# 1. MQ的作用及问题

MQ的作用主要有三个解耦，异步，削峰

## 1.1 MQ的作用

### 1.1.1 解耦

如果多个模块或者系统中，掺杂在一起很复杂，维护起来比较麻烦，就可以运用MQ到这个业务中拆解模块。

### 1.1.2 异步

这个很好理解，比如用户的操作日志的维护，可以不用同步处理，节约响应时间。

### 1.1.3 削峰

在高峰期的时候，系统每秒的请求量达到 5000，那么调用 MySQL 的请求也是 5000，一般情况下 MySQL 的请求大概在 2000 左右，那么在高峰期的时候，数据库就被打垮了，那系统就不可用了。此时引入MQ，在系统 A 前面加个 MQ，用户请求先到 MQ，系统 A 从 MQ 中每秒消费 2000 条数据，这样就把本来 5000 的请求变为 MySQL 可以接受的请求数量了，可以保证系统不挂掉，可以继续提供服务。MQ 里的数据可以慢慢的把它消费掉。

## 1.2 使用了MQ会有什么问题

- 增加了系统的复杂度：因为一个系统引入了MQ之后会造成系统的复杂性的提升，复杂性提升后，增加MQ的维护成本
- 降低的系统的可用性：复杂性的提升意味这系统可用性的降低，因为MQ一旦出现问题就会造成系统出现问题。
- 一致性问题：因为MQ是异步处理消息，需要处理类似于消息丢失以及重复消费的问题，一旦处理不好就会造成重复消费问题。

## 1.3 如何避免MQ消息堆积

### 1.3.1 产生MQ消息堆积的原因

1. 生产者投递消息的速率与我们消费者消费的速率完全不匹配。
2. 生产者投递消息的速率>消费者消费的速率 导致我们消息会堆积在我们 mq 服务器端中，没有及时的被消费者消费 所以就会产生消息堆积的问题
3. 注意的是：rabbitmq 消费者我们的消息消费如果成功的话 消息会被立即删除。 kafka 或者 rocketmq 消息消费如果成功的话，消息是不会立即被删除。

### 1.3.2 解决办法

1. 提高消费者消费的速率；（对我们的消费者实现集群）
2. 如果已经堆积，先消费暂存，再缓慢处理业务；

## 1.4 为什么会出现重复消费?

MQ的消息流程主要有两个阶段来完成，发送消息到消息队列以及消息队列将消息投递到消费者，因为各种网络原因会造成发送方与以及接收方消息重试，就会造成重复消费

### 1.4.1 发送方消息重试

因为网络原因以及MQ自身原因会导致发送的消息未成功投递到MQ，如果出现未成功投递就会重复投递消息，这个时候是正常的，但是可能因为网络原因导致消息确认消息延时，实际上已经投递到了MQ但是确认消息没有被发送方接收到，就会重新发起消息重试，这样就会造成消息重复，这种是发送方消息重试，但是发送方重试次数是有限制的，如果达到一定次数后还是失败如果没有做其他处理就会造成消息丢失。

### 1.4.2 消费方消息重试

当消息正常投递到MQ后，就需要消费消息了，正常情况下消息会被发送到消费方进行消费，消费方一般需要开启手动确认来确定消息一定会被消费掉，但是可能网络原因导致消息未被消费，这个时候消费方就会重试消费消息，如果是出现消费超时或者异常就会进行消息重复消费，如果异常没有处理好导致消息重复消费可能造成业务出现幂等性问题，但是如果做了幂等性问题，多次消费还是失败就会造成消息丢失。

## 1.5 如果出现重复消费如何解决

**唯一主键**

手动给每一条消息增加一个唯一的消息ID，使用消息ID作为唯一主键，如果消息重复，消息插入不进去，用这种方式来解决消息重复

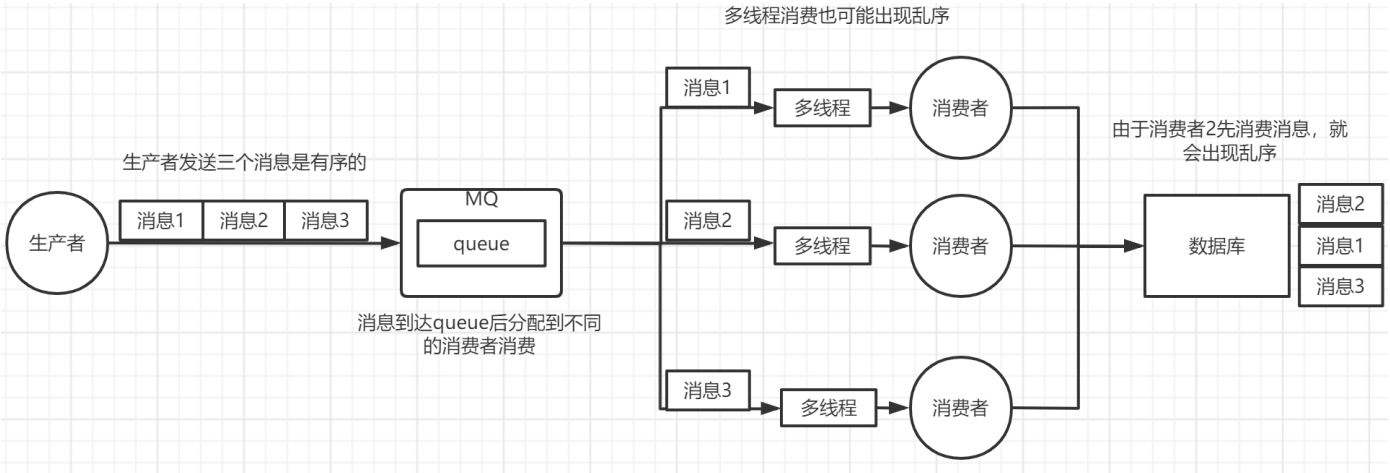
**使用幂等**

使用幂等方式来解决重复消费问题，手动给每一条消息增加一个唯一的消息ID，让消费业务逻辑幂等。

## 1.6 如何保证消息顺序性

### 1.6.1 问题解析

一个queue，有多个consumer去消费，这样就会造成顺序的错误，consumer从MQ里面读取数据是有序的，但是每个consumer的执行时间是不固定的，无法保证先读到消息的consumer一定先完成操作，这样就会出现消息并没有按照顺序执行，造成数据顺序错误，并且执行的时候是多线程执行的，并不能保证执行的顺序性。



#### 1.6.2 解决办法

单线程消费

打包发送

## 1.7 如何实现延时消息

延时消息就是一个消息需要延时多长时间才能够发送，比如用户支付订单，如果多长时间未支付就会取消订单，这个就属于延时消息

**延时消息实现方式**

RabbitMQ

可以采用RabbitMQ的死信队列来实现延时队列，RabbitMQ可以针对Queue和Message设置 x-message-tt，来控制消息的生存时间，如果超时，则消息变为dead letter，dead letter会被投递到死信交换器，然后通过死信队列将消息发送出去


**RocketMQ**

RocketMQ可以实现18个等级的消息延时，但是不可以实现任意时间的消息延时，使用RocketMQ的延时消息只需要按照正常消息发送，并指定延时等级即可，简单高效，并且这个延时时间可以在RocketMQ的配置参数中进行配置。

# 2. 怎样选型MQ

## 2.1 需求分析

### 2.1.1 功能需求

需不需要延迟队列?

#### 2.1.2 性能需求

吞吐量如何。

#### 2.1.3 可用性需求

允不允许消息丢失。

#### 2.1.4 易用性需求

包括学习成本、初期的开发部署成本、日常的运维成本等。

### 2.2 横向对比

| 特性 | ActiveMQ | RabbitMQ | Kafka | RocketMQ |
| --- | --- | --- | --- | --- |
| PRODUCER-COMSUMER | 支持 | 支持 | 支持 | 支持 |
| PUBLISH-SUBSCRIBE | 支持 | 支持 | 支持 | 支持 |
| REQUEST-REPLY | 支持 | 支持 | - | 支持 |
| API完备性 | 高 | 高 | 高 | 低（静态配置） |
| 多语言支持 | 支持，JAVA优先 | 语言无关 | 支持，JAVA优先 | 支持 |
| 单机吞吐量 | 万级 | 万级 | 十万级 | 单机万级 |
| 消息延迟 | 毫秒级 | 微秒级 | 毫秒级 | 毫秒级 |
| 可用性 | 高（主从） | 高（主从） | 很高（分布式） | 非常高（分布式） |
| 消息丢失 | - | 低 | 理论上不会丢失 | 理论上不会丢失 |
| 消息重复 | - | 可控制 | 理论上会有重复 | 允许重复 |
| 文档的完备性 | 高 | 高 | 高 | 中 |
| 提供快速入门 | 有 | 有 | 有 | 无 |
| 首次部署难度 | - | 低 | 中 | 高 |

### 2.3 选型参考

| MQ | 描述 |
| --- | --- |
| RabbitMQ | 简单、上手容易、系统规模不大、低延迟、无堆积的情况。 |
| RocketMQ | 互联网面向业务开发、队列较多、并发较高场景。 |
| Kafka | 日志、允许丢失、重视吞吐量、业务队列没那么多的场景。 |
| ActiveMQ | java开发，简单，稳定，性能不如前面三个。不推荐 |

## 2.4 业务为什么使用rocketmq 不用kafka

因为kafka的诞生是作为大数据的一个中间件来使用的，MQ只是kafka的一个功能

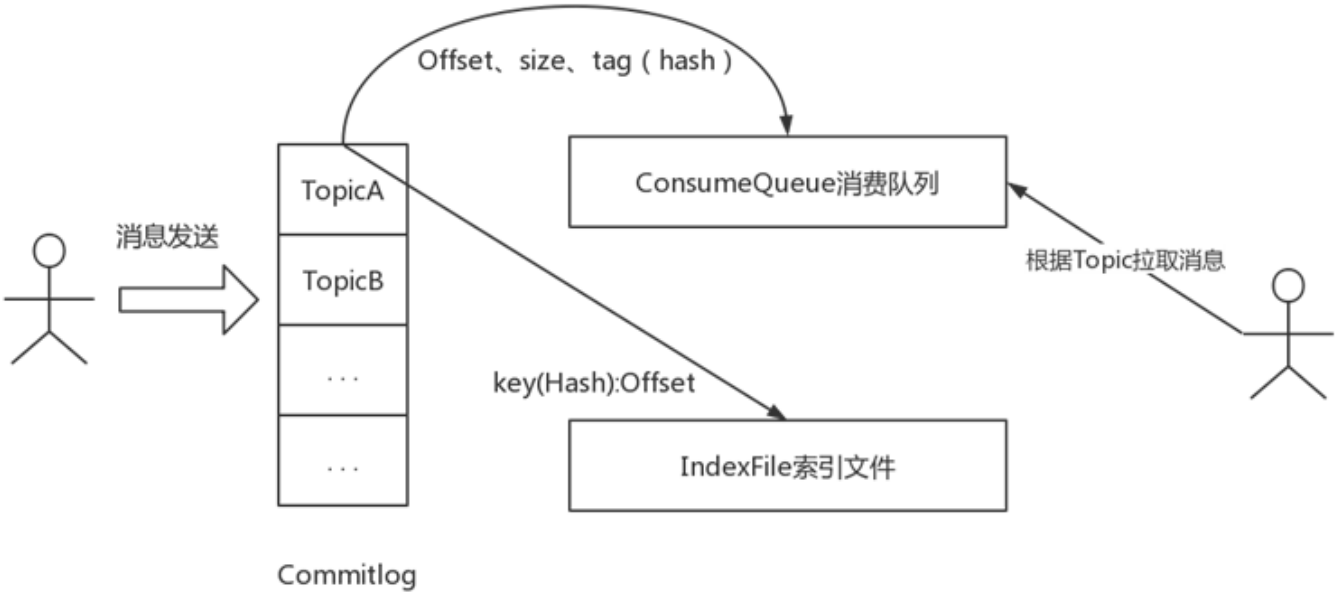
RocketMQ有很多kafka不具备的功能，比如：严格顺序消息，延时消息，服务端tag过滤，能够至少保证消费一个的可靠性策略，消息失败后重试时间随着次数递增等都是很多业务所需要的，并且RocketMQ是经过了双十一的验证，所以很多人说**RocketMQ是专门为业务**而生的。

## 2.5 为什么kafka不能支持大量的topic

这句话如果严格意义上说的话应该是kafka在topic-partition过多而不是单纯的topic过多

这个要从kafka的存储结构来说起，kafka的最小存储单元是一个partition，一个partition由一个数据文件以及一个索引文件组成，partition的存储文件是通过顺序写的方式来保证写入磁盘的效率的，但是如果非常多的partition需要磁盘写入，那么就可能造成顺序写变成随机写，因为磁盘的IO流量就那麽大，同时由很多个partition需要写入，就会变成随机写，性能反而会急剧下降，并且partition过多还是导致元数据管理，副本同步的困难，这些都是导致kafka不能支持大量topic的原因。

## 2.6 为什么RcoketMQ可以支持大量的Topic




这个需要从RocketMQ的存储结构来说，RocketMQ的所有的数据存储在一个commitlog

### 2.6.1 下面是RocketMQ和kafka的topic的测试报告

| 产品 | Topic数量 | 发送端并发数 | 发送端RT（ms） | 发送端TPS | 消费端TPS | |
|---|---|---|---|---|---|---|
| RocketMQ | 64 | 800 | 8 | 9w | 8.6w | |
| | 128 | 800 | 9 | 7.8w | 7.7w | |
| | 256 | 800 | 10 | 7.5w | 7.5w | |
| Kafka | 64 | 800 | 5 | 13.6w | 13.6w | |
| | 128 | 256 | 23 | 8500 | 8500 | |
| | 256 | 256 | 133 | 2215 | 2352 | |