RocketMQ深入分析

1 消息存储

目前的MQ中间件从存储模型来，分为需要持久化和不需要持久化的两种模型，现在大多数的是支持持久化存储的，比如ActiveMQ、RabbitMQ、Kafka、RocketMQ。ZeroMQ却不需要支持持久化存储而业务系统也大多需要MQ有持久存储的能力，这样可以大大增加系统的高可用性。

从存储方式和效率来看，文件系统高于KV存储，KV存储又高于关系型数据库，直接操作文件系统肯定是最快的，但如果从可靠性的角度出发直接操作文件系统是最低的，而关系型数据库的可靠性是最高的。

1.1 存储介质类型和对比

> 常用的存储类型分为关系型数据库存储、分布式KV存储 和 文件系统存储
>
> 目前的高性能磁盘，顺序写速度可以达到 600MB/s，足以满足一般网卡的传输速度，而磁盘随机读写的速度只有约 100KB/s，与顺序写的性能相差了 6000 倍。故好的消息队列系统都会采用顺序写的方式

	关系型数据库存储	分布式KV存储	文件系统存储
简介	选用 JDBC 方式实现消息持久化，只需要简单地配置 xml 即可实现 JDBC 消息存储	kv存储即 Key-Value 型存储中间件，如 Redis 和 RocksDB，将消息存储到这些中间件中	将消息存储到文件系统中
性能	存在性能瓶颈，如mysql在单表数据量达到千万级别的情况下，IO读写性能下降	通过高并发的中间件存储和处理消息，速度必然优于数据库存储方式	将消息刷盘至所部属虚拟化/物理机的文件系统来实现消息持久化，效率更高
可靠性	该方案十分依赖DB，一旦DB出现故障，MQ消息无法落盘存储，从而导致线上故障	相较DB来说更加安全可靠	除非部署 MQ 的机器本身或是本地磁盘挂了，否则一般不会出现无法持久化的问题
项目使用	ActiveMQ	Redis、RockDB	RocketMQ、Kafaka、RabbitMQ

存储效率：文件系统 > 分布式KV存储 > 关系型数据库DB

开发难度和集成：关系型数据库DB < 分布式KV存储 < 文件系统

1.2 顺序读写和随机读写

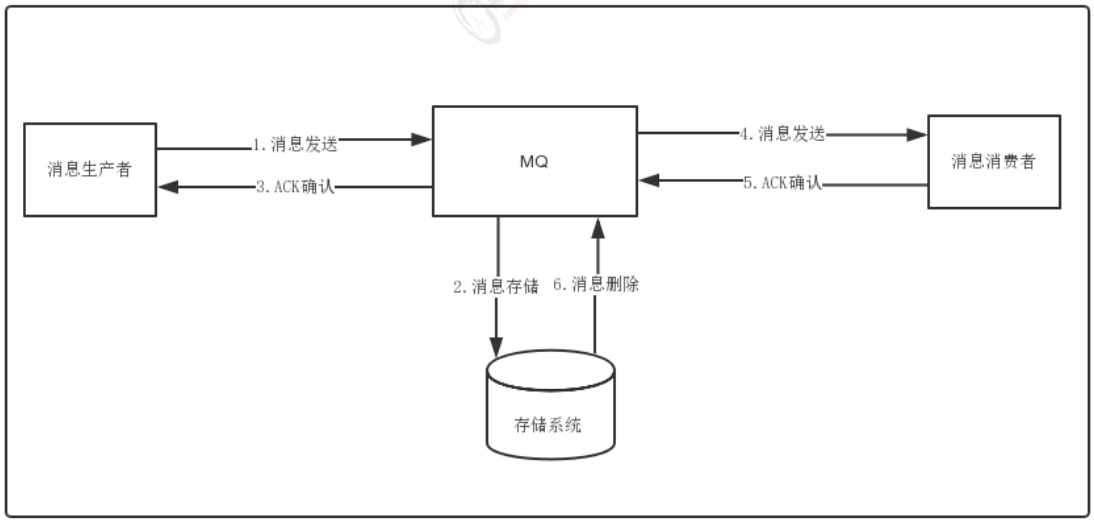
> 顺序读写和随机读写对于机械硬盘来说为什么性能差异巨大

	顺序读写	随机读写
文件数目	读取一个大文件	读取多个小文件
比较	顺序读写只读取一个大文件，耗时更少	随机读写需要打开多个文件，写进行多次的训导和旋转延迟，标绿远低于顺序读写
文件预读	顺序读写时磁盘会预读文件，即在读取的起始地址连续读取多个页面，若被预读的页面被使用，则无需再去读取	由于数据不在一起，无法预读
比较	在大并发的情况下，磁盘预读能够免去大量的读操作，处理速度肯定更快	磁盘需要不断的寻址，效率很低
写入数据	写入新文件时，需要寻找磁盘可用空间	写入新文件时，需要寻找磁盘可用空间。但由于一个文件的存储量更小，这个操作触发频率更多
比较	顺序读写创建新文件，只需要创建一个大文件就可以用很久	随机读写可能频繁创建文件。创建文件时需要进行寻找磁盘可用空间等一些列操作，肯定更加耗时

1.3 消息存储机制

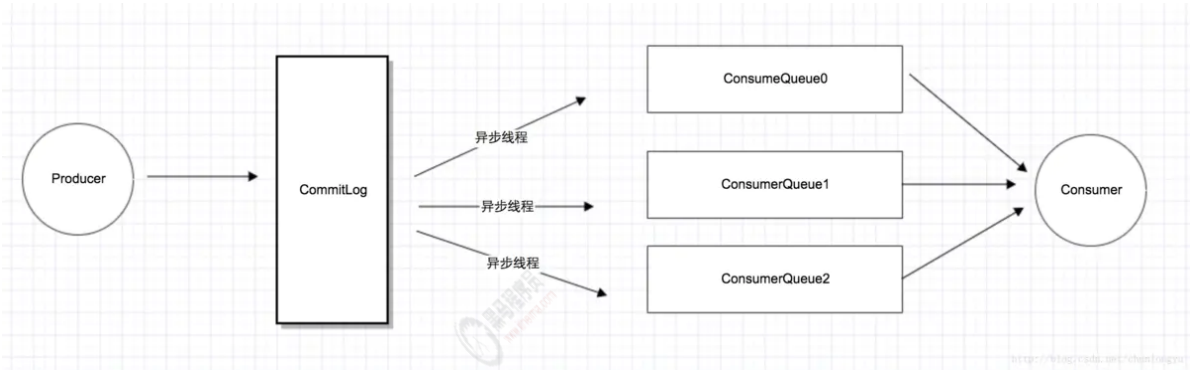
> 由于消息队列有高可靠性的要求，故要对队列中的数据进行持久化存储。

1. 消息生产者先向 MQ 发送消息
2. MQ 收到消息，将消息进行持久化，并在存储系统中新增一条记录
3. 返回ACK(确认字符)给生产者
4. MQ 推送消息给对应的消费者，等待消费者返回ACK(确认字符，确认消费)
5. 若这条消息的消费者在等待时间内成功返回ACK，则 MQ 认为消息消费成功，删除存储中的消息
6. 若 MQ 在指定时间内没有收到ACK，则认为消息消费失败，会尝试重新推送消息



1.4 消息存储设计

RocketMQ采用了单一的日志文件，即把同1台机器上面所有topic的所有queue的消息，存放在一个文件里面，从而避免了随机的磁盘写入。



所有消息都存在一个单一的CommitLog文件里面，然后有后台线程异步的同步到ConsumeQueue，再由Consumer进行消费。

这里之所以可以用“异步线程”，也是因为消息队列天生就是用来“缓冲消息”的。只要消息到了CommitLog，发送的消息也就不会丢。只要消息不丢，那就有了“充足的回旋余地”，用一个后台线程慢慢同步到ConsumeQueue，再由Consumer消费。

1.5 消息存储结构

消息主体以及元数据的存储主体，存储Producer端写入的消息主体内容,消息内容不是定长的。RocketMQ 采取一些机制，尽量向 CommitLog 中顺序写，但是随机读。单个文件大小默认1G ，可通过在 broker 置文件中设置 mapedFileSizeCommitLog 属性来改变默认大小。

1.5.1 CommitLog

> CommitLog是存储消息内容的存储主体，Producer发送的消息都会顺序写入CommitLog文件

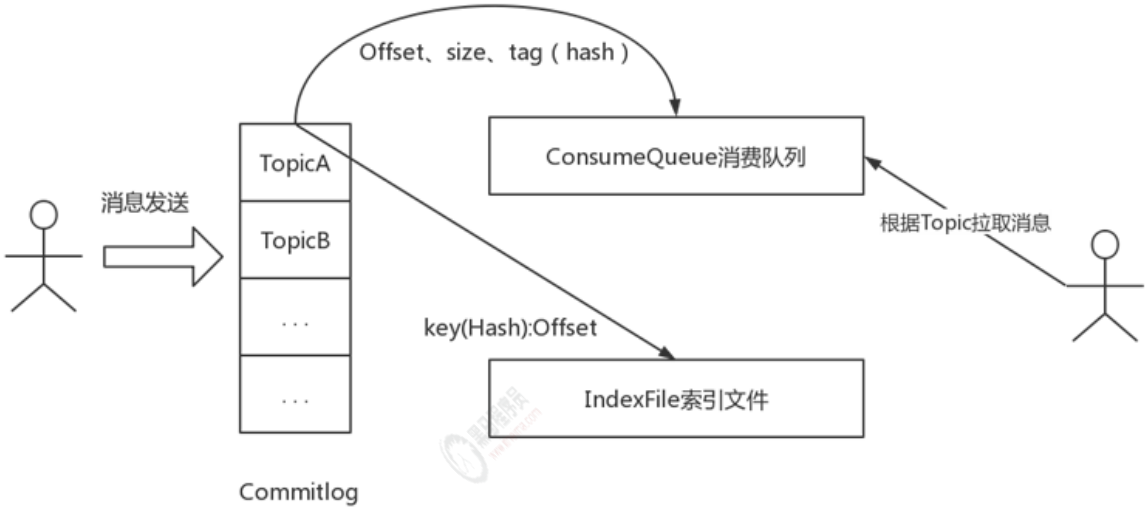
CommitLog 以物理文件的方式存放，每台 Broker 上的 CommitLog 被本机器所有 ConsumeQueue 共享，文件地址：$ {user.home} \store$ { commitlog} \ $ { fileName}。

文件名长度为20位，左边补零，剩余为起始偏移量，比如00000000000000000000代表了第一个文件，起始偏移量为0，文件大小为1G=1073741824；当第一个文件写满了，第二个文件为00000000001073741824，起始偏移量为1073741824，以此类推。消息主要是顺序写入日志文件，当文件满了，写入下一个文件。

1.5.2 ConsumeQueue

> consumequeue文件可以看成是基于topic的commitlog索引文件。

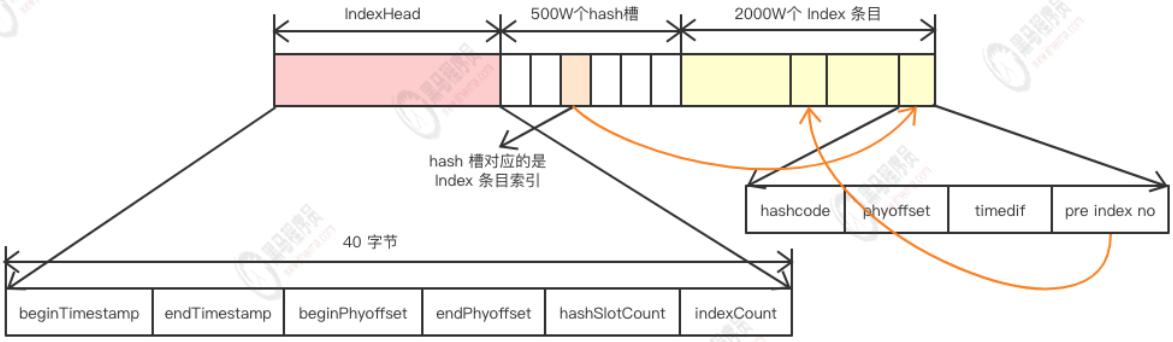
RocketMQ基于主题订阅模式实现消息的消费，消费者关心的是主题下的所有消息。但是由于不同的主题的消息不连续的存储在commitlog文件中，如果只是检索该消息文件可想而知会有多慢，为了提高效率，对应的主题的队列建立了索引文件，为了加快消息的检索和节省磁盘空间，每一个consumequeue条目存储了消息的关键信息commitog文件中的偏移量、消息长度、tag的hashcode值。



1.5.3 IndexFile

index 存的是索引文件，用于为生成的索引文件提供访问服务，这个文件用来加快消息查询的速度，通过消息Key值查询消息真正的实体内容。消息消费队列 RocketMQ 专门为消息订阅构建的索引文件，提高根据主题与消息检索 消息的速度，使用 Hash 索引机制，具体是 Hash 槽与 Hash 冲突的链表结构。

在实际的物理存储上，文件名则是以创建时的时间戳命名的，固定的单个IndexFile文件大小约为400M，一个IndexFile可以保存 2000W个索引



1.5.4 Config

> config 文件夹中 存储着 Topic 和 Consumer 等相关信息。主题和消费者群组相关的信息就存在在此。

- topics.json : topic 配置属性
- subscriptionGroup.json :消息消费组配置信息。
- delayOffset.json： 延时消息队列拉取进度。
- consumerOffset.json： 集群消费模式消息消进度。consumerFilter.json： 主题消息过滤信息。

2 消费进度管理

业务实现消费回调的时候，当且仅当此回调函数返回`ConsumeConcurrentlyStatus.CONSUME_SUCCESS`，RocketMQ才会认为这批消息（默认是1条）是消费完成的

如果这时候消息消费失败，例如数据库异常，余额不足扣款失败等一切业务认为消息需要重试的场景，只要返回ConsumeConcurrentlyStatus.RECONSUME_LATER，RocketMQ就会认为这批消息消费失败了。

为了保证消息是肯定被至少消费成功一次，RocketMQ会把这批消费失败的消息重发回Broker（topic不是原topic而是这个消费租的RETRY topic），在延迟的某个时间点（默认是10秒，业务可设置）后，再次投递到这个ConsumerGroup。而如果一直这样重复消费都持续失败到一定次数（默认16次），就会投递到DLQ死信队列。应用可以监控死信队列来做人工干预。

2.1 从哪里开始消费

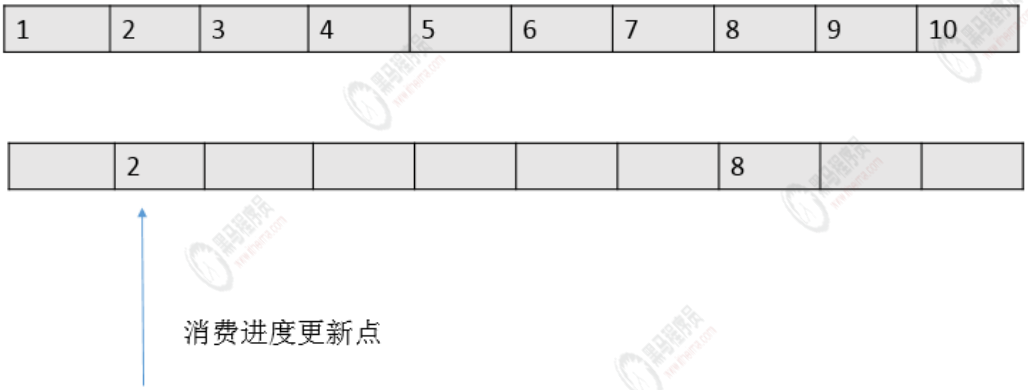
当新实例启动的时候，PushConsumer会拿到本消费组broker已经记录好的消费进度，如果这个消费进度在Broker并没有存储起来，证明这个是一个全新的消费组，这时候客户端有几个策略可以选择：

```
CONSUME_FROM_LAST_OFFSET //默认策略，从该队列最尾开始消费，即跳过历史消息
CONSUME_FROM_FIRST_OFFSET //从队列最开始开始消费，即历史消息（还储存在broker的）全部消费一遍
CONSUME_FROM_TIMESTAMP//从某个时间点开始消费，和setConsumeTimestamp()配合使用，默认是半个小时以前
```

2.2 消息ACK机制

RocketMQ是以consumer group+queue为单位是管理消费进度的，以一个consumer offset标记这个这个消费组在这条queue上的消费进度。如果某已存在的消费组出现了新消费实例的时候，依靠这个组的消费进度，就可以判断第一次是从哪里开始拉取的。

每次消息成功后，本地的消费进度会被更新，然后由定时器定时同步到broker，以此持久化消费进度。但是每次记录消费进度的时候，只会把一批消息中最小的offset值为消费进度值，如下图：



2.3 重复消费问题

这定时方式和传统的一条message单独ack的方式有本质的区别。性能上提升的同时，会带来一个潜在的重复问题——由于消费进度只是记录了一个下标，就可能出现拉取了100条消息如 2101-2200的消息，后面99条都消费结束了，只有2101消费一直没有结束的情况。

在这种情况下，RocketMQ为了保证消息肯定被消费成功，消费进度职能维持在2101，直到2101也消费结束了，本地的消费进度才能标记2200消费结束了（注：consumerOffset=2201）。

在这种设计下，就有消费大量重复的风险。如2101在还没有消费完成的时候消费实例**突然退出（机器断电，或者被kill）**。这条queue的消费进度还是维持在2101，当queue重新分配给新的实例的时候，新的实例从broker上拿到的消费进度还是维持在2101，这时候就会又从2101开始消费，2102-2200这批消息实际上已经被消费过还是会投递一次。

对于这个场景，RocketMQ暂时无能为力，所以业务必须要保证消息消费的幂等性，这也是RocketMQ官方多次强调的态度。

2.4 重复消费验证

2.4.1 查看当前消费进度

> 检查队列消费的当前进度

```
cat consumerOffset.json
```

```
{
    "offsetTable": {
        "topicTest@rocket_test_consumer_group": {
            0: 33,
            1: 32,
            2: 32,
            3: 33
        },
        "%RETRY%rocket_test_consumer_group@rocket_test_consumer_group": {
            0: 6
        }
    }
}
```

> 通过consumerOffset.json我们可以知道当前topicTest主题的queue0消费到偏移量为28

2.4.2 消费者发送消息

> 消费者发送消息，并查看各个队列消息的偏移量

```
发送queueId:[2],偏移量offset:[32],发送状态:[SEND_OK]
发送queueId:[3],偏移量offset:[33],发送状态:[SEND_OK]
发送queueId:[0],偏移量offset:[33],发送状态:[SEND_OK]
发送queueId:[1],偏移量offset:[32],发送状态:[SEND_OK]
发送queueId:[2],偏移量offset:[33],发送状态:[SEND_OK]
发送queueId:[3],偏移量offset:[34],发送状态:[SEND_OK]
发送queueId:[0],偏移量offset:[34],发送状态:[SEND_OK]
发送queueId:[1],偏移量offset:[33],发送状态:[SEND_OK]
发送queueId:[2],偏移量offset:[34],发送状态:[SEND_OK]
发送queueId:[3],偏移量offset:[35],发送状态:[SEND_OK]
```

> 我们发现队列2的偏移量最小为29

> 消费的时候最小偏移量不提交，其他都正常

```
//队列2的偏移量为29的数据在等待
if (ext.getQueueId() == 2 && ext.getQueueOffset() == 29) {
    System.out.println("消息消费耗时较厂接收queueId:[" + ext.getQueueId() + "],偏移量
offset:[" + ext.getQueueOffset() + "]");
    //等待 模拟假死状态
    try {
        Thread.sleep(Integer.MAX_VALUE);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

> 运行查看日志

```
接收queueId:[3],偏移量offset:[34],接收时间:[1608880793658],消息=[Hello Java demo
RocketMQ 5]
接收queueId:[3],偏移量offset:[33],接收时间:[1608880793658],消息=[Hello Java demo
RocketMQ 1]
接收queueId:[3],偏移量offset:[35],接收时间:[1608880793658],消息=[Hello Java demo
RocketMQ 9]
接收queueId:[1],偏移量offset:[32],接收时间:[1608880794684],消息=[Hello Java demo
RocketMQ 3]
接收queueId:[0],偏移量offset:[34],接收时间:[1608880794687],消息=[Hello Java demo
RocketMQ 6]
接收queueId:[1],偏移量offset:[33],接收时间:[1608880794685],消息=[Hello Java demo
RocketMQ 7]
接收queueId:[2],偏移量offset:[33],接收时间:[1608880794689],消息=[Hello Java demo
RocketMQ 4]
接收queueId:[0],偏移量offset:[33],接收时间:[1608880794685],消息=[Hello Java demo
RocketMQ 2]
模拟服务宕机无法确认消息,接收queueId:[2],偏移量offset:[32]
接收queueId:[2],偏移量offset:[34],接收时间:[1608880794691],消息=[Hello Java demo
RocketMQ 8]
```

> 我们发现只有队列2的偏移量为29的消息消费超时，其他都已经正常消费
>
> 我们再查看下consumerOffset.json

```
cat consumerOffset.jso
```

```
{
    "offsetTable": {
        "topicTest@rocket_test_consumer_group": {
            0: 33,
            1: 32,
            2: 32,
            3: 33
        },
        "%RETRY%rocket_test_consumer_group@rocket_test_consumer_group": {
            0: 6
        }
    }
}
```

> 我们发现rocketMQ 整个消费记录都没有被提交，所以下次消费会全部再次消费

2.4.3 再次消费

> 去掉延时代码继续消费

```
接收queueId:[3],偏移量offset:[35],接收时间:[1608880958530],消息=[Hello Java demo
RocketMQ 9]
接收queueId:[3],偏移量offset:[33],接收时间:[1608880958530],消息=[Hello Java demo
RocketMQ 1]
接收queueId:[3],偏移量offset:[34],接收时间:[1608880958530],消息=[Hello Java demo
RocketMQ 5]
接收queueId:[2],偏移量offset:[32],接收时间:[1608880959539],消息=[Hello Java demo
RocketMQ 0]
接收queueId:[2],偏移量offset:[33],接收时间:[1608880959560],消息=[Hello Java demo
RocketMQ 4]
接收queueId:[0],偏移量offset:[33],接收时间:[1608880959561],消息=[Hello Java demo
RocketMQ 2]
接收queueId:[2],偏移量offset:[34],接收时间:[1608880959561],消息=[Hello Java demo
RocketMQ 8]
接收queueId:[1],偏移量offset:[32],接收时间:[1608880959564],消息=[Hello Java demo
RocketMQ 3]
接收queueId:[1],偏移量offset:[33],接收时间:[1608880959564],消息=[Hello Java demo
RocketMQ 7]
接收queueId:[0],偏移量offset:[34],接收时间:[1608880959566],消息=[Hello Java demo
RocketMQ 6]
```

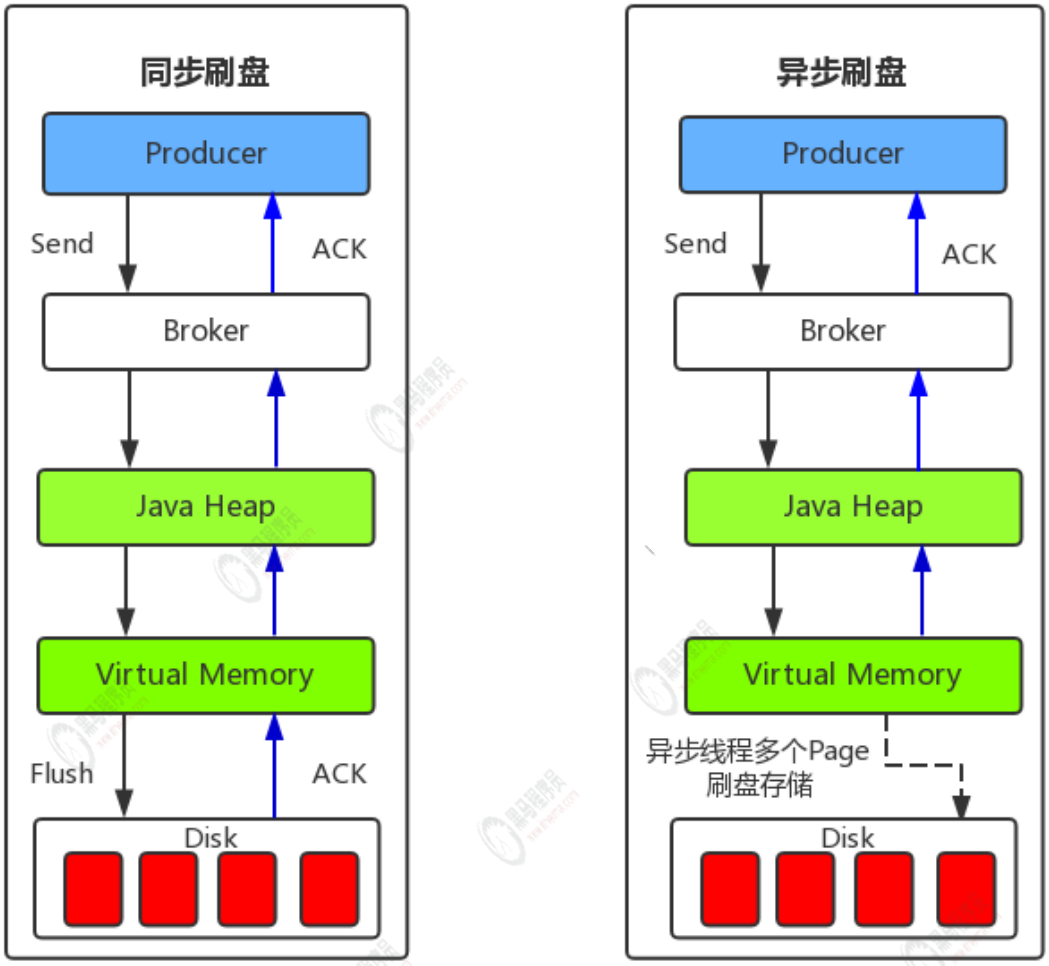
> 我们发现消息被重复消费了一遍

3 文件刷盘机制

> RocketMQ 的消息是存储在磁盘上的，这样做有两个优点：

- 保证断电后恢复
- 让存储的消息量超出内存的限制

RocketMQ 存储与读写是基于 JDK NIO 的内存映射机制，具体使用 MappedByteBuffer（基于 MappedByteBuffer 操作大文件的方式，其读写性能极高）RocketMQ 的消息是存储到磁盘上的，这样既能保证断电后恢复，又可以让存储的消息 超出内存的限制 RocketMQ 为了提高性能，会尽可能地保证磁盘的顺序写 消息在通过 Producer 写人 RocketMQ 的时候，有两种写磁盘方式：



3.1 同步刷盘方式

如上图所示，只有在消息真正持久化至磁盘后，RocketMQ的Broker端才会真正地返回给Producer端一个成功的ACK响应。同步刷盘对MQ消息可靠性来说是一种不错的保障，但是性能上会有较大影响，一般适用于金融业务应用领域。

3.2 异步刷盘

能够充分利用OS的PageCache的优势，只要消息写入PageCache即可将成功的ACK返回给Producer端。消息刷盘采用后台异步线程提交的方式进行，降低了读写延迟，提高了MQ的性能和吞吐量。异步和同步刷盘的区别在于，异步刷盘时，主线程并不会阻塞，在将刷盘线程wakeup后，就会继续执行。

3.3 刷盘方式对比

	同步刷盘	异步刷盘
消息情况	在返回写成功状态时，消息已经被写入磁盘中。即消息被写入内存的PAGECACHE 中后，立刻通知刷新线程刷盘，等待刷盘完成，才会唤醒等待的线程并返回成功状态	在返回写成功状态时，消息可能只是被写入内存的 PAGECACHE 中。当内存的消息量积累到一定程度时，触发写操作快速写入
性能	需要等待刷盘才能返回结果	消息写入内存后立刻返回结果，吞吐量更高
可靠性	可以保持MQ的消息状态和生产者/消费者的消息状态一致	Master宕机，磁盘损坏的情况下，会丢失少量的消息, 导致MQ的消息状态和生产者/消费者的消息状态不一致

4 过期文件删除

4.1 为什么删除过期文件

由于RocketMQ操作CommitLog、ConsumeQueue文件是基于文件内存映射机制，并且在启动的时候会将所有的文件加载，为了避免内存与磁盘的浪费、能够让磁盘能够循环利用、避免因为磁盘不足导致消息无法写入等引入了文件过期删除机制。

4.2 删除文件的思路

RocketMQ顺序写CommitLog文件、ComsumeQueue文件，所有的写操作都会落到最后一个文件上，因此在当前写文件之前的文件将不会有数据插入，也就不会有任何变动，因此可通过时间来做判断，比如超过72小时未更新的文件将会被删除

注意：RocketMQ删除过期文件时不会关注该文件的内容是否全部被消费

4.3 如何防止重复投递

> 客户端通过broker的消费进度确定自己需要拉取那些消息

消息的存储是一直存在于CommitLog中的。而由于CommitLog是以文件为单位（而非消息）存在的，CommitLog的设计是只允许顺序写的，且每个消息大小不定长，所以这决定了消息文件几乎不可能按照消息为单位删除（否则性能会极具下降，逻辑也非常复杂）。所以消息被消费了，消息所占据的物理空间并不会立刻被回收。

> 但消息既然一直没有删除，那RocketMQ怎么知道应该投递过的消息就不再投递?

答案是客户端自身维护——客户端拉取完消息之后，在响应体中，broker会返回下一次应该拉取的位置，PushConsumer通过这一个位置，更新自己下一次的pull请求。这样就保证了正常情况下，消息只会被投递一次。

4.4 过期文件删除流程

> 删除过期文件的整体流程如下：

- 开启定时任务每10s扫描是否有文件需要删除
- 有三种情况会进入删除文件操作：到了deleteWhere指定的时间点（默认是凌晨4点）、磁盘不足、手动触发
- 对于磁盘不足的情况，当磁盘使用率大于磁盘空间警戒线水位（默认是90%），会阻止消息写入，当超过85%时会强制删除文件（需要设置允许强制删除参数，否者不生效），其他两种情况都只能删除过期的文件（文件最后更新时间+文件最大的存活时间 < 当前时间）
- 当被删除的文件存在引用时，会有一个文件删除缓存时间，在这段时间内，该文件不会被删除，主要是留给引用该文件程序一些时间，当超过了文件删除缓存时间后，每次都会将该文件的引用减少

1000，直到减少小于等于0后才释放该文件引用的相关资源，然后将该文件放入一个“文件删除集合”中

- 一次连续删除文件中间会存在一定的间隔，不会连续释放文件相关的资源
- 一次连续删除的文件和不大于10
- 将“文件删除集合”中的文件从

5 高可用

5.1 NameServer 高可用

由于 NameServer 节点是无状态的，且各个节点直接的数据是一致的，故存在多个 NameServer 节点的情况下，部分 NameServer 不可用也可以保证 MQ 服务正常运行

5.1.1 BrokerServer 高可用

> RocketMQ是通过 Master 和 Slave 的配合达到 BrokerServer 模块的高可用性的，一个 Master 可以配置多个 Slave，同时也支持配置多个 Master-Slave 组。

当其中一个 Master 出现问题时：

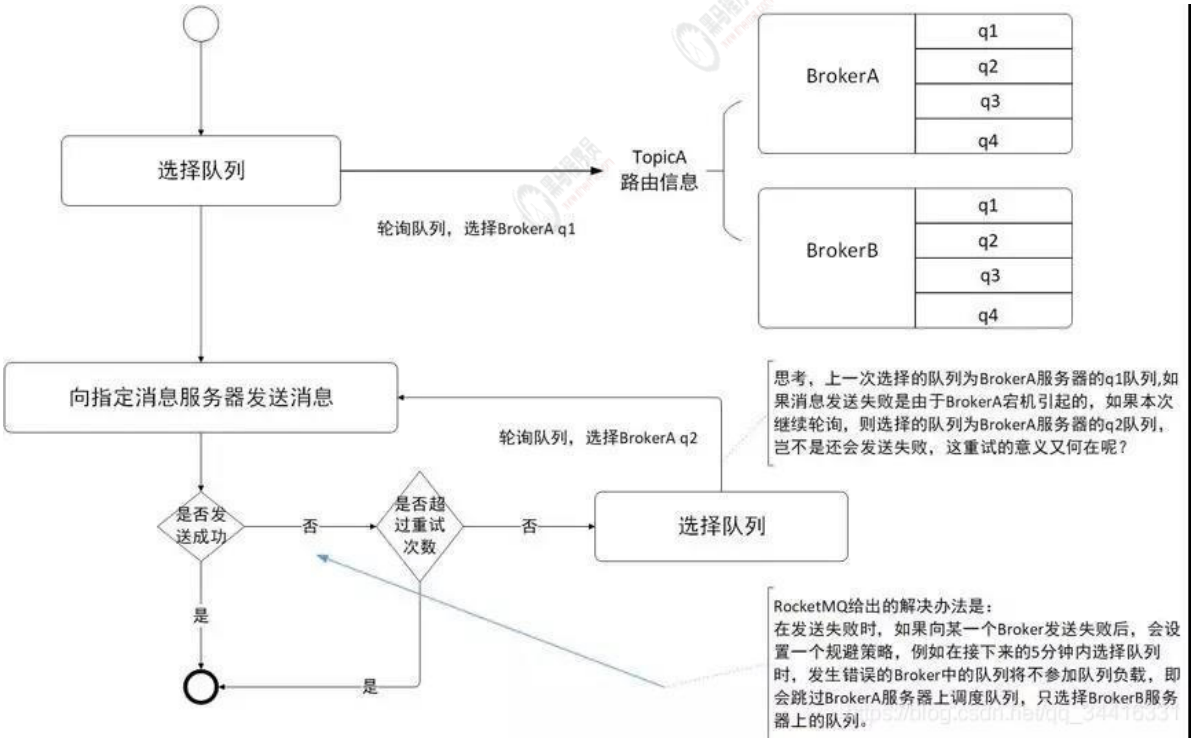
- 由于Slave只负责读，当 Master 不可用，它对应的 Slave 仍能保证消息被正常消费
- 由于配置多组 Master-Slave 组，其他的 Master-Slave 组也会保证消息的正常发送和消费

5.2 消息消费高可用

Consumer 的高可用是依赖于 Master-Slave 配置的，由于 Master 能够支持读写消息，Slave 支持读消息，当 Master 不可用或繁忙时， Consumer 会被自动切换到从 Slave 读取(自动切换，无需配置)。故当 Master 的机器故障后，消息仍可从 Slave 中被消费

5.3 消息发送高可用

在创建Topic的时候，把Topic的多个Message Queue创建在多个Broker组上（相同Broker名称，不同 brokerId的机器组成一个Broker组），这样当一个Broker组的Master不可用后，其他组的Master仍然可用，Producer仍然可以发送消息。 RocketMQ目前还不支持把Slave自动转成Master，如果机器资源不足， 需要把Slave转成Master，则要手动停止Slave角色的Broker，更改配置文 件，用新的配置文件启动Broker。



5.4 消息主从复制

5.4.1 同步复制和异步复制

> 若一个 Broker 组有一个 Master 和 Slave，消息需要从 Master 复制到 Slave 上，有同步复制和异步复制两种方式

	同步复制	异步复制
概念	即等 Master 和 Slave 均写成功后才反馈给客户端写成功状态	只要 Master 写成功，就反馈客户端写成功状态
可靠性	可靠性高，若 Master 出现故障，Slave 上有全部的备份数据，容易恢复	若 Master 出现故障，可能存在一些数据还没来得及写入 Slave，可能会丢失
效率	由于是同步复制，会增加数据写入延迟，降低系统吞吐量	由于只要写入 Master 即可，故数据写入延迟较低，吞吐量较高

5.4.2 配置方式

> 可以对 broker 配置文件里的 brokerRole 参数进行设置，提供的值有：

- **ASYNC_MASTER**：异步复制
- **SYNC_MASTER**：同步复制
- **SLAVE**：表明当前是从节点，无需配置 brokerRole

5.4.3 实际应用

> 在实际应用中，由于同步刷盘方式会频繁触发磁盘写操作，明显降低性能，故通常配置为：

- **刷盘方式**：ASYNC_FLUSH(异步刷盘)
- **主从复制**：SYNC_MASTER(同步复制)

异步刷盘能够避免频繁触发磁盘写操作，除非服务器宕机，否则不会造成消息丢失。

主从同步复制能够保证消息不丢失，即使 Master 节点异常，也能保证 Slave 节点存储所有消息并被正常消费掉。

6 业务上保障

6.1 为什么从业务上保证

6.1.1 消息丢失问题

RocketMQ虽然号称消息不会丢失，但是还是有几率存在MQ宕机以及rocketMQ使用上的问题可能存在消息丢失等，对于类似于支付确认的消息一般来说是一条都不允许丢失的

6.1.2 消息幂等性

> 在网络环境中，由于网络不稳定等因素，消息队列的消息有可能出现重复，大概有以下几种：

6.1.2.1 发送时消息重复

当一条消息已被成功发送到服务端并完成持久化，此时出现了网络闪断或者客户端宕机，导致服务端对客户端应答失败。 如果此时生产者意识到消息发送失败并尝试再次发送消息，消费者后续会收到两条内容相同并且 Message ID 也相同的消息。

6.1.2.2 投递时消息重复

消息消费的场景下，消息已投递到消费者并完成业务处理，当客户端给服务端反馈应答的时候网络闪断。 为了保证消息至少被消费一次，消息队列 RocketMQ 的服务端将在网络恢复后再次尝试投递之前已被处理过的消息，消费者后续会收到两条内容相同并且 Message ID 也相同的消息。

6.1.2.3 负载均衡时消息重复

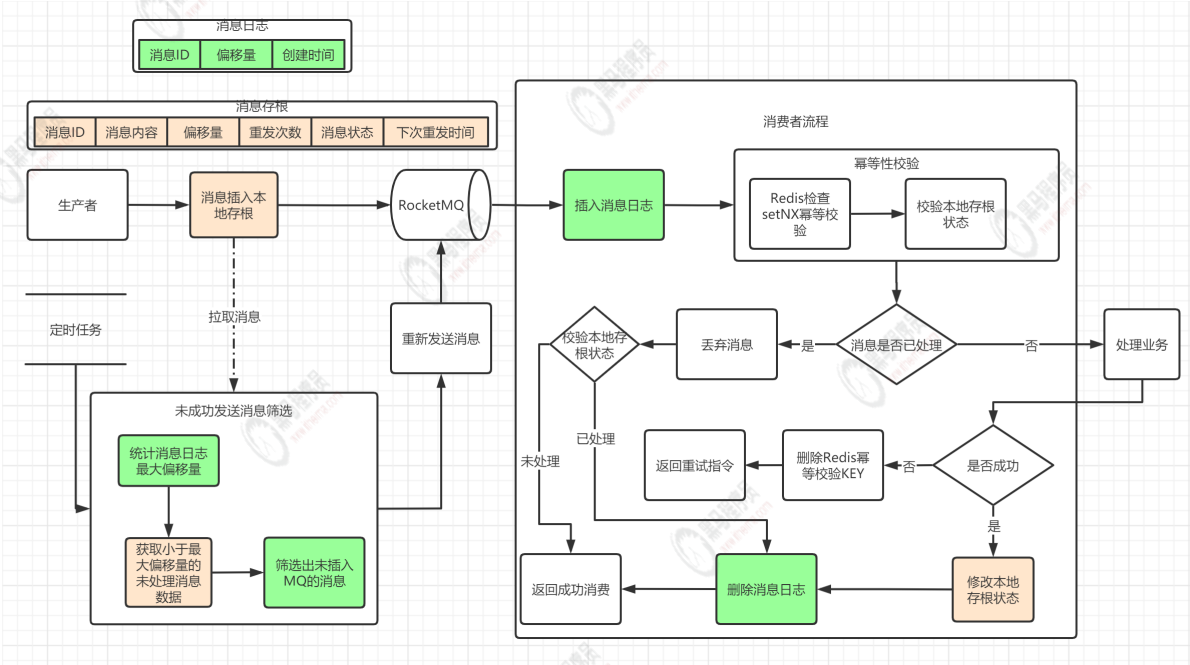
> 包括但不限于网络抖动、Broker 重启以及订阅方应用重启

当消息队列 RocketMQ 的 Broker 或客户端重启、扩容或缩容时，会触发 Rebalance，此时消费者可能会收到重复消息。

结合三种情况，可以发现消息重发的最后结果都是，消费者接收到了重复消息，那么，我们只需要在消费者端统一进行幂等处理就能够实现消息幂等。

6.2 整体架构

> 上面我们是从MQ本身来保证消息的的可靠性，下面我们从业务上来分析如何保障MQ的可靠性，Mq 的消息成功投递和消费是比较难的，这里提供一个思路，如何保证消息成功投递并且消息是幂等的。



6.2.1 表结构设计

> 这里面涉及到两张表

6.2.1.1 消息存根

> 消息发送到 mq 之前先把消息存储到 mongodb 或者 mysql 中，消息有一个存根是为了避免消息在 mq 中丢失后就找不到发送的消息，主要需要保存几个关键信息:

- messageId： 消息流水号，用来在消息的生产端和消费端串联使用，根据这个id找到消费端和生产端唯一的消息
- messageContent：消息内容
- count：发送到 broker 的次数，如果超过特定次数就不往 broker 发
- status：消息状态，已确认、未确认、已作废，如果消费端已经确认就需要修改该状态
- offset：偏移量，发送的时候记录消息的偏移量。

- resendTime：重发时间，需要加上消息可能在队列的堆积时间，否则可能造成消息还未被消费到就被重发了
- createDatre：记录消息的发送时间

6.2.1.2 消息日志

> 为了保障消息能够成功发送到MQ，需要在拉取到消息后的第一时间将消息ID保存到消息日志表，用来让发送端知道消息消息是否成功发送到了MQ，需要包含一下字段：

- messageId： 消息流水号，为了和消息存根来保持一致
- offset：用来记录当前接收到的消息的偏移量，重发时需要通过统计最大偏移量来确定消费者是否堆积来确定是否重发
- createDate：用来记录当前消息接收到的时间

6.3 幂等性校验

6.3.1 Redis幂等性校验

> 首先进行Redis的setNX进行幂等性校验，有以下情况

- 成功：代表redis幂等性校验通过，为了防止Redis数据误删或者过期，需要进行数据库的幂等性校验，检查消息存根是否是未处理状态，如果是未处理状态则进行业务，则说明消息是未处理状态，否则幂等性校验失败
- 失败：代表redis幂等性校验未通过，则不通过数据库直接进行校验失败处理

6.3.2 DB幂等性校验

> 如果Redis校验没有通过，还需要DB进行幂等性校验，有以下情况

- 成功：说明消息确定时未处理的，需要进行处理
- 失败：说明Redis缓存已经过期或者误删，这里做兜底处理

6.3.3 校验失败处理

> 幂等性校验失败则说明消息是重复，存在一下两种情况

6.3.3.1 Redis校验失败

> 如果直接Redis的setNX校验失败，说明是重复消息，但是这个时候消息是不知道消息是否处理完成，有以下两种情况：

- 消息处理完成：需要删除消息日志以及返回成功消费的标志
- 消息未处理完成：这个时候说明已经有一个线程在处理消息了，直接结束并返回成功消费的标志。

> 正确处理逻辑如下

这个时候的处理业务应该是Redis校验失败后，但是并不能确定消息是否真的已经处理完成还是处理中的消息，需要先检查消息存根状态：

- 如果消息处理成功需要删除消息日志，返回成功标志。
- 如果没有处理成功，则说明另一个线程正在处理，直接返回成功标志。

6.3.3.2 DB校验失败

这个时候说明消息已经处理完成了，redis并且已经通过setNX已经设置标志了，需要删除消息日志，并且返回成功标志。

6.3.4 校验成功

幂等性校验成功后就需要处理业务操作了

6.4 业务处理

> 业务操作分为两种情况：

6.4.1 操作成功

如果操作成功，需要修改消息存根状态，并且删除消息日志，然后返回成功标志

6.4.2 操作失败（异常）

如果操作失败，需要消费端重试，这个时候删除redis的setNX的值，并且返回重试指令，让消费端进行重试，但是重试可能一直不成功，RocketMQ的消费端重试机制，达到上限后会投递到死信队列，后期需要人工处理。

6.5 定时重发消息

6.5.1 重发消息筛选

> 需要符合一下条件的数据才会被筛选出来

- 消息存根中未确认的消息
- 消息存根中发送次数小于最大发送次数的消息
- 消息存根中下次发送时间大于当前时间的消息
- 消息存根中的偏移量小于消息日志中的最大偏移量的消息
- 消息存根中的消息ID在消息日志中不存在的消息

6.5.2 重发消息

> 符合以上条件的消息需要进行重发，调用MQProducer客户端进行重发消息，重发完成后还需要做一下事情

- 修改消息的重发次数，当前次数+1
- 修改消息的下次重发时间，通过规则计算需要间隔多长时间才需要重发
- 修改消息的偏移量，设置为当前发送消息的的偏移量

6.5.3 人工处理

如果消息的发送次数达到最大的发送次数，将无法进行重发，需要人工进行处理，可以通过发邮件以及其他的方式通知开发人员进行后续的人工处理