想知道RocketMQ原理，看这一篇就够了

RocketMQ 是阿里巴巴在 2012 年开源的分布式消息中间件，经历过多次“双十一”万亿级数据洪峰的考验，它的性能、稳定性和可靠性都是值得信赖的。 本文主要介绍RocketMQ的实现原理，分了四大模块NameServer，Prodcuer，Consumer和Broker，难度系统如下表：

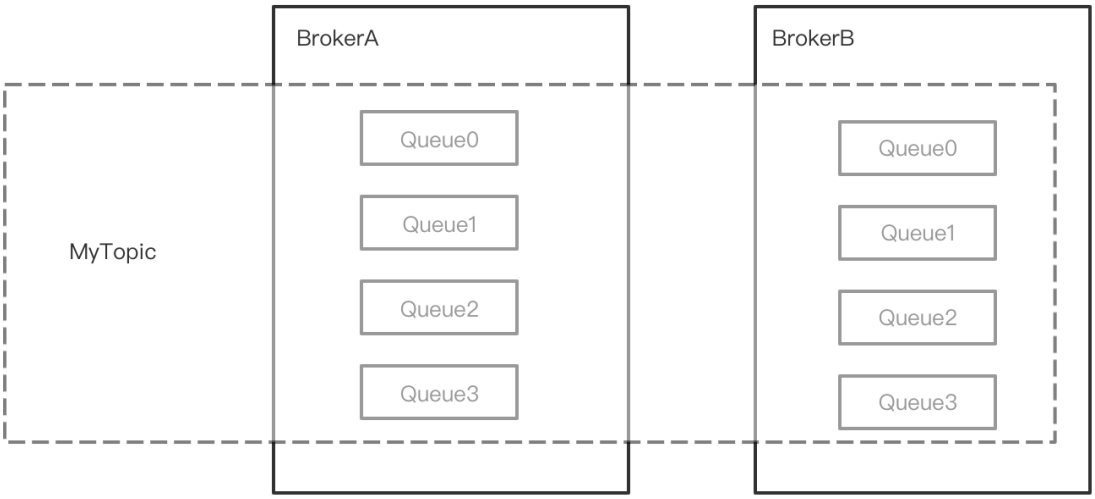
模块名称	难度系统
Nameserver	☆
prodcuer	☆☆
consumer	☆☆☆
Broker	☆☆☆☆

> 写文章也不易，你的一个赞就是对我最大的支持，也欢迎大家留言区讨论

为了增加阅读体验，本文尽量采用图文表达，源码采用附录的形式放在最后，希望通过本文，你能对RocketMQ的工作原理有整体的认识，但RocketMQ本身较为复杂，光看技术文章只能理解和领会一个大概，更多地还是需要自己多撸源码、Debug以及多实践才能对其有一个较为深入的理解。

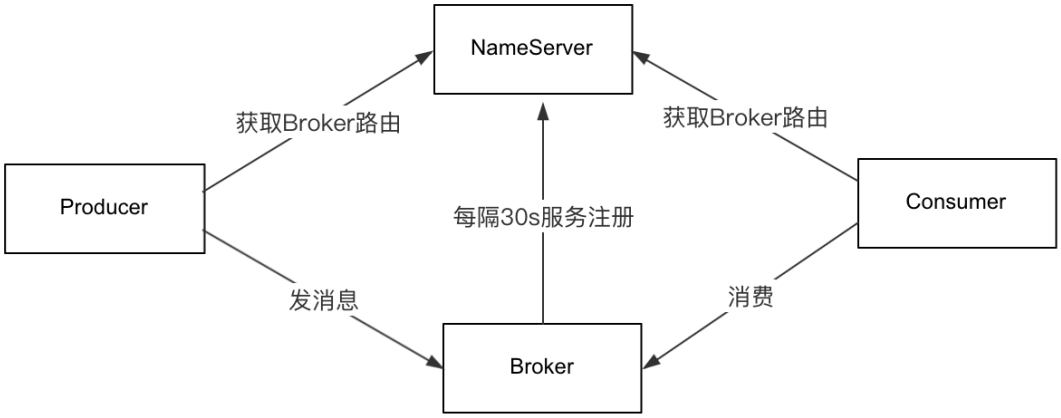
> 印第安人有一句谚语：I hear and I forget. I see and I remember. I do and I understand

我们先来对RocketMQ的名词有个初步对概念，然后我们带着这些概念，来开始四大模块的学习
Producer：消息生产者，作用就是发消息。
Consumer：消息消费者，作用就是消费生产者发的消息。
Consumer Group：消费者组，订阅了相同Topic的多个 Consumer 实例组成一个消费者组。
Topic：Topic 是一种消息的逻辑分类，是生产者在发送消息和消费者在拉取消息的类别。
Message：Message 就是我们发的消息，一个 Message 必须指定 Topic。
Name Server：为 Producer 和 Consumer 提供路由信息。
Broker：接收来自生产者的消息，储存以及为消费者拉取消息的请求做好准备。 Broker分为Master与Slave，一个Master可以对应多个Slave，但是一个Slave只能对应一个Master。
BrokerName和BrokerId：Master与Slave的对应关系通过指定相同的BrokerName，不同的BrokerId来定义，BrokerId为0表示Master，非0表示Slave。
Queue：是Topic在一个Broker上的分片等分为指定份数后的其中一份，是负载均衡过程中资源分配的基本单元。
下图描述了Topic，Broker和Queue的关系，MyTopic第一次分片，将消息存放在BrokerA和BrokerB中，Broke本身又对消息进行二次分片，存到了四个不同的Queue队列中。



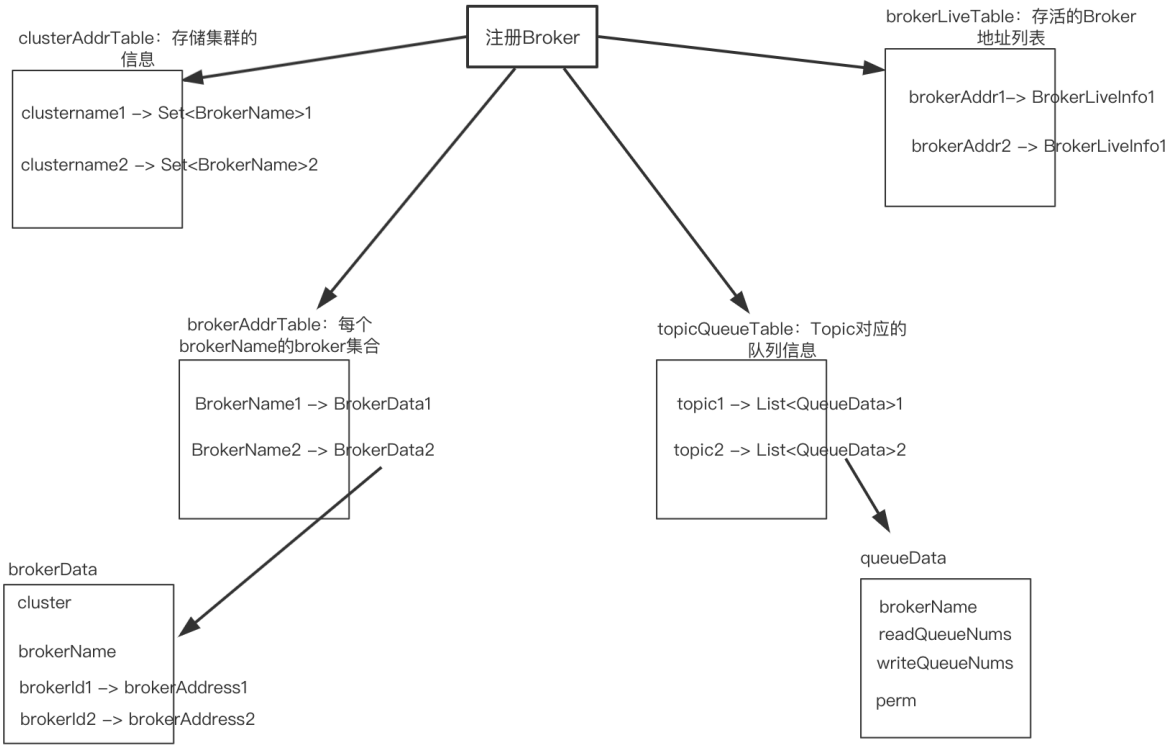
Nameserver协调者

Nameserver设计非常简单，是四大模块里最容易理解的一个，Nameserver的作用是注册中心，类似于Zookeeper，但又有区别于它的地方（PS：早期的RocketMQ版本用的就是ZK）。每个Nameserver节点互相之间是独立的，没有任何信息交互，也就不存在任何的选主或者主从切换之类的问题，因此Nameserver与Zookeeper相比更轻量级。单个NameServer节点中存储所有Broker列表（包括master和slave）并和Broker保持长链接心跳。下图为nameserver的整体工作流程，主要功能分为两个：注册发现和路由剔除。



注册发现

注册和发现主要是针对Broker，因为 NameServer节点互不通信，所以Broker在启动的时候会和NameServer所有节点建立长连接，每隔30s发送一次心跳，包含 BrokerId、Broker地址、Broker名称等Broker集群的信息，NameServer接收到心跳包后，会将整个消息集群的数据存入到RouteInfoManager对象，搞清楚RouteInfoManager，就搞清楚了NameServer的注册和发现。RouteInfoManager一共有4个table：topicQueueTable，brokerAddrTable，clusterAddrTable和brokerLiveTable，结构如下图，源码见附录1.1RouteInfoManager。



NameServer在处理心跳包的时候，存在多个Broker同时操作一张Broker表，为了防止并发修改Broker表导致不安全，路由注册操作引入了ReadWriteLock读写锁，这个设计亮点允许多个消息生产者并发读，保证了消息发送时的高并发，但是同一时刻NameServer只能处理一个Broker心跳包，多个心跳包串行处理。这也是读写锁的经典使用场景，即读多写少。

NameServer在路由注册或者路由剔除过程中，并不会主动推送会客户端的，需要由客户端拉取主题的最新路由信息，而是由定时拉取主题最新的路由。当客户端的拉取请求过来后，NameServer会从topicQueueTable，brokerAddrTable，clusterAddrTable中填充TopicRouteData中的List< QueueData >,List< BrokerData >，其中QueueData为topic对应的消息队列，BrokerData为可以发送的Broker地址。

路由剔除

RocketMQ有两个触发点来触发路由删除：
1）NameServer中有一个定时任务，每隔10秒扫描一下brokerLiveTable，如果某个Broker的心跳包最新时间戳距离当前时间超多120秒，也会判定Broker失效并将其移除。
2）正常情况下，如果Broker关闭，则会与NameServer断开长连接，Netty的通道关闭监听器会监听到连接断开事件，然后会将这个Broker信息剔除掉。
无论是哪种方式触发路由删除，最终都是关闭与Broker的长连接，同时更新topicQueueTable，brokerAddrTable，clusterAddrTable。

Producer生产者

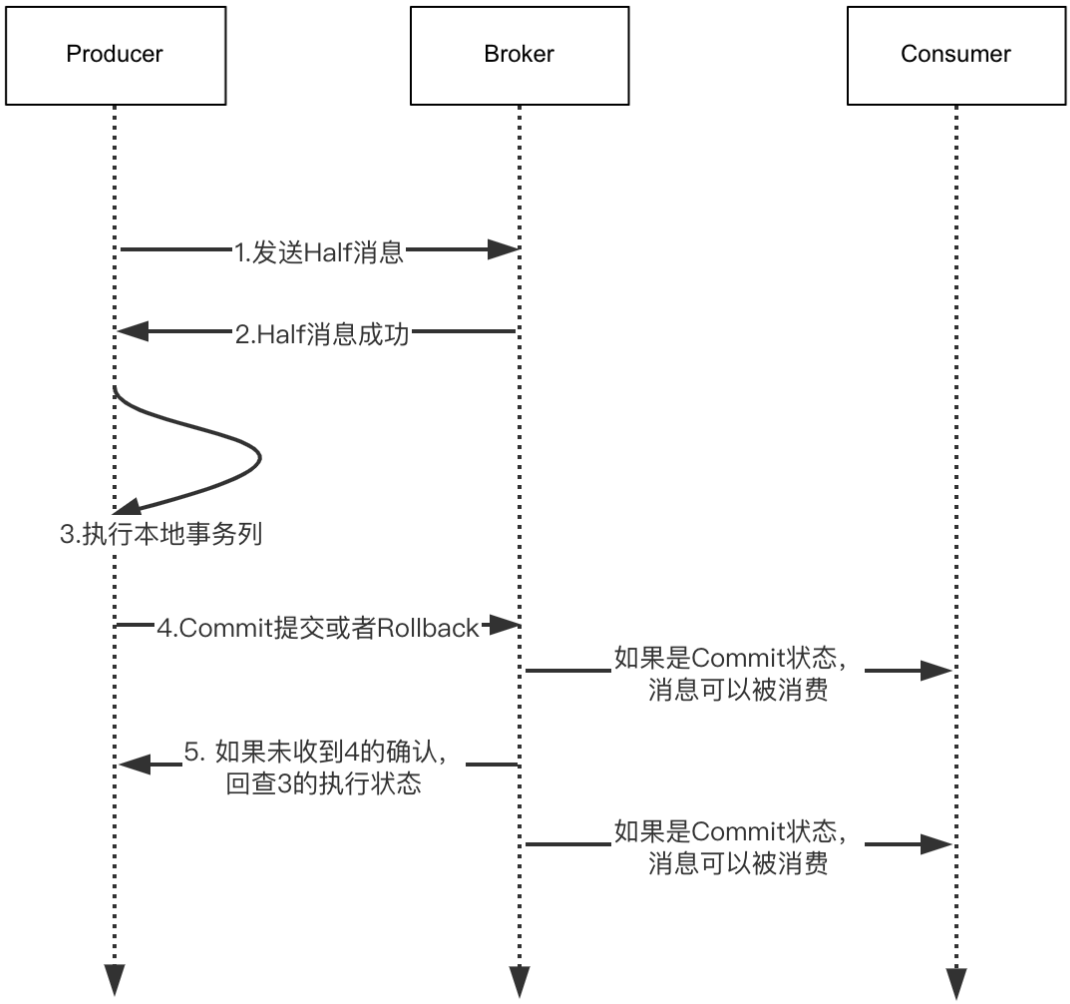
Producer在四大模块里面也比较好理解，核心作用就是发消息，本节先介绍一下Producer与NameServer、Broker的关系，以及支持发送的消息类型和发送方式，最后介绍一下发送过程。

	连接	轮询时间	心跳
Producer与NameServer	单个生产者者和一台nameserver保持长连接，定时查询topic配置信息，如果该NameServer挂掉，生产者会自动连接下一个nameserver	生产者每隔30秒从nameserver获取所有topic的最新队列情况	与nameserver没有心跳
Producer与Broker	单个生产者和该生产者关联的所有Broker保持长连接。	生产者每隔30秒向所有Broker发送心跳	没有轮询

消息类型

支持四种消息类型

- 普通消息：没有什么特殊的地方，就是普通消息
- 延迟消息：延时消息在投递时，需要设置指定的延时级别，即等到特定的时间间隔后消息才会被消费者消费。mq服务端 ScheduleMessageService中，为每一个延迟级别单独设置一个定时器，定时(每隔1秒)拉取对应延迟级别的消费队列。目前RocketMQ不支持任意时间间隔的延时消息，只支持特定级别的延时消息，即 "1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m 30m 1h 2h"
- 顺序消息：对于指定的一个 Topic，Producer保证消息顺序的发到一个队列中，消费的时候需要保证队列的数据只有一个线程消费。
- 事务消息：通过两阶段提交、状态定时回查来保证消息一定发到broker。具体流程见下图



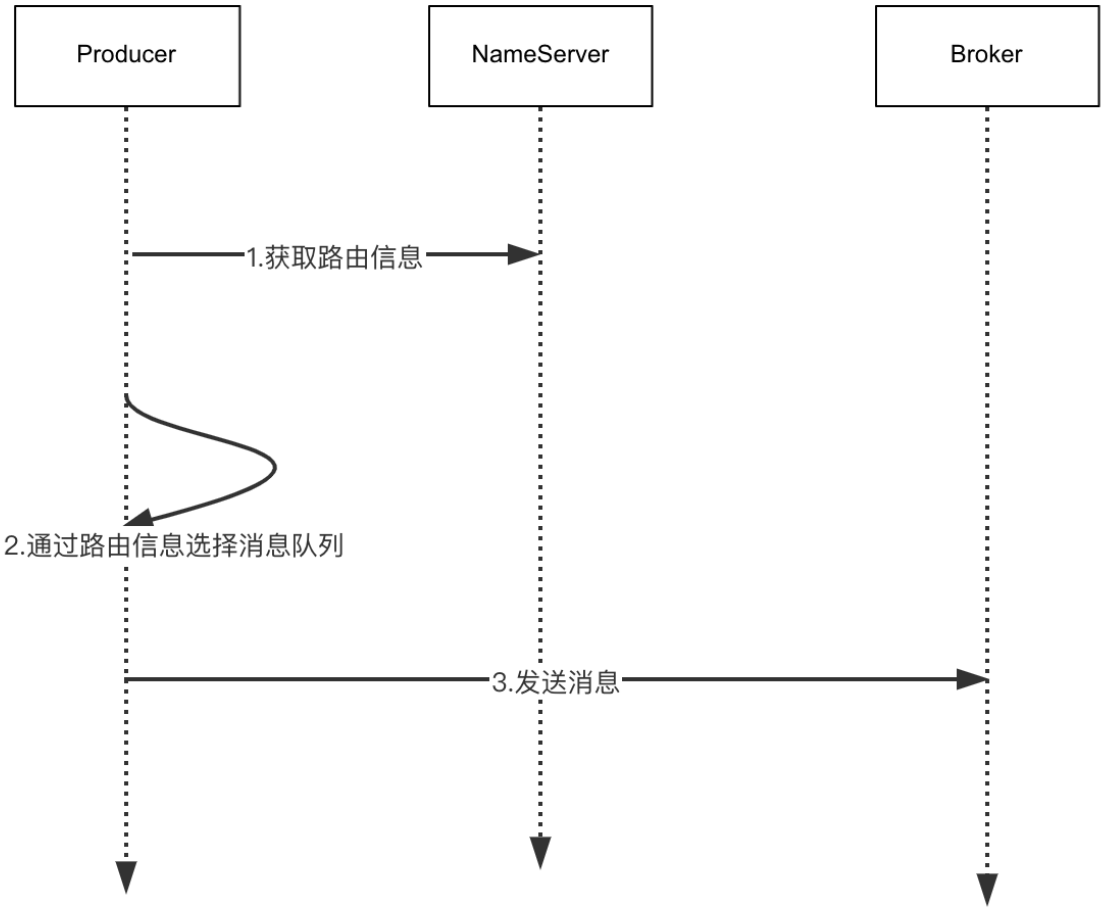
发送方式

RocketMQ支持三种消息发送方式

- 可靠同步发送：同步发送是指消息发送方发出数据后，会在收到接收方发回响应之后才发下一个数据包的通讯方式。
- 可靠异步发送：异步发送是指发送方发出数据后，不等接收方发回响应，接着发送下个数据包的通讯方式。 MQ 的异步发送，需要用户实现异步发送回调接口（SendCallback）。消息发送方在发送了一条消息后，不需要等待服务器响应即可返回，进行第二条消息发送。发送方通过回调接口接收服务器响应，并对响应结果进行处理。
- 单向（Oneway）发送：特点为发送方只负责发送消息，不等待服务器回应且没有回调函数触发，即只发送请求不等待应答。 此方式发送消息的过程耗时非常短，一般在微秒级别。

发送流程

我们再来说一下普通消息的具体发送流程，整体流程见下图



1）Producer在发送消息的时候，如果本地路由表中未缓存Topic的路由信息，则调用DefaultMQProducerImpl.tryToFindTopicPublishInfo方法向NameServer发送获取路由的请求，更新本地路由表，并每隔30s从NameServer更新本地路由表。源码见附录2.1和2.2
2）前面我们有介绍过NameServer返回的路由信息是TopicRouteData，里面包含List< QueueData >,List< BrokerData >，Producer在拿到路由信息后，轮询topic下的所有的QueueData，再根据BrokerName找到BrokerData信息，完成路由查询。源码见附录2.3
3）消息发送。将消息发送到指定到Broker节点上。

生产环境下为什么不能自动创建Topic?
autoCreateTopicEnable该配置用于在Topic不存在时自动创建，会造成的问题是自动新建的Topic只会存在于一台Broker上，后续所有对该Topic的请求都会局限在单台Broker上，造成单点压力。
所以生产环境需要用命令行工具手动创建Topic，可以用集群模式去创建（这样集群里面每个broker的queue的数量相同），也可以用单个broker模式去创建（这样每个broker的queue数量可以不一致）。
另外我们需要注意Consumer数量要小于等于queue的总数量，这在consumer的负载均衡那一节会详细介绍。

Consumer消费者

Consumer理解起来稍微复杂一些，他的核心作用就是消费消息。RocketMQ 是基于发布订阅模型的消息中间件。所谓的发布订阅就是说，Consumer 订阅了 某个 Topic，当 Producer 发该 Topic的消息时，Consumer 就能收到该条消息。

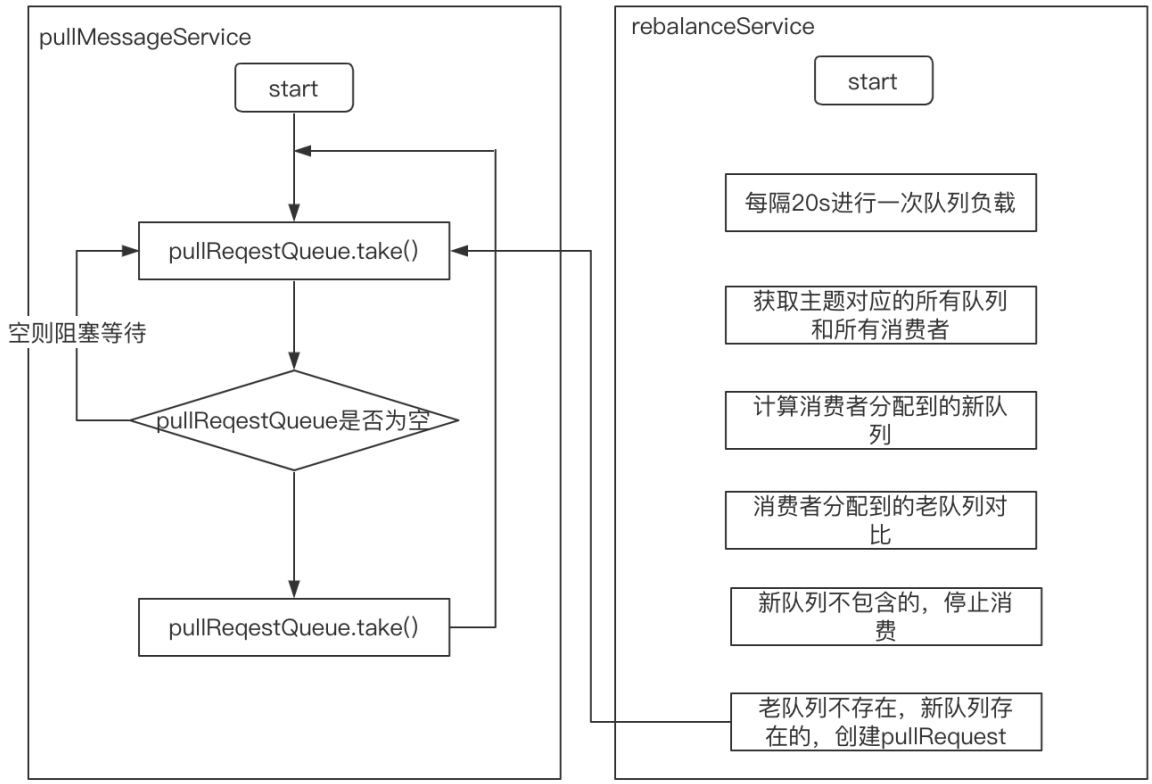
RocketMQ有两种消费模式：BROADCASTING广播模式，CLUSTERING集群模式，默认的是 集群消费模式。
广播消费指的是：一条消息被多个Consumer消费，即使这些Consumer属于同一个ConsumerGroup
集群消费模式：一条消息只能被ConsumerGroup里面的一个Consumer消费。

	连接	轮询时间	心跳
Consumer与NameServer	单个消费者和一台nameserver保持长连接，定时查询topic配置信息，如果该nameserver挂掉，消费者会自动连接下一个nameserver	消费者每隔30秒从nameserver获取所有topic的最新队列情况	与nameserver没有心跳
Consumer与Broker	单个Consumer和该生产者关联的所有Broker保持长连接。	无	Consumer每隔30秒向所有Broker发送心跳，一旦连接断开，Broker会立即感知到，并向该消费者分组的所有消费者发出通知，分组内消费者重新分配队列继续消费

拉取流程

对于任何一款消息中间件而言，消费者客户端一般有两种方式从消息中间件获取消息并消费：
（1）Push方式：由消息中间件主动地将消息推送给消费者；采用Push方式，实时性高，可以尽快的将消息发送给消费者进行消费。但是在消费者的处理消息的能力较弱的时候，而MQ不断地向消费者Push消息，消费者端的缓冲区可能会溢出，导致异常。
（2）Pull方式：由消费者客户端主动向消息中间件拉取消息；采用Pull方式，可控性好，如何设置Pull消息的频率需要重点去考虑。如果每次Pull的时间间隔比较久，会增加消息的延迟，若每次Pull的时间间隔较短，但是在一段时间内MQ中并没有任何消息可以消费，那么会产生很多无效的Pull请求的RPC开销，影响MQ整体的网络性能。

RocketMQ的消费方式都是基于拉模式拉取消息的，而且还采用了一种长轮询机制，来平衡上面Push/Pull方式的各自缺点。我们先来说一下拉取消息的流程，见下图：



消息的拉取主要依靠两个线程pullMessageService和rebalanceService。pullMessageService负责从broker拉取待消费的消息，rebalanceService负责定期调整consumer端负载均衡（在下一节会介绍），rebalanceService和pullMessageService相互配合使用，前者负责将新加入messageQueue拉取任务加入到pullRequestQueue阻塞队列中，后者负责从阻塞队列中获取任务并执行。pullMessageService和rebalanceService源码见附录3.1和3.2.

pullMessageService在拉取完消息之后，并不参与消息的消费，而是提交给消费线程后直接返回，拉取流程结束。我们主要介绍一下拉取流程里面的pullAPIWrapper.pullKernelImpl方法，这个方法是构造Broker请求参数，了解请求参数对我们理解consumer有很大到帮助， pullKernelImpld 的核心请求参数见下表：

MessageQueue mq	从哪个消息队列拉取消息
offset	消息拉取偏移量
maxNums	本次拉取最大消息条数，默认32条
brokerSuspendMaxTimeMillis	允许Broker挂起的时间，默认15s
timeoutMillis	消息拉取的超时时间
CommuniactionbMode	消息拉取模式，默认异步
PullCallback pullCallback	从Broker拉取到消息后的回调

Broker返回的数据：

PullStatus pullStatus	拉取结果
nextBeginOffset	下次拉取偏移量
minOffset	消息队列最小偏移量
maxOffset	消息队列最大偏移量
List msgFoundList	消息列表

从这俩个表我们就可以看出，Consumer会告诉Broker要拉取哪个队列，拉取偏移量和最大调数，Broker返回之后消息列表并告诉Consumer下次拉取的偏移量nextBeginOffset。

前面我们还说到，RocketMQ的消息消费方式采用了长轮询方式，兼具了Push和Pull的优点，即Consumer发送pull请求，Broker端接受请求，如果发现队列里没有新消息，不立即返回，而是持有这个请求一段时间（通过设置超时时间来实现），在这段时间内轮询Broker队列内是否有新的消息，如果有新消息，就利用现有的连接返回消息给消费者；如果这段时间内没有新消息进入队列，则返回空。 这样消费消息的主动权既保留在Consumer端，也不会出现Broker积压大量消息后，短时间内推送给Consumer大量消息使Consumer因为性能问题出现消费不及时的情况。

消息负载

消息负载是由rebalanceService线程实现的，每隔20s进行一次负载均衡。
1）从缓存中获取topic对应的消息队列mqSet和所有消费者cidAll。
2）按照负载均衡策略分配消息队列，常有的策略有：
AllocateMessageQueueAveragely平均算法，默认策略。
AllocateMessageQueueAveragelyByCircle：环形平均算法。
AllocateMessageQueueByConfig：根据配置负载均衡算法，根据配置，为每一个消费者配置固定的消息队列。
AllocateMessageQueueByMachineRoom：根据机房负载均衡算法。
AllocateMessageQueueConsistentHash：一致性哈希负载均衡算法。

举个栗子，假设队列大小是8（编号0-7），消费者数量3（编号0-2），分配结果就是AllocateMessageQueueAveragely的结果：
消费者0：队列0，1，2；
消费者1：队列3，4，5；
消费者2：队列6，7。

AllocateMessageQueueAveragelyByCircle的结果：
消费者0：队列0，3，6；
消费者1：队列1，4，7；
消费者2：队列2，5。

3）和原来旧的队列比较，如果旧的队列被删除，则停止旧队列的消息，如果是新增队列，创建PullRequest，扔到阻塞队列里面。

新创建的PullRequest，从哪里开始消费呢?
消费进度保存在Broker端，从Broker获取ConsumerGroup的消费offset，如果offset>0，继续消费即可，如果offset返回-1，则表示第一次消费，根据ConsumeFromWhere这个参数的配置计算从哪开始消费，参数配置见下表：

参数	描述
CONSUME_FROM_LAST_OFFSET	第一次启动从队列最后位置消费
CONSUME_FROM_FIRST_OFFSET	第一次启动从队列初始位置消费
CONSUME_FROM_TIMESTAMP	第一次启动从指定时间点位置消费

无论那种负载均衡的算法，我们可以看出，Consumer 数量要小于等于queue的总数量，由于Topic下的queue会被相对均匀的分配给Consumer，如果 Consumer 超过queue的数量，那多余的 Consumer 将没有queue可以消费消息。

消息消费

拉取消息的时候，pullmessage方法会将从Broker拉取的消息存入到ProcessQueue对象，提交给消费线程，消费线程会监听业务消费的结果，然后将结果返回给Broker，共有两种状态：
ConsumeConcurrentlyStatus, //消费成功
RECONSUME_LATER; //消费失败，一段时间后重试。

如果Consumer端返回ConsumeConcurrentlyStatus的时候，Broker更新对应的offset。
如果Consumer端消费返回RECONSUME_LATER或者在超时时间内没有返回给Broker消费状态，那么Broker会自动重试。重试次数和重试时间间隔可以在broker.conf文件中配置，默认"1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m 30m 1h 2h"

Broker消息存储

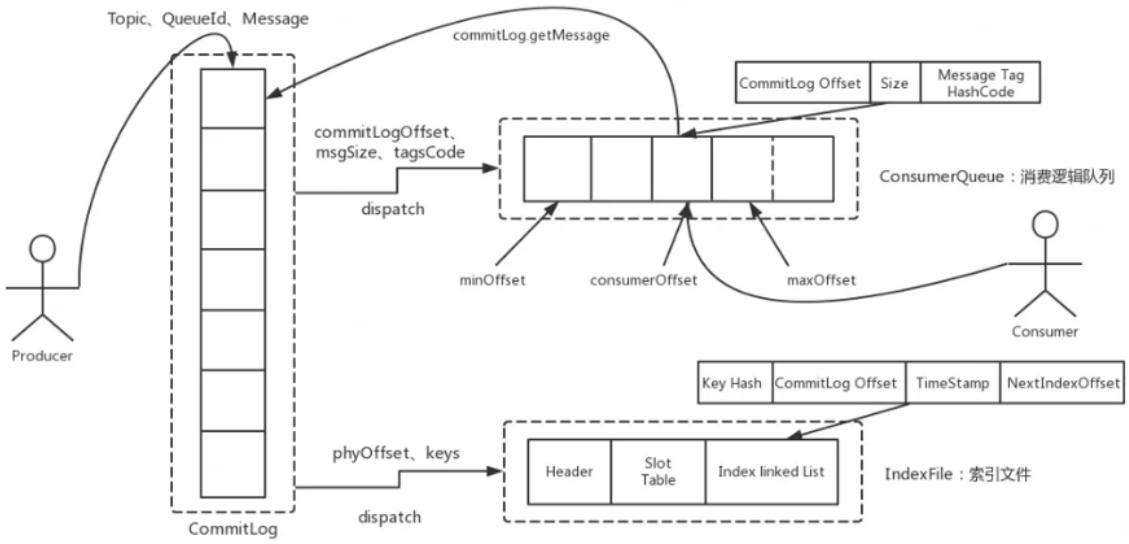
Broker的作用是存储MQ消息并持久化，RocketMQ作为一款高性能的中间件，存储设计是核心，存储设计的核心就是IO交互了。Broker是四个模块里最复杂，最不好理解的一个模块。

当前业界几款主流的MQ消息队列采用的存储方式主要有以下三种方式：
（1）分布式KV存储：这类MQ一般会采用诸如levelDB、RocksDB和Redis来作为消息持久化的方式，由于分布式缓存的读写能力要优于DB，所以在对消息的读写能力要求都不是比较高的情况下，采用这种方式倒也不失为一种可以替代的设计方案。
（2）文件系统：目前业界较为常用的几款产品（RocketMQ/Kafka/RabbitMQ）均采用的是消息刷盘至所部署虚拟机/物理机的文件系统来做持久化（刷盘一般可以分为异步刷盘和同步刷盘两种模式）。
（3）关系型数据库DB：Apache下开源的另外一款MQ—ActiveMQ（默认采用的KahaDB做消息存储）可选用JDBC的方式来做消息持久化，通过简单的xml配置信息即可实现JDBC消息存储。
从存储效率来说， 文件系统>分布式KV存储>关系型数据库DB，直接操作文件系统肯定是最快和最高效的，但是如果从可靠性来讲的话，关系型数据库DB>分布式KV存储>文件系统。

文件结构

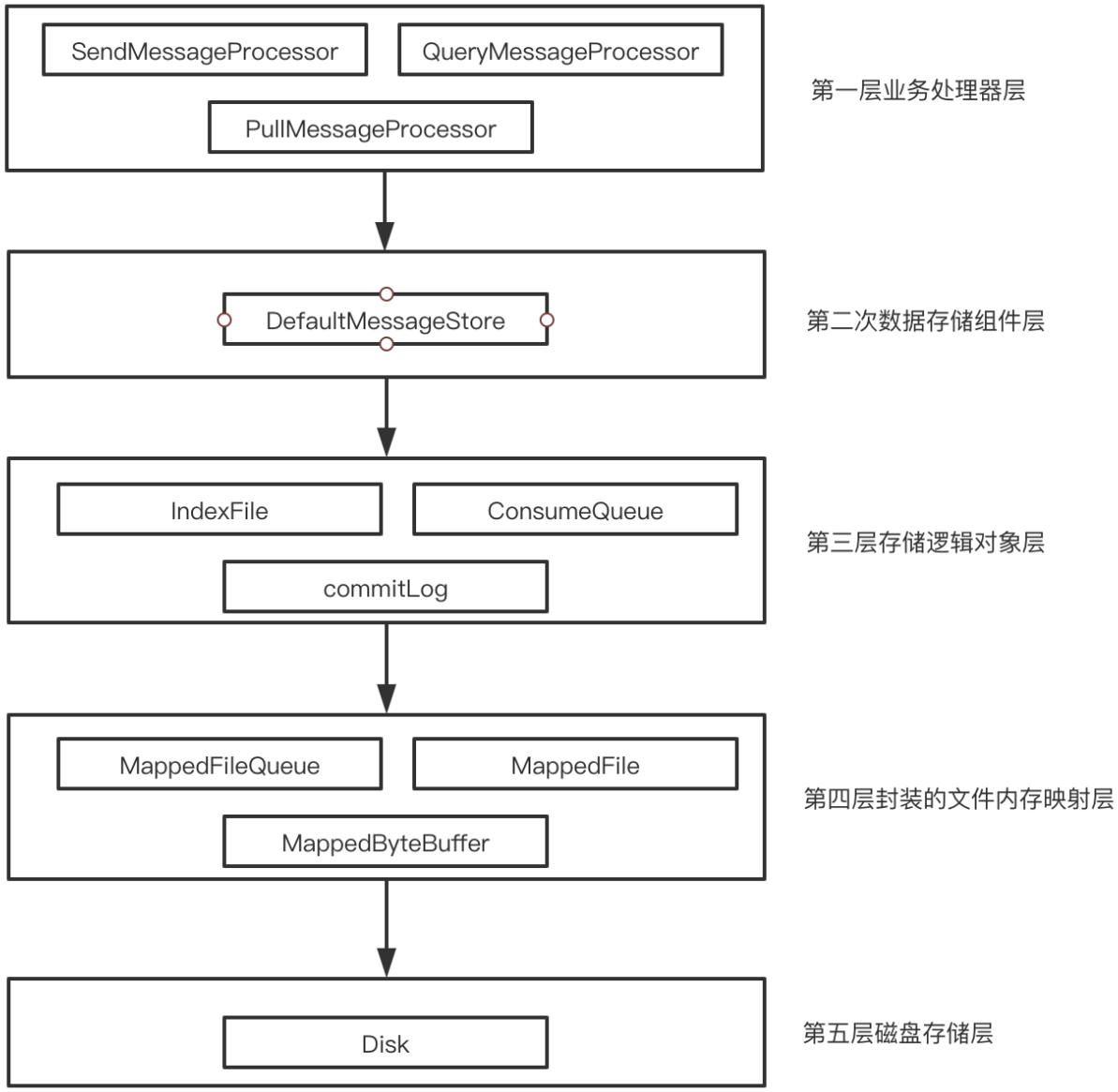
RocketMQ采用的是文件存储，如下图所示（图片来自网络）：

- CommitLog是消息存储文件，所有消息主题的消息都存储在CommitLog文件中；
- ConsumeQueue是消息消费队列文件，消息达到commitlog文件后将被异步转发到消息消费队列，供消息消费者消费；
- IndexFile是消息索引文件，主要存储的是key和offset的对应关系。



存储模型

文件存储模型层次结构如下图所示，根据类别和作用从概念模型上大致可以划分为5层



（1）RocketMQ业务处理器层：Broker端对消息进行读取和写入的业务逻辑入口，这一层主要包含了业务逻辑相关处理操作，比如前置的检查和校验步骤、decode反序列化、构造Response返回对象等，读写操作是通过DefaultMessageStore提供的API来实现。

（2）RocketMQ数据存储组件层：该层是RocketMQ的存储核心类—DefaultMessageStore，其为RocketMQ消息数据文件的访问入口，业务层可以调用DefaultMessageStore的方法完成对文件的读写，底层实现不需要关心。比如通过该类的putMessage()和getMessage()方法完成对CommitLog文件的读写操作，另外，在该组件初始化时候，还会启动很多存储相关的后台服务线程。

（3）RocketMQ存储逻辑对象层：该层主要包含了RocketMQ数据文件存储直接相关的三个模型类IndexFile、ConsumerQueue和CommitLog。IndexFile为索引数据文件提供访问服务，ConsumerQueue为逻辑消息队列提供访问服务，CommitLog则为消息存储的日志数据文件提供访问服务。

（4）封装的文件内存映射层：RocketMQ主要采用JDK NIO中的MappedByteBuffer和FileChannel两种方式完成数据文件的读写。其中，采用MappedByteBuffer这种内存映射磁盘文件的方式完成对大文件的读写，在RocketMQ中将该类封装成MappedFile类。

（5）磁盘存储层：主要指的是部署RocketMQ服务器所用的磁盘，存储了CommitLog，ConsumeQueue，IndexFile等文件。

存储流程

我们按照这些层级，来介绍一下消息存储的流程 SendMessageProcessor.asyncSendMessage->DefaultMessageStore.putMessage->commitLog.putMessage->mappedFile.appendMessage

1）Producer端将消息发送到Broker端后，先进入了第一层业务处理层的SendMessageProcessor.asyncSendMessage方法，这个方法做了一些前置的检查和校验步骤，调用了DefaultMessageStore.putMessage方法进入了第二层。 2）第二层数据存储层主要做了一些校验和失败监控，然后调用第三层逻辑对象层的commitLog.putMessage方法。
3）第三层逻辑对象层主要是调用第四层mappedFileQueue和MappedFile的api，完成commitLog的写入，这里简单介绍一下写入流程：

- 先申请putMessageLock.lock()，也就是消息存储到commitLog是串行的。
- 获取mappedFileQueue最后一个mappedFile，如果为null，则说明是第一次发消息，用偏移量为0创建第一个commit文件。
- 调用 mappedFile.appendMessage将消息追加到mappedFile，然后根据返回值构造putMessageResult。

4）第四层在文件映射内存层，为第三层提供了两个api，mappedFileQueue.getLastMappedFile，mappedFile.appendMessage方法，appendMessage只是将消息写进了内存，MappedFile会调用flush方法，将数据刷写到磁盘，永久存储。这里我不在详细介绍appendMessage的实现原理，具体可以看代码附录4.2。

RocketMQ之所以称为高性能，除了有合理的存储结构，IO的读写效率也非常关键。MappedFile类提供了读写的操作服务，号称读写文件的速度接近内存，这个是怎么实现的呢？ 这里面要说到的就是java nio中引入了一种基于MappedByteBuffer操作大文件的方式，其读写性能极高，因为这块并不属于MQ的内容，这里就简单介绍一下MappedByteBuffer，想要深入了解的同学，可以自己去研究一下jdk源码。首先我们需要先知道一些术语。

MMC：CPU的内存管理单元。
物理内存：即内存条的内存空间。
虚拟内存：计算机系统内存管理的一种技术。它使得应用程序认为它拥有连续的可用的内存（一个连续完整的地址空间），而实际上，它通常是被分隔成多个物理内存碎片，还有部分暂时存储在外部磁盘存储器上，在需要时进行数据交换。
页面文件：操作系统反映构建并使用虚拟内存的硬盘空间大小而创建的文件。
缺页中断：当程序试图访问已映射在虚拟地址空间中但未被加载至物理内存的一个分页时，由MMC发出的中断。如果操作系统判断此次访问是有效的，则尝试将相关的页从虚拟内存文件中载入物理内存。

当调用MappedByteBuffer的get方法时，会返回一个地址address,通过address就能够操作文件。第一次访问address所指向的内存区域，导致缺页中断，中断响应函数会在交换区中查找相对应的页面，如果找不到（也就是该文件从来没有被读入内存的情况），则从硬盘上将文件指定页读取到物理内存中（非jvm堆内存）。 如果在拷贝数据时，发现物理内存不够用，则会通过虚拟内存机制（swap）将暂时不用的物理页面交换到硬盘的虚拟内存中。对于数据文件的读取，如果一次读取文件时出现未命中PageCache的情况，OS从物理磁盘上访问读取文件的同时，会顺序对其他相邻块的数据文件进行预读取（ps：顺序读入紧随其后的少数几个页面）。这样，只要下次访问的文件已经被加载至PageCache时，读取操作的速度基本等于访问内存。
另外当缺页中断发生时，将文件从磁盘拷贝至用户态的进程空间内，传统的IO要经历至少三次数据拷贝才可以把数据读出来，而MappedByteBuffer只需要一次数据拷贝，效率和性能非常高。

第四层文件映射内存层的MappedFile正是用了MappedByteBuffer的完成了对文件的封装，使RocketMQ可以高效的读写文件。

源码附录

1. Nameserver

1. RouteInfoManager org.apache.rocketmq.namesrv.routeinfo.RouteInfoManager

```
public class RouteInfoManager {
   private static final InternalLogger log =
InternalLoggerFactory.getLogger(LoggerName.NAMESRV_LOGGER_NAME);
   private final static long BROKER_CHANNEL_EXPIRED_TIME = 1000 * 60 * 2;
   private final ReadWriteLock lock = new ReentrantReadWriteLock();
   private final HashMap<String/* topic */, List<QueueData>>
topicQueueTable;
   private final HashMap<String/* brokerName */, BrokerData>
brokerAddrTable;
   private final HashMap<String/* clusterName */, Set<String/* brokerName
*/>> clusterAddrTable;
   private final HashMap<String/* brokerAddr */, BrokerLiveInfo>
brokerLiveTable;
   private final HashMap<String/* brokerAddr */, List<String>/* Filter
Server */> filterServerTable;

   public RouteInfoManager() {
       this.topicQueueTable = new HashMap<String, List<QueueData>>(1024);
       this.brokerAddrTable = new HashMap<String, BrokerData>(128);
       this.clusterAddrTable = new HashMap<String, Set<String>>(32);
       this.brokerLiveTable = new HashMap<String, BrokerLiveInfo>(256);
       this.filterServerTable = new HashMap<String, List<String>>(256);
   }
   ....
}
```

2. Producer

1. tryToFindTopicPublishInfo

   org.apache.rocketmq.client.impl.producer.DefaultMQProducerImpl#tryToFindTopicPublishInfo

```
private TopicPublishInfo tryToFindTopicPublishInfo(final String topic)
{
   TopicPublishInfo topicPublishInfo =
this.topicPublishInfoTable.get(topic);
   if (null == topicPublishInfo || !topicPublishInfo.ok()) {
```

```
        this.topicPublishInfoTable.putIfAbsent(topic, new TopicPublishInfo());
        this.mQClientFactory.updateTopicRouteInfoFromNameServer(topic);
        topicPublishInfo = this.topicPublishInfoTable.get(topic);
    }
    if (topicPublishInfo.isHaveTopicRouterInfo() || topicPublishInfo.ok()) {
        return topicPublishInfo;
    } else {
        this.mQClientFactory.updateTopicRouteInfoFromNameServer(topic, true, this.defaultMQProducer);
        topicPublishInfo = this.topicPublishInfoTable.get(topic);
        return topicPublishInfo;
    }
}
```

2. TopicPublishInfo

   org.apache.rocketmq.client.impl.producer.TopicPublishInfo

```
public class TopicPublishInfo {
  private boolean orderTopic = false;
  private boolean haveTopicRouterInfo = false;
  private List<MessageQueue> messageQueueList = new ArrayList<MessageQueue>();
  private volatile ThreadLocalIndex sendWhichQueue = new ThreadLocalIndex();
  private TopicRouteData topicRouteData;
...
}
```

3. topicRouteData2TopicPublishInfo

   org.apache.rocketmq.client.impl.factory.MQClientInstance#topicRouteData2TopicPublishInfo

```
public static TopicPublishInfo topicRouteData2TopicPublishInfo(final String topic, final TopicRouteData route) {
  TopicPublishInfo info = new TopicPublishInfo();
  info.setTopicRouteData(route);
  if (route.getOrderTopicConf() != null && route.getOrderTopicConf().length() > 0) {
      String[] brokers = route.getOrderTopicConf().split(";");
      for (String broker : brokers) {
          String[] item = broker.split(":");
          int nums = Integer.parseInt(item[1]);
          for (int i = 0; i < nums; i++) {
              MessageQueue mq = new MessageQueue(topic, item[0], i);
              info.getMessageQueueList().add(mq);
```

```
            }
        }
        info.setOrderTopic(true);
    } else {
        List<QueueData> qds = route.getQueueDatas();
        Collections.sort(qds);
        for (QueueData qd : qds) {
            if (PermName.isWriteable(qd.getPerm())) {
                BrokerData brokerData = null;
                for (BrokerData bd : route.getBrokerDatas()) {
                    if (bd.getBrokerName().equals(qd.getBrokerName())) {
                        brokerData = bd;
                        break;
                    }
                }
                if (null == brokerData) {
                    continue;
                }
                if
(!brokerData.getBrokerAddrs().containsKey(MixAll.MASTER_ID)) {
                    continue;
                }
                for (int i = 0; i < qd.getWriteQueueNums(); i++) {
                    MessageQueue mq = new MessageQueue(topic,
qd.getBrokerName(), i);
                    info.getMessageQueueList().add(mq);
                }
            }
        }
        info.setOrderTopic(false);
    }
    return info;
}
```

3. Consumer

1. PullMessageService org.apache.rocketmq.client.impl.consumer.PullMessageService

```
public class PullMessageService extends ServiceThread {
    private final InternalLogger log = ClientLogger.getLog();
    private final LinkedBlockingQueue<PullRequest> pullRequestQueue =
new LinkedBlockingQueue<PullRequest>();
    private final MQClientInstance mQClientFactory;
    private final ScheduledExecutorService scheduledExecutorService =
Executors
        .newSingleThreadScheduledExecutor(new ThreadFactory() {
            @Override
            public Thread newThread(Runnable r) {
```

```
9               return new Thread(r,
"PullMessageServiceScheduledThread");
10          }
11      });
12
13    public PullMessageService(MQClientInstance mQClientFactory) {
14        this.mQClientFactory = mQClientFactory;
15    }
16
17    public void executePullRequestLater(final PullRequest pullRequest,
final long timeDelay) {
18        if (!isStopped()) {
19            this.scheduledExecutorService.schedule(new Runnable() {
20                @Override
21                public void run() {
22
 PullMessageService.this.executePullRequestImmediately(pullRequest);
23                }
24            }, timeDelay, TimeUnit.MILLISECONDS);
25        } else {
26            log.warn("PullMessageServiceScheduledThread has shutdown");
27        }
28    }
29
30    public void executePullRequestImmediately(final PullRequest
pullRequest) {
31        try {
32            this.pullRequestQueue.put(pullRequest);
33        } catch (InterruptedException e) {
34            log.error("executePullRequestImmediately
pullRequestQueue.put", e);
35        }
36    }
37
38    public void executeTaskLater(final Runnable r, final long timeDelay)
{
39        if (!isStopped()) {
40            this.scheduledExecutorService.schedule(r, timeDelay,
TimeUnit.MILLISECONDS);
41        } else {
42            log.warn("PullMessageServiceScheduledThread has shutdown");
43        }
44    }
45
46    public ScheduledExecutorService getScheduledExecutorService() {
47        return scheduledExecutorService;
48    }
49
50    private void pullMessage(final PullRequest pullRequest) {
```

```
51        final MQConsumerInner consumer =
this.mQClientFactory.selectConsumer(pullRequest.getConsumerGroup());
52        if (consumer != null) {
53            DefaultMQPushConsumerImpl impl = (DefaultMQPushConsumerImpl)
consumer;
54            impl.pullMessage(pullRequest);
55        } else {
56            log.warn("No matched consumer for the PullRequest {}, drop
it", pullRequest);
57        }
58    }
59
60    @Override
61    public void run() {
62        log.info(this.getServiceName() + " service started");
63
64        while (!this.isStopped()) {
65            try {
66                PullRequest pullRequest = this.pullRequestQueue.take();
67                this.pullMessage(pullRequest);
68            } catch (InterruptedException ignored) {
69            } catch (Exception e) {
70                log.error("Pull Message Service Run Method exception",
e);
71            }
72        }
73
74        log.info(this.getServiceName() + " service end");
75    }
76
77    @Override
78    public void shutdown(boolean interrupt) {
79        super.shutdown(interrupt);
80        ThreadUtils.shutdownGracefully(this.scheduledExecutorService,
1000, TimeUnit.MILLISECONDS);
81    }
82
83    @Override
84    public String getServiceName() {
85        return PullMessageService.class.getSimpleName();
86    }
```

}

2 . RebalanceService
org.apache.rocketmq.client.impl.consumer.RebalanceService

```
1  public class RebalanceService extends ServiceThread {
2   private static long waitInterval =
```

```
        Long.parseLong(System.getProperty(
            "rocketmq.client.rebalance.waitInterval", "20000"));
    private final InternalLogger log = ClientLogger.getLog();
    private final MQClientInstance mqClientFactory;

    public RebalanceService(MQClientInstance mqClientFactory) {
        this.mqClientFactory = mqClientFactory;
    }

    @Override
    public void run() {
        log.info(this.getServiceName() + " service started");

        while (!this.isStopped()) {
            this.waitForRunning(waitInterval);
            this.mqClientFactory.doRebalance();
        }

        log.info(this.getServiceName() + " service end");
    }

    @Override
    public String getServiceName() {
        return RebalanceService.class.getSimpleName();
    }
}
```

4. Broker

1. DefaultMessageStore#putMessage
   org.apache.rocketmq.store.DefaultMessageStore#putMessage

```
public PutMessageResult putMessage(MessageExtBrokerInner msg) {
    PutMessageStatus checkStoreStatus = this.checkStoreStatus();
    if (checkStoreStatus != PutMessageStatus.PUT_OK) {
        return new PutMessageResult(checkStoreStatus, null);
    }

    PutMessageStatus msgCheckStatus = this.checkMessage(msg);
    if (msgCheckStatus == PutMessageStatus.MESSAGE_ILLEGAL) {
        return new PutMessageResult(msgCheckStatus, null);
    }

    long beginTime = this.getSystemClock().now();
    PutMessageResult result = this.commitLog.putMessage(msg);
    long elapsedTime = this.getSystemClock().now() - beginTime;
    if (elapsedTime > 500) {
```

```
16        log.warn("not in lock elapsed time(ms)={}, bodyLength={}",
elapsedTime, msg.getBody().length);
17    }
18
19    this.storeStatsService.setPutMessageEntireTimeMax(elapsedTime);
20
21    if (null == result || !result.isOk()) {
22
 this.storeStatsService.getPutMessageFailedTimes().incrementAndGet();
23    }
24
25    return result;
26  }
```

2. MappedFile#appendMessagesInner
org.apache.rocketmq.store.MappedFile#appendMessagesInner

```
1   public AppendMessageResult appendMessagesInner(final MessageExt
messageExt, final AppendMessageCallback cb) {
2     assert messageExt != null;
3     assert cb != null;
4
5     int currentPos = this.wrotePosition.get();
6
7     if (currentPos < this.fileSize) {
8         ByteBuffer byteBuffer = writeBuffer != null ?
writeBuffer.slice() : this.mappedByteBuffer.slice();
9         byteBuffer.position(currentPos);
10        AppendMessageResult result;
11        if (messageExt instanceof MessageExtBrokerInner) {
12            result = cb.doAppend(this.getFileFromOffset(), byteBuffer,
this.fileSize - currentPos, (MessageExtBrokerInner) messageExt);
13        } else if (messageExt instanceof MessageExtBatch) {
14            result = cb.doAppend(this.getFileFromOffset(), byteBuffer,
this.fileSize - currentPos, (MessageExtBatch) messageExt);
15        } else {
16            return new
AppendMessageResult(AppendMessageStatus.UNKNOWN_ERROR);
17        }
18        this.wrotePosition.addAndGet(result.getWroteBytes());
19        this.storeTimestamp = result.getStoreTimestamp();
20        return result;
21    }
22    log.error("MappedFile.appendMessage return null, wrotePosition: {}
fileSize: {}", currentPos, this.fileSize);
23    return new AppendMessageResult(AppendMessageStatus.UNKNOWN_ERROR);
24  }
25
```

参考资料

https://www.jianshu.com/p/42330afbe53a https://www.jianshu.com/p/bbaf72d160ca https://www.jianshu.com/p/b73fdd893f98 https://www.jianshu.com/p/6c2cb0d2bfcd https://www.jianshu.com/p/f071d5069059 https://www.jianshu.com/p/c717cb26752e https://segmentfault.com/q/1010000020877020/ https://www.cnblogs.com/xuwc/p/9034352.html https://www.cnblogs.com/wxd0108/p/6041829.html https://cloud.tencent.com/developer/article/1451157 https://mp.weixin.qq.com/s/mq7t2ocIBxt96haE82HpFQ https://blog.csdn.net/hosaos/article/details/100053480 https://blog.csdn.net/hosaos/java/article/details/100053480 https://blog.csdn.net/tales522/java/article/details/88085494