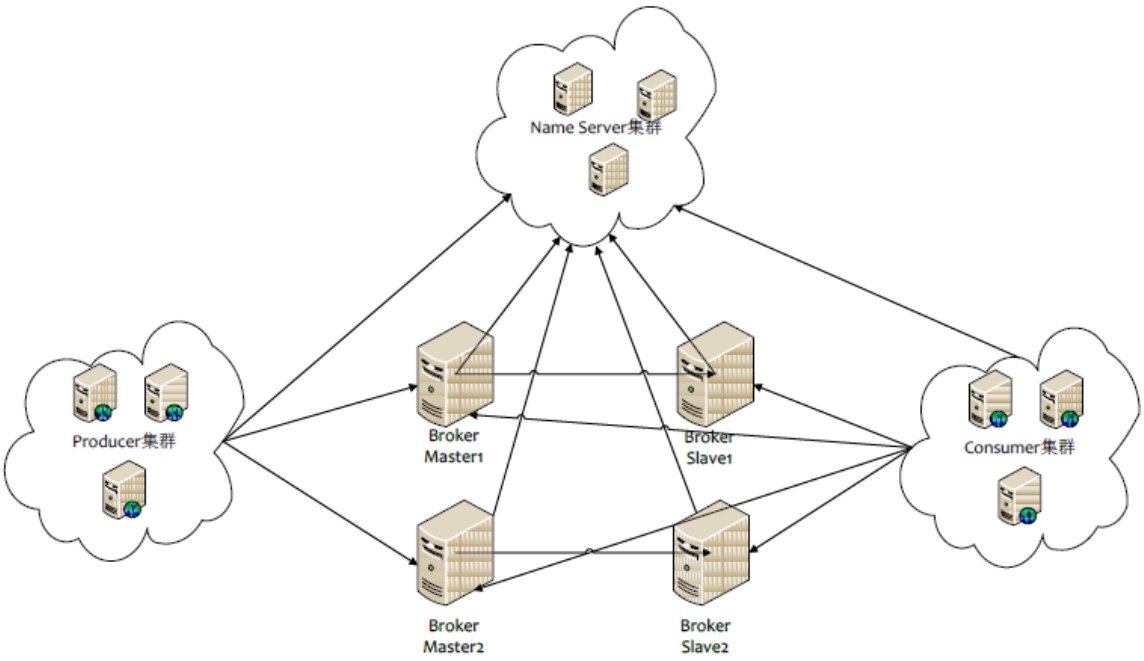
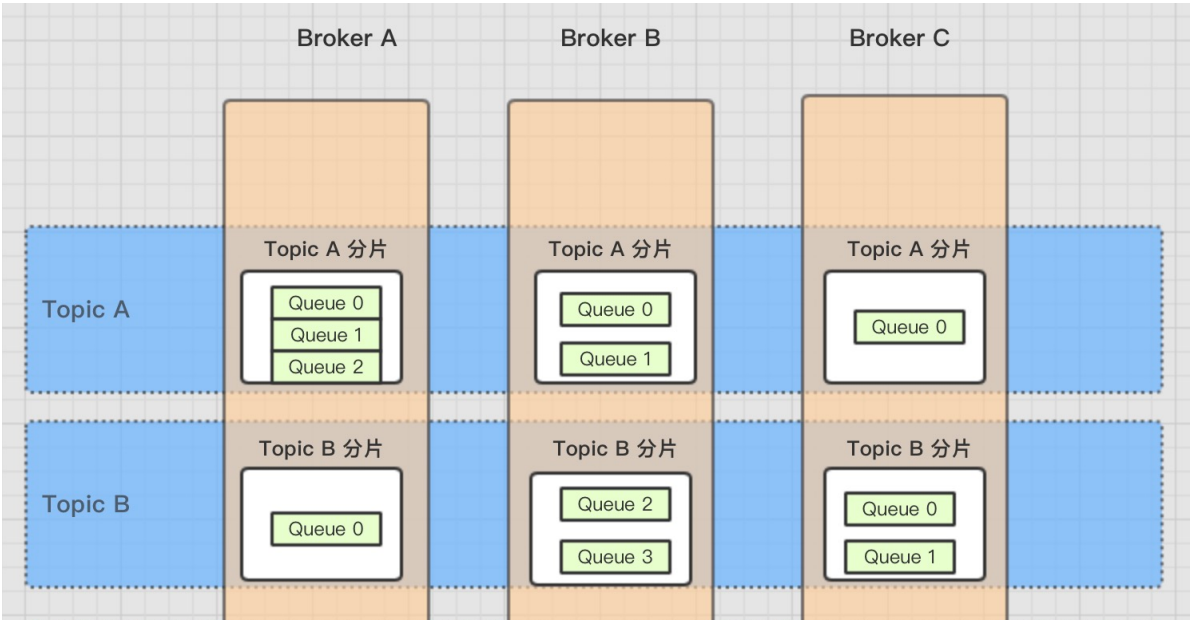
RocketMQ





> `Producer通过NameServer获取所有Broker的路由信息`，根据负载均衡策略选择将消息发到哪个Broker，然后调用Broker接口提交消息。
>
> Consumer通过NameServer获取所有broker的路由信息后，向Broker发送Pull请求来获取消息数据。Consumer可以以两种模式启动，**广播（Broadcast）和集群（Cluster），广播模式下，一条消息会发送给所有Consumer，集群模式下消息只会发送给一个Consumer。**
>
> Topic用于将消息按主题做划分，**Producer将消息发往指定的Topic，Consumer订阅该Topic就可以收到这条消息。**Topic跟发送方和消费方都没有强关联关系，发送方可以同时往多个Topic投放消息，消费方也可以订阅多个Topic的消息。在RocketMQ中，**Topic是一个上逻辑概念。消息存储不会按Topic分开。**

> Message代表一条消息，使用`MessageId`唯一识别，用户在发送时可以设置messageKey，便于之后查询和跟踪。一个 Message 必须指定 Topic，相当于寄信的地址。Message 还有一个可选的 Tag 设置，以便消费端可以基于 Tag 进行过滤消息。也可以添加额外的键值对，例如你需要一个业务 key 来查找 Broker 上的消息，方便在开发过程中诊断问题。
>
> Tag标签可以被认为是对 Topic 进一步细化。一般在相同业务模块中通过引入标签来标记不同用途的消息。
>
> Broker是RocketMQ的核心模块，`负责接收并存储消息`，同时提供Push/Pull接口来将消息发送给Consumer。Consumer可选择从Master或者Slave读取数据。多个主/从组成Broker集群，集群内的Master节点之间不做数据交互。Broker同时提供消息查询的功能，可以通过MessageID和MessageKey来查询消息。Borker会将自己的Topic配置信息实时同步到NameServer。
>
> **Topic和Queue是1对多的关系，一个Topic下可以包含多个Queue**，主要用于负载均衡。发送消息时，用户只指定Topic，Producer会根据Topic的路由信息选择具体发到哪个Queue上。Consumer订阅消息时，会根据负载均衡策略决定订阅哪些Queue的消息。
>
> RocketMQ在存储消息时会为每个Topic下的每个Queue生成一个消息的索引文件，每个Queue都对应一个Offset**记录当前Queue中消息条数。**
>
> NameServer可以看作是RocketMQ的注册中心，它管理两部分数据：集群的Topic-Queue的路由配置；Broker的实时配置信息。其它模块通过Nameserv提供的接口获取最新的Topic配置和路由信息。

Topic是一个逻辑上的概念，实际上Message是在每个Broker上以Queue的形式记录。`Queue不是真正存储Message的地方，真正存储Message的地方是在CommitLog`。Queue存储的是Message的索引，根据索引至commitLog取出真正的消息体

一个Topic的数据可能会存在多个Broker中。

一个Broker存在多个Queue。

单个的Queue也可能存储多个Topic的消息。

1.Name Server与Producer、Consumer、Broker交互

Name Server是一个几乎无状态节点，可集群部署，节点之间无任何信息同步。

每一个Broker与Name Server中所有节点建立长连接，定时注册topic信息到所有的Name Server节点

Producer/Consumer与Name Server集群中的其中一个节点（随机选择）建立长连接，定期从Name Server取Topic路由信息。

Broker每隔30s向所有的Name Server发送心跳，心跳包含了Topic的配置信息；

Name Server每隔10s扫描所有存活的Broker，发现某一个Broker两分钟内没有发送心跳包，则与其断开连接；

默认情况下，Producer或者Consumer每隔30s从Name Server获取所有的Topic的最新的队列信息，这意味者如果某个Broker宕机，Producer或者Consumer要过30s才能感知到。

发送MQ消息

第一步：**找到Topic所在的路由信息**

1. 先从本地缓存的路由表中查询；

2. 没有找到的话，便向NameSrv发起请求，更新本地路由表，再次查询。
3. 如果仍然没有查询到，表明Topic没有事先配置，则用MQ自身默认的**Topic TBW102**向NameSrv发起查询，返回TBW102的路由信息，暂时作为该Topic的路由。

> 如果某Broker配置了 autoCreateTopicEnable，允许自动创建Topic，那么在该Broker启动后，便会向自己的路由表中插入TBW102这个Topic，并注册到NameSrv，表明处理该Topic类型的消息。

第二步：**选择某个Queue用来发送消息**

简单点说，主要逻辑就是在消息发送的基础上加上了**超时机制**及**重试机制。**当选择某个Queue发送消息失败后，只要还没有超时，且没有超出最大重试次数，就会再次选择某个Queue进行重试。

```
public MessageQueue selectOneMessageQueue(String lastBrokerName) {
  if (lastBrokerName != null) {
    int index = this.sendWhichQueue.getAndIncrement();
    for (int i = 0; i < this.messageQueueList.size(); i++) {
      int pos = Math.abs(index++) % this.messageQueueList.size();
      MessageQueue mq = this.messageQueueList.get(pos);
      if (!mq.getBrokerName().equals(lastBrokerName)) {
        return mq;
      }
    }

    return null;
  }

  int index = this.sendWhichQueue.getAndIncrement();
  int pos = Math.abs(index) % this.messageQueueList.size();
  return this.messageQueueList.get(pos);
}
```

选择Queue的具体逻辑在topicPublishInfo.selectOneMessageQueue（lastBrokerName）中。这里在调用时传入了lastBrokerName。核心就是选择一个与lastBrokerName姓名不同的Queue从而避免再次选择了lastBrokerName所在的Queue。相当于一种轮询的负载均衡方式。

第三步：发送MQ消息（消息发送的核心过程）

核心就是通过第二步选择的Queue获取对应的Broker的地址，通过该地址实现网络传输；

消息的网络传输在 DefaultMQProducerImpl的sendKernelImpl方法；

拿到Broker地址后，要将消息内容及其他信息封装进请求头：

```
SendMessageRequestHeader requestHeader = new SendMessageRequestHeader();
requestHeader.setProducerGroup(this.defaultMQProducer.getProducerGroup());
requestHeader.setTopic(msg.getTopic());
requestHeader.setDefaultTopic(this.defaultMQProducer.getCreateTopicKey());
requestHeader.setDefaultTopicQueueNums(this.defaultMQProducer.getDefaultTopicQue
ueNums());
requestHeader.setQueueId(mq.getQueueId());
......
```

接着调用MQClientAPIImpl的sendMessage（）方法：

```
 SendResult sendResult =
this.mQClientFactory.getMQClientAPIImpl().sendMessage(//
                    brokerAddr,// 1
                    mq.getBrokerName(),// 2
                    msg,// 3
                    requestHeader,// 4
                    timeout,// 5
                    communicationMode,// 6
                    sendCallback// 7
                    );
```

sendMessage（）内部便是创建请求，调用封装的Netty进行网络传输了。

有三种发送方式：单向、同步、异步

这里解释下三种发送方式。

- 单向：只管发送，不管是否发送成功；
- 同步：阻塞至拿到发送结果；
- 异步：发送后直接返回，在回调函数中等待发送结果。

看到这儿，消息的发送就已经结束了，成功的从生产者传输到了Broker。

整个发送流程：

1：获取路由信息

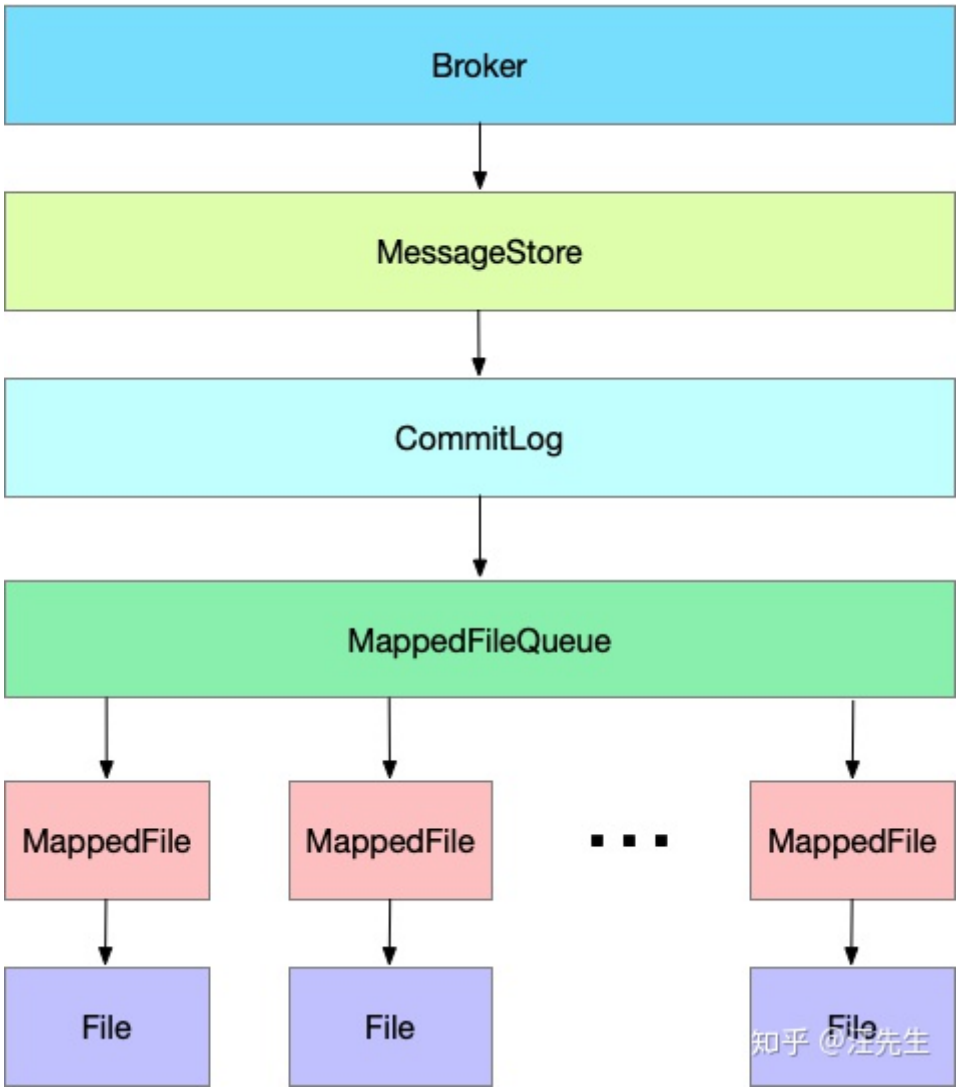
2：按照负载均衡方式，选择路由

3：根据选择出的路由，发送消息到Broker。

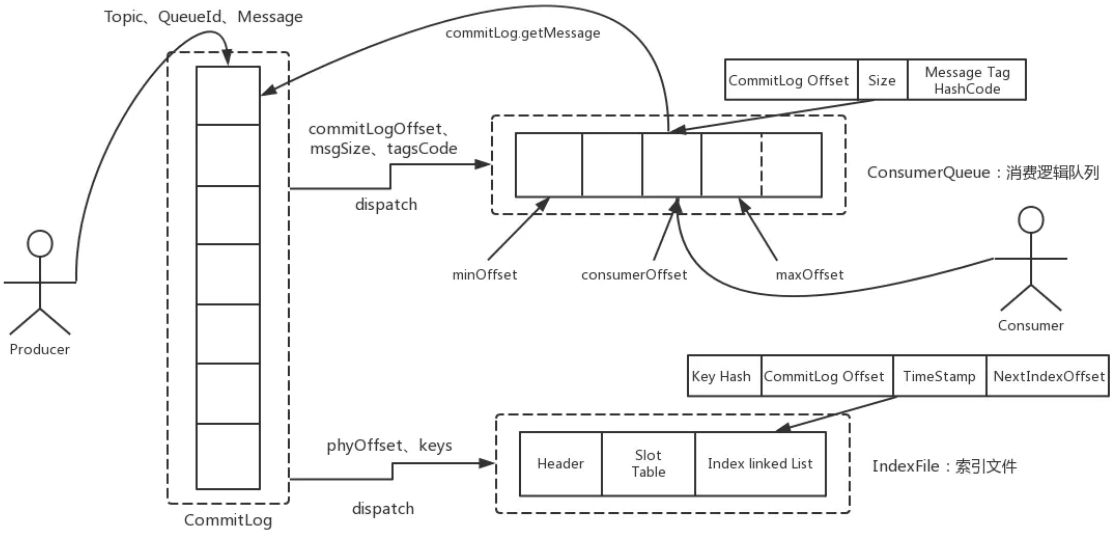
消息存储

> 如何设计消息的存储？
>
> - 首先我们肯定需要有块内存缓冲区，用来接收消息
> - 但是内存毕竟有限，当消息大量堆积的时候，全放在内存肯定是不合适的，所以我们肯定需要将消息从内存写到文件中。
> - 如果所有消息全都存放到一个文件中，消息检索会很耗时，过期消息的清理也会很麻烦，所以消息肯定要进行多文件存储。

每个Broker都对应有一个MessageStore，专门用来存储发送到它的消息，不过MessageStore本身不是文件，只是存储的一个抽象，MessageStore 中保存着一个 CommitLog，CommitLog 维护了一个MappedFileQueue，而MappedFileQueue 中又维护了多个 MappedFile，每个MappedFile都会映射到文件系统中一个文件，这些文件才是真正的存储消息的地方，MappedFile的文件名为它记录的第一条消息的全局物理偏移量。

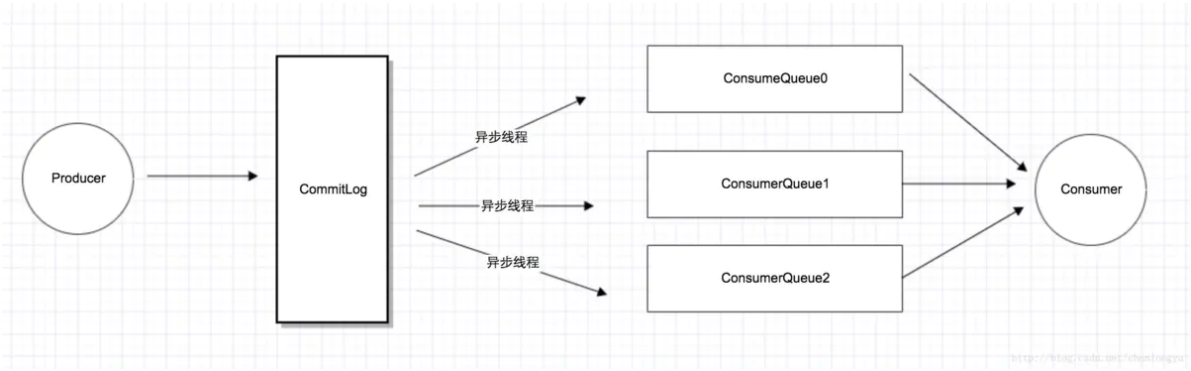


RocketMQ消息存储整体架构

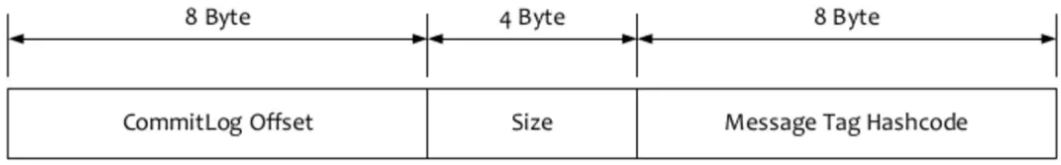


> RocketMQ采用的是混合型的存储结构，即为Broker单个实例下所有的队列共用一个日志数据文件（即为CommitLog）来存储。

> Producer发送消息至Broker端，然后Broker端使用同步或者异步的方式对消息刷盘持久化，保存至CommitLog中。只要消息被刷盘持久化至磁盘文件CommitLog中，那么Producer发送的消息就不会丢失。
>
> RocketMQ的具体做法是，使用Broker端的后台服务线程—ReputMessageService不停地分发请求并异步构建ConsumeQueue（逻辑消费队列）和IndexFile（索引文件）数据；ConsumeQueue（逻辑消费队列）作为消费消息的索引，保存了指定Topic下的队列消息在CommitLog中的起始物理偏移量offset，消息大小size和消息Tag的HashCode值。而IndexFile（索引文件）则只是为了消息查询提供了一种通过key或时间区间来查询消息的方法。
>
> **消息生产与消息消费相互分离**，Producer端发送消息最终写入的是CommitLog（消息存储的日志数据文件），**Consumer端先从ConsumeQueue（消息逻辑队列）读取持久化消息的起始物理位置偏移量offset、大小size和消息Tag的HashCode值，随后再从CommitLog中进行读取待拉取消费消息的真正实体内容部分；**这就导致对于commitLog来说写入消息虽然是顺序写，但是读却变成了随机读。



> **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。**
>
> 文件名规则：
>
> 文件名以已有存储容量依次递增，类似如下：
> 00000000000000000000
> 00000000001073741824
> 00000000002147483648



> CommitLog Offset是指这条消息在Commit Log文件中的实际偏移量
> Size存储中消息的大小
> Message Tag HashCode存储消息的Tag的哈希值：主要用于订阅时消息过滤（订阅时如果指定了Tag，会根据HashCode来快速查找到订阅的消息）
>
> RocketMq的删除机制。RocketMq并不会立即删除消息，所以消息是可以被重复消费的。
> RocketMq的消息时定期清除，默认3天。

消息刷盘

消息刷盘是指将内存中producer提交的message刷到磁盘文件上，实现持久化，RocketMQ实现了两种刷盘方式：同步刷盘与异步刷盘

> 同步刷盘的意思就是说当消息追加到内存后，就立即刷到文件中存储。
>
> 异步刷盘是指内存中的消息不是立即刷新到磁盘文件而是由定时任务定期异步刷新到磁盘
>
> 同步刷盘可以保证可靠性，但是对性能有影响；
>
> 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。

RocketMQ消息可靠性如何保证?

> 可能影响可靠性的问题：生产者丢失消息、MQ持久化消息丢失、消费者丢失消息
>
> 生产者丢失消息的可能点在于程序发送失败抛异常了没有重试处理，或者发送的过程成功但是过程中网络闪断MQ没收到，消息就丢失了。我们需要做的就是保证消息一定是成功发送了，如果同步发送的话这个很好确认，只有在消息成功发送了才会收到确认的ACK消息，异步的话如果要保证消息不会丢失需要有回调处理，过异步发送+回调通知+本地消息表的形式我们就可以做出一个解决方案。消息发送了，将消息落库到本地消息表并且设置状态为已发送，如果收到回调消息通知发送成功了，那么将本地消息表对应的消息的状态设置为已成功，同时启动一个定时任务系统，从本地消息表查询状态为成功的消息进行删除操作，同时查询长时间处于未成功状态的消息进行重试操作。
>
> MQ消息丢失是指消息在内存中还未及时刷盘到磁盘文件时，宕机了导致消息丢失。对于RocketMQ来说同步刷盘不会出现这个问题，但是异步刷盘可能会有这个问题；
>
> 消费者丢失消息是指消费者还未消费消息或者消费者收到消息未成功消费的时候，MQ认为消息已被消费没有重发消息。这种情况可以通过增加消费者ACK确认机制，只有MQ收到消费者ACK确认才认为该条消息消费成功，否则进行重试。重试超过一定次数进入死信队列人工处理；

消息积压如何处理?

> 消费者出错，消息不断进行重试，而且还有大量的消息不断进来导致消息积压，这种情况要考程序的问题，如果可以定位到程序中的原因可以修改程序然后重新上线处理；如果暂时无法定位到原因，可以写一个临时的consumer消费方案，把消息转发给一个新的topic和MQ资源，当然这个新的topic的机器资源最好是单独申请的可以抗住当前积压的消息。

Broker Master与Broker Slave如何让同步数据?

> 消息在master和slave之间的同步是根据raft协议来进行的：
>
> 1. 在broker收到消息后，会被标记为uncommitted状态
> 2. 然后会把消息发送给所有的slave
> 3. slave在收到消息之后返回ack响应给master
> 4. master在收到超过半数的ack之后，把消息标记为committed
> 5. 发送committed消息给所有slave，slave也修改状态为committed

RocketMQ Broker集群部署

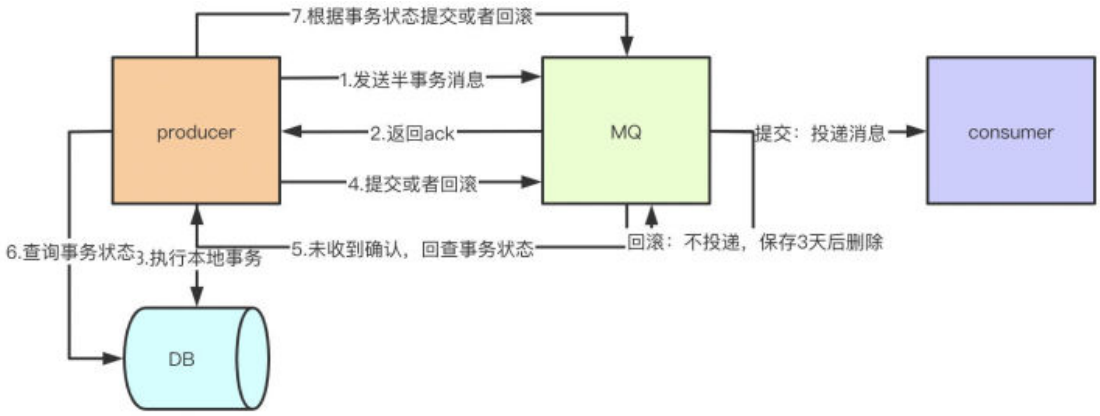
> Broker分为多master，多master多slave同步，多master多slave异步这三种部署方案。
>
> 其中master节点支持数据的读/写即生产者写以及消费者读，而slave节点仅仅支持读而不支持写。
>
> 多master无slave：
>
> 优点：配置简单
>
> 缺点：单个机器宕机之后，这台机器未被消费的消息在机器恢复之前无法订阅，消息实时性会收到影响；

> 多master多slave，异步复制
>
> 优点：每一个master配置一个slave，有多对master-slave，HA采用异步复制的方式，slave节点定期与master同步数据，这样的话master与slave可能存在一个短暂的消息延迟，这个时间等于master/slave同步数据的时间间隔。这样即使master宕机也可以从slave节点读取订阅的消息；
>
> 缺点：由于是异步复制所以假设master宕机的时候还有部分数据未同步至slave节点，那么这部分数据就必须等到master节点恢复之后再同步至slave节点；
>
> 多master多slave，同步复制
>
> 优点：与异步复制相比，同步复制的时候每次master节点写入一条数据该数据会实时同步至slave节点，这样的话即使master节点宕机也不会有数据未同步至slave节点；
>
> 缺点：可靠性的提升必然会导致性能的下降，同步复制的话由于要实时将数据同步至slave节点，所以会使得MQ集群的性能降低；

RocketMQ为什么速度快吗?

> 1. 我们在写入commitlog的时候是顺序写入的，这样比随机写入的性能就会提高很多
> 2. 写入commitlog的时候并不是直接写入磁盘，而是先写入操作系统的PageCache
> 3. 最后由操作系统异步将缓存中的数据刷到磁盘

RocketMQ事务消息?

> 
>
>
> 1. 生产者先发送一条半事务消息到MQ
> 2. MQ收到消息后返回ack确认
> 3. 生产者开始执行本地事务
> 4. 如果事务执行成功发送commit到MQ，失败发送rollback
> 5. 如果MQ长时间未收到生产者的二次确认commit或者rollback，MQ对生产者发起消息回查
> 6. 生产者查询事务执行最终状态
> 7. 根据查询事务状态再次提交二次确认

https://www.cnblogs.com/qdhxhz/p/11094624.html

https://zhuanlan.zhihu.com/p/260351128

> RocketMQ拉模式

推模式

推模式指的是消息从 Broker 推向 Consumer，即 Consumer 被动的接收消息，由 Broker 来主导消息的发送。

我们来想一下推模式有什么好处？

消息实时性高，Broker 接受完消息之后可以立马推送给 Consumer。

对于消费者使用来说更简单，简单啊就等着，反正有消息来了就会推过来。

推模式有什么缺点？

推送速率难以适应消费速率，推模式的目标就是以最快的速度推送消息，当生产者往 Broker 发送消息的速率大于消费者消费消息的速率时，随着时间的增长消费者那边可能就“爆仓”了，因为根本消费不过来啊。当推送速率过快就像 DDos 攻击一样消费者就傻了。

并且不同的消费者的消费速率还不一样，身为 Broker 很难平衡每个消费者的推送速率，如果要实现自适应的推送速率那就需要在推送的时候消费者告诉 Broker，我不行了你推慢点吧，然后 Broker 需要维护每个消费者的状态进行推送速率的变更。

这其实就增加了 Broker 自身的复杂度。

所以说推模式难以根据消费者的状态控制推送速率，适用于消息量不大、消费能力强要求实时性高的情况下。

拉模式

拉模式指的是 Consumer 主动向 Broker 请求拉取消息，即 Broker 被动的发送消息给 Consumer。

我们来想一下拉模式有什么好处？

拉模式主动权就在消费者身上了，**消费者可以根据自身的情况来发起拉取消息的请求**。假设当前消费者觉得自己消费不过来了，它可以根据一定的策略停止拉取，或者间隔拉取都行。

拉模式下 Broker 就相对轻松了，它只管存生产者发来的消息，至于消费的时候自然由消费者主动发起，来一个请求就给它消息呗，从哪开始拿消息，拿多少消费者都告诉它，它就是一个没有感情的工具人，消费者要是没来取也不关它的事。

拉模式可以更合适的进行消息的批量发送，基于推模式可以来一个消息就推送，也可以缓存一些消息之后再推送，但是推送的时候其实不知道消费者到底能不能一次性处理这么多消息。而拉模式就更加合理，它可以参考消费者请求的信息来决定缓存多少消息之后批量发送。

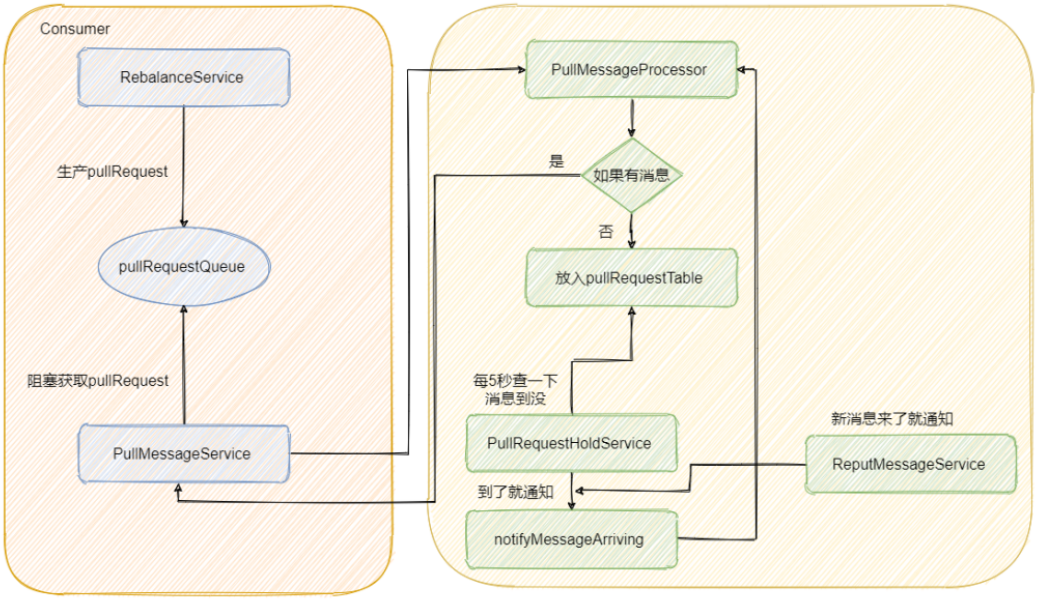
拉模式有什么缺点？

消息延迟，毕竟是消费者去拉取消息，但是消费者怎么知道消息到了呢？所以它只能不断地拉取，但是又不能很频繁地请求，太频繁了就变成消费者在攻击 Broker 了。因此需要降低请求的频率，比如隔个 2 秒请求一次，你看着消息就很有可能延迟 2 秒了。

消息忙请求，忙请求就是比如消息隔了几个小时才有，那么在几个小时之内消费者的请求

RocketMQ选择了拉模式

RocketMQ 和 Kafka 都是利用“长轮询”来实现拉模式



Rocketmq在consumer端是拉取消息的方式，它会在客户端维护一个PullRequestQueue，这个是一个阻塞队列（LinkedBlockingQueue），内部的节点是PullRequest，每一个PullRequest代表了一个消费的分组单元

Push模式大体实现思路是这样的：当消费者发送请求到Broker拉取消息的时候，如果有新的消息可以消费，会立马返回消息到消费者进行消费，消费后会接着发送请求到Broker拉取消息。

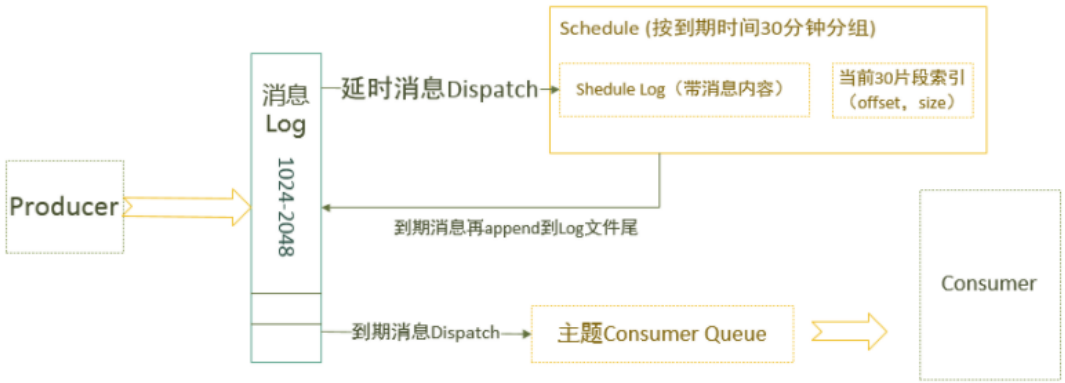
也就说Push模式下，处理完一批消息后会理解再发送请求给Broker拉取下一批消息，所以时效性更好，看起来就像是Broker在实时推送消息。

当请求发送到Broker发现没有需要消费的消息时，就会让请求线程挂起，默认挂起15秒，然后会有另一个后台线程每隔一段时间判断一下是否有新消息需要消费，一旦发现了新的消息，就会去唤醒挂起的线程，将消息返回给消费者进行消费，然后消费完毕再次发送请求拉取消息。

消息先写入Log中，再dispatch到Delay log， 按照到期时间半小时分组，同时会在内存中建立当前半小时的索引

时钟每秒移动，根据内存索引，把到期时间<=当前时间的消息重新读出，append到Log中，变成正常的消息，正常投递

比如201903251600XXXXX 文件都是16:00到16:30 这半个小时的到期的消息



- **延迟队列的核心思路**

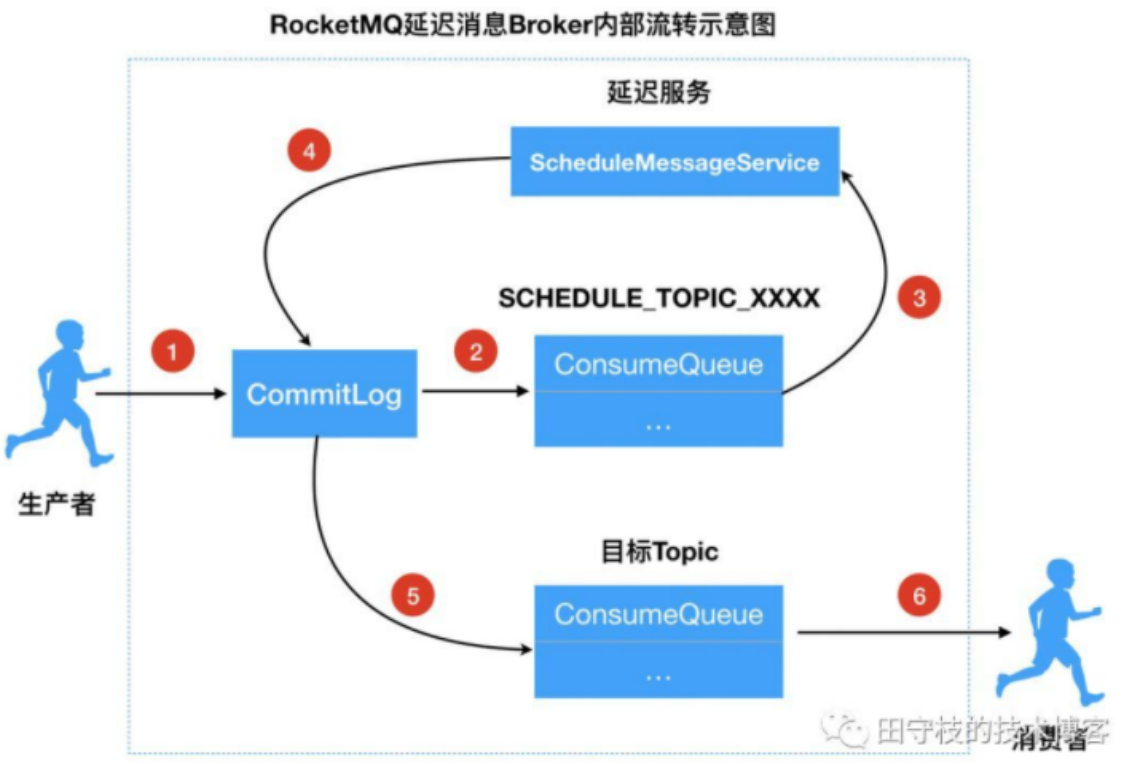
  所有的延迟消息由producer发出之后，都会存放到同一个topic（SCHEDULE_TOPIC_XXXX）下，不同的延迟级别会对应不同的队列序号，当延迟时间到之后，由定时线程读取转换为普通的消息存的真实指定的topic下，此时对于consumer端此消息才可见，从而被consumer消费。

- **延迟消息存放的结构**

```
consumequeue
├── SCHEDULE_TOPIC_XXXX
│   ├── 0
│   │   └── 00000000000000000000
│   ├── 1
│   │   └── 00000000000000000000
│   ├── 2
│   │   └── 00000000000000000000
│   ├── 3
│   │   └── 00000000000000000000
│   ├── 4
```

其中不同的延迟级别放在不同的队列序号下（queueId=delayLevel-1）。每一个延迟级别对应的延迟消息转换为普通消息的位置标识存放在~/store/config/delayOffset.json文件内。key为对应的延迟级别，value对应不同延迟级别转换为普通消息的offset值。

```
{
    "offsetTable":{1:1,2:1,3:11,4:1,5:1,6:1,7:1,8:1,9:1,10:1,11:1,12:1,13:1,14:1,15:0,16:0,17:
}
```



1. 修改消息Topic名称和队列信息
2. 转发消息到延迟主题的CosumeQueue中
3. 延迟服务消费SCHEDULE_TOPIC_XXXX消息
4. 将信息重新存储到CommitLog中
5. 将消息投递到目标Topic中
6. 消费者消费目标topic中的数据