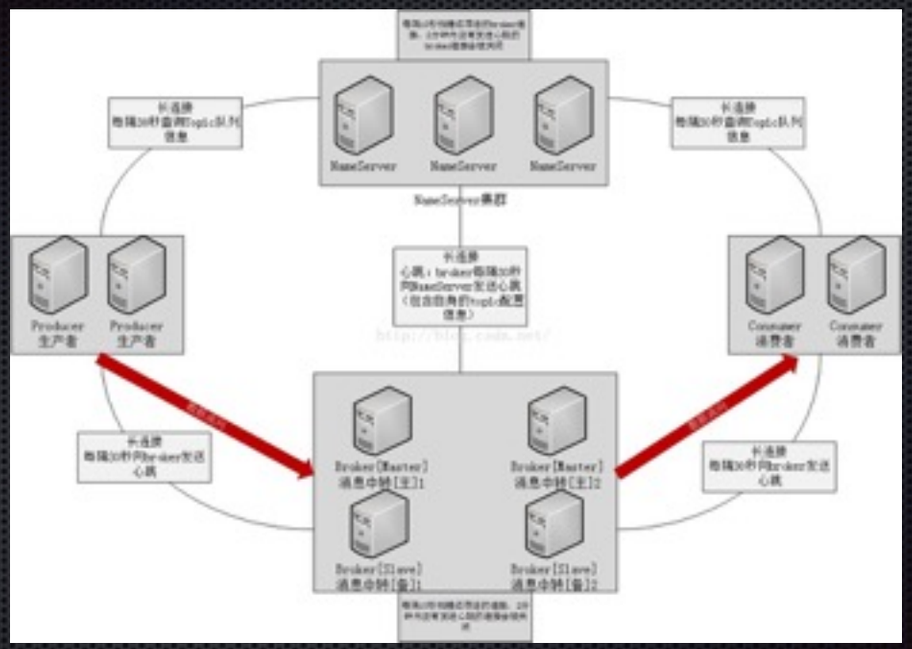
RocketMQ

yongping.ren@ygomi.com

什么是RocketMQ

- 开源的分布式消息和流数据平台。

- 低延迟(高压下，1毫秒内的响应延迟超过99.6%。)
- 高可用
- 万亿级消息容量保证
- 高吞吐量（批量传输）


NameServer
NameServer
NameServer
NameServer集群
Producer
生产者
Producer
生产者
Consumer
消费者
Consumer
消费者
Broker[Master]
消息中转[主]1
Broker[Master]
消息中转[主]2
Broker[Slave]
消息中转[备]1
Broker[Slave]
消息中转[备]2

参与者

- Producer（消息生产者）
- Consumer（消息消费者）
- Broker Server（代理服务器）
- Name Server（名字服务）

Producer

负责生产消息，一般由业务系统负责生产消息。一个消息生产者会把业务应用系统里产生的消息发送到broker服务器。RocketMQ提供多种发送方式，同步发送、异步发送、顺序发送、单向发送。同步和异步方式均需要Broker返回确认信息，单向发送不需要。

Consumer

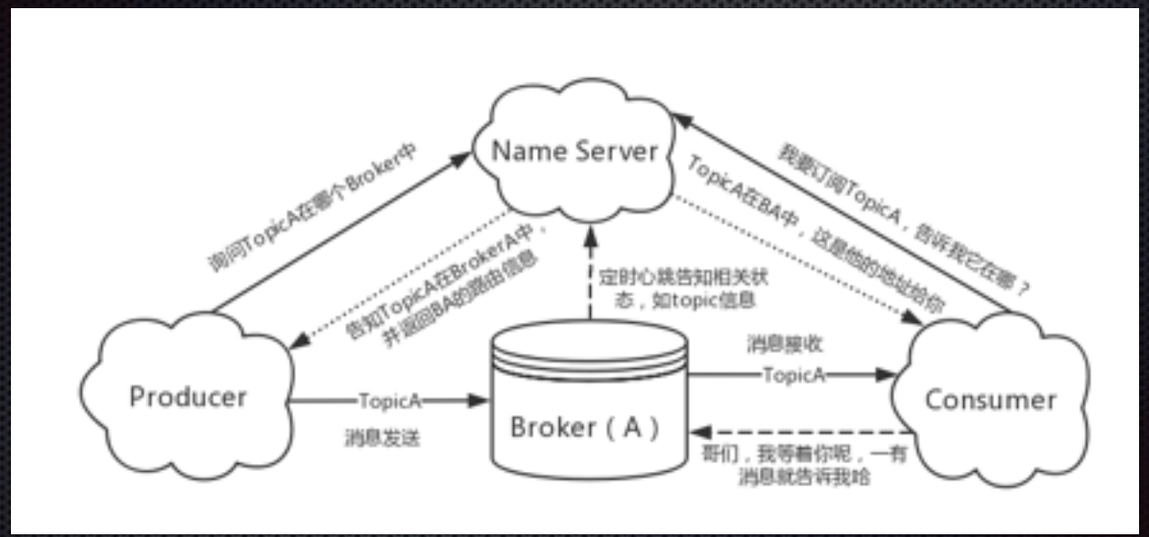
负责消费消息，一般是后台系统负责异步消费。一个消息消费者会从Broker服务器拉取消息、并将其提供给应用程序。从用户应用的角度而言提供了两种消费形式：拉取式消费、推动式消费。

Broker Server

消息中转角色，负责存储消息、转发消息。代理服务器在RocketMQ系统中负责接收从生产者发送来的消息并存储、同时为消费者的拉取请求作准备。代理服务器也存储消息相关的元数据，包括消费者组、消费进度偏移和主题和队列消息等。

Name Server

NameServer是一个非常简单的Topic路由注册中心，其角色类似Dubbo中的zookeeper，支持Broker的动态注册与发现。主要包括两个功能：Broker管理，NameServer接受Broker集群的注册信息并且保存下来作为路由信息的基本数据。然后提供心跳检测机制，检查Broker是否还存活；路由信息管理，每个NameServer将保存关于Broker集群的整个路由信息和用于客户端查询的队列信息。然后Producer和Conumser通过NameServer就可以知道整个Broker集群的路由信息，从而进行消息的投递和消费。NameServer通常也是集群的方式部署，各实例间相互不进行信息通讯。Broker是向每一台NameServer注册自己的路由信息，所以每一个NameServer实例上面都保存一份完整的路由信息。当某个NameServer因某种原因下线了，Broker仍然可以向其它NameServer同步其路由信息，Producer,Consumer仍然可以动态感知Broker的路由的信息。

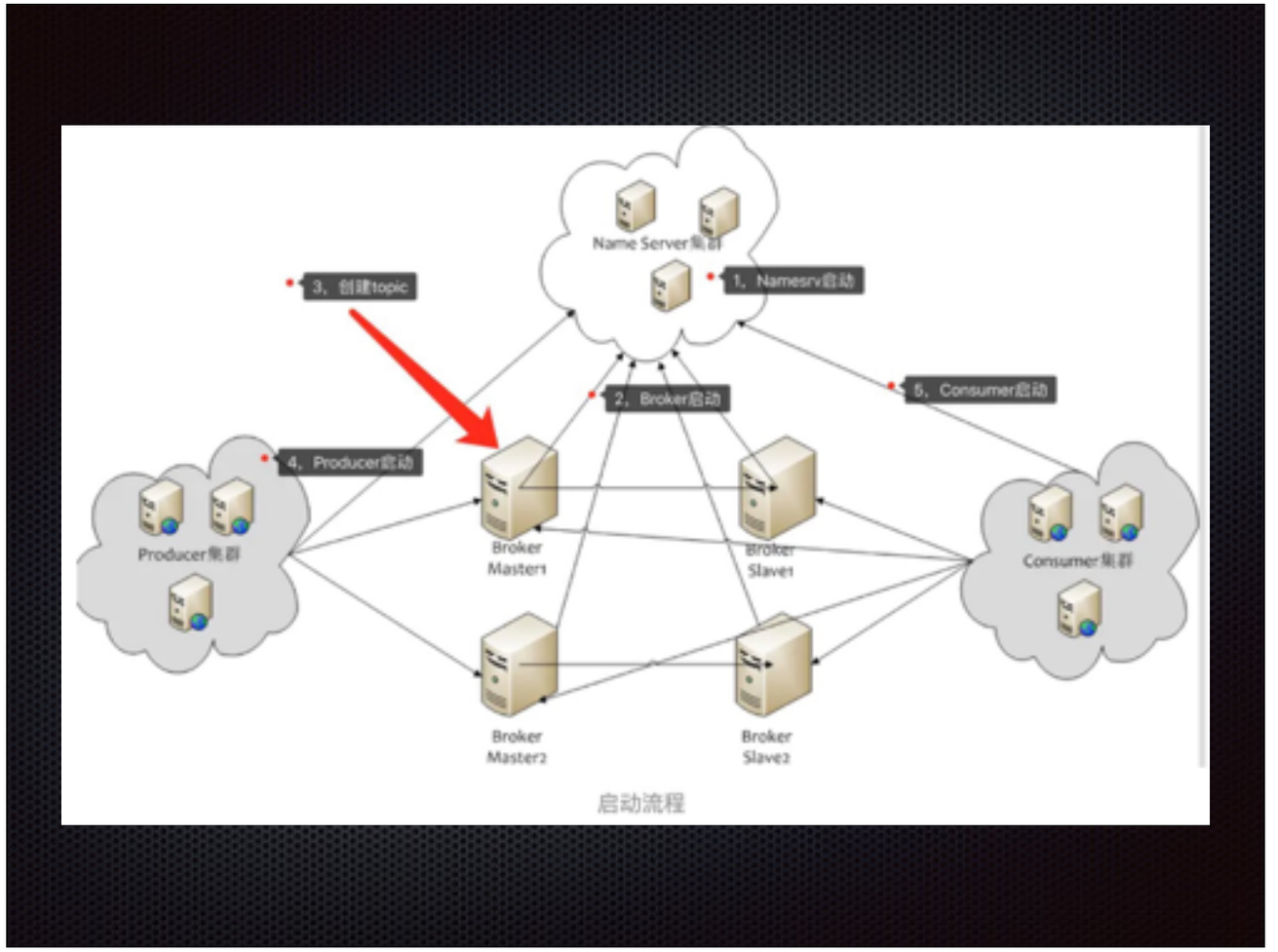

Name Server
询问TopicA在哪个Broker中
告知TopicA在BrokerA中，
并返回BA的路由信息
定时心跳告知相关状
态，如topic信息
我要订阅TopicA，告诉我它在哪？
TopicA在BA中，这是他的地址给你
Producer
TopicA
消息发送
Broker（A）
消息接收
TopicA
Consumer
哥们，我等着你呢，一有
消息就告诉我哈

部署特点

- NameServer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。
- Broker部署相对复杂，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，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。Master也可以部署多个。每个Broker与NameServer集群中的所有节点建立长连接，定时注册Topic信息到所有NameServer。 注意：当前RocketMQ版本在部署架构上支持一Master多Slave，但只有BrokerId=1的从服务器才会参与消息的读负载。
- Producer与NameServer集群中的其中一个节点（随机选择）建立长连接，定期从NameServer获取Topic路由信息，并向提供Topic 服务的Master建立长连接，且定时向Master发送心跳。Producer完全无状态，可集群部署。
- Consumer与NameServer集群中的其中一个节点（随机选择）建立长连接，定期从NameServer获取Topic路由信息，并向提供Topic服务的Master、Slave建立长连接，且定时向Master、Slave发送心跳。Consumer既可以从Master订阅消息，也可以从Slave订阅消息，消费者在向Master拉取消息时，Master服务器会根据拉取偏移量与最大偏移量的距离（判断是否读老消息，产生读I/O），以及从服务器是否可读等因素建议下一次是从Master还是Slave拉取。

工作流程

- 启动NameServer，NameServer起来后监听端口，等待Broker、Producer、Consumer连上来，相当于一个路由控制中心。
- Broker启动，跟所有的NameServer保持长连接，定时发送心跳包。心跳包中包含当前Broker信息(IP+端口等)以及存储所有Topic信息。注册成功后，NameServer集群中就有Topic跟Broker的映射关系。
- 收发消息前，先创建Topic，创建Topic时需要指定该Topic要存储在哪些Broker上，也可以在发送消息时自动创建Topic。
- Producer发送消息，启动时先跟NameServer集群中的其中一台建立长连接，并从NameServer中获取当前发送的Topic存在哪些Broker上，轮询从队列列表中选择一个队列，然后与队列所在的Broker建立长连接从而向Broker发消息。
- Consumer跟Producer类似，跟其中一台NameServer建立长连接，获取当前订阅Topic存在哪些Broker上，然后直接跟Broker建立连接通道，开始消费消息。


Name Server集群
1，Namesrv启动
2，Broker启动
3，创建topic
4，Producer启动
5，Consumer启动
Producer集群
Broker
Master1
Broker
Slave1
Broker
Master2
Broker
Slave2
Consumer集群

启动流程

消息存储

- CommitLog
- ConsumeQueue
- IndexFile

CommitLog

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。单个文件大小默认1G ，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；

ConsumeQueue

消息消费队列，引入的目的主要是提高消息消费的性能，由于RocketMQ是基于主题topic的订阅模式，消息消费是针对主题进行的，如果要遍历commitlog文件中根据topic检索消息是非常低效的。Consumer即可根据ConsumeQueue来查找待消费的消息。其中，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。consumequeue文件可以看成是基于topic的commitlog索引文件，故consumequeue文件夹的组织方式如下：topic/queue/file三层组织结构，具体存储路径为：$HOME/store/consumequeue/{topic}/{queueId}/{fileName}。同样consumequeue文件采取定长设计，每一个条目共20个字节，分别为8字节的commitlog物理偏移量、4字节的消息长度、8字节tag hashcode，单个文件由30W个条目组成，可以像数组一样随机访问每一个条目，每个ConsumeQueue文件大小约5.72M

IndexFile

IndexFile（索引文件）提供了一种可以通过key或时间区间来查询消息的方法。Index文件的存储位置是：$HOME \store\index${fileName}，文件名fileName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，IndexFile的底层存储设计为在文件系统中实现HashMap结构，故rocketmq的索引文件其底层实现为hash索引。

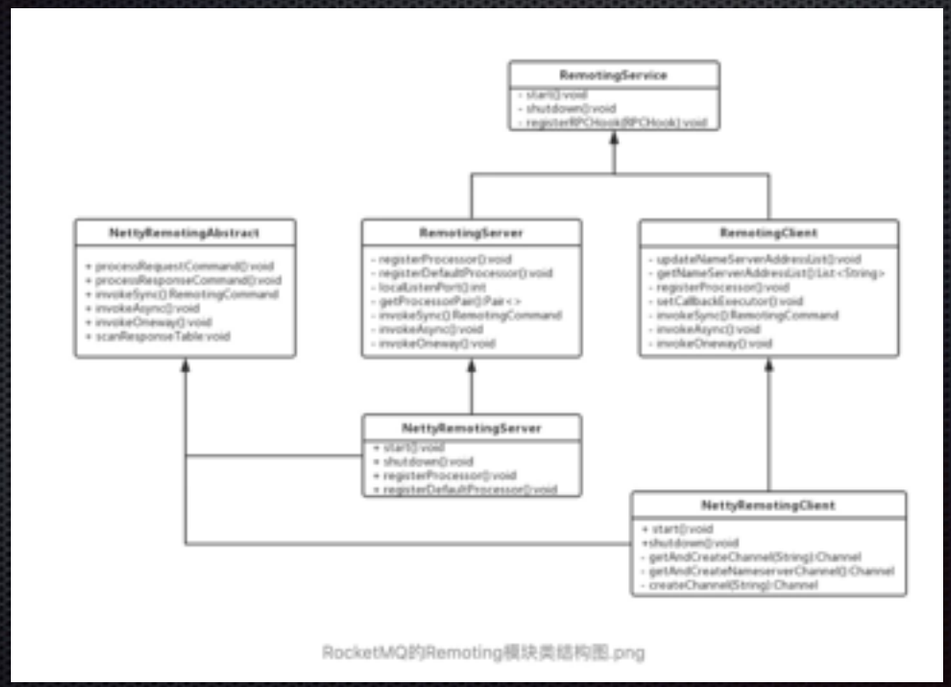
消息刷盘

- 同步刷盘：只有在消息真正持久化至磁盘后RocketMQ的Broker端才会真正返回给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该模式较多。
- 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。

通信机制

- Broker启动后需要完成一次将自己注册至NameServer的操作；随后每隔30s时间定时向NameServer上报Topic路由信息。
- 消息生产者Producer作为客户端发送消息时候，需要根据消息的Topic从本地缓存的TopicPublishInfoTable获取路由信息。如果没有则更新路由信息会从NameServer上重新拉取，同时Producer会默认每隔30s向NameServer拉取一次路由信息。
- 消息生产者Producer根据2）中获取的路由信息选择一个队列（MessageQueue）进行消息发送；Broker作为消息的接收者接收消息并落盘存储。
- 消息消费者Consumer根据2）中获取的路由信息，并再完成客户端的负载均衡后，选择其中的某一个或者某几个消息队列来拉取消息并进行消费。

Remoting通信模块的类结构图



RocketMQ的Remoting模块类结构图.png

消息的协议设计与编码解码

在Client和Server之间完成一次消息发送时，需要对发送的消息进行一个协议约定，因此就有必要自定义RocketMQ的消息协议。同时，为了高效地在网络中传输消息和对收到的消息读取，就需要对消息进行编解码。在RocketMQ中，RemotingCommand这个类在消息传输过程中对所有数据内容的封装，不但包含了所有的数据结构，还包含了编码解码，如下：
Broker向NameServer发送一次心跳注册的报文

```
[
code=103,//这里的103对应的code就是broker向nameserver注册自己的消息
language=JAVA,
version=137,
opaque=58,//这个就是requestId
flag(B)=0,
remark=null,
extFields={
    brokerId=0,
    clusterName=DefaultCluster,
    brokerAddr=ip1: 10911,
    haServerAddr=ip1: 10912,
    brokerName=LAPTOP-SMF2CKDN
},
serializeTypeCurrentRPC=JSON
```

- code: 请求操作码，应答方根据不同的请求码进行不同的业务处理
- language: 实现的语言
- version: 程序的版本
- opaque: 相当于reqeustId，在同一个连接上的不同请求标识码，与响应消息中的相对应
- flag: 区分是普通RPC还是onewayRPC的标志
- remark: 传输自定义文本信息
- extFields: 自定义扩展信息

消息编码

```java
public ByteBuffer encode() {
    // 1> header length size
    int length = 4;

    // 2> header data length
    byte[] headerData = this.headerEncode();
    length += headerData.length;

    // 3> body data length
    if (this.body != null) {
        length += body.length;
    }

    ByteBuffer result = ByteBuffer.allocate(4 + length);

    // length
    result.putInt(length);

    // header length
    result.put(markProtocolType(headerData.length, serializeTypeCurrentRPC));

    // header data
    result.put(headerData);

    // body data;
    if (this.body != null) {
        result.put(this.body);
    }

    result.flip();

    return result;
}
```

RocketMQ中Remoting协议格式.png

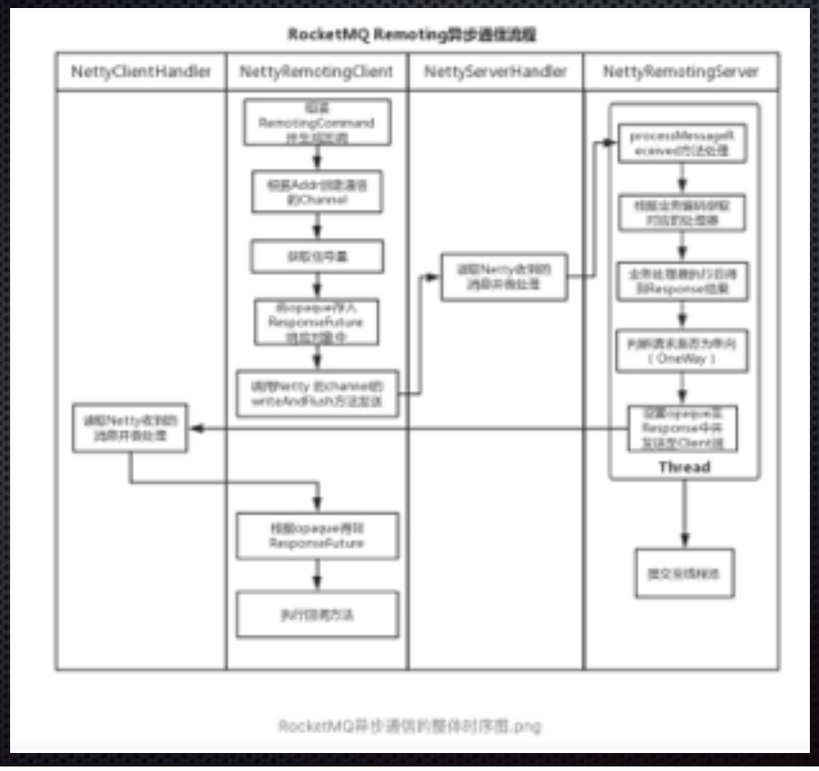
消息解码

```java
public static RemotingCommand decode(final ByteBuffer byteBuffer) {
    int length = byteBuffer.limit();
    int oriHeaderLen = byteBuffer.getInt();
    int headerLength = getHeaderLength(oriHeaderLen);

    byte[] headerData = new byte[headerLength];
    byteBuffer.get(headerData);

    RemotingCommand cmd = headerDecode(headerData, getProtocolType(oriHeaderLen));

    int bodyLength = length - 4 - headerLength;
    byte[] bodyData = null;
    if (bodyLength > 0) {
        bodyData = new byte[bodyLength];
        byteBuffer.get(bodyData);
    }
    cmd.body = bodyData;

    return cmd;
}
```

消息的通信方式

- 同步(sync)
- 异步(async)
- 单向(oneway)

RocketMQ异步通信流程



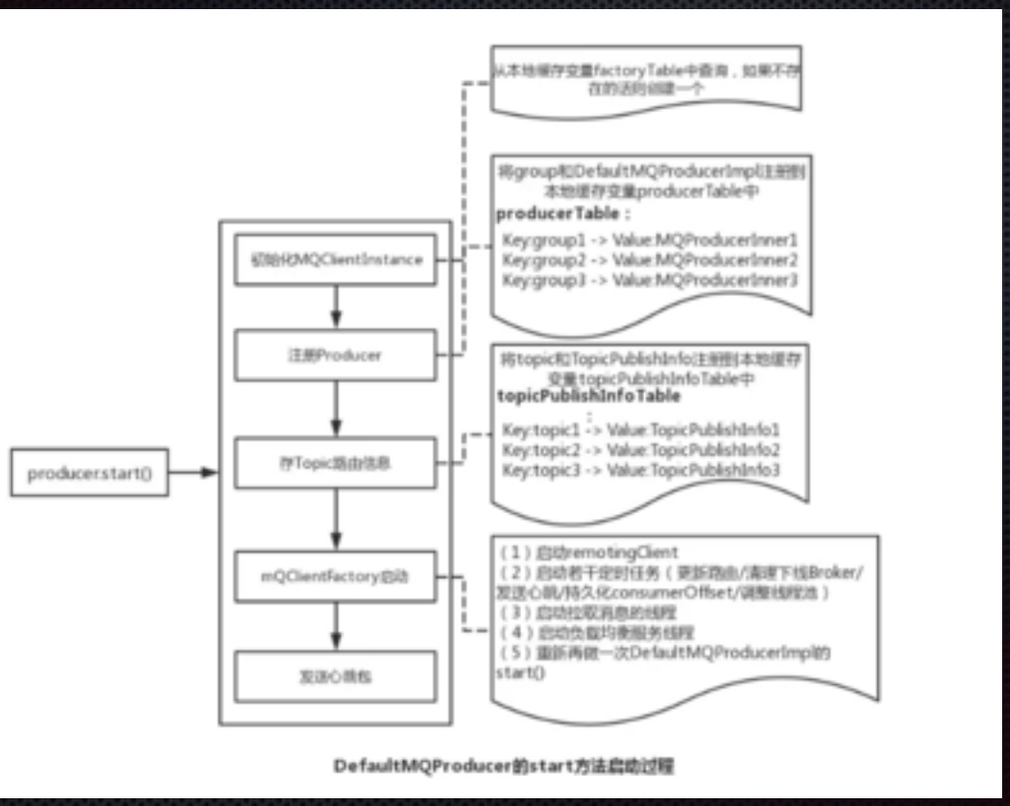
RocketMQ异步通信的整体时序图.png

消息发送

```java
/**
 * Start this producer instance.
 * </p>
 *
 * <strong>
 * Much internal initializing procedures are carried out to make this instance prepared, thus, it's a must to invoke
 * this method before sending or querying messages.
 * </strong>
 * </p>
 *
 * @throws MQClientException if there is any unexpected error.
 */
@Override
public void start() throws MQClientException {
    this.defaultMQProducerImpl.start();
}
```

DefaultMQProducer的启动流程

- 初始化得到MQClientInstance实例对象，并注册至本地缓存变量—producerTable中；
- 将默认Topic（“TBW102”）保存至本地缓存变量—topicPublishInfoTable中；
- MQClientInstance实例对象调用自己的start()方法，启动一些客户端本地的服务线程，如拉取消息服务、客户端网络通信服务、重新负载均衡服务以及其他若干个定时任务（包括，更新路由/清理下线Broker/发送心跳/持久化consumerOffset/调整线程池），并重新做一次启动（这次参数为false）；
- 最后向所有的Broker代理服务器节点发送心跳包；



DefaultMQProducer的start方法启动过程

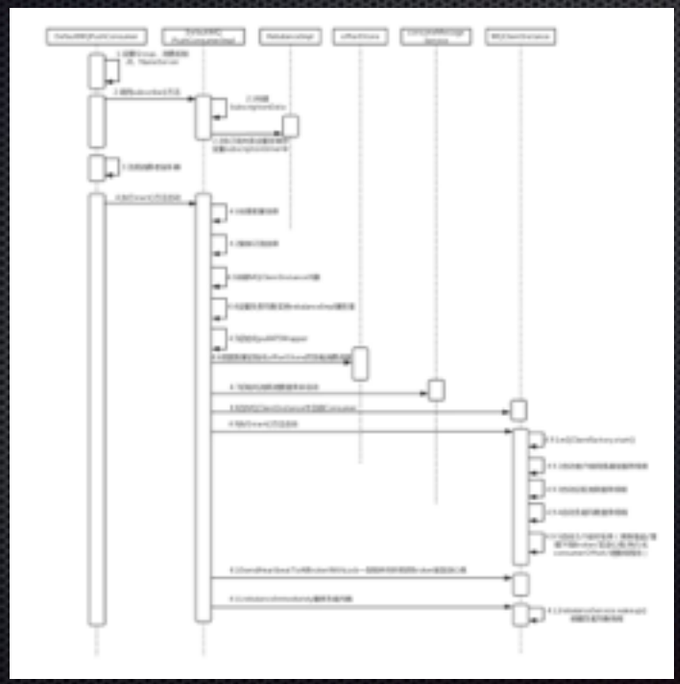
消息消费

- Push方式：由消息中间件（MQ消息服务器代理）主动地将消息推送给消费者；采用Push方式，可以尽可能实时地将消息发送给消费者进行消费。
- Pull方式：由消费者客户端主动向消息中间件（MQ消息服务器代理）拉取消息。
- 思考：Push和Pull两种消息消费方式有各自的特点。如果长时间没有消息，而消费者端又不停的发送Pull请求不就会导致RocketMQ中Broker端负载很高吗？那么在RocketMQ中如何解决以做到高效的消息消费呢?

RocketMQ消息消费的长轮询机制

研究可知，RocketMQ的消费方式都是基于拉模式拉取消息的，而在这其中有一种长轮询机制（对普通轮询的一种优化），来平衡上面Push/Pull模型的各自缺点。基本设计思路是：消费者如果第一次尝试Pull消息失败（比如：Broker端没有可以消费的消息），并不立即给消费者客户端返回Response的响应，而是先hold住并且挂起请求（将请求保存至pullRequestTable本地缓存变量中），然后Broker端的后台独立线程－PullRequestHoldService会从pullRequestTable本地缓存变量中不断地去取，具体的做法是查询待拉取消息的偏移量是否小于消费队列最大偏移量，如果条件成立则说明有新消息达到Broker端（这里，在RocketMQ的Broker端会有一个后台独立线程－ReputMessageService不停地构建ConsumeQueue/IndexFile数据，同时取出hold住的请求并进行二次处理），则通过重新调用一次业务处理器－PullMessageProcessor的处理请求方法－processRequest()来重新尝试拉取消息（此处，每隔5S重试一次，默认长轮询整体的时间设置为30s）。

RocketMQ中消费者Push方式的启动流程



- 设置consumerGroup、NameServer服务地址、消费起始偏移地址并根据参数Topic构建Consumer端的SubscriptionData（订阅关系值）；
- 在Consumer端注册消费者监听器，当消息到来时完成消费消息；
- 启动defaultMQPushConsumerImpl实例，主要完成前置校验、复制订阅关系（将defaultMQPushConsumer的订阅关系复制至rebalanceImpl中，包括retryTopic（重试主题）对应的订阅关系）、创建MQClientInstance实例、设置rebalanceImpl的各个属性值、pullAPIWrapper包装类对象的初始化、初始化offsetStore实例并加载消费进度、启动消息消费服务线程以及在MQClientInstance中注册consumer等任务；
- 启动MQClientInstance实例，其中包括完成客户端网络通信线程、拉取消息服务线程、负载均衡服务线程和若干个定时任务的启动；
- 向所有的Broker端发送心跳（采用加锁方式）；
- 唤醒负载均衡服务线程在Consumer端开始负载均衡；

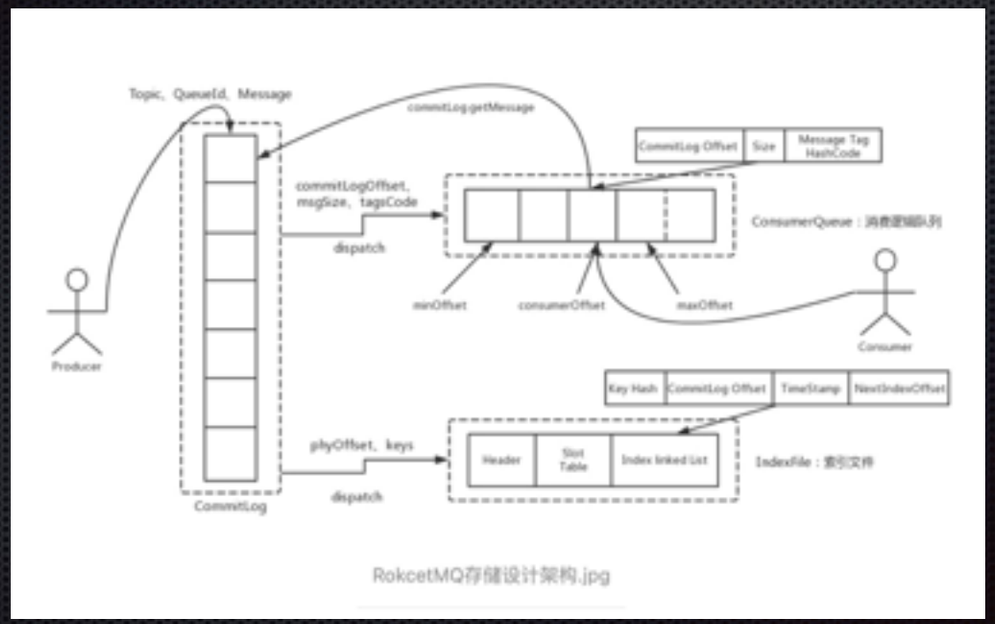
RocketMQ的Push消费模式流程简析

从严格意义上说，RocketMQ并没有实现真正的消息消费的Push模式，而是对Pull模式进行了一定的优化，一方面在Consumer端开启后台独立的线程－PullMessageService不断地从阻塞队列－pullRequestQueue中获取PullRequest请求并通过网络通信模块发送Pull消息的RPC请求给Broker端。另外一方面，后台独立线程－rebalanceService根据Topic中消息队列个数和当前消费组内消费者个数进行负载均衡，将产生的对应PullRequest实例放入阻塞队列－pullRequestQueue中。这里算是比较典型的生产者-消费者模型，实现了准实时的自动消息拉取。然后，再根据业务反馈是否成功消费来推动消费进度。
在Broker端，PullMessageProcessor业务处理器收到Pull消息的RPC请求后，通过MessageStore实例从commitLog获取消息。如果第一次尝试Pull消息失败（比如Broker端没有可以消费的消息），则通过长轮询机制先hold住并且挂起该请求，然后通过Broker端的后台线程PullRequestHoldService重新尝试和后台线程ReputMessageService的二次处理。

消息存储

- 分布式KV存储
- 文件系统
- 关系型数据库DB

RocketMQ消息存储整体架构



RocketMQ消息存储架构深入分析

RocketMQ的混合型存储结构针对Producer和Consumer分别采用了数据和索引部分相分离的存储结构，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。正因为如此，Consumer也就肯定有机会去消费这条消息，至于消费的时间可以稍微滞后一些也没有太大的关系。退一步地讲，即使Consumer端第一次没法拉取到待消费的消息，Broker服务端也能够通过长轮询机制等待一定时间延迟后再次发起拉取消息的请求。

这里，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。然后，Consumer即可根据ConsumerQueue来查找待消费的消息了。其中，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

消息过滤

- Tag过滤方式：Consumer端在订阅消息时除了指定Topic还可以指定TAG，如果一个消息有多个TAG，可以用||分隔。其中，Consumer端会将这个订阅请求构建成一个 SubscriptionData，发送一个Pull消息的请求给Broker端。Broker端从RocketMQ的文件存储层—Store读取数据之前，会用这些数据先构建一个MessageFilter，然后传给Store。Store从 ConsumeQueue读取到一条记录后，会用它记录的消息tag hash值去做过滤，由于在服务端只是根据hashcode进行判断，无法精确对tag原始字符串进行过滤，故在消息消费端拉取到消息后，还需要对消息的原始tag字符串进行比对，如果不同，则丢弃该消息，不进行消息消费
- SQL92的过滤方式：这种方式的大致做法和上面的Tag过滤方式一样，只是在Store层的具体过滤过程不太一样，真正的 SQL expression 的构建和执行由rocketmq-filter模块负责的。每次过滤都去执行SQL表达式会影响效率，所以RocketMQ使用了BloomFilter避免了每次都去执行。SQL92的表达式上下文为消息的属性。

负载均衡

- Producer的负载均衡
- Consumer的负载均衡

Producer的负载均衡

Producer端在发送消息的时候，会先根据Topic找到指定的TopicPublishInfo，在获取了TopicPublishInfo路由信息后，RocketMQ的客户端在默认方式下selectOneMessageQueue()方法会从TopicPublishInfo中的messageQueueList中选择一个队列（MessageQueue）进行发送消息。具体的容错策略均在MQFaultStrategy这个类中定义。这里有一个sendLatencyFaultEnable开关变量，如果开启，在随机递增取模的基础上，再过滤掉not available的Broker代理。所谓的"latencyFaultTolerance"，是指对之前失败的，按一定的时间做退避。例如，如果上次请求的latency超过550Lms，就退避3000Lms；超过1000L，就退避60000L；如果关闭，采用随机递增取模的方式选择一个队列（MessageQueue）来发送消息，latencyFaultTolerance机制是实现消息发送高可用的核心关键所在。

Consumer的负载均衡

在RocketMQ中，Consumer端的两种消费模式（Push/Pull）都是基于拉模式来获取消息的，而在Push模式只是对pull模式的一种封装，其本质实现为消息拉取线程在从服务器拉取到一批消息后，然后提交到消息消费线程池后，又“马不停蹄”的继续向服务器再次尝试拉取消息。如果未拉取到消息，则延迟一下又继续拉取。在两种基于拉模式的消费方式（Push/Pull）中，均需要Consumer端在知道从Broker端的哪一个消息队列—队列中去获取消息。因此，有必要在Consumer端来做负载均衡，即Broker端中多个MessageQueue分配给同一个ConsumerGroup中的哪些Consumer消费。

集群

- 单Master模式
- 多Master模式
- 多Master多Slave模式-异步复制
- 多Master多Slave模式-同步双写

单Master模式

- 风险较大，一旦Broker重启或者宕机时，会导致整个服务不可用。不建议线上环境使用,可以用于本地测试。

多Master模式

- 优点：配置简单，单个Master宕机或重启维护对应用无影响，在磁盘配置为RAID10时，即使机器宕机不可恢复情况下，由于RAID10磁盘非常可靠，消息也不会丢（异步刷盘丢失少量消息，同步刷盘一条不丢），性能最高；
- 缺点：单台机器宕机期间，这台机器上未被消费的消息在机器恢复之前不可订阅，消息实时性会受到影响。

多Master多Slave模式-异步复制

- 优点：即使磁盘损坏，消息丢失的非常少，且消息实时性不会受影响，同时Master宕机后，消费者仍然可以从Slave消费，而且此过程对应用透明，不需要人工干预，性能同多Master模式几乎一样；
- 缺点：Master宕机，磁盘损坏情况下会丢失少量消息。

多Master多Slave模式-同步双写

- 优点：数据与服务都无单点故障，Master宕机情况下，消息无延迟，服务可用性与数据可用性都非常高；
- 缺点：性能比异步复制模式略低（大约低10%左右），发送单个消息的RT会略高，且目前版本在主节点宕机后，备机不能自动切换为主机。