

选择push 还是 pull

腾讯云的 CMQ 支持 Pull、Push 两种方式，而这两种方式的适用场景是什么呢？我们简要分析下Push 和 Pull模型，在不同场景下各自存在的利弊。

Push和Pull的区别

所谓 Push 模型，即当 Producer 发出的消息到达后，服务端马上将这条消息投递给Consumer；而 Pull 则是服务端收到这条消息后什么也不做，只是等着 Consumer 主动到自己这里来读，即 Consumer 这里有一个“拉取”的动作。

场景1：Producer 的速率大于 Consumer 的速率

对于 Producer 速率大于 Consumer 速率的情况，有两种可能性需要讨论，第一种是Producer 本身的效率就要比 Consumer 高（比如说，Consumer 端处理消息的业务逻辑可能很复杂，或者涉及到磁盘、网络等I/O操作）；另一种则是Consumer出现故障，导致短时间内无法消费或消费不畅。

Push 方式由于无法得知当前 Consumer 的状态，所以只要有数据产生，便会不断地进行推送，在以上两种情况下时，可能会导致 Consumer 的负载进一步加重，甚至是崩溃（比如生产者是 flume 疯狂抓日志，消费者是 HDFS+hadoop，处理效率跟不上）。除非Consumer 有合适的反馈机制能够让服务端知道自己的状况。

而采取 Pull 的方式问题就简单了许多，由于 Consumer 是主动到服务端拉取数据，此时只需要降低自己访问频率就好了。举例：如前端是 flume 等日志收集业务，不断往 CMQ 生产消息，CMQ 往后端投递，后端业务如数据分析等业务，效率可能低于生产者。

场景2：强调消息的实时性

采用 Push 的方式时，一旦消息到达，服务端即可马上将其推送给服务端，这种方式的实时性显然是非常好的；而采用 Pull 方式时，为了不给服务端造成压力（尤其是当数据量不足时，不停的轮询显得毫无意义），需要控制好自己轮询的间隔时间，但这必然会给实时性带来一定的影响。

场景3：Pull 的长轮询

Pull 模式有什么问题呢？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 取到消息则需要等待一段时间再重新 pull。

但等待时间就很难判定了。你可能会说，我可以有xx动态拉取时间调整算法，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 CMQ 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。

在腾讯云的 CMQ 里，有一种优化的做法-长轮询，来平衡 pull/push
模型各自的缺点。基本方式是：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，把连接拉起，返回最新消息。

场景4：部分或全部Consumer不在线

在消息系统中，Producer 和 Consumer 是完全解耦的，Producer 发送消息时，并不要求Consumer
一定要在线，对于 Consumer 也是同样的道理，这也是消息通信区别于 RPC 通信的主要特点；但是对于
Consumer不在线的情况，却有很多值得讨论的场景。

首先，在 Consumer 偶然宕机或下线的情况下，Producer 的生产是可以不受影响的，Consumer
上线后，可以继续之前的消费，此时消息数据不会丢失；但是如果 Consumer
长期宕机或是由于机器故障无法再次启动，就会出现问题，即服务端需不需要为 Consumer
保留数据，以及保留多久的数据等等。

采用 Push 方式时，因为无法预知 Consumer 的宕机或下线是短暂的还是持久的，如果一直为该 Consumer 保留自宕机开始的所有历史消息，那么即便其他所有的 Consumer 都已经消费完成，数据也无法清理掉，随着时间的积累，队列的长度会越来越大，此时无论消息是暂存于内存还是持久化到磁盘上（采用 Push 模型的系统，一般都是将消息队列维护于内存中，以保证推送的性能和实时性，这一点会在后边详细讨论），都将对

CMQ 服务端造成巨大压力，甚至可能影响到其他 Consumer
的正常消费，尤其当消息的生产速率非常快时更是如此；但是如果不保留数据，那么等该 Consumer
再次起来时，则要面对丢失数据的问题。

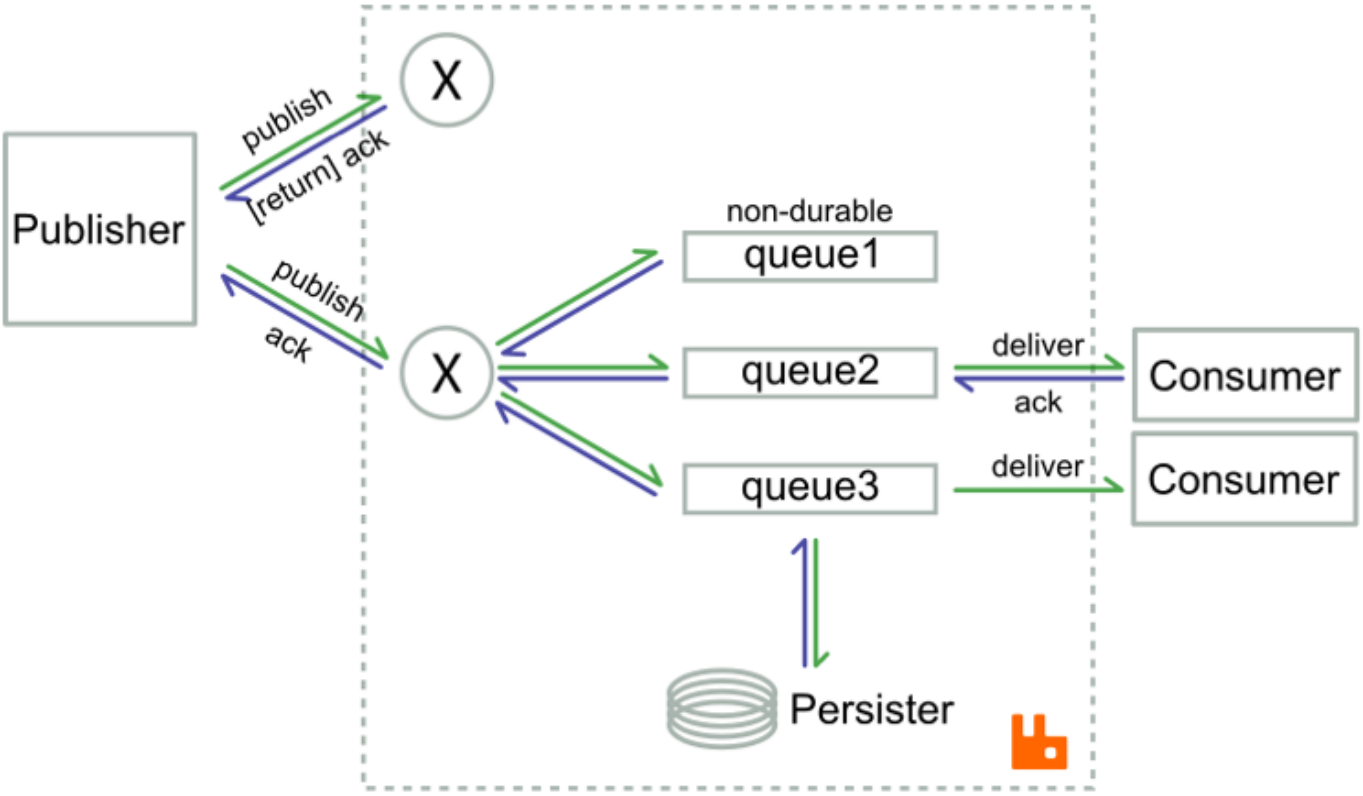
折中的方案是：CMQ 给数据设定一个超时时间，当 Consumer
宕机时间超过这个阈值时，则清理数据；但这个时间阈值也并太容易确定。

在采用 Pull 模型时，情况会有所改善；服务端不再关心 Consumer
的状态，而是采取“你来了我才服务”的方式，Consumer
是否能够及时消费数据，服务端不会做任何保证（也有超时清理时间）。

消息去重

对于重复消息，最好的方法是消息可重入（消息重复消费对业务无影响）。做不到可重入时，需要在消费端去重。

一、重复消息出现的原因



网络异常、服务器宕机等原因都有可能导致消息丢失。CMQ
为了做到不丢消息、可靠交付，采用了消息生产、消费确认机制。

生产消息确认：生产者向 CMQ 发送消息后，等待 CMQ 回复确认；CMQ
将消息持久化到磁盘后，向生产者返回确认成功。否则在生产者请求超时、CMQ
返回失败等情况下，生产者需要向 CMQ 重发消息。

消费者确认：CMQ 向消费者交付消息后，将消息置为不可见；在消息不可见时间内，消费者使用句柄删除消息。如果消息未被删除，且不可见时间超时，消息将重新可见。

由于消息确认机制是“至少一次交付（at least
once）”，在网络抖动、生产者/消费者异常等情况下，就会出现生产者重复生产、消费者重复消费的情况。

二、去重方案

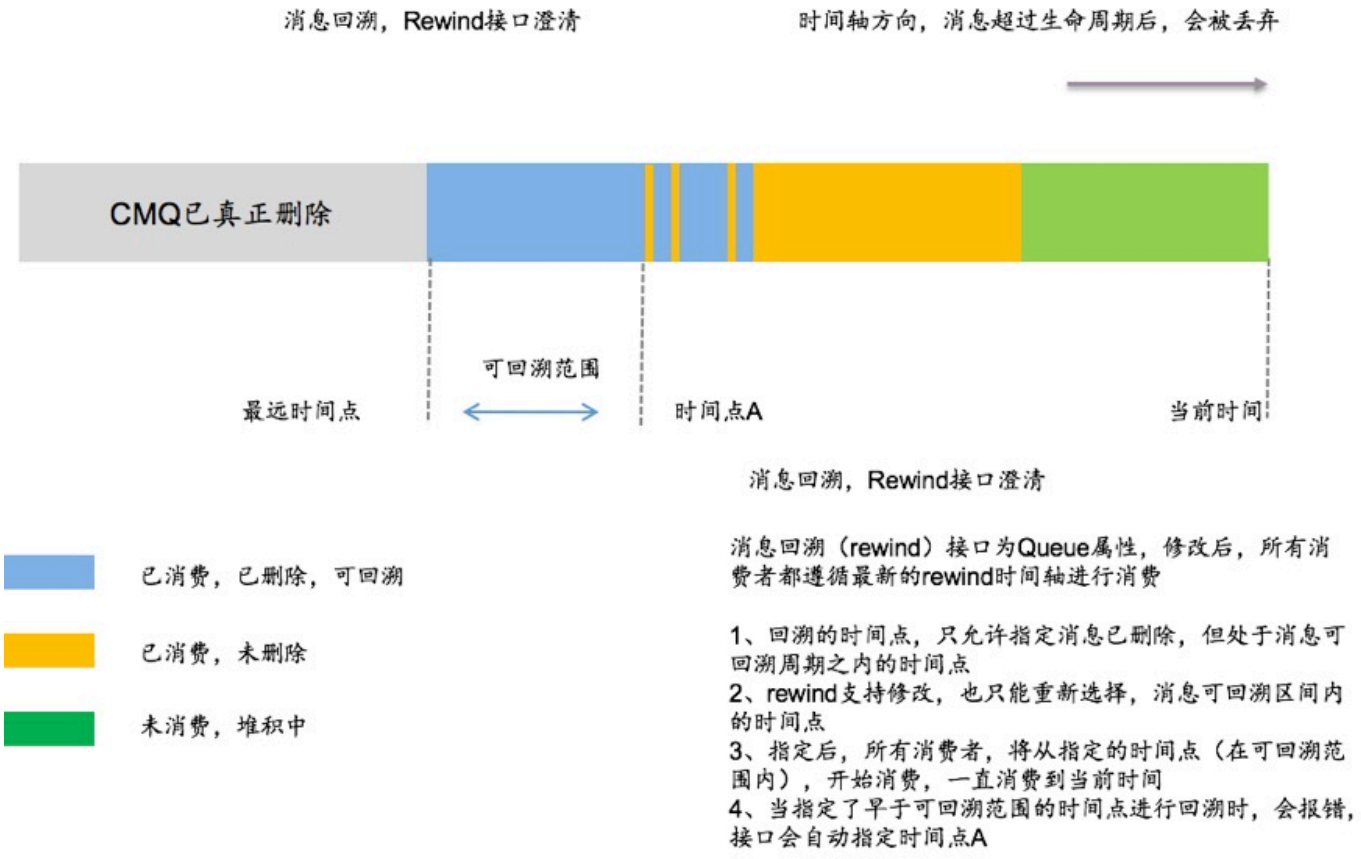
要去重，先要识别重复消息。通常的做法是在生产消息时，业务方在消息体中插入去重key，消费时通过该去重key 来识别重复消息。去重key 可以是由 <生产者 IP + 线程 ID + 时间戳 + 时间内递增值> 组成的唯一值。

只有一个消费者时，您可以将消费过的 去重key 缓存（如 KV 等），然后每次消费时检查 去重key 是否已消费过。去重key 缓存可以根据消息最大有效时间来淘汰。CMQ 提供了队列当前最小未消费消息的时间（min_msg_time），您可以使用该时间和业务生产消息最大重试时间来确定缓存淘汰时间。
存在多个消费者时，去重key 缓存就需要是分布式的。

• 根据消息最大有效时间，计算 key 过期时间：
current_time - max_retention_time - max_retry_time - max_network_time

• 根据 CMQ 最小未消费时间，计算 key 过期时间：
min_msg_time - max_retry_time - max_network_time

说明：



CMQ 可配置消息最大有效时间为15天，业务可根据实际情况调整。
CMQ队列当前最小未消费消息时间，即上图中最远时间点。该时间之前的消息都已经被删除，之后的消息可能未被删除。

三、举例说明

避免重复提交：

场景：A 为生产者，B 为消费者，中间是 CMQ。A 已完成10元转账操作，且将消息发送给 CMQ，CMQ 也已成功收到。此时网络闪断或者客户端 A 宕机导致服务端应答给客户端 A 失败。A 会认为发送失败，从而再次生产消息。这会造成重复提交。

解决方法：A 在生产消息时，加入 time 时间戳等信息，生成唯一的 去重key。若生产者 A 由于网络问题判断当前发送失败，重试时，去重key 沿用第一次发送的 去重key。此时消费者 B可通过去重key 判断并做去重。
（该案例也说明了不能使用 CMQ 的 message id 来去重，因为这两条消息有不同的 ID，但却有相同的 body。）

这里要注意的是，生产者A，在发送消息之前，要将去重key做持久化（写磁盘等，避免掉电后丢失）

避免多条相同 body 的消息被过滤：

场景：A 给 B 转账10元，一共发起5次，每一次提交的 body 内容是一样的。如果消费者粗暴用 body 做去重判断，就会把5次请求，当做1次请求来处理。

解决方法：A 在生产消息时，加入 time 时间戳等信息。此时哪怕消息 body 一样，生成的 去重key 都是不同的，这样就满足了多次发送同样内容的需求。