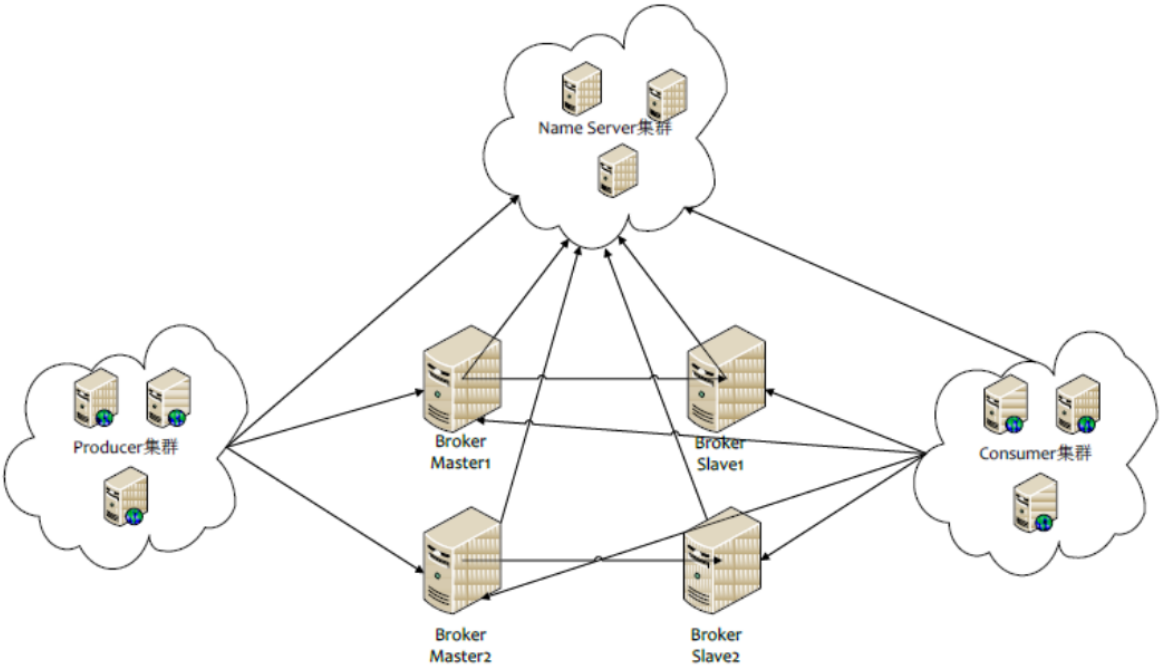
RocketMQ

0、前言



NameServer：

维持心跳和提供Topic-Broker的关系数据，多个Namesrv之间相互没有通信，单台Namesrv宕机不影响其他Namesrv与集群；即使整个Namesrv集群宕机，已经正常工作的Producer，Consumer，Broker仍然能正常工作，但新起的Producer, Consumer，Broker就无法工作，nameserver不会有频繁的读写，所以性能开销非常小，稳定性很高

Broker：

Broker与Namesrv的心跳机制：单个Broker跟所有Namesrv保持心跳请求，心跳间隔为30秒，心跳请求中包括当前Broker所有的Topic信息

高可靠并发读写服务：所有发往broker的消息，有同步刷盘和异步刷盘机制，同步刷盘时，消息写入物理文件才会返回成功，因此非常可靠；异步刷盘时，只有机器宕机，才会产生消息丢失，broker挂掉可能会发生，但是机器宕机崩溃是很少发生的，除非突然断电。

负载均衡:Broker上存Topic信息，Topic由多个队列组成，队列会平均分散在多个Broker上，而Producer的发送机制保证消息尽量平均分布到所有队列中，最终效果就是所有消息都平均落在每个Broker上

高可用：集群部署时一般都为主备，Broker名相同的一组Master/Slave Broker，其中包含一个Master Broker（Broker Id为0）和0~N个Slave Broker（Broker Id不为0），备机实时从主机同步消息，如果其中一个主机宕机，备机提供消费服务，但不提供写服务。

Producer

Producer启动时，也需要指定Namesrv的地址，从Namesrv集群中选一台Master建立长连接，生产者每30秒从Namesrv获取Topic跟Broker的映射关系，更新到本地内存中。再跟Topic涉及的所有Broker建立长连接

生产者发送时，会自动轮询当前所有可发送的broker，一条消息发送成功，下次换另外一个broker发送，以达到消息平均落到所有的broker上。假如某个Broker宕机，意味生产者最长需要30秒才能感知到。在这期间会向宕机的Broker发送消息。当一条消息发送到某个Broker失败后，会往该broker自动再重发2次，假如还是发送失败，则抛出发送失败异常。业务捕获异常，重新发送即可。客户端里会自动轮询另外一个Broker重新发送，这个对于用户是透明的

消息发送方式分为，同步发送，异步发送，单向发送

Consumer

消费者启动时需要指定Namesrv地址，与其中一个Namesrv建立长连接。消费者每隔30秒从nameserver获取所有topic的最新队列情况

Consumer跟Broker是长连接，会每隔30秒发心跳信息到Broker。Broker端每10秒检查一次当前存活的Consumer，若发现某个Consumer 2分钟内没有心跳，就断开与该Consumer的连接，并且向该消费组的其他实例发送通知，触发该消费者集群的负载均衡。

消费者得到master宕机通知后，转向slave消费（重定向，对于2次开发者透明），但是slave不能保证master的消息100%都同步过来了，因此会有少量的消息丢失。但是消息最终不会丢的，一旦master恢复，未同步过去的消息会被消费掉。

消费分为集群消费和广播消费。

Producer Group

用来表示一个发送消息应用，一个Producer Group下包含多个Producer实例，可以是多台机器，也可以是一台机器的多个进程，或者一个进程的多个Producer对象。一个Producer Group可以发送多个Topic消息。

Consumer Group

用来表示一个消费消息应用，一个Consumer Group下包含多个Consumer实例，可以是多台机器，也可以是多个进程，或者是一个进程的多个Consumer对象。一个Consumer Group下的多个Consumer以均摊方式消费消息，如果设置为广播方式，那么这个Consumer Group下的每个实例都消费全量数据。

【要点】

- Broker都注册到Nameserver上
- Producer发消息的时候会从Nameserver上获取发消息的topic信息
- Producer向提供服务的所有master NameServer建立长连接，且定时向master发送心跳
- Consumer通过NameServer集群获得Topic的路由信息
- Consumer会与所有的Master和所有的Slave都建立连接进行监听新消息

【TAG】

子主题，二级类型，比如交易消息又可以分为：交易创建消息、交易完成消息等，一条消息可以没有 **Tag** 。

发送/消费 消息的时候可以指定tag/key来进行过滤消息，支持通配符。

【KEY】

每个消息在业务层面的唯一标识码，要设置到 keys 字段，方便将来定位消息丢失问题。服务器会为每个消息创建索引(哈希索引)，应用可以通过 topic，key来查询这条消息内容，以及消息被谁消费。由于是哈希索引，请务必保证key 尽可能唯一，这样可以避免潜在的哈希冲突。

//订单Id

String orderId= "20034568923546";

message.setKeys(orderId);

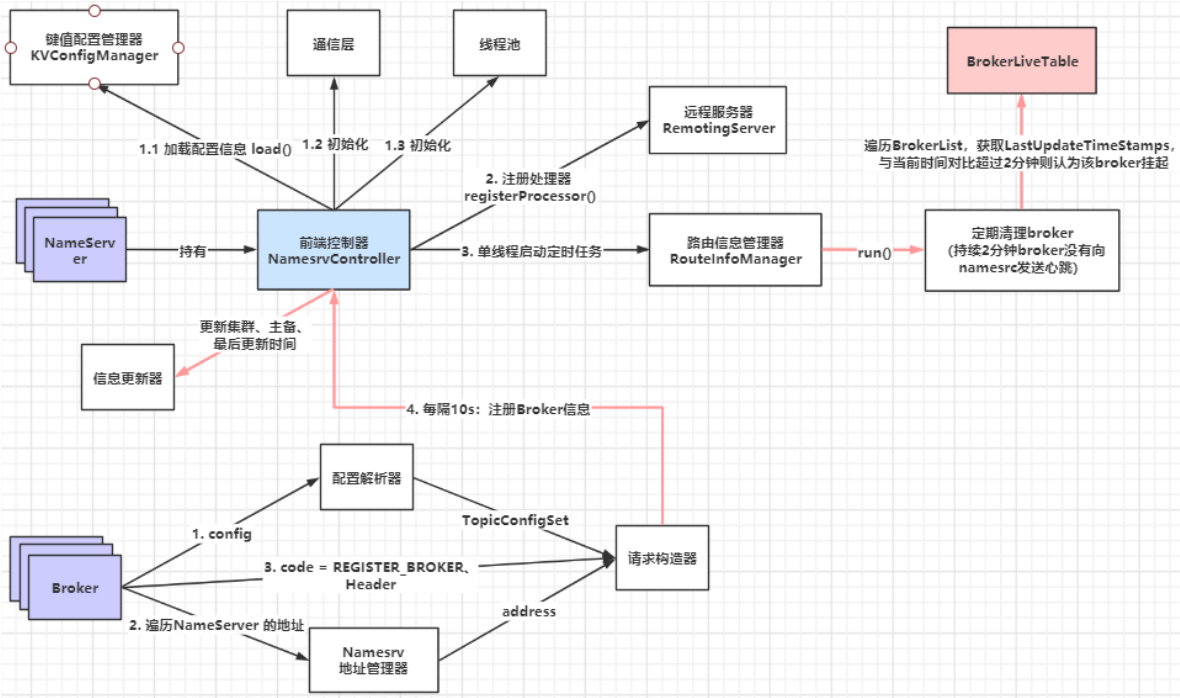
1、NameServer

1. NameServer的功能

- 每个broker启动的时候会向namesrv注册
- Producer发送消息的时候根据topic获取路由到broker的信息
- Consumer根据topic获取topic的路由到broker的信息

2. NameServer的启动流程

【流程图】



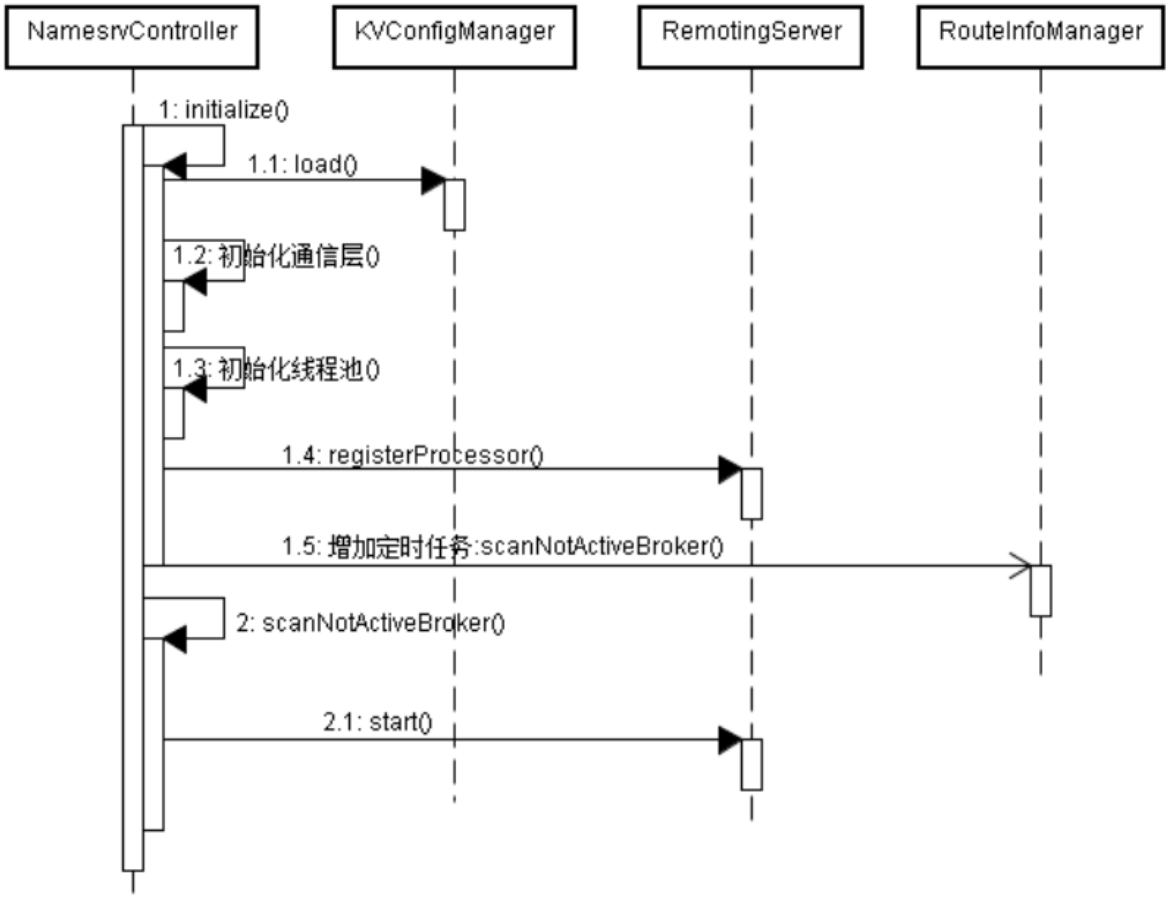
NameServer持有一个核心控制器NamesrcController，由该控制器主持NameServer的启动流程。

启动时，首先会加载配置信息，初始化通信层，底层是netty框架，所以是创建一个nettyClient，用于和生产者、消费者、Broker的通信。

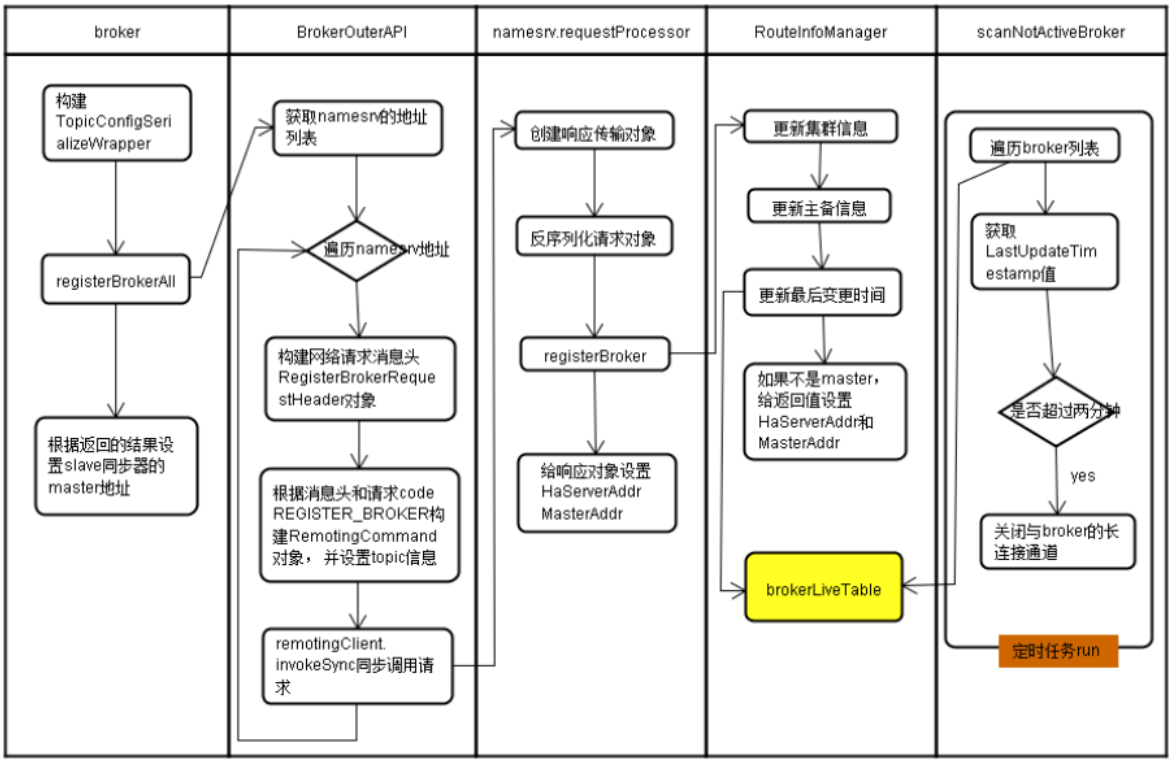
然后是初始化线程池，单线程启动一个定时任务，定时清理不活跃的Broker，具体就是更新一个Broker活跃表BrokerLivedTable，将不活跃的对象剔除出去。通过调用 RouteInfoManger 的 scanNotActiveBroker 方法来定时清理不活动的broker，默认两分钟如果Broker没有向 namesrv 发送心跳更新 BrokerLiveInfo 时间戳的，则认为是过期，关闭channel连接。

> **核心控制器（加载配置信息、初始化通信层，创建 netty 客户端）**
>
> **线程池定时任务（定期清理不活跃的 Broker，2分钟没心跳包来往的客户端）**

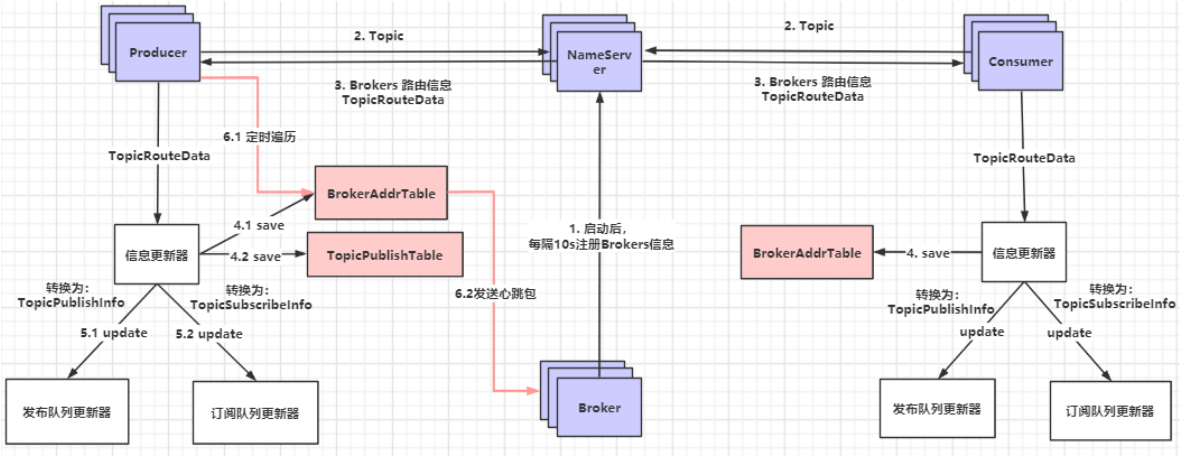
【时序图】



【活动图】



3. NameServer 与 Client 之间的心跳



【NameServer 与 Broker 的心跳】

Broker启动时会启动定时任务，每隔十秒钟会向所有namesrv发送心跳请求，同时注册topic信息到namesrv。

Namesrv 接收 borker 心跳 DefaultRequestProcessor 的 REGISTER_BROKE 事件处理，进行：

- 注册 broker 的 topic 信息
- 构建或者更新 BrokerLiveInfo 的时间戳

【Producer 与 NameServer的心跳】

Producer 会定时根据 Topic 到 NameServer 拉取 Broker 的路由信息到本地并进行更新，比如会更新 Broker 的地址（BrokerAddrTable），更新 Topic 的信息（TopicPublishTable）等。

Producer 会定时遍历所有的 Broker（BrokerAddrTable），与之建立心跳，并发送 Producer 的信息（procduer的 group）。

【Consumer 与 NameServer 的心跳】

Consumer 会根据 Topic 到 NameServer 拉取 Broker 的路由信息到本地并进行更新。

【Broker 和 Client 的心跳】

Producer 和 Consumer 会定时遍历所有的 Broker（BrokerAddrTable），与之建立心跳，并发送 Producer 的信息（procduer的 group）和 Consumer 的信息（订阅关系，哪些消费者订阅了哪些主题）。如果2分钟没有心跳，Broker 会将对应的客户端清理出去。

【小结】

每个 Broker 启动后会定时向 namesrv 注册 topic 信息。

Producer 和 Consumer 会定时（30s）到 namesrv 拉取 Broker 的路由信息（Topic 跟 Broker的映射关系）到本地并进行更新。

Producer 和 Consumer 启动后会定时和所有 Broker 建立心跳，发送 Producer 的信息、和 Consumer 的订阅信息（订阅关系，哪些消费者订阅了哪些主题）。

4.路由信息

路由信息 RouteInfo 由 RouteInfoManager管理，主要包括：

1）Broker 信息：Map<BrokerName, BrokerData>
brokerData 由 brokerName 和它的 broker ids 和 address组成，id表示是master还是slave，id= 0为master ，大于0为slave。brokerName表示一组broker，如：一个叫brokerName=broker-a， 可能包括一个master跟它的多个slave。

{broker-a，[(0, ip1)、(1, ip2)、(2, ip3)]}

{broker-b，[(0, ip1)、(1, ip2)、(2, ip3)]}

> **Broker 的地址信息，如果 Broker 是集群，那就是一个 Broker 的名字代表一个集群，对应有 n 个 Broker 的地址以及表示是主还是从的 id。**

2）Topic 信息：Map<topic, List>
queueData由brokerName， 读队列数，写队列数，权限值组成。

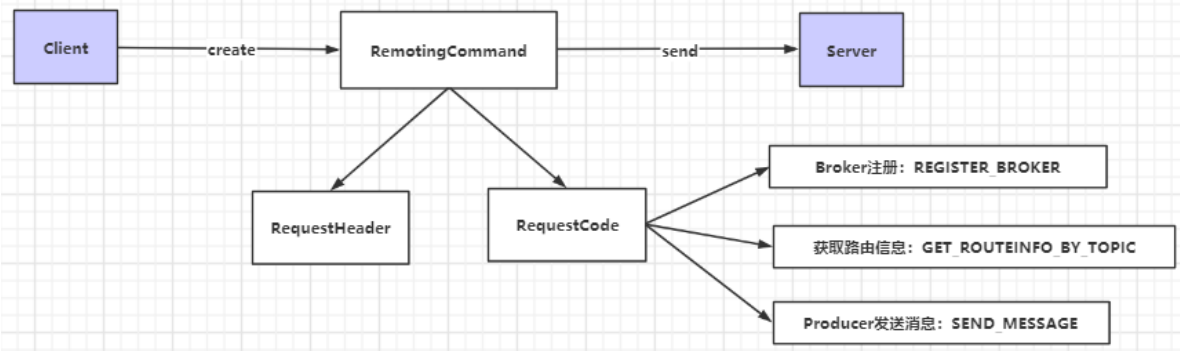
> **topic 与对应的 brokerName 和读写队列信息**

3）集群信息： Map<clusterName, Set>

将broker按照集群分组

4）活跃 Broker 信息：Map<brokerAddr, BrokerLiveInfo>
BrokerLiveInfo 代表一个活的broker链接由最后更新时间，一个链接channel，数据版本和Ha地址组成。

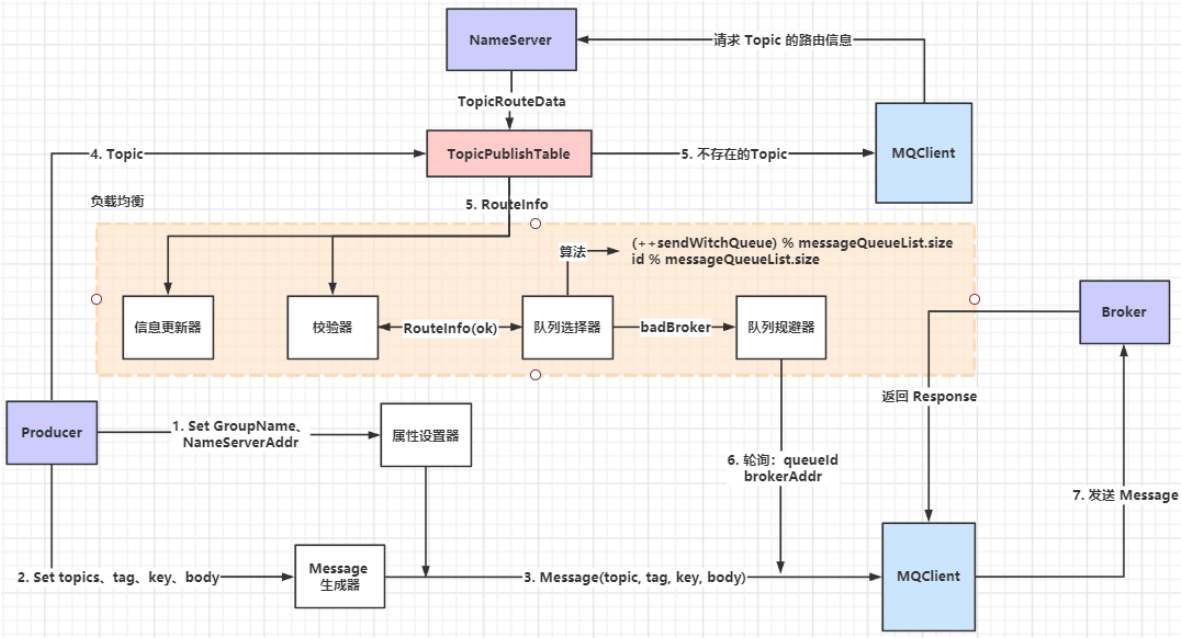
5.通信协议



2、Producer

1.Producer 发送普通消息

【流程图】



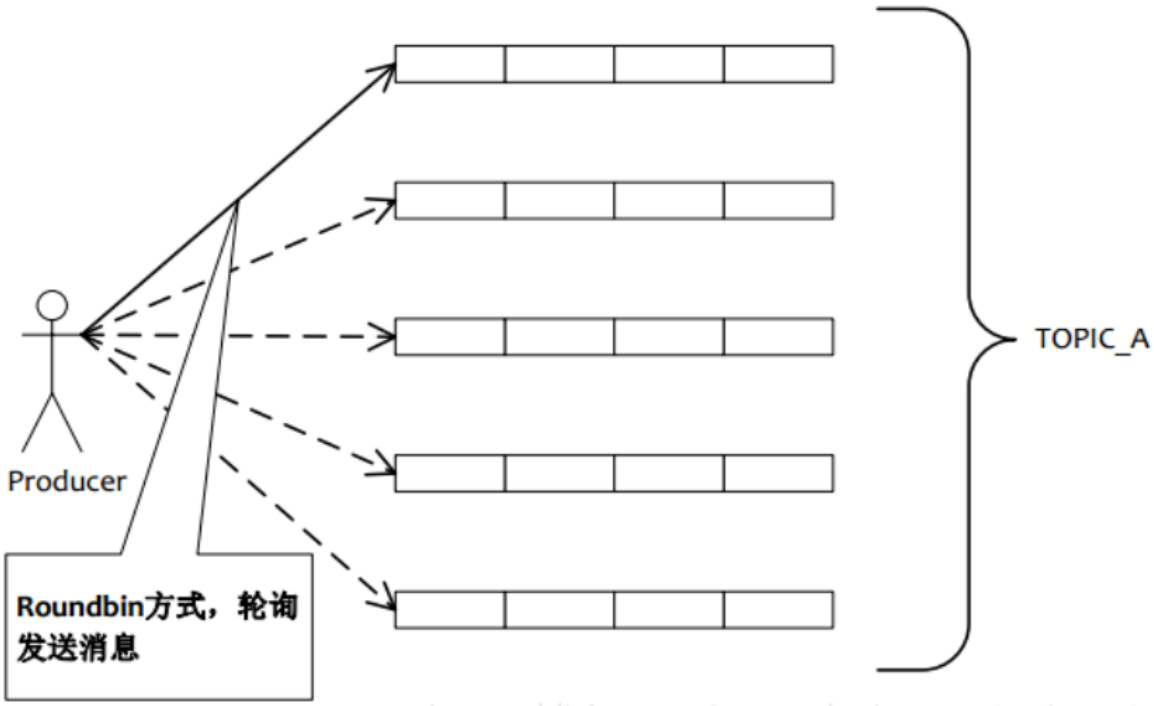
消息发送时，首先初始化一些参数配置，比如消息的组名 GroupName 以及 NameServer 的地址等。

然后根据 Topic 获取对应的 Broker 的路由信息，先从本地的路由信息表 TopicPublishInfoTable 中查找，如果没有找到目标的 Topic，再发心跳包到 NameServer，查找该 Topic 的路由信息，拉取到本地再进行相应的更新。

获取到路由信息后，准备发送到目标 Broker，这时可能存在一个 Topic 对应多个 Broker 多个 Queue 的情况，所以需要进行负载均衡。常见的算法有两种，一种是正常情况下的队列轮询算法，一种是要顺序消费时，通过相同的某个关键字，比如可以是订单id，然后对消息队列个数求余来达到相同订单id发往同一个队列的目的。

> **获取路由信息（本地获取、远程获取）**
>
> **负载均衡（轮询、哈希或求余）**

【负载均衡】



【队列轮询】
Producer 从 namesrv 获取的到 Topic 的路由信息 TopicPublishInfo，比如List messageQueueList 代表 Topic_A 的所有的队列，自增整形队列下标是AtomicIntegersendWhichQueue，调用 selectOneMessageQueue 方法用来选择一个发送队列，(++sendWitchQueue)% messageQueueList.size为队列集合的下标，每次获取 queue 都会通过 sendWhichQueue 加一来实现对所有 queue 的轮询。

【队列规避】

轮询时还会进行一个队列规避的动作，意思是上一次发送失败的 Broker 不参与此负载均衡，所以上次失败的 Broker 对应的 Queue 就会被规避。以保证一定的高可用性。

【发送重试】
Producer 通过 selectOneMessageQueue 方法获取一个 MessagQueue(topic, brokerName, queueId)对象，向指定topic的指定broker的指定queue发送消息。若：

- 失败次数不到重试次数上限
- 发送时间未超时

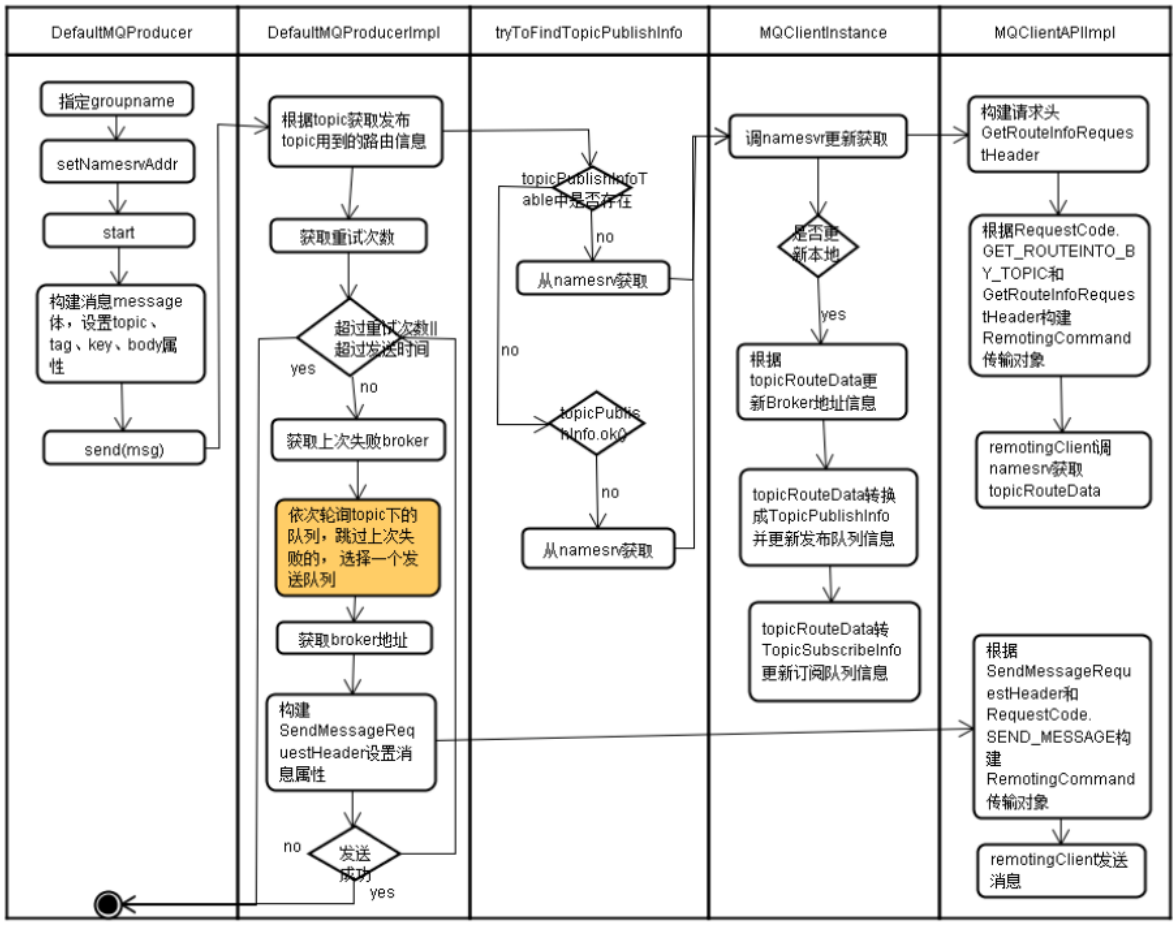
则换个队列继续尝试。

选出合适的队列后，通过 nettyClient 通信层发往 Broker。

【发送方式】

sendOneway > sendCallBack > send批量 > send单条

【活动图】



2.Producer 发送事务消息

【示例】

Bob向Smith转账100块。

单机

操作	耗时	总耗时
锁定Bob账户	0.01ms	5.04ms
锁定Smith账户	0.01ms	
检查Bob账户是否有100块	1ms	
Bob账户减去100块	2ms	
Smith账户加上100块	2ms	
解锁Smith账户	0.01ms	
解锁Bob账户	0.01ms	

事物时间顺序

网络

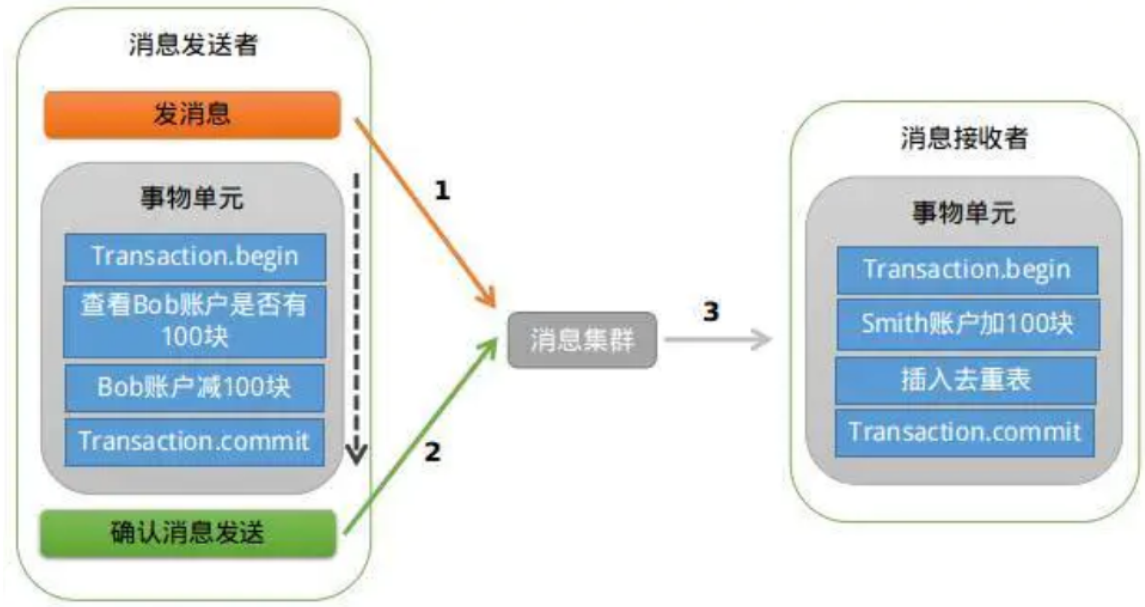
操作	耗时	总耗时
锁定Bob账户	0.01ms	11.04ms
通过网络锁定Smith账户	2+0.01ms	
检查Bob账户是否有100块	1ms	
Bob账户减去100块	2ms	
通过网络Smith账户加上100块	2+2ms	
通过网络解锁Smith账户	2+0.01ms	
解锁Bob账户	0.01ms	

事物时间顺序

解决

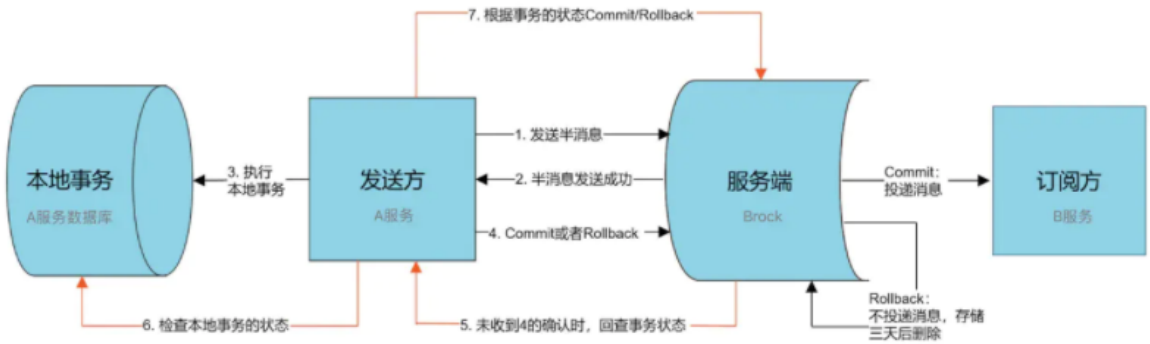
将大事务拆分成多个小事务异步执行。这样基本上能够将跨机事务的执行效率优化到与单机一致。转账的事务就可以分解成如下两个小事务：

【实现】



RocketMQ第一阶段发送Prepared消息时，会拿到消息的地址，第二阶段执行本地事物，第三阶段通过第一阶段拿到的地址去访问消息，并修改状态。细心的你可能又发现问题了，如果确认消息发送失败了怎么办？RocketMQ会定期扫描消息集群中的事物消息，这时候发现了Prepared消息，它会向消息发送者确认，Bob的钱到底是减了还是没减呢？如果减了是回滚还是继续发送确认消息呢？RocketMQ会根据发送端设置的策略来决定是回滚还是继续发送确认消息。这样就保证了消息发送与本地事务同时成功或同时失败。

【示意图】



Half Message(半消息)

是指暂不能被 Consumer 消费的消息。Producer 已经把消息成功发送到了 Broker 端，但此消息被标记为暂不能投递状态，处于该种状态下的消息称为半消息。需要 Producer 对消息的二次确认后，Consumer 才能去消费它。

消息回查

由于网络闪段，生产者应用重启等原因。导致 Producer 端一直没有对 Half Message(半消息) 进行 二次确认。这时Brock服务器会定时扫描长期处于半消息的消息，会主动询问 Producer端 该消息的最终状态(Commit或者Rollback),该消息即为 消息回查。

流程

A服务先发送个Half Message给Brock端，消息中携带 B服务 即将要+100元的信息。

当A服务知道Half Message发送成功后，那么开始第3步执行本地事务。

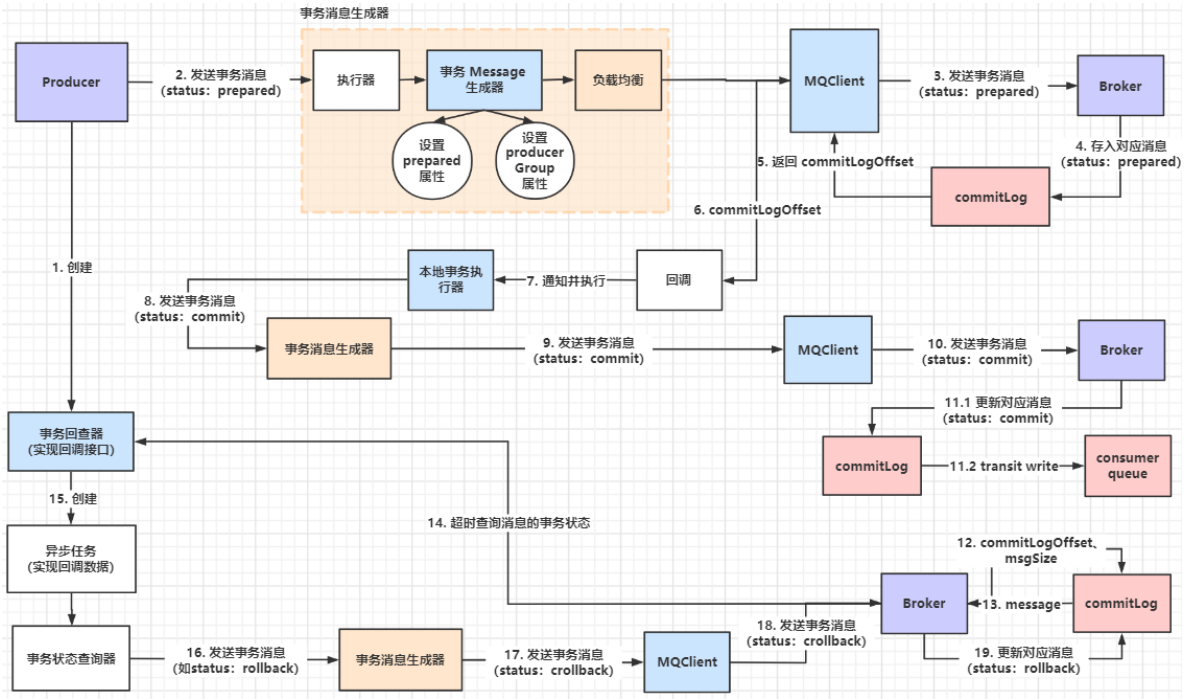
执行本地事务(会有三种情况1、执行成功。2、执行失败。3、网络等原因导致没有响应)

如果本地事务成功，那么Product向Brock服务器发送Commit,这样B服务就可以消费该message。

如果本地事务失败，那么Product向Brock服务器发送Rollback,那么就会直接删除上面这条半消息。

如果因为网络等原因迟迟没有返回失败还是成功，那么会执行RocketMQ的回调接口,来进行事务的回查。

【流程图】



发送事务消息，是两阶段提交的过程，这个过程 Producer 会向 Broker 发送两条消息，而 Consumer 最终消费一条消息的过程。

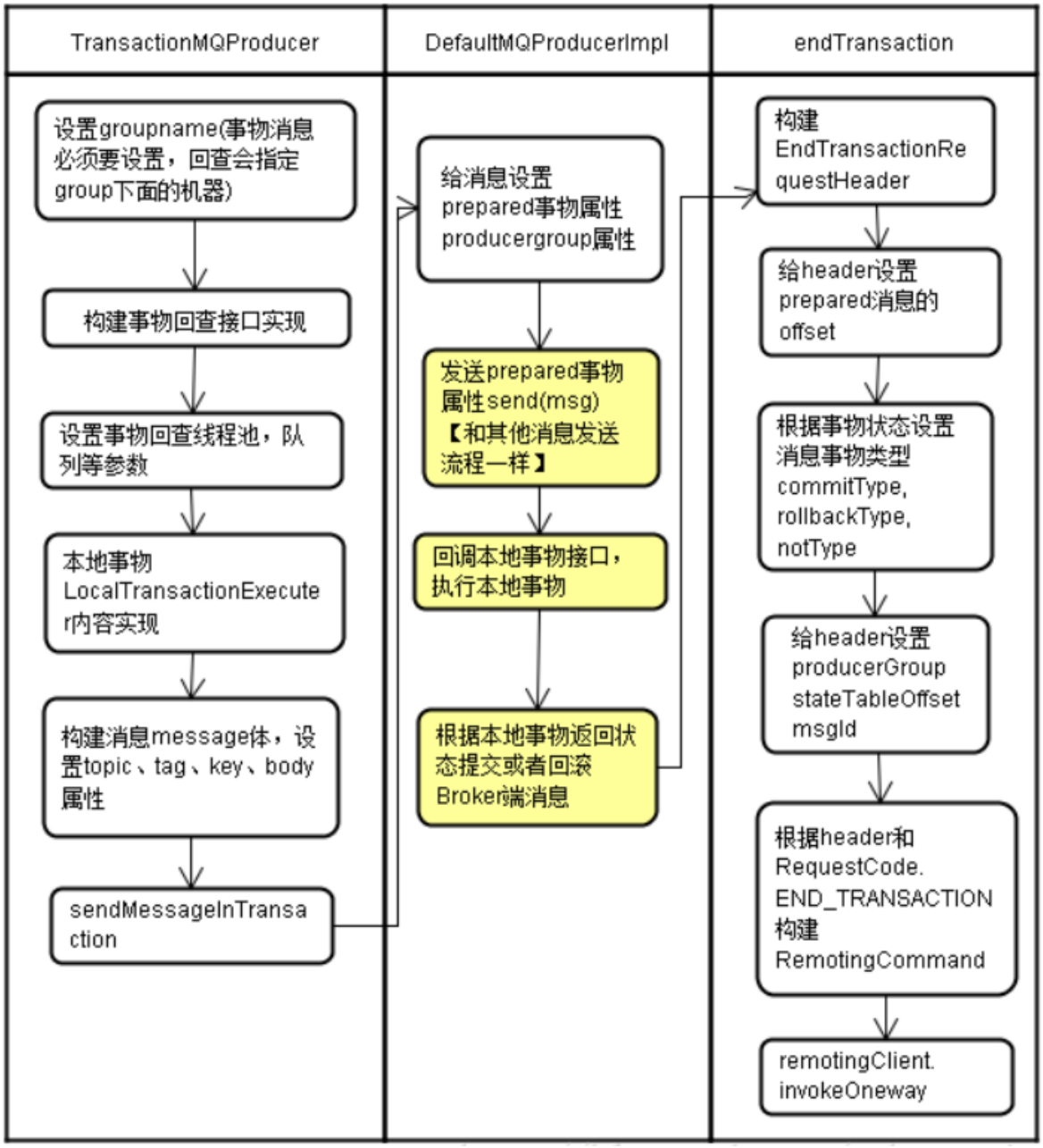
通俗地说，首先 Producer 先发消息和 Broker 说我准备好执行事务操作了，此时发过去的消息带有一个状态标识是 prepared，Broker 收到 prepared 状态后回复 Producer 表示已知悉，此时消息并未提交到消费队列，消费者并不能消费。

Producer 收到回复后，开始执行事务，执行完后再和 Broker 说我完成事务了，此时发过去的消息带有一个状态标识是 commit ，表示可以提交了。然后Broker 收到后就将消息提交到 Consumer Queue 消费队列中，此时对消费者可见，消费者才能真正消费。

如果由于其他原因，超过一定时间后，Producer 没有对消息进行二次确认，那么 Broker 会通过我们实现的一个回调接口进行状态的回查，主动向 Producer 询问消息的状态。Producer 查询后，再向 Broker 汇报，如果本地事务失败，那么 Product 向 Brock 发送 状态Rollback，那么 Broker 就会直接删除上面这条半消息。

> **Producer 发送 prepared 消息、执行本地事务**
>
> **Producer 发送 commit 消息、Broker 提交到日志文件和消费队列**
>
> **Broker 通过回调接口主动查询 Producer 消息状态**

【活动图】



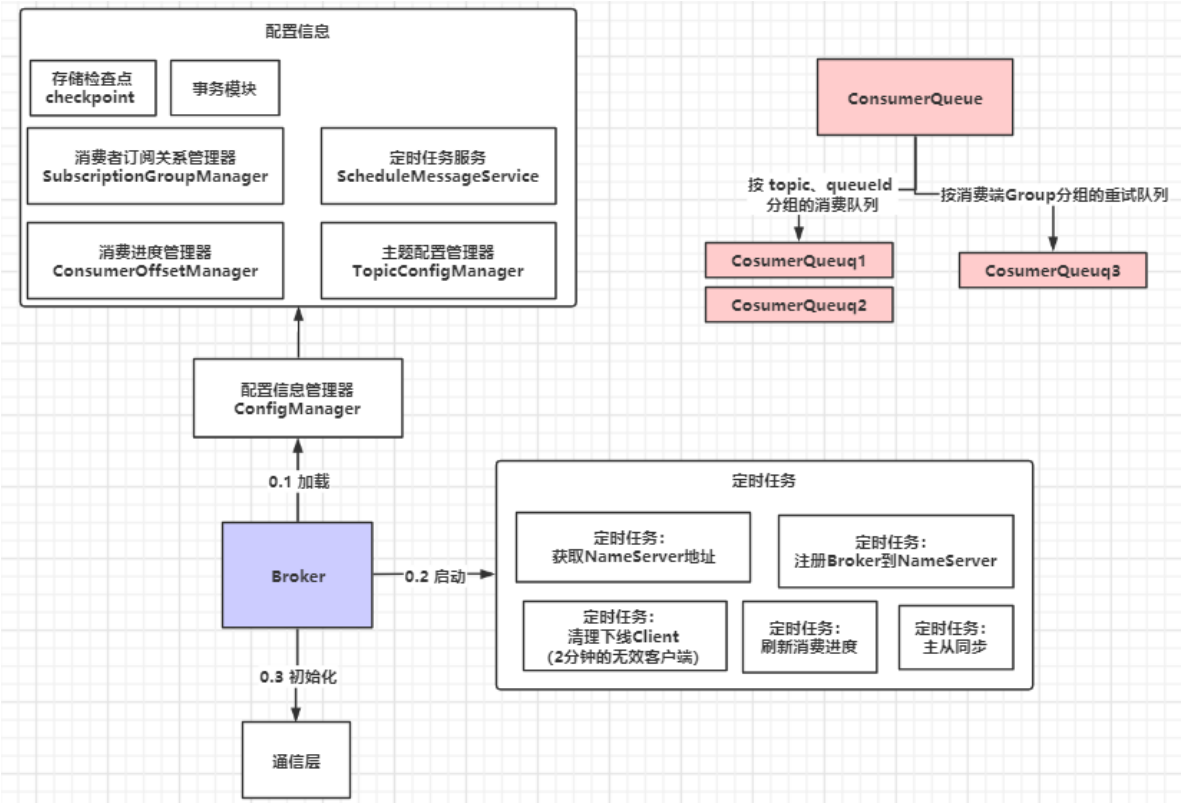
3.优先级

由于RocketMQ所有消息都是持久化的，所以如果按照优先级来排序，开销会非常大，因此RocketMQ没有特意支持消息优先级，但是可以通过变通的方式实现类似功能，即单独配置一个优先级高的队列，和一个普通优先级的队列， 将不同优先级的消息发送到不同队列即可。

3、Broker

1.Broker的启动流程

【流程图】



Broker 启动时首先会加载一连串的配置信息，比如 Topic 的消费进度，Topic 的配置信息（Topic 和 Broker 的对应关系），消费者的订阅信息（哪些消费者订阅了哪些 Topic 的哪些 Queue），以及将内存中的数据刷新到磁盘时的存储检查点等。

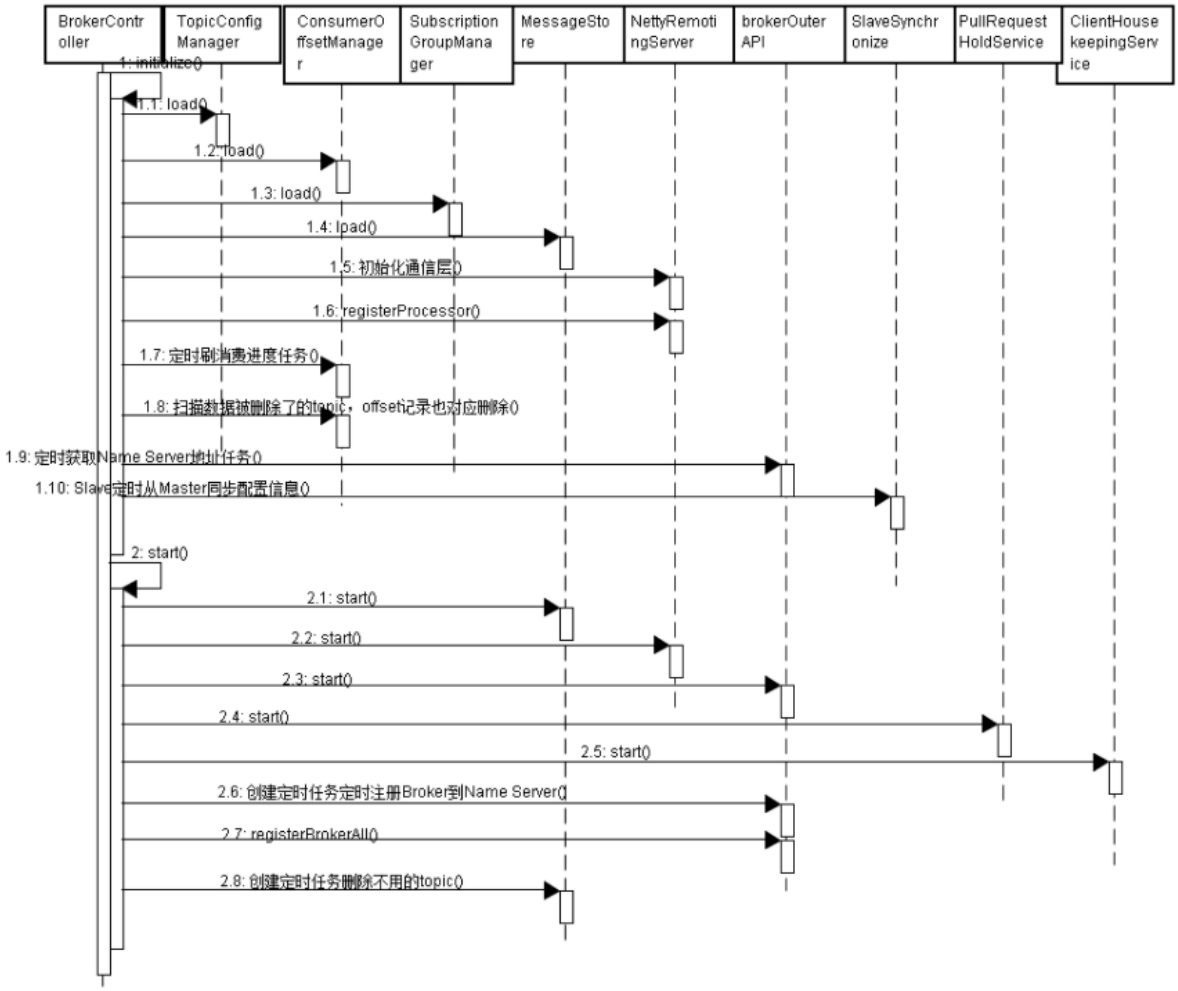
第二步是启动一些定时任务，比如

- 定时获取 NameServer 地址并更新
- 定时注册 Broker 和 Topic 的信息（二者的对应关系）到 NameServer中
- 定时清理下线的客户端，比如每2分钟会和 Producer 和 Consumer 维持心跳，如果超时没有心跳，则会认为客户端下线了，会将其清理出本地维护的状态表中
- 定时刷新消费进度
- 定时主从同步

第三部是初始化通信层，建立NettyClient客户端

> **加载配置信息（topic 消费进度、topic 配置信息、消费者订阅信息）**
>
> **启动定时任务（获取 nsv 地址、注册 Broker 和 Topic 信息到 nsv、清理客户端、刷新消费进度、主从同步）**
>
> **初始化通信层，建立 Netty 客户端**

【时序图】



2.Broker的文件组织形式

【TopicConfig】

```
{
 "TopicTest":{
  "perm":6,
  "readQueueNums":8,
  "topicFilterType":"SINGLE_TAG",
  "topicName":"TopicTest",
  "writeQueueNums":8
 }
}
```

【Consume Queue】

消息的逻辑队列，相当于字典的目录用来指定消息在消息的真正的物理文件commitLog上的位置，每个topic下的每个queue都有对应的consumequeue文件。

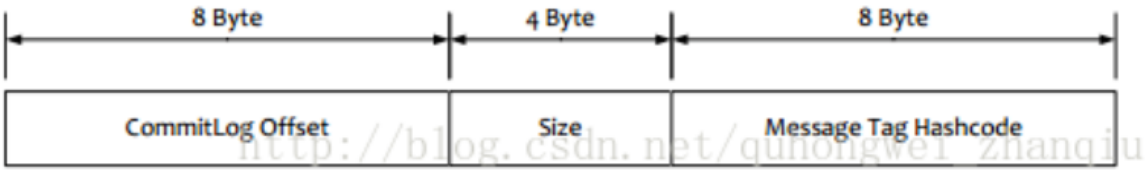
文件地址：${user.home} \store\consumequeue${topicName}${queueId}${fileName}

consume queue中存储单元是一个20字节定长的数据，是顺序写顺序读

（1） commitLogOffset 是指这条消息在 commitLog 文件的实际偏移量

（2） size就是指消息大小

（3） 消息tag的哈希值



topic queueId 组织，比如TopicA配了读写队列0, 1,那么TopicA和Queue=0组成一个ConsumeQueue,TopicA和Queue=1组成一个另一个ConsumeQueue.

- 按消费端group分组重试队列，如果消费端消费失败，发送到retry消费队列中
- 按消费端group分组死信队列，如果消费端重试超过指定次数，发送死信队列
- 每个ConsumeQueue可以由多个文件组成无限队列被MapedFileQueue对象管理

```
`-- consumequeue
    |-- %DLQ%ConsumerGroupA
    |    `-- 0
    |         `-- 00000000000006000000
    |-- %RETRY%ConsumerGroupA
    |    `-- 0
    |         `-- 00000000000000000000
    |-- %RETRY%ConsumerGroupB
    |    `-- 0
    |         `-- 00000000000000000000
    |-- SCHEDULE_TOPIC_XXXX
    |    |-- 2
    |    |    `-- 00000000000006000000
    |    |-- 3
    |    |    `-- 00000000000006000000
    |-- TopicA
    |    |-- 0
    |    |    |-- 00000000002604000000
    |    |    |-- 00000000002610000000
    |    |    `-- 00000000002616000000
    |    |-- 1
    |    |    |-- 00000000002610000000
    |    |    `-- 00000000002616000000
```

【 CommitLog 】

CommitLog 消息存放物理文件，每台 broker 上的 commitLog 被本机器所有 queue 共享不做区分。

```
文件地址：${user.home} \store\${commitlog}\${fileName}

一个消息存储单元长度是不定的，顺序写但是随机读
```

```
消息存储结构：

=        4  //4个字节代表这个消息的大小

                    +       4 //四个字节的MAGICCODE = daa320a7

                    +       4  //消息体BODY CRC  当broker重启recover时会校验

                    +       4 //queueId  你懂得

                    +       4  //flag  这个标志值rocketmq不做处理，只存储后透传

                    +     8  //QUEUEOFFSET这个值是个自增值不是真正的consume queue的偏移量，可以代表这个队列中消息的个数，要通过这个值查找到consume queue中数据，QUEUEOFFSET * 20才是偏移地址

                    +       8       //PHYSICALOFFSET 代表消息在commitLog中的物理起始地址偏移量

                    +     4       //SYSFLAG消息标志，指明消息是事物事物状态等等消息特征

                    +     8       //BORNTIMESTAMP 消息产生端(producer)的时间戳

                    +     8      //BORNHOST          消息产生端(producer)地址(address:port)

                    +       8       //STORETIMESTAMP 消息在broker存储时间

                    +       8       //STOREHOSTADDRESS 消息存储到broker的地址(address:port)

                    +       8      //RECONSUMETIMES消息被某个订阅组重新消费了几次（订阅组之间独立计数）,因为重试消息发送到了topic名字为%retry%groupName的队列queueId=0的队列中去了

                    +       8       //Prepared Transaction Offset 表示是prepared状态的事物消息

                    +       4  +  bodyLength // 前4个字节存放消息体大小值，  后bodylength大小空间存储了消息体内容

                    +       1 +  topicLength  //一个字节存放topic名称能容大小， 后存放了topic的内容

         +       2 + propertiesLength    // 2个字节（short）存放属性值大小， 后存放propertiesLength大小的属性数据
```

【 MapedFile 】

MapedFile 是 PageCache 文件的封装，操作物理文件在内存中的映射以及将内存数据持久化到物理文件中，代码中写死了要求os系统的页大小为4k, 消息刷盘根据参数（commitLog默认至少刷4页, consumeQueue默认至少刷2页）才刷 。

【 MapedFileQueue 】

MapedFileQueue 存储队列，数据定时删除，无限增长。

队列有多个文件（MapedFile）组成，由集合对象List表示升序排列，前面讲到文件名即是消息在此文件的中初始偏移量，排好序后组成了一个连续的消息队。

```
当消息到达broker时，需要获取最新的MapedFile写入数据，调用MapedFileQueue的
getLastMapedFile获取，此函数如果集合中一个也没有创建一个，如果最后一个写满了也创建一个新的。

MapedFileQueue在获取getLastMapedFile时，如果需要创建新的MapedFile会计算出下一个
MapedFile文件地址，通过预分配服务AllocateMapedFileService异步预创建下一个MapedFile文件，
这样下次创建新文件请求就不要等待，因为创建文件特别是一个1G的文件还是有点耗时的，

getMinOffset获取队列消息最少偏移量，即第一个文件的文件起始偏移量
getMaxOffset获取队列目前写到位置偏移量
getCommitWhere刷盘刷到哪里了
```

. `MessageStoreConfig`:存储相关的配置

```
[root@a7de77ad073b store]# tree
.
├── abort
├── checkpoint
├── commitlog
│   └── 00000000000000000000
├── config
│   ├── consumerFilter.json
│   ├── consumerFilter.json.bak
│   ├── consumerOffset.json
│   ├── consumerOffset.json.bak
│   ├── delayOffset.json
│   ├── delayOffset.json.bak
│   ├── subscriptionGroup.json
│   ├── subscriptionGroup.json.bak
│   ├── topics.json
│   └── topics.json.bak
├── consumequeue
│   └── testTopic
│       └── 3
│           └── 00000000000000000000
├── index
│   └── 20181117090154588
└── lock
```

1. $1 获取消息的事务类型
2. $2 对于事务消息中UNKNOW、COMMIT消息，处理topic和queueId, 同时备份real_topic，real_queueId
3. $3 获取最新的mappedFile文件，有可能为空
4. $4 给写mappedFile加锁（默认自旋锁）
5. $5 mappedFile为空时创建mappedFile文件， 创建的mappedFile文件offset为0

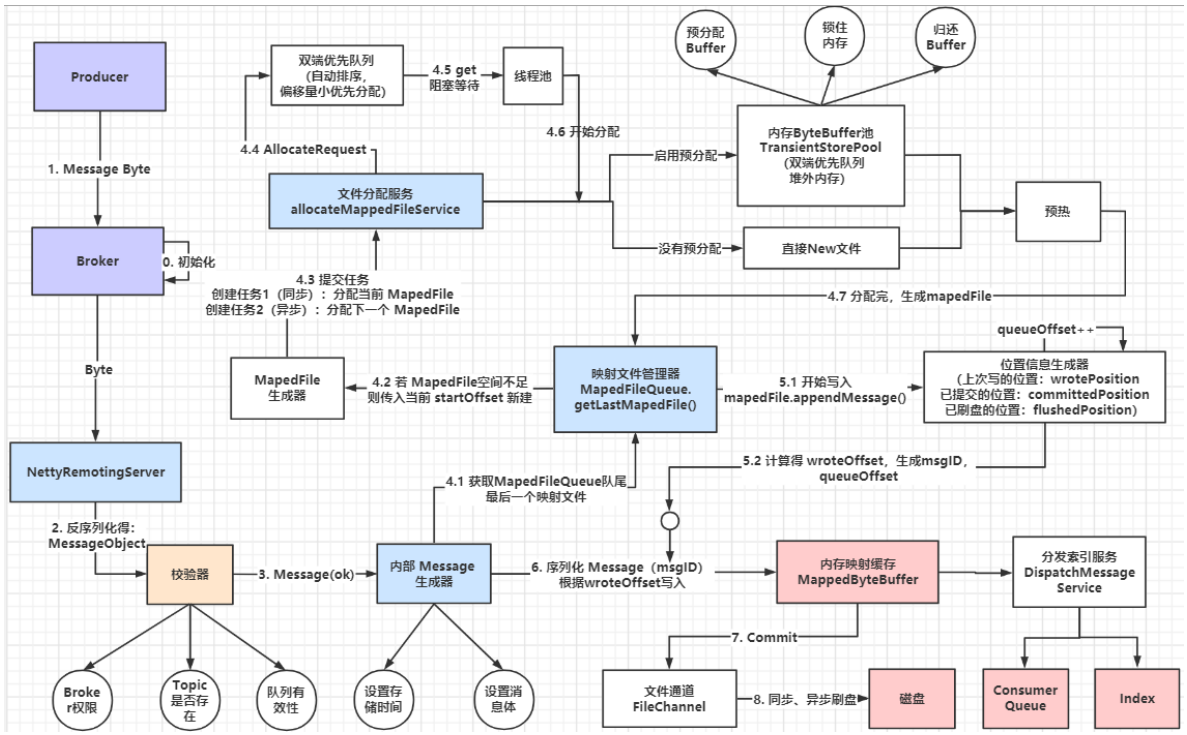
6. $6 在mappedFile中append消息，下面具体说明

```
$1 获取当前写入位置
$2 创建写缓存，放入文件的写入位置
$3 判断是单条消息还是批量消息
$4 同步写消息，fileSize-currentPos即为该文件还剩下的空白大小
$5 写完消息，累加文件当前位置
```

1. $7 根据mappedFile写消息的结果
    - ok, 直接break
    - 文件剩下的空间不够写了，重新创建一个mappedFile文件, 重新写消息
    - msg大小，properties大小，未知错误，返回错误类型
2. $8 执行刷盘
3. $9 执行主从同步

3.Broker的存储流程

【流程图】



1）Consume Queue、CommitLog

消息的逻辑队列，相当于字典的目录用来指定消息在消息的真正的物理文件 commitLog 上的位置，每个 topic 下的每个 queue 都有对应的 consumequeue 文件。consume queue 中存储单元是一个20字节定长的数据，是顺序写顺序读。commitLogOffset 是指这条消息在 commitLog 文件的实际偏移量，size就是指消息大小。可以根据消费端的逻辑偏移量首先根据定长20字节找到消费队列中的某个索引记录单元，然后取出其中的实际偏移量，以及消息的长度，就可以定位到是哪个 commitLog 文件，并且通过与文件大小求余的方式定位到具体位置，再根据消息的大小取出完整的消息。

> **consumequeue 、20字节、逻辑偏移、实际偏移、size**

2）MapedFile

MapedFile 是 PageCache 文件的封装，操作物理文件在内存中的映射以及将内存数据持久化到物理文件中。每个 commitlog 和 consumequeue 文件都对应若干个 MapedFile 对象，并最终形成一个 MapedFileQueue 存储队列，数据定时删除，无限增长。

> **commitlog、consumequeue、MapedFileQueue**

【文件组织形式】

Broker 的存储文件组织形式主要涉及到两个文件形式，一个是数据的存储文件 commitLog，它可以类比于 kafka 的 log 文件，在 Broker 接收到 Producer 的数据后，首先数据会存放到 commitLog 中。但相对于 kafka 的 log 文件，不同的是 kafka 是以分区的形式来保存各分区的数据的，全量数据被瓜分到不同 patition 的若干个 log 文件中，而 rocketmq 是将全量的数据都保存到 commitLog 文件中，即所有 Topic 的所有 Queue 的数据都发到 commitLog 文件，超过1G则新建，文件名就是该文件的最小起始偏移量，是顺序写，随机读的。

【索引文件】

顺序写可以减少寻道时间，提高写性能。但是为了解决随机读的性能问题，rocketmq 又引入了索引文件 consumerQueue 和 index 文件，主要是前者。consumerQueue 将按照 Topic 和 Queue 的维度进行划分，比如一个 topic 下有几个 queue，就会对应有几个 consumerQueue，同样地在磁盘上也会以文件夹和文件的组织方式各自对应起来，比如一个 topic 是一个文件夹，topic 下有几个代表 queue 的文件夹，文件夹中有若干个 consumerQueue 文件，同样地超过1G就新建。这样的话，就可以将全量的消息按照不同的 Topic 和 Queue 来归类，划分到各自对应的 ConsumerQueue 中。

【消息的查找】

当消息发送到commitLog后，再转发给 ConsumerQueue ，ConsumerQueue 为每个消息建立索引，所以一个 ConsumerQueue 中有若干个对应于消息的索引单元，索引中主要存储的是消息在 commitLog 中的全局物理偏移量，以及消息的长度size，还有 tag 的 hashCode，做成 hashCode 是为了保证每个索引单元有固定的长度是20个字节，这样就可以很方便灵活地在 Broker 端或者在 Consumer 端维护 CosunmerQueue 中的这个消费进度，很容易根据 consumerQueueOffset 和定长20字节定位到一个索引单元，再根据这个索引单元的 commitOffset 找到是属于哪个commit 文件，再根据偏移量和文件大小求余以及 size 找到真正的消息，进行消费。

而索引文件 Index，存储的是消息的 key 和 对应的在 consumerQueue 中的逻辑偏移量，是为了加快消息查找的效率，如果想要查找某个消息，就需要提供这个key，然后系统首先在 index 文件中取出在 consumerQueue 中的逻辑偏移量，再根据物理偏移量去 commitLog 中找到真正的消息。

【消息校验】

所以 Broker 接收消息并存储的过程，首先是 Broker 启动后加载配置，启动定时任务。然后接收 Producer 发来的消息，进行反序列化和校验，主要是校验消息的合法性和要存储的队列是否存在，以及 Broker 是否有权限存储，在主从模型中，Poducer 是通过将消息发给 Master，然后主从再通过定时任务去同步的，而从只负责读不写，如果此时是从的话，就会校验自己的权限是否可以处理这个请求，而 Consumer 则可以向所有的 Broker 发起消费请求。

【获取 MappedFile】

消息校验完后就会去 MappedFileQueue 中取得队尾元素，即取得队尾的文件。因为CommitLog消息存储、ConsumeQueue等通常会记录大量的数据，一个MappedFile具有固定大小（默认1G），所以一个MappedFile不能记录所有的内容，于是CommitLog、ConsumeQueue通常会使用多个MappedFile记录数据，RocketMQ则使用MappedFileQueue组织一系列的MappedFile，处在MappedFileQueue队尾的通常是刚写满的或者还有部分空间或者刚分配的MappedFile，每次写操作时，都会从队尾拿到最后一个MappedFile进行写。

【文件分配 - 优先阻塞队列】

此时如果队尾文件空间充足可以写入，则返回该文件指针；如果空间不够用，则需要创建文件并申请空间的分配。分配的具体流程是交给一个文件分配服务来完成的，这个服务持有一个双端优先阻塞队列，队列中的元素是一个个的分配请求 AllocateRequest，同时这个服务是一个单线程，阻塞式地等待队列中的请求到来，一旦有请求到来，就唤醒这个线程，开始处理，所以说他是阻塞的。而由于每次请求对 mappedFile 的操作都需要申请写锁，这里是读写锁分离的场景，所以请求进来时是有序的，为了保证处理请求时，文件的创建也要有序，所以AllocateRequest 对象通过实现了Comparable接口的方式，来使得放入队列中的元素可以自动排序，偏移量最小的排最前面，因为偏移量越小说明消息越早来，其对应的文件应该越早分配，以此来保证一个有序，所以说他是优先阻塞队列。

【两任务提交】

而刚才说到文件分配服务，在实际场景中，系统会提交两个文件分配的请求任务，一个是同步地创建当前的 MappedFile，一个是异步地创建下一个 MappedFile，这样下次需要新建 MappedFile 时就可以直接拿到已经预创建好的 MappedFile，提高了写的性能。

【预分配】

刚才说到文件的分配其实还涉及到一个预分配的过程，预分配就是系统会有一个缓存空间叫暂存池 TransientStorePool ，它持有一个双端队列，然后这个队列先预分配了一些内存空间 byteBuffer，这些内存空间是堆外分配的直接内存，在发生缺页中断时会发生与磁盘空间的置换，那么数据就没了。所以为了利用这部分内存，同时避免被置换，就用到了预分配的主要功能，就是分配内存后锁住，避免操作系统虚拟内存的置换。如果没有开启这个功能，将直接创建一个文件。

其实就是将 MappedFile 文件（包括 CommitLog 和 ConsumerQueue）缓存在操作系统的页缓存 PageCache 中，提高消息的读写性能，并且将这些内存锁住，防止操作系统发生缺页中断时将这部分内存置换到磁盘，丢失数据。

【预热】

文件预热使得内存提前分配，并锁定在内存中，在写入消息时不必再进行内存分配。

写入1G字节假值0是为了让系统分配物理内存空间，如果没有这些假值，系统不会实际分配物理内存，防止在写入消息时发生缺页异常。 锁定内存，防止其被交换到swap空间。

使用mmap()内存分配时，只是建立了进程虚拟地址空间，并没有分配虚拟内存对应的物理内存。当进程访问这些没有建立映射关系的虚拟内存时，处理器自动触发一个缺页异常，进而进入内核空间分配物理内存、更新进程缓存表，最后返回用户空间，恢复进程运行。

【预热流程】

- 创建新的缓冲区
- 将文件1G用字节0填充
- 进行内存锁定
- 同步异步刷盘

【预读、分发、刷盘】

分配完成后会进行预读，就是根据局部性原理，一次读多个磁盘页的数据到内存中，加大数据读取的命中率，提高读性能。最后是刷盘，通过 mappedFile 文件记录的相关位置信息计算出此次要写入磁盘的位置，位置信息比如是上次写的位置，已提交的位置，以及磁盘刷新的位置等。如果启用预分配功能，则先把缓存中的数据提交到文件通道 FileChannel，再通过同步或异步刷入磁盘。在刷盘发生前，还会将这个消息分发给索引服务，也就是我们一开始说的消息先存储在 commitLog，然后转发到 consumerQueue 和 Index 文件中，就是通过这个服务来完成的。

【PageCache】

是OS对文件的缓存，用于加速对文件的读写。一般来说，程序对文件进行顺序读写的速度几乎接近于内存的读写访问，这里的主要原因就是在于OS使用PageCache机制对读写访问操作进行了性能优化，将一部分的内存用作PageCache。

（1）**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。

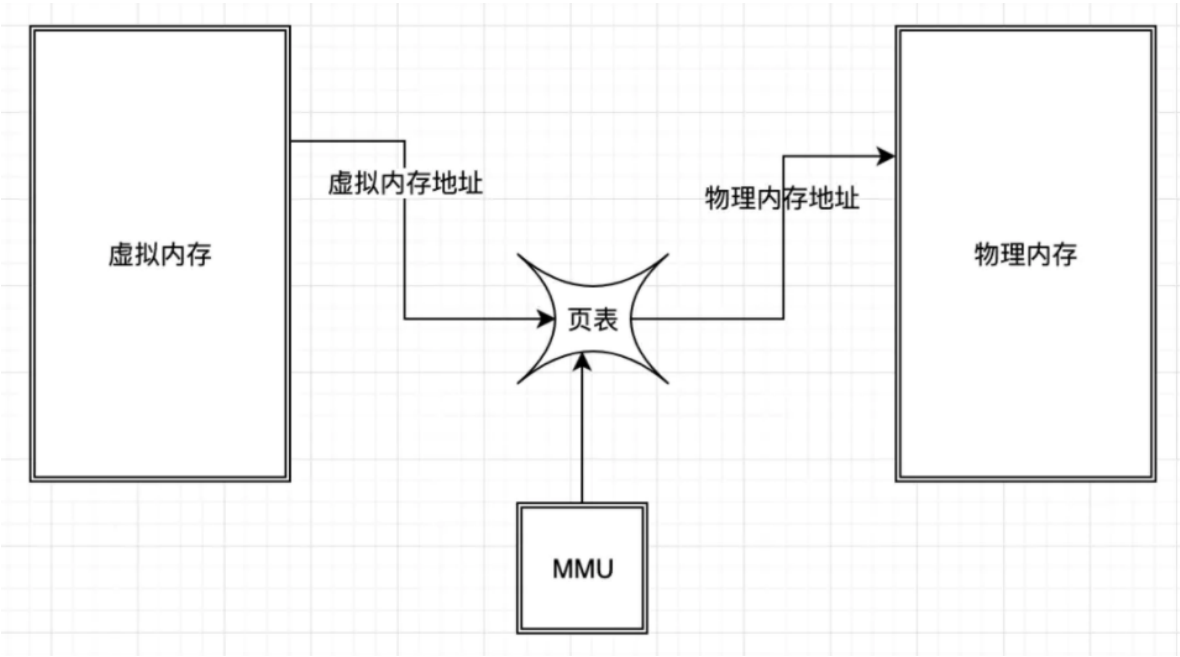
（2）**对于数据文件的写入**，OS会先写入至Cache内，随后通过异步的方式由pdflush内核线程将Cache内的数据刷盘至物理磁盘上。对于文件的顺序读写操作来说，读和写的区域都在OS的PageCache内，此时读写性能接近于内存。

RocketMQ的大致做法是，通过MMAP系统调用将磁盘文件映射到OS的虚拟内存页缓存中，写消息的时候首先写入PageCache（虚拟内存），同时锁住这部分内存防止缺页置换，优先通过异步刷盘的方式将消息批量的做持久化（同时也支持同步刷盘）；订阅消费消息时（对CommitLog操作是随机读取），由于PageCache的局部性热点原理且整体情况下还是从旧到新的有序读，因此大部分情况下消息还是可以直接从Page Cache中读取，不会产生太多的缺页（Page Fault）中断而从磁盘读取。

【MMAP】

内存与磁盘的数据传输，通常都是需要通过调用系统函数来完成的，而这个过程需要进行进程的切换，开销很大，所以 rocketMq 采取了一种 零拷贝的方式，和 kafka 一样，采用了系统的MMAP机制，通过这种系统调用，可以将文件映射到内存上，相当于在用户空间和内核空间的储存媒介上打通了一个通路，使得进程可以像访问普通内存一样访问文件，不用调用write和read等系统调用，省去了用户空间和内核空间来回拷贝数据的开销。

【内存映射】



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

虚拟地址空间的内部又被分为内核空间和用户空间两部分，进程在用户态时，只能访问用户空间内存；只有进入内核态后，才可以访问内核空间内存

MMU

MMU是Memory Management Unit的缩写，中文名是内存管理单元，它是中央处理器（CPU）中用来管理虚拟存储器、物理存储器的控制线路，同时也负责虚拟地址映射为物理地址

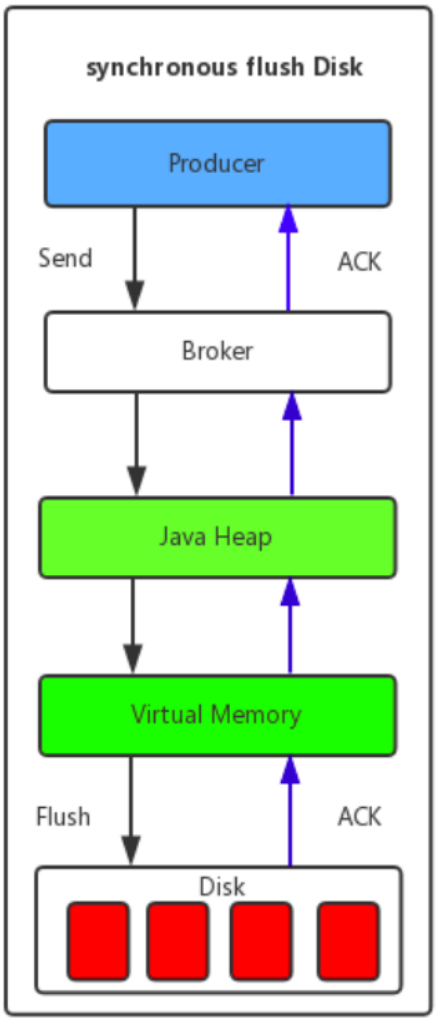
页表

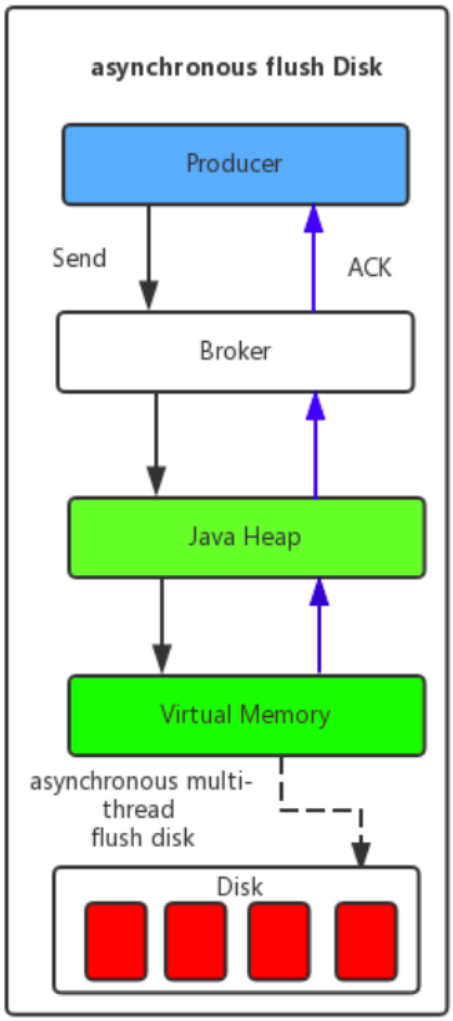
是虚拟内存系统用来存储逻辑地址和物理地址之间映射的数据结构

内存映射mmap

将虚拟地址映射到物理地址

【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。

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。

pageCache定义

Page cache 也叫页缓冲或文件缓冲，是由好几个磁盘块构成，大小通常为4k，在64位系统上为8k，构成的几个磁盘块在物理磁盘上不一定连续，文件的组织单位为一页， 也就是一个page cache大小，文件读取是由外存上不连续的几个磁盘块，到buffer cache，然后组成page cache，然后供给应用程序。

pageCache加载

操作系统操作I/O时，会先在pageCache中查找，如果未命中，则启动磁盘I/O，并把磁盘文件中的数据加载到pageCache的一个空闲快中，然后在copy到用户缓冲区

pageCache预读

对于每个文件的第一个读请求操作，系统在读入所请求页面的同时会顺序读入后面少数几个页面

pageCache与RocketMQ的关联

MQ读取消息依赖系统PageCache，PageCache命中率越高，读性能越高

ConsumeQueue逻辑消费队列是顺序读取，在pageCache机制的预读取作用下，ConsumeQueue的读性能会比较高近乎内存，即使在有消息堆积情况下也不会影响性能。

Mmap内存映射技术—MappedByteBuffer

另外，RocketMQ主要通过MappedByteBuffer对文件进行读写操作。其中，利用了NIO中的FileChannel模型直接将磁盘上的物理文件直接映射到用户态的内存地址中（这种Mmap的方式减少了传统IO将磁盘文件数据在操作系统内核地址空间的缓冲区和用户应用程序地址空间的缓冲区之间来回进行拷贝的性能开销），将对文件的操作转化为直接对内存地址进行操作，从而极大地提高了文件的读写效率

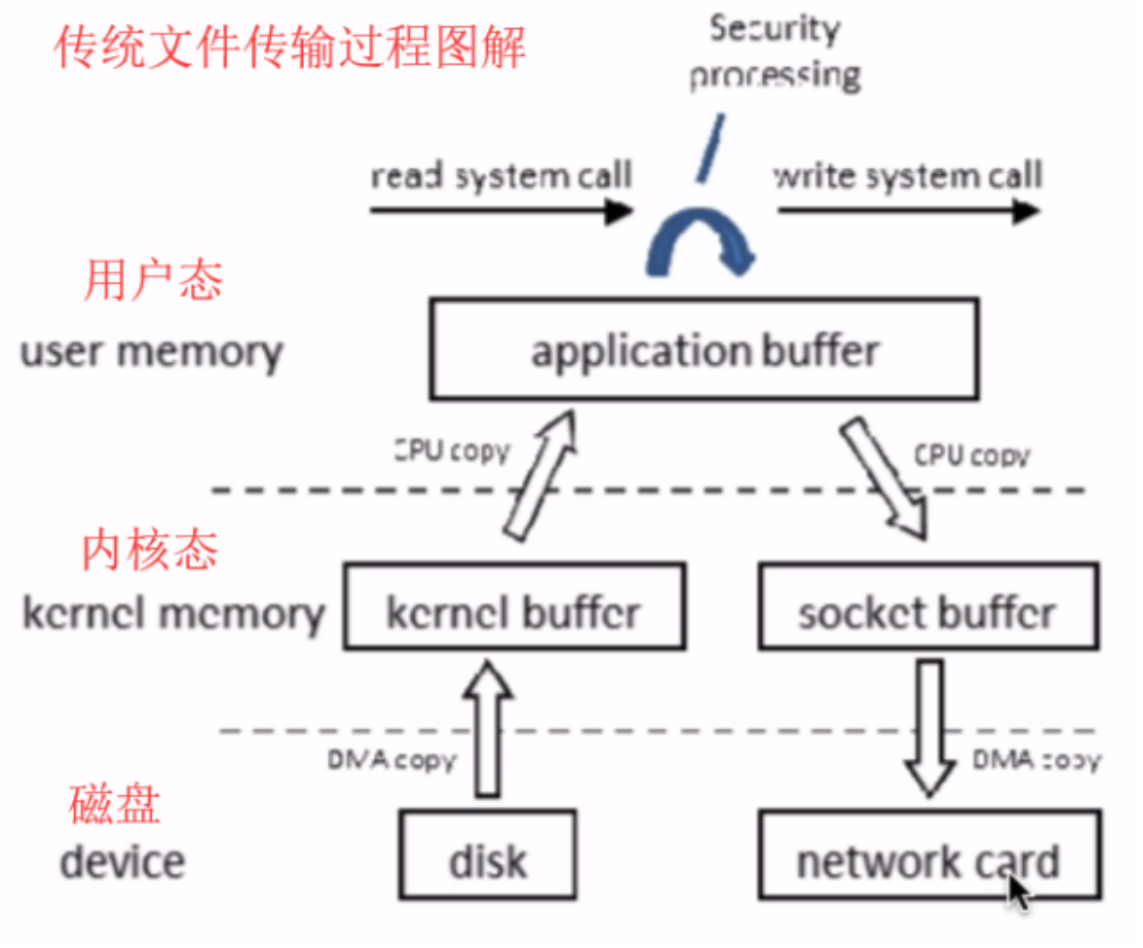
使用mmap内存映射的限制

- 每次只能映射1.5左右的文件至用户态的虚拟内存，这也是为何RocketMQ默认设置单个CommitLog日志数据文件为1G的原因
- MMAP 使用的是虚拟内存，和 PageCache 一样是由操作系统来控制刷盘的，虽然可以通过 force() 来手动控制，但这个时间把握不好，在小内存场景下会很令人头疼。
- 会存在内存占用率较高和文件关闭不确定性的问题

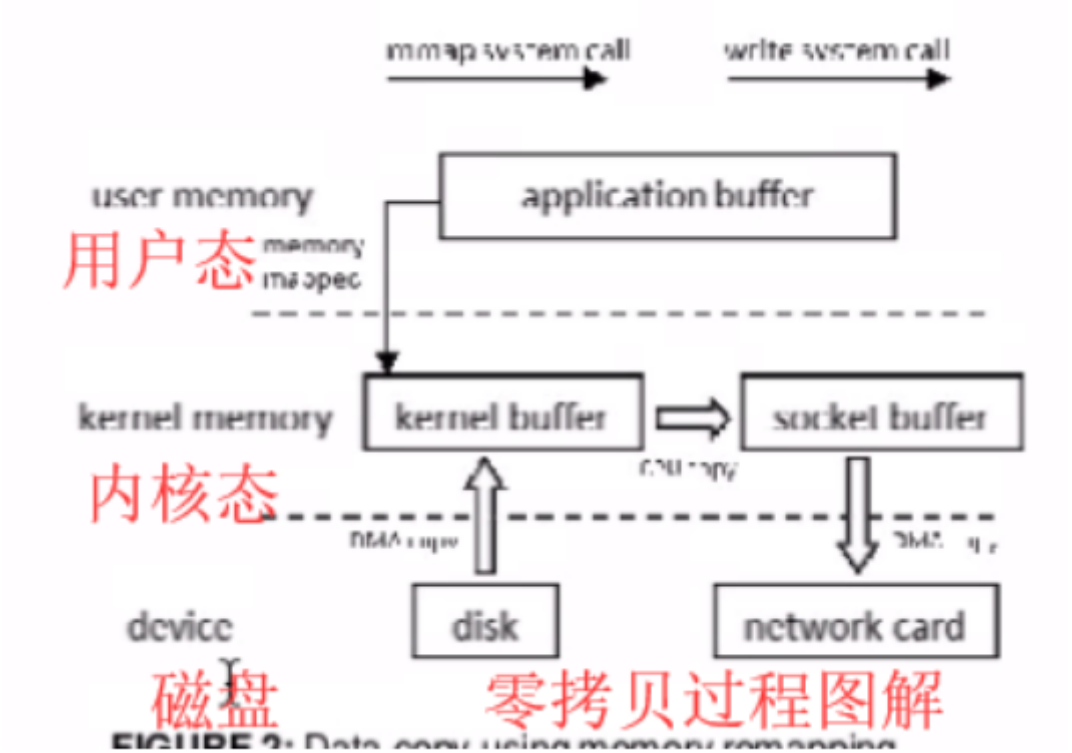
零拷贝

linux有两个上下文(内核态、用户态), 传统的将一个file读取并发送出去会经历4个过程。
read时：

1. 将文件从磁盘copy到kernel(内核)态
2. cpu将kernrl态的数据copy到user(用户)态
   write时：
3. user态的内容会copy到kernel态的socket的buffer中
4. 将kernel中buffer的数据copy到网卡中传送
   我们可以发现2、3完全是多余的步骤，而且上下文之间的切换是很耗性能的。



ZeroCopy：内核直接把磁盘的数据传输到socket，而不是通过应用程序去传输。减少了不必要的内核缓冲区和用户缓冲区间的拷贝，从而提升了性能。
零拷贝技术有mmap及sendfile；sendfile大文件传输快，mmap小文件传输快。MMQ发送的消息通常都很小，rocketmq就是以mmap+write方式实现的。像kafka、netty都采用了零拷贝技术。



RocketMQ 的主从同步机制如下：

A. 首先启动Master并在指定端口监听；

B. 客户端启动，主动连接Master，建立TCP连接；

C. 客户端以每隔5s的间隔时间向服务端拉取消息，如果是第一次拉取的话，先获取本地commitlog文件中最大的偏移量，以该偏移量向服务端拉取消息；

D. 服务端解析请求，并返回一批数据给客户端；

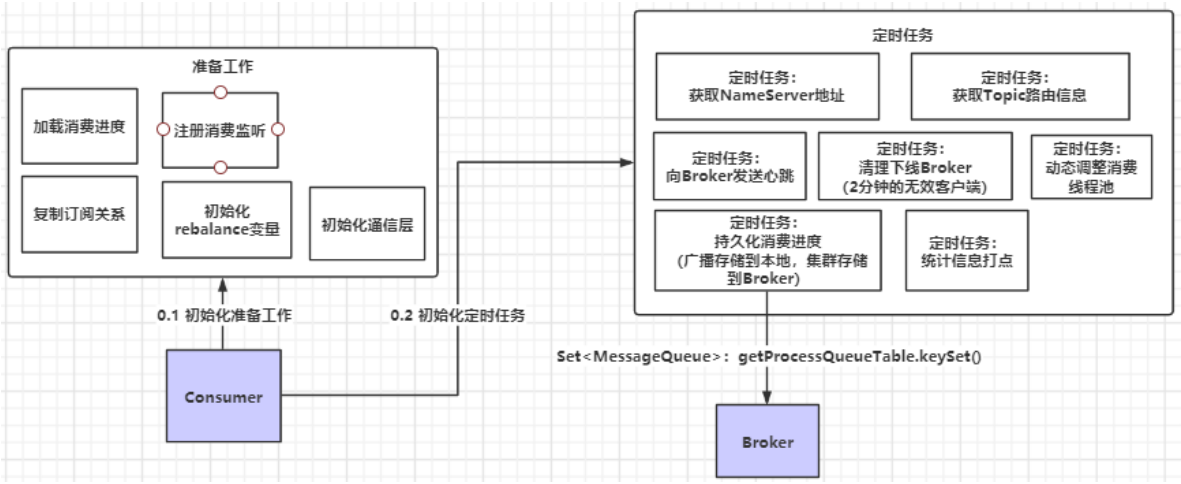
E. 客户端收到一批消息后，将消息写入本地commitlog文件中，然后向Master汇报拉取进度，并更新下一次待拉取偏移量；

F. 然后重复第3步；

4、Consumer

【流程图】

初始化



定时获取nameserver地址

定时从nameserver获取topic路由信息

定时清理下线的borker

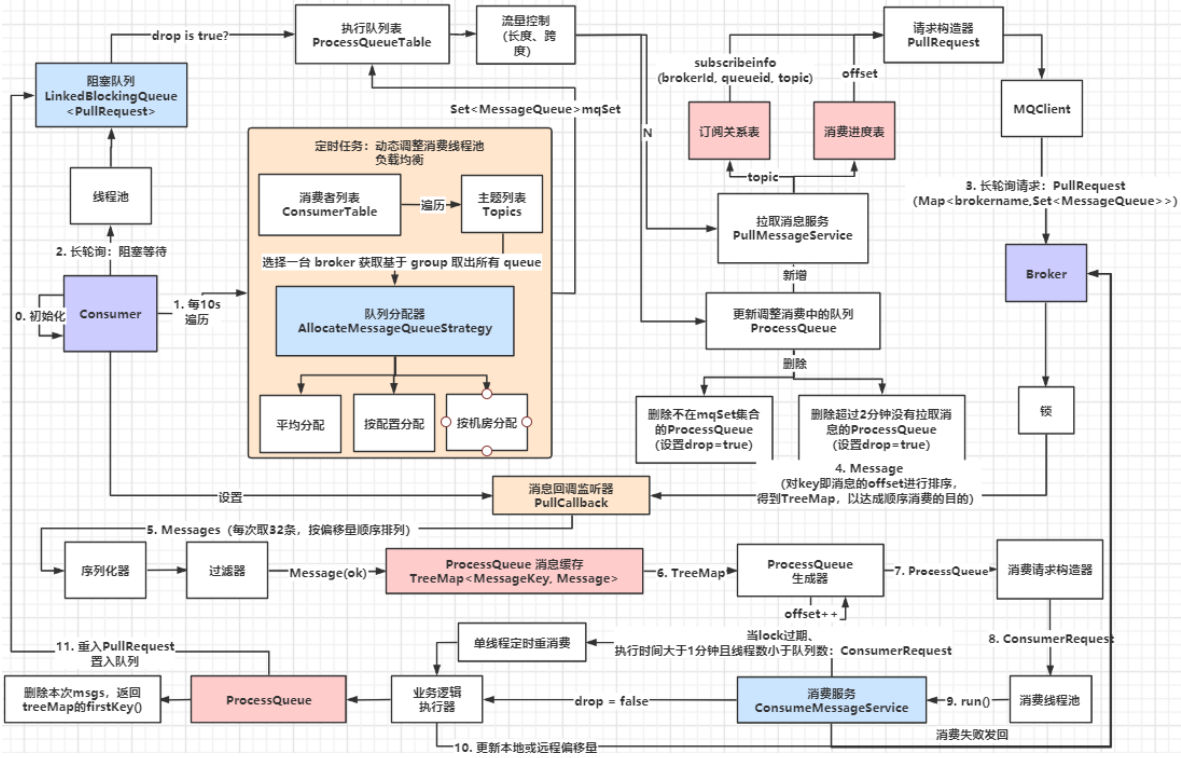
定时向所有broker发送心跳信息，（包括订阅关系）

动态调整消费线程池

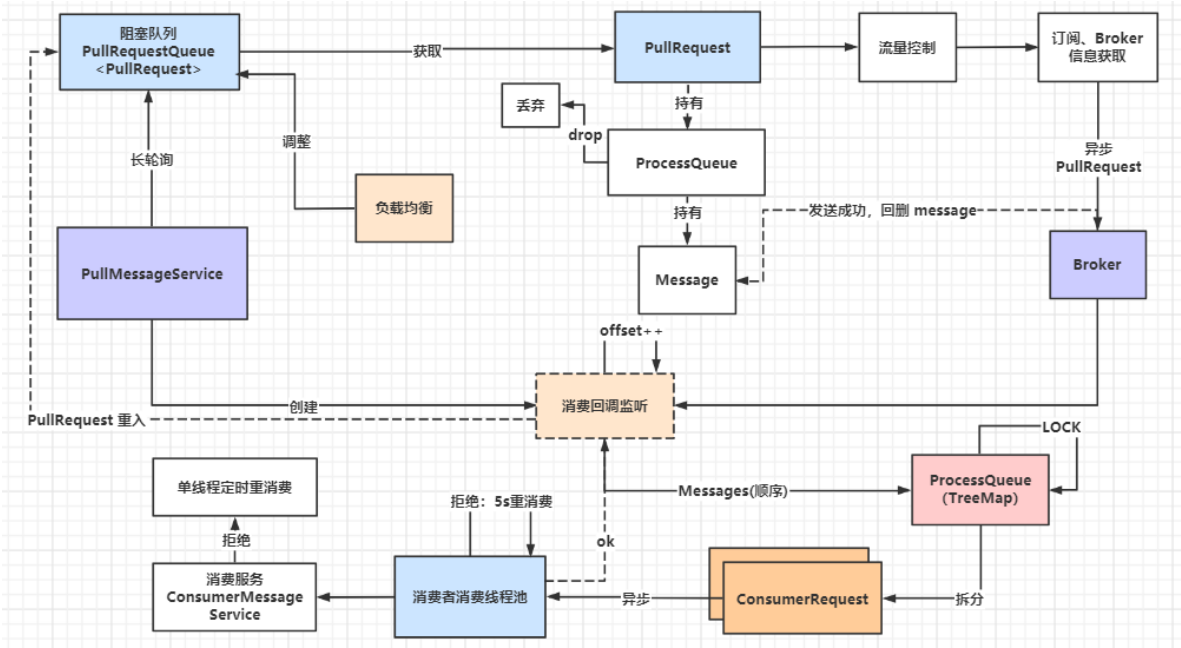
定时持久化Consumer消费进度（广播存储到本地，集群存储到Broker）

统计信息打点

详细流程



主要流程



【长轮询机制】

消息消费的过程，主要是通过长轮询机制来实现的。消费端有个阻塞队列，这个队列存放着要到 Broker 端拉取消息的请求。消费者在启动时会启动拉消息服务 PullMessageService，轮询探查队列中有没有请求，有的话就取出然后向 Broker 发送请求，没有的话就阻塞直到有拉取任务放入队列，重新唤醒线程处理。

【请求产生的时机】

而队列中产生拉取请求任务的时机有两个，一个是当执行完一次拉取任务后，Broker 返回消息并且放入消费线程池消费完了，又会将 PullRequest 对象重新放入到 PullRequestQueue 队列中，进行下一次的长轮询请求。另一个是消费者启动时会启动一个定时任务，每隔20s会进行一次消费者的负载均衡，因为启动时消费端也会启动定时任务，定时去 NameServer 通过 Topic 拉取对应的路由信息到本地，以及会上报自己消费端的信息到 NameServer ，所以如果有 Broker 或者 消费者不可用或新增了，这个时候就会拉取到最新的 Topic 和 Broker 的订阅信息和路由信息然后更新到消费端的本地，这时消费端在负载均衡时，就可以查询本地的路由表，根据最新的信息来负载均衡，如果新分配的队列中，不包含当

前正在处理的队列 ProcessQueue（当前消费的快照），则停止当前的队列的消费；如果有新增的要消费的队列时，就为该队列创建一个拉取请求 PullRequest 放入请求队列中进行长轮询，所以这就是拉取任务 PullRequest 产生的两个时机。

【流量控制】

拉取服务 PullService 从队列中获取到请求后，会经过一个流量的控制，主要的维度有两个，一个是队列中的长度，超过一定限度后则放弃本次拉取请求，延时后再重新将请求加入到队列中，这一考量是为了避免出现某一条消息阻塞住导致消息堆积；另一个维度是队列中的跨度，跨度指的是队列中消息最大和最小偏移量的距离，如果过大也视为是有消息被阻塞导致进度无法推进，也会进行流量控制。经过流控后就会获取本地的相关订阅信息和Broker路由信息，获取Broker地址异步发送。

【顺序消费】

当 Broker 收到 producer 的消息后，将根据消息的逻辑偏移地址先到消费队列中取得对应的物理偏移地址，然后再定位到实际的commitlog中的位置，再根据消息长度取出消息并返回。按照偏移量排序再发往消费端，触发消费端的消息回调接口，然后再进行消费，由于是排好序过来的，所以消费时也保证了这个顺序性。

【进入消费】

消息发到回调接口对象 callback ，后者将异步把消息先存入 ProccessQueue中。这个对象可以看做是消息消费的快照文件，存储了消息消费过程中的一些信息，以便系统进行一个全面的监控，比如有消息后面要放到线程池去处理，就需要了解消息的堆积情况，比如消息处理时是否超时，以及消息是否需要被丢弃（drop）等，在 Broker 返回消息后，如果当前处理队列查看到消息不是由本队列处理的，即不是当前队列感兴趣的消息时，就会丢弃。这个执行队列还持有一个TreeMap和锁，拉取到的消息先放入这个TreeMap中，通过锁来进行多线程并发访问TreeMap的控制。

然后生成消费请求，如果拉取到的消息过长，就会生成多个 ConsumerRequest，放入消费者的消费线程池中。如果提交出现异常，则延迟5s再提交。使用线程池可以对消息拉取和消费进行一个解耦，而且拉取到消息后是通过异步方式去消费的，提高了消费端的吞吐量。异步提交消费任务后，会更新消费进度，广播模式是保存在消费端，因为每个消费端消费消息时是相互独立的，所以每个消费者需要有自己的消费进度，集群模式下消息是被同个消费组的消费者均分平摊的，所以多个消费者需要共同感知消费进度，此时进度要发给Broker去维护。

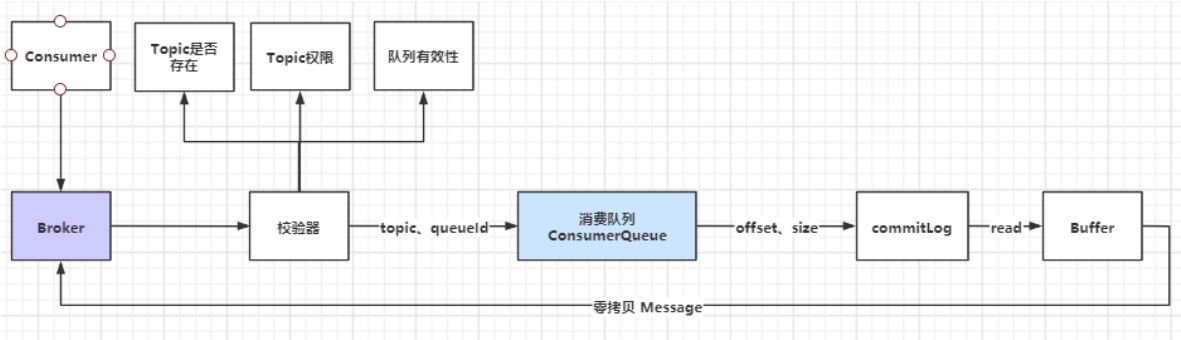
【消息重试重消费】

提交到线程池的任务，会启动线程启动消费服务来消费，执行队列 ProcessQueue 持有一把锁，如果消费的时间过长导致锁过期，或者当前线程池的数量小于执行队列的数量，那么消费服务会另起一个单线程，将这些失败的请求放到线程中，等待定时重新消费，这样是为了避免某个执行队列产生饥饿现象，太久没有消费请求到达。同时会回复需要重消费的ACK给Broker，Broker收到后重置该消息的主题为重试，设置重试次数，然后存到commitlog，等到消费端定时任务重新发起消费。

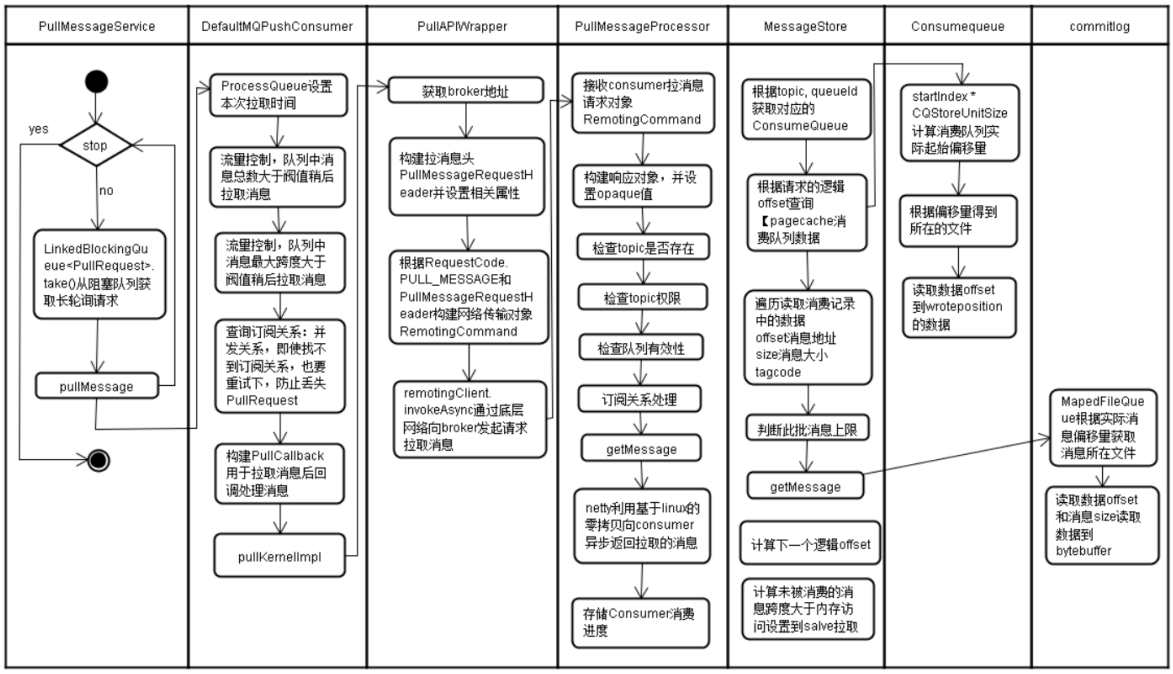
【ACK】

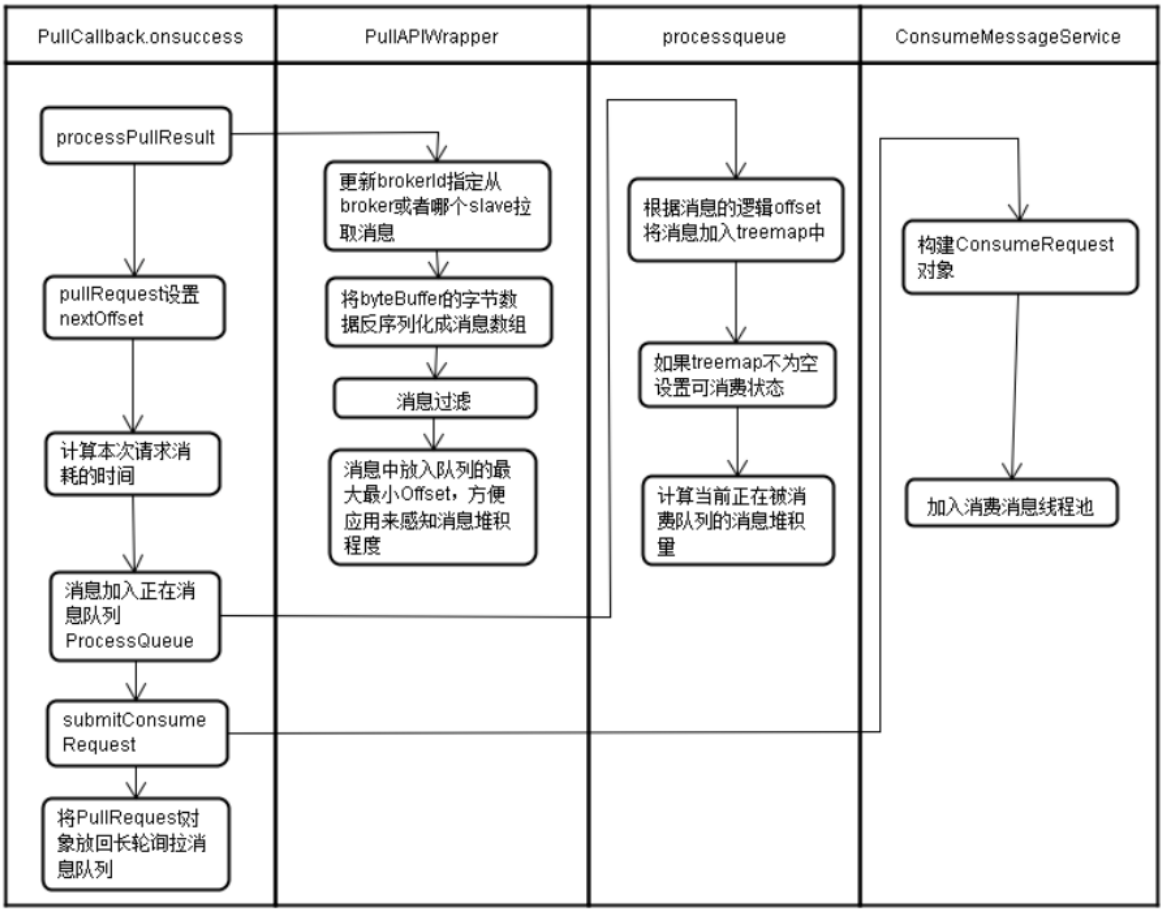
首先要明确一点：**ACK机制是发生在Consumer端的，不是在Producer端的。**也就是说Consumer消费完消息后要进行ACK确认，如果未确认则代表是消费失败，这时候Broker会进行重试策略（仅集群模式会重试）。ACK的意思就是：Consumer说：ok，我消费成功了。这条消息给我标记成已消费吧。

消息消费

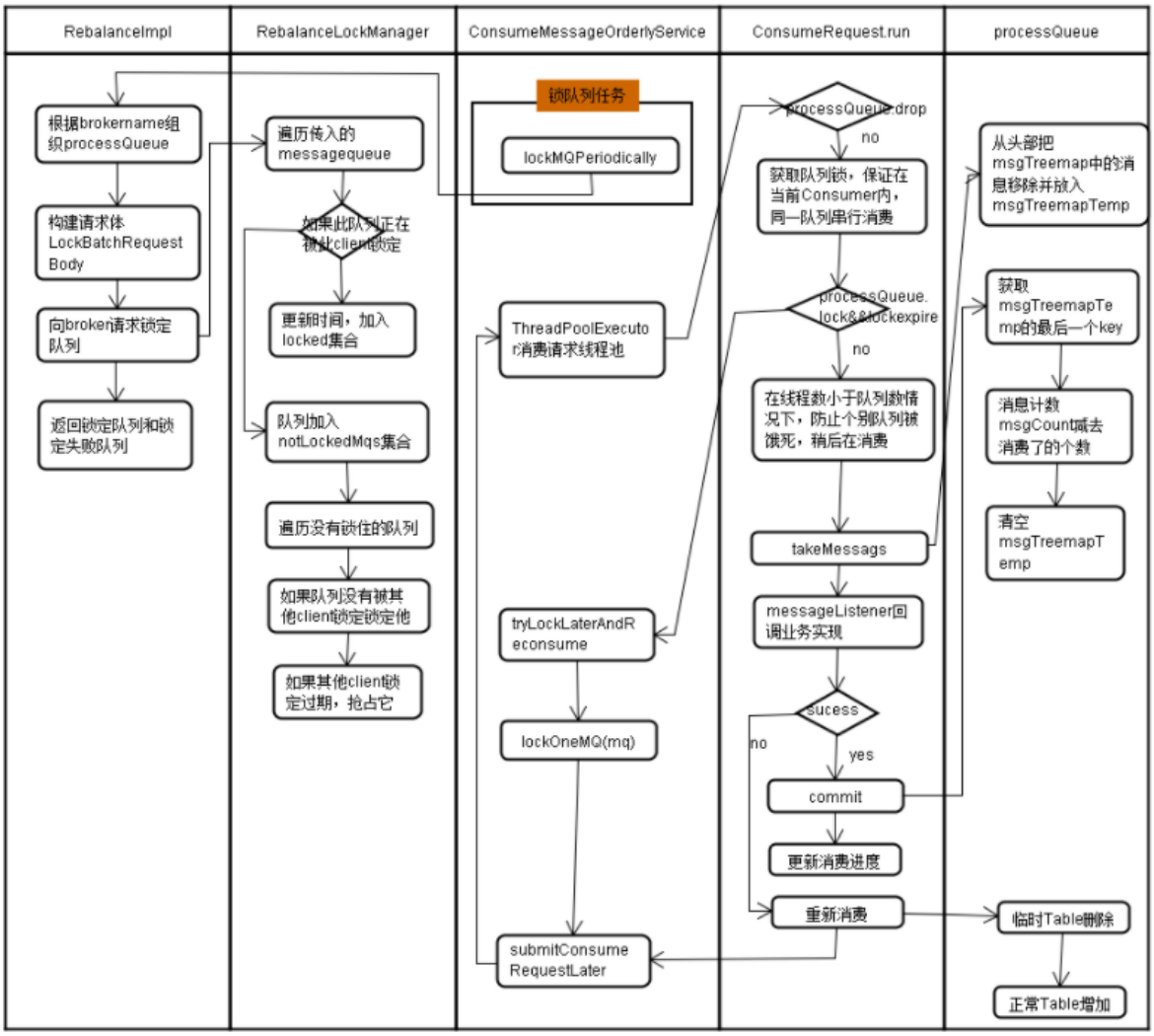

Consumer
Topic是否存在
Topic权限
队列有效性
Broker
校验器
topic、queueId
消费队列 ConsumerQueue
offset、size
commitLog
read
Buffer
零拷贝 Message

【活动图】


PullMessageService
DefaultMQPushConsumer
PullAPIWrapper
PullMessageProcessor
MessageStore
Consumequeue
commitlog
yes
stop
no
LinkedBlockingQueue<PullRequest>.take()从阻塞队列获取长轮询请求
pullMessage
ProcessQueue设置本次拉取时间
流量控制，队列中消息总数大于阈值稍后拉取消息
流量控制，队列中消息最大跨度大于阈值稍后拉取消息
查询订阅关系：并发关系，即使找不到订阅关系，也要重试下，防止丢失PullRequest
构建PullCallback用于拉取消息后回调处理消息
pullKernelImpl
获取broker地址
构建拉消息头PullMessageRequestHeader并设置相关属性
根据RequestCode.PULL_MESSAGE和PullMessageRequestHeader构建网络传输对象RemotingCommand
remotingClient.invokeAsync通过底层网络向broker发起请求拉取消息
接收consumer拉消息请求对象RemotingCommand
构建响应对象，并设置opaque值
检查topic是否存在
检查topic权限
检查队列有效性
订阅关系处理
getMessage
netty利用基于linux的零拷贝向consumer异步返回拉取的消息
存储Consumer消费进度
根据topic, queueId获取对应的ConsumeQueue
根据请求的逻辑offset查询【pagecache消费队列数据
遍历读取消费记录中的数据offset消息地址size消息大小tagcode
判断此批消息上限
getMessage
计算下一个逻辑offset
计算未被消费的消息跨度大于内存访问设置到salve拉取
startIndex * CQStoreUnitSize计算消费队列实际起始偏移量
根据偏移量得到所在的文件
读取数据offset到wroteposition的数据
MapedFileQueue根据实际消息偏移量获取消息所在文件
读取数据offset和消息size读取数据到bytebuffer

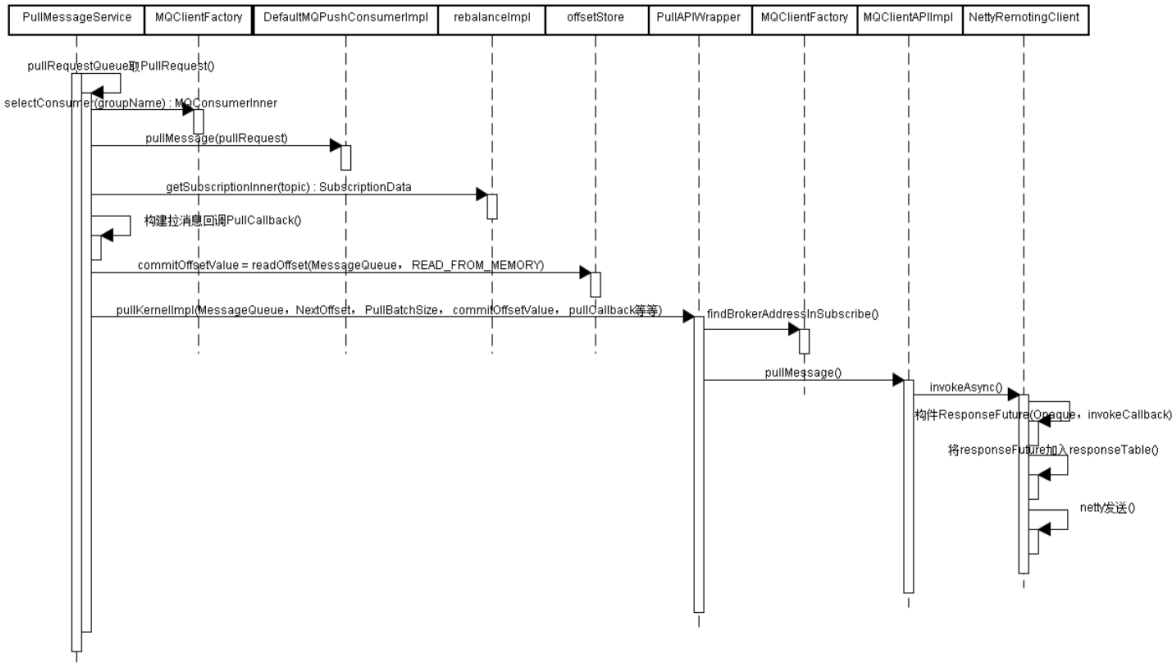

PullCallback.onsuccess
PullAPIWrapper
processqueue
ConsumeMessageService
processPullResult
pullRequest设置nextOffset
计算本次请求消耗的时间
消息加入正在消息队列ProcessQueue
submitConsumeRequest
将PullRequest对象放回长轮询拉消息队列
更新brokerId指定从broker或者哪个slave拉取消息
将byteBuffer的字节数据反序列化成消息数组
消息过滤
消息中放入队列的最大最小Offset，方便应用来感知消息堆积程度
根据消息的逻辑offset将消息加入treemap中
如果treemap不为空设置可消费状态
计算当前正在被消费队列的消息堆积量
构建ConsumeRequest对象
加入消费消息线程池

顺序消费

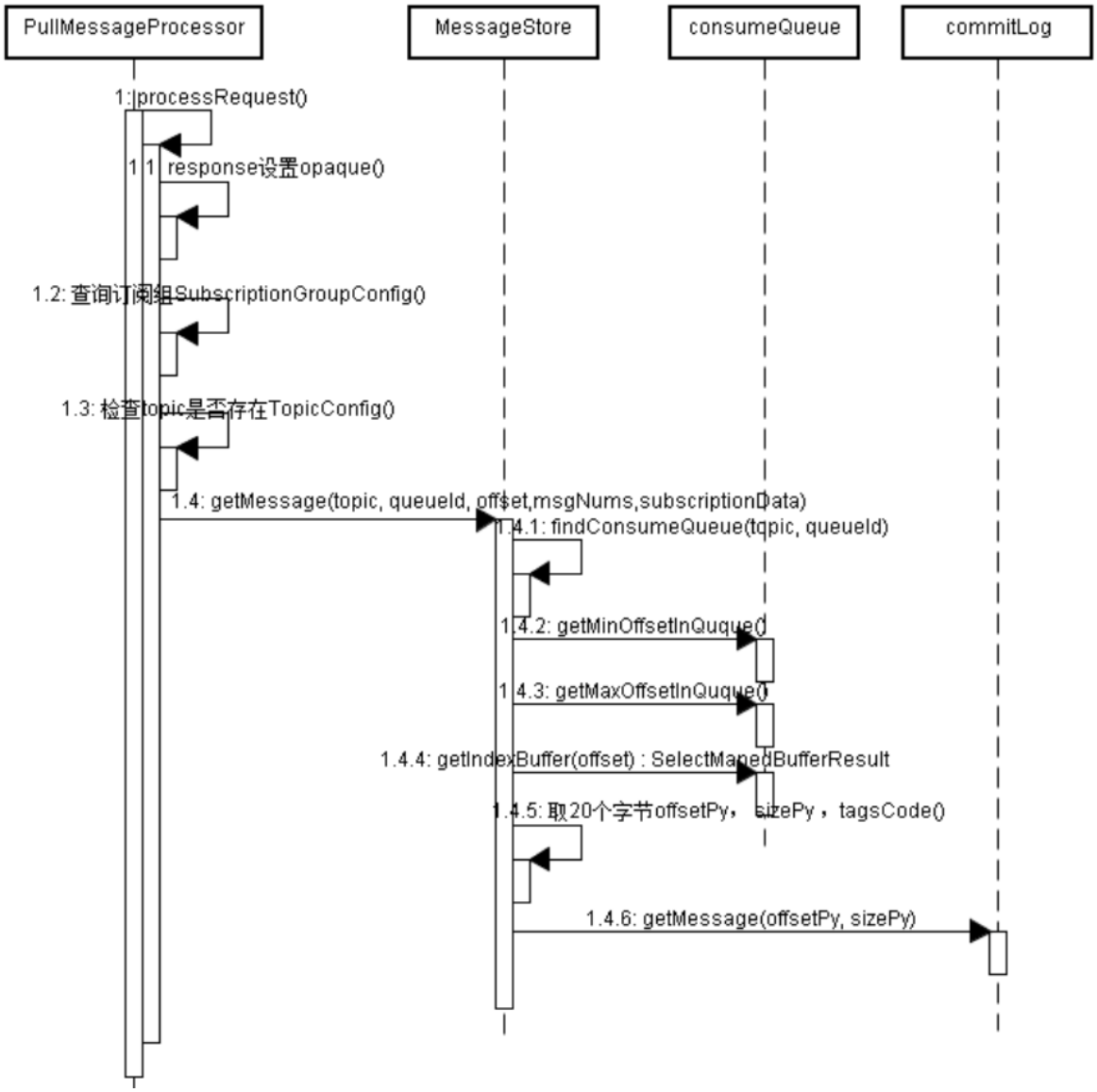


【时序图】

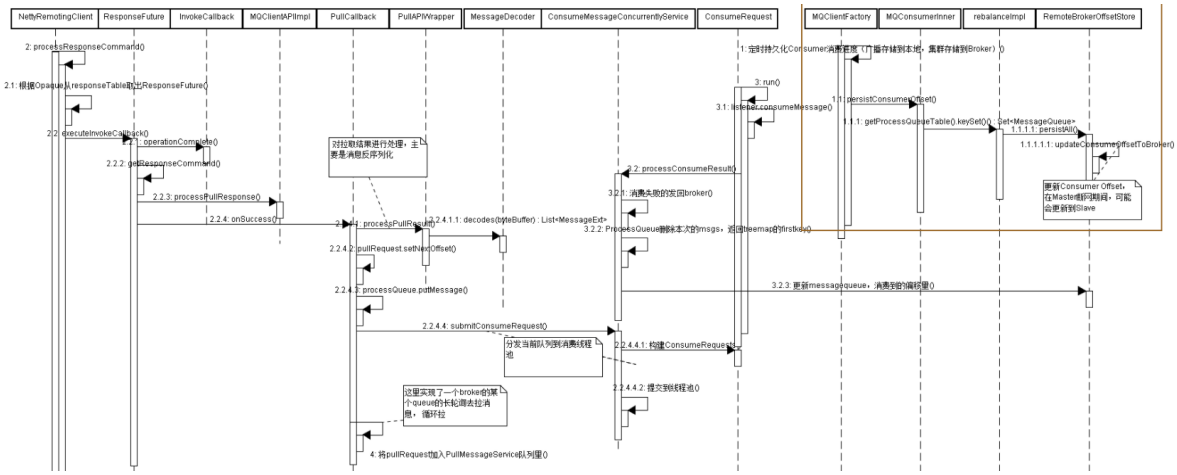
Consumer -> Broker



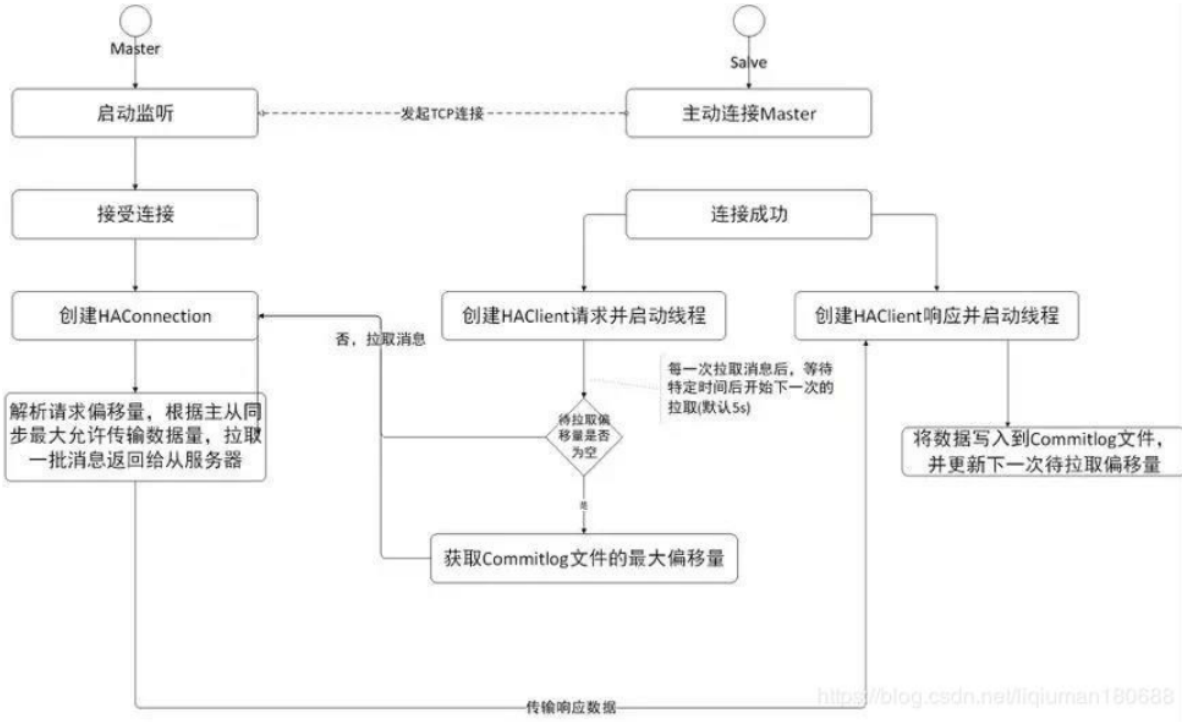
Broker 接收



Broker - > Consumer



【主从同步】



5、总结

【NameServer】

工作机制（作用）

每个 Broker 启动后会定时向 namesrv 注册 topic 信息。

Producer 和 Consumer 会定时（30s）到 namesrv 拉取 Broker 的路由信息（Topic 跟 Broker的映射关系）到本地并进行更新。

Producer 和 Consumer 启动后会定时和所有 Broker 建立心跳，发送 Producer 的信息、和 Consumer 的订阅信息（订阅关系，哪些消费者订阅了哪些主题）。

NameServer持有一个核心控制器NamesrcController，由该控制器主持NameServer的启动流程。

启动流程

启动时，首先会加载配置信息，初始化通信层，底层是netty框架，所以是创建一个nettyClient，用于和生产者、消费者、Broker的通信。

然后是初始化线程池，单线程启动一个定时任务，定时清理不活跃的Broker，具体就是更新一个Broker活跃表BrokerLivedTable，将不活跃的对象剔除出去。通过调用 RouteInfoManger 的 scanNotActiveBroker 方法来定时清理不活动的broker，默认两分钟如果Broker没有向 namesrv 发送心跳更新 BrokerLiveInfo 时间戳的，则认为是过期，关闭channel连接。

> **核心控制器（加载配置信息、初始化通信层，创建 netty 客户端）**
>
> **线程池定时任务（定期清理不活跃的 Broker，2分钟没心跳包来往的客户端）**

【Producer】

发送普通消息

消息发送时，首先初始化一些参数配置，比如消息的组名 GroupName 以及 NameServer 的地址等。

然后根据 Topic 获取对应的 Broker 的路由信息，先从本地的路由信息表 TopicPublishInfoTable 中查找，如果没有找到目标的 Topic，再发心跳包到 NameServer，查找该 Topic 的路由信息，拉取到本地再进行相应的更新。

获取到路由信息后，准备发送到目标 Broker，这时可能存在一个 Topic 对应多个 Broker 多个 Queue 的情况，所以需要进行负载均衡。常见的算法有两种，一种是正常情况下的队列轮询算法，一种是要顺序消费时，通过相同的某个关键字，比如可以是订单id，然后对消息队列个数求余来达到相同订单id发往同一个队列的目的。

轮询时还会进行一个队列规避的动作，意思是上一次发送失败的 Broker 不参与此负载均衡，所以上次失败的 Broker 对应的 Queue 就会被规避。以保证一定的高可用性。发送失败则会重试，发送方式有 sendOneway > sendCallBack > send批量 > send单条 。

> **获取路由信息（本地获取、远程获取）**
>
> **负载均衡（轮询、哈希或求余）**

发送事务消息

发送事务消息，是两阶段提交的过程，这个过程 Producer 会向 Broker 发送两条消息，而 Consumer 最终消费一条消息的过程。

通俗地说，首先 Producer 先发消息和 Broker 说我准备好执行事务操作了，此时发过去的消息带有一个状态标识是 prepared，Broker 收到 prepared 状态后回复 Producer 表示已知悉，此时消息并未提交到消费队列，消费者并不能消费。

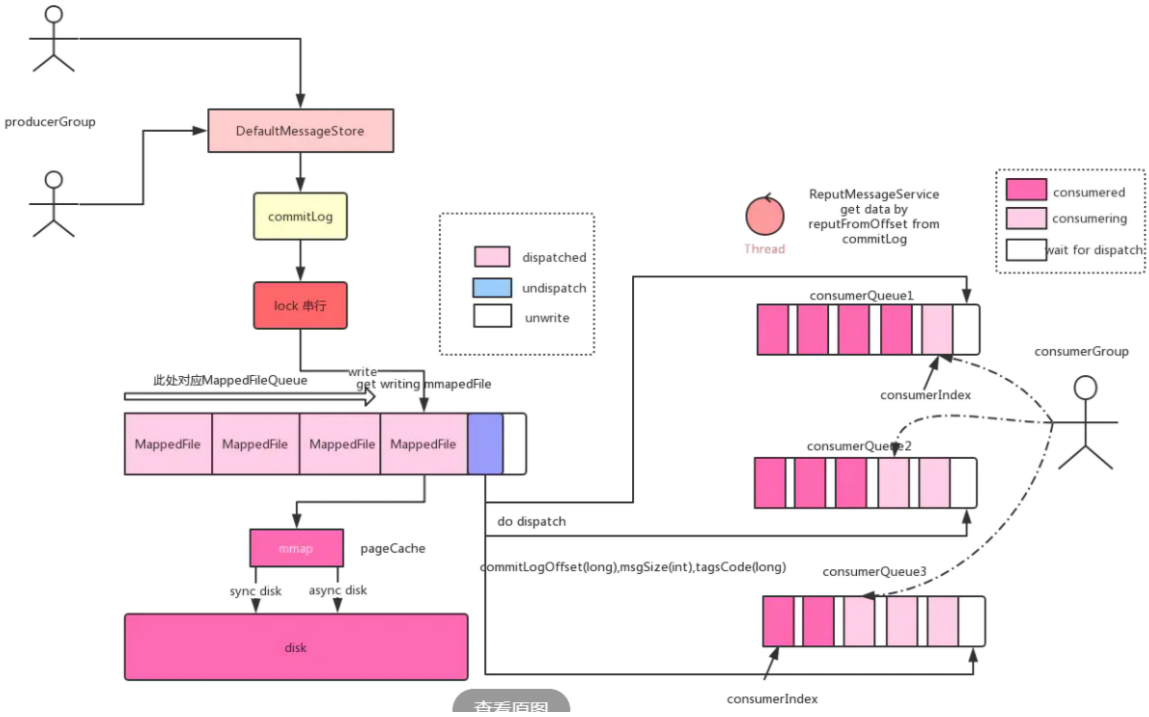
Producer 收到回复后，开始执行事务，执行完后再和 Broker 说我完成事务了，此时发过去的消息带有一个状态标识是 commit ，表示可以提交了。然后Broker 收到后就将消息提交到 Consumer Queue 消费队列中，此时对消费者可见，消费者才能真正消费。

如果由于其他原因，超过一定时间后，Producer 没有对消息进行二次确认，那么 Broker 会通过我们实现的一个回调接口进行状态的回查，主动向 Producer 询问消息的状态。Producer 查询后，再向 Broker 汇报，如果本地事务失败，那么 Product 向 Brock 发送 状态Rollback，那么 Broker 就会直接删除上面这条半消息。

消费者执行后也会发送消息到队列中，然后生产者去消费，如果失败，则要进行重试。

> **Producer 发送 prepared 消息、执行本地事务**
>
> **Producer 发送 commit 消息、Broker 提交到日志文件和消费队列**
>
> **Broker 通过回调接口主动查询 Producer 消息状态**

【Broker】



Broker 启动流程

Broker 启动时首先会加载一连串的配置信息，比如 Topic 的消费进度，Topic 的配置信息（Topic 和 Broker 的对应关系），消费者的订阅信息（哪些消费者订阅了哪些 Topic 的哪些 Queue），以及将内存中的数据刷新到磁盘时的存储检查点等。

第二步是启动一些定时任务，比如

- 定时刷新消费进度
- 定时注册 Broker 和 Topic 的信息（二者的对应关系）到 NameServer中
- 定时清理下线的客户端，比如每2分钟会和 Producer 和 Consumer 维持心跳，如果超时没有心跳，则会认为客户端下线了，会将其清理出本地维护的状态表中
- 定时获取 NameServer 地址并更新
- 定时主从同步

第三部是初始化通信层，建立NettyClient客户端

> **加载配置信息（topic 消费进度、topic 配置信息、消费者订阅信息）**
>
> **启动定时任务（获取 nsv 地址、注册 Broker 和 Topic 信息到 nsv、清理客户端、刷新消费进度、主从同步）**
>
> **初始化通信层，建立 Netty 客户端**

存储模型

Broker 的存储文件组织形式主要涉及到两个文件形式，一个是数据的存储文件 commitLog，它可以类比于 kafka 的 log 文件，在 Broker 接收到 Producer 的数据后，首先数据会存放到 commitLog 中。但相对于 kafka 的 log 文件，不同的是 kafka 是以分区的形式来保存各分区的数据的，全量数据被瓜分到不同 patition 的若干个 log 文件中，而 rocketmq 是将全量的数据都保存到 commitLog 文件中，即所有 Topic 的所有 Queue 的数据都发到 commitLog 文件，超过1G则新建，文件名就是该文件的最小起始偏移量，是顺序写，随机读的。

> **commitLog 类比 log，分区保存各log，全量保存一个 commitLog**

索引文件

顺序写可以减少寻道时间，提高写性能。但是为了解决随机读的性能问题，rocketmq 又引入了索引文件 consumerQueue 和 index 文件，主要是前者。consumerQueue 将按照 Topic 和 Queue 的维度进行划分，比如一个 topic 下有几个 queue，就会对应有几个 consumerQueue，这样的话，就可以将全量的消息按照不同的 Topic 和 Queue 来归类，划分到各自对应的 ConsumerQueue 中。

> **consumerQueue 将按照 Topic 和 Queue 的维度进行划分**

消息的查找

当消息发送到commitLog后，再异步转发给 ConsumerQueue ，ConsumerQueue 为每个消息建立索引，所以一个 ConsumerQueue 中有若干个对应于消息的索引单元，索引中主要存储的是消息在 commitLog 中的全局物理偏移量，以及消息的长度size，这样就可以根据消费端的逻辑偏移量首先根据定长20字节找到消费队列中的某个索引记录单元，然后取出其中的实际偏移量，以及消息的长度，就可以定位到是哪个 commitLog 文件，并且通过与文件大小求余的方式定位到具体位置，再根据消息的大小取出完整的消息。

而索引文件 Index，存储的是消息的 key 和 对应的在 consumerQueue 中的逻辑偏移量，是为了加快消息查找的效率，如果想要查找某个消息，就需要提供这个key，然后系统首先在 index 文件中取出在 consumerQueue 中的逻辑偏移量，再根据物理偏移量去 commitLog 中找到真正的消息。

> **逻辑偏移量、20字节、实际偏移量、索引单元、size**

获取 MappedFile

Broker 收到消息后首先需要把消息写入文件，在 RockerMQ 中与文件操作相关的对象是 MapedFile ，MapedFile 是 PageCache 文件的封装，操作物理文件在内存中的映射以及将内存数据持久化到物理文件中。每个 commitlog 和 consumequeue 文件都对应若干个 MapedFile 对象，并最终形成一个 MapedFileQueue 存储队列，数据定时删除，无限增长。处在 MappedFileQueue 队尾的通常是刚写满的或者还有部分空间，所以每次写操作时，都会从队尾拿到最后一个 MappedFile 进行写。

> **PageCache 文件的封装、MapedFileQueue、队尾**

文件分配 - 优先阻塞队列

此时如果队尾文件空间充足可以写入，则返回该文件指针；如果空间不够用，则需要创建文件并申请空间的分配。分配的具体流程是交给一个文件分配服务来完成的，这个服务持有一个双端优先阻塞队列，队列中的元素是一个个的分配请求 AllocateRequest，同时这个服务是一个单线程，阻塞式地等待队列中的请求到来，一旦有请求到来，就唤醒这个线程，开始处理，所以说他是阻塞的。而由于每次请求对 mappedFile 的操作都需要申请写锁，这里是读写锁分离的场景，所以请求进来时是有序的，为了保证处理请求时，文件的创建也要有序，所以AllocateRequest 对象通过实现了Comparable接口的方式，来使得放入队列中的元素可以自动排序，偏移量最小的排最前面，因为偏移量越小说明消息越早来，其对应的文件应该越早分配，以此来保证一个有序，所以说他是优先阻塞队列。

> **空间分配、双端优先队列、读写锁、偏移量排序**
>
> 文件的分配服务会为到来的请求创建文件，并将磁盘的地址空间与当前内存的虚拟地址映射起来。请求到来的时候首先是进入一个队列中，这个队列根据消息偏移量来排队，这样的好处就是使得先来的消息先分配，达到一个按顺序分配的目的。因为偏移量越小说明消息越早来。
>
> 分配的时候只是将两个地址进行一个简单的映射，但实际上并没有分配内存空间，这样的话当真正访问到这片虚拟内存地址时就会产生缺页中断，然后从磁盘加载数据进来，这样的效率就太低了，所以 rocketmq 采取了一种预热的机制，在文件分配好以后，就先写入 1G 的比特位的 0 值，以此来提前触发 OS 为当前的虚拟地址空间分配内存，才不会等到真正访问到的时候才发生缺页中断然后去分配内存。

两任务提交

而刚才说到文件分配服务，在实际场景中，系统会提交两个文件分配的请求任务，一个是同步地创建当前的 MappedFile，一个是异步地创建下一个 MappedFile，这样下次需要新建 MappedFile 时就可以直接拿到已经预创建好的 MappedFile，提高了写的性能。

预热

因为 rocketmq 使用 mmap 机制来将磁盘的地址映射到进程的虚拟地址中，使得程序可以直接将数据写入内存，然后再映射到磁盘中。但一开始新建文件，文件进行分配初始化的时候，只是简单地将文件磁盘地址和进程虚拟地址做了个映射，并没有为文件实际分配物理内存，当程序真正访问这片内存时产生缺页异常，这时候才会将文件的内容拷贝到page cache，但如果每次等到要写入的时候才发生缺页异常再将磁盘的数据加载到内存，效率会很低。所以 rocketmq 采用一种预热的机制，就是新建文件后会立马写入1G字节的0值，每个文件最大1G，这样的话就可以让系统提前为文件分配物理内存空间，防止在写入消息时发生缺页异常。同时预热后，又通过 mlock 系统调用锁住这部分内存，防止其被交换到交换区。

堆外内存

写消息时先写到堆外内存，纯内存操作非常快。读消息时是从pagecache中读，相当于实现了读写分离。

刷盘

通过MMAP系统调用将磁盘文件映射到OS的虚拟内存页缓存中，写消息的时候首先写入PageCache（虚拟内存），然后使用 MMAP 系统调用将内存数据持久化到磁盘，这样省去了数据在用户态和内核态的来回拷贝的开销，即零拷贝机制，同时锁住这部分内存防止缺页置换。最后是刷盘，通过 mappedFile 文件记录的相关位置信息计算出此次要写入磁盘的位置，位置信息比如是上次写的位置，已提交的位置，以及磁盘刷新的位置等。

> **写页缓存、零拷贝到磁盘、刷盘**

1）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。

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。

主从同步

- 首先启动 Master 主服务器并在指定端口监听；
- 从服务器启动，主动连接 Master 主服务器，建立TCP连接；
- 从服务器以每隔5s的间隔时间向服务端拉取消息，如果是第一次拉取的话，先获取本地commitlog文件中最大的偏移量，以该偏移量向服务端拉取消息；
- 主服务器解析请求，并返回一批数据给从服务器；
- 从服务器收到一批消息后，将消息写入本地commitlog文件中，然后向主服务器汇报拉取进度，并更新下一次待拉取偏移量；
- 然后重复第3步；

> **主服务器监听、从服务器连接**
>
> **从服务器定时拉取，第一次发送最大偏移量；主服务器接收并回发**
>
> **从服务器写入 commitlog，再汇报拉取进度偏移量，并更新本地偏移量**

【Consumer】

消费者启动后，会启动一些定时任务。

首先是定时地从 NameSrv 中拉取 Topic 的路由信息保存到本地，并且提取每个 Topic 对应的队列信息保存在本地的订阅信息表中，为后面的负载均衡作准备。然后是定时地和 Boker 保持心跳，并且把当前的消费者节点信息、订阅关系以及所属组的信息发给 Broker，所以如果有新的消费者产生的话，Broker 端会保存这些消费者的信息。

然后是每隔 20s 定时地针对每个消费组做一次负载均衡，要做负载均衡的话需要两样东西，一个是所有 topic 的队列信息，另一个是当前消费组内的所有消费者id。其中 Topic 的队列信息已经在定时任务中从 NameSrv 取得，而至于当前消费者组的所有消费者 id，就需要向 Broker 发送请求，通过发送 Topic 和 消费者组给 Broker，Broker 将对应的消费者 id 列表返回，因为在消费者启动时就把自己的订阅信息、组信息等发给 Broker，所以这里 Broker 是能返回这些消费者 id 的。

获取到队列信息和消费者id后，分别对队列信息和消费者 id 进行字典排序，然后通过负载均衡算法重新计算出每个消费者所对应的消费队列，默认的负载均衡算法是平均分配法。如果是第一次分配的话，那么就会为当前消费者的若干消费队列都一一生成拉取请求，然后放入一个阻塞队列中，这个队列由一个消息拉取服务轮询去取任务。如果不是第一次负载均衡的话，那么可能存在由于 Broker 或者 Consumer 的新增或减少而产生了新的消费分配列表，然后消费端会对比新旧两张分配表，如果发现有新增的队列则为新增的队列新建一个请求放入请求队列中，如果发现有减少的队列，那么就会停止相应的已经不存在的队列的消费，比如在拉取服务从队列中拉取请求的时候，发现这个请求对应的队列在 Broker 端已经不存在了，就会丢弃这个请求。

当负载均衡完毕后，消息拉取服务就会从阻塞队列中逐个取出拉取请求，然后经过一个流量的控制，主要的维度有两个，一个是队列中的长度，超过一定长度后消费者会认为消息中产生了堆积，则放弃本次拉取请求，延时一段时间后再重新将请求加入到队列中；另一个维度是队列中的跨度，跨度指的是队列中消息最大和最小偏移量的距离，如果过大也视为是有消息被堆积也会放弃此次拉取请求，这里流量控制的目的是为了让消费者能够根据自己的消费能力去拉取消息，防止一下子拉取太多超过了自己的消费水平从而堆积起来而造成压力和性能瓶颈。经过流控后就会获取本地的Broker路由信息，获取Broker地址异步发送。当消息拉取服务拉取到消息后会将消息异步交给消息消费服务，然后就把请求体这个对象重新放回那个阻塞的任务列表，也就是说一个请求发送到 Broker 拉取到消息后就会立即放入阻塞队列，然后又马不停蹄的进行下一次拉取，消费端就是用这种方式来保证消息拉取的实时性，使其拉取性能接近于从 Broker 主动推送到消费端的性能。

Broker收到拉取新消息请求后，采用一种长轮询的方式来返回消息。指的是如果消费队列里没有新消息，并不急于返回，而是每隔5s查看消息是否到达，如果到达了则到 commitLog 中取出消息返回给消费者，此时消费者是处于阻塞等待状态的，意思是被 Broker 挂起，如果连续检查3次都没有消息的话，此时 Broker 就返回一个空结果。但这样最多只检查3次，每次消费者需要等待5s，实时性很差，所以 rocketmq 有另一个机制，就是如果在3次检查的过程中，有消息到达的话，就会立即复用当前连接，然后把一批消息根据偏移量排好序发给消费者。

消费者的消息拉取服务拉取到消息后，会先创建一个执行队列，然后把消息存入这个执行队列中的一个 treeMap 容器中，key 就是消息偏移量，value 就是消息实体。接着拉取服务会把这个执行队列异步交给消费者服务，然后就发送下一个请求，不用等消费服务处理完提高了吞吐量。接着消费服务会在消费线程池中取得一个线程来获取执行队列中的消息，这里是通过取锁互斥的方式来取 treeMap 的消息的，而由于从 Broker 返回的是有序的消息且存在 treeMap 中也是有序的，所以这里就保证了一个顺序消费。

如果消费失败了，会发回给broker存入延迟队列中，每个消费者在启动时默认订阅延迟队列，这样消费失败的消息在一段时候后又能够重新消费。

6、Q&A

Q1、RocketMQ 的特点和优点

是一个**队列模型**的**消息中间件**，具有**高性能**、**高可靠**、**高可用**、**高实时**、**高堆积**、**多功能**特点。

Q2、优点展开

高性能在于

- 顺序写盘(CommitLog)
- 零拷贝
- 跳跃读(预读：尽量命中PageCache)
- 预热、堆外内存写
- 索引文件组织形式
- 生产者消费者分布式集群部署，可以灵活地扩容，提高系统吞吐

高可靠在于

- 发送失败和消费失败重试
- 生产者和消费者的负载均衡，保证在Broker不可用时及时更新相应的配置
- 保证消息至少被消费一次保证不丢
- 同步异步刷盘将消息持久化，同步刷盘是等待持久化成功返回成功后生产者再进行下一次发送，可靠性最
- 主从复制模式，同步和异步，同步模式将等待从机同步完主机后才返回给生产者，可靠性最高

高可用在于

- 主从模式集群部署，多主多从，提高容灾能力

高实时在于

- 双线程长轮询

高堆积在于

- RocketMQ会定期删除过期的消息，使得内存Buffer抽象成一个无限长度的队列

其他功能

- 支持顺序、回溯、定时消费

Q3、影响消息可靠性的几种情况

1. Broker正常关闭
2. Broker异常Crash
3. OS Crash
4. 机器掉电，但是能立即恢复供电情况。
5. 机器无法开机（可能是cpu、主板、内存等关键设备损坏）
6. 磁盘设备损坏。

(1)、(2)、(3)、(4)四种情况都属于硬件资源可立即恢复情况，RocketMQ在这四种情况下能保证消息不丢，或者丢失少量数据（依赖刷盘方式是同步还是异步）。

(5)、(6)属于单点故障，且无法恢复，一旦发生，在此单点上的消息全部丢失。RocketMQ在这两种情况下，通过**异步复制**，可保证99%的消息不丢，但是仍然会有极**少量**的消息可能丢失。通过同步**双写技术**可以完全避免单点，**同步双写势必会影响性能**，适合对消息可靠性要求极高的场合，例如与Money相关的应用。

Q4、RocketMQ 怎么保证的消息不丢失？可靠性保证？

1）生产者这边

生产者这边有两种方式。一种是设置发送方式为同步发送，发送后会同步阻塞等待结果，当消息发送失败时，设置重试次数进行重试，这种比其他两种发送方式异步发送和单向发送要可靠。另一种方式是部署多个 broker 节点，当某个 broker 奔溃不可用了，就会造成生产者发送失败，那么在重试时，重新选择路由时，就会自动跳过那些宕机的 broker，其对应的队列不会参与轮询，所以就可以重新选择正常的 broker 进行发送，保证消息能正常发送。

2）broker 这边

broker 这边也有两种方式。

第一种方式是broker 这边可以配置刷盘模式为同步刷盘。默认情况是异步刷盘，Broker 收到消息后会先存到缓存中然后立马通知生产者说消息收到了，然后生产者就继续发下一次消息。然后 Broker 起个线程异步的去持久化到磁盘中，但如果 Broker 还没持久化到磁盘就宕机的话，消息就丢失了，所以需要同步刷盘，收到消息存到缓存并且刷新到磁盘中后才会返回成功给生产者，这种方式可靠性最高，但性能比异步刷盘要低，一般用于金融或者支付领域。

第二种方式是主从配置，并且同步双写。配置一主多从或者多主多从，然后数据及时从主节点同步到从节点，避免单点故障造成数据丢失，而且主节点宕机后消费者可以继续从从节点读取数据，保证了消费的实时性，最新的版本已经支持在多个从机中自动选举master来实现真正的高可用。同时设置同步复制的策略，即同步双写。主从同步有异步复制和同步复制两种。异步复制指的是只要主节点写了数据就可以返回成功给生产者，这种方式性能高，吞吐量大，但可靠性差，可能会丢失数据。而同步复制就是等主从节点都写数据后才返回成功给生产者，这种最可靠。一般项目中配置成异步刷盘和同步复制的策略。

3）消费者这边

一种方式是手动提交ack。指的是消费者会先把消息拉取到本地，然后进行业务逻辑，业务逻辑完成后手动进行ack确认，这时候才会真正的代表消费完成。然后再保证一个消费的幂等性，消费完把消息id存入缓存，再次消费时检查一下是否消费过，避免重复消费。另一种方式是设置重试次数，如果消费消息失败了，没有进行ack确认，则会自动重试

Q5、实时性

在消息不堆积情况下，消息到达Broker后，能**立刻**到达Consumer。RocketMQ使用**长轮询Pull**方式，可保证消息非常实时，消息实时性不低于Push。

Q7、RocketMQ的动态伸缩

Producer、Consumer、队列都可以**分布式**，**可以横向扩展**，**提高系统吞吐**

具有很好动态伸缩能力(非顺序消息)，伸缩性体现在**Topic**和**Broker**两个维度。
Topic维度：假如一个Topic的消息量特别大，但**集群水位压力还是很低**，就可以扩大该**Topic的队列数**，Topic的队列数跟发送、消费速度成正比。
Broker维度：如果**集群水位很高**了，**需要扩容**，直接加机器**部署Broker**就可以。Broker起来后向

Namesrv注册，Producer、Consumer通过Namesrv 发现新Broker，立即跟该Broker直连，收发消息。

Q8、Broker的Buffer满了怎么办?

RocketMQ的**内存Buffer抽象**成一个**无限长度**的队列，不管有多少数据进来都能装得下，这个无限是有**前提**的，Broker会**定期删除过期**的数据，例如Broker只保存3天的消息，那么这个Buffer虽然长度无限，但是3天前的数据会被从队尾删除。

Q9、顺序消费

能够保证严格的**消息顺序**，**生产者**可以通过**hash**将同类消息发往相同队列，**存储者**通过**阻塞优先队列**，实现了按偏移量排序来处理文件分配的请求，保证了消息的写的顺序，然后发给消费者的批量消息也是按照偏移量排好序的，保证了消费的顺序

Q10、回溯消费

回溯消费是指**Consumer已经消费成功的消息**，由于业务上需求**需要重新消费**，要支持此功能，Broker在向Consumer投递成功消息后，**消息仍然需要保留。**并且重新消费一般是按照时间维度，例如由于Consumer系统故障，恢复后需要重新消费1小时前的数据，那么Broker要提供一种机制，可以按照时间**维度来回退消费进度。**
RocketMQ支持按照时间回溯消费，时间维度精确到毫秒，**可以向前回溯，也可以向后回溯。**

Q11、定时消息

定时消息是指消息发到Broker后，不能**立刻**被Consumer消费，要到**特定的时间**点或者等待特定的时间后才能被消费。 RocketMQ支持定时消息，但是不支持任意时间精度，支持特定的level，例如定时5s，10s，1m等。

Q12、Broker的搭建方式

1.单个Master模式 : 这种方式风险较大，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

2.多个Master模式： 一个集群无 Slave，全是 Master，例如 2 个 Master 戒者 3 个 Master
　　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，同步刷盘一条会丢）。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。
```

3.多Msater多slave模式，异步复制： 每个 Master 配置一个 Slave，有多对 Master-Slave，HA 采用异步复制方式，主备有短暂消息延迟，毫秒级。
　　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，因为 Master 宕机后，消费者仍然可以从 Slave 消费，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，会丢失少量消息
```

4.多Master多slave模式，同步双写： 每个 Master 配置一个 Slave，有多对 Master-Slave，HA 采用同步双写方式，主备都写成功，向应用返回成功。

优点：数据不服务都无单点，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。目前主宕机后，备机不能能自动切换为主机，后续会支持自动切换功能。

Q13、多个mq如何选型?

MQ	描述
Rabbit MQ	erlang开发，对消息堆积的支持并不好，当大量消息积压的时候，会导致 RabbitMQ 的性能急剧下降。每秒钟可以处理几万到十几万条消息。
Rocket MQ	java开发，面向互联网集群化功能丰富，对在线业务的响应时延做了很多的优化，大多数情况下可以做到毫秒级的响应，每秒钟大概能处理几十万条消息。
Kafka	Scala开发，面向日志功能丰富，性能最高。当你的业务场景中，每秒钟消息数量没有那么多的时候，Kafka 的时延反而会比较高。所以，Kafka 不太适合在线业务场景。
Active MQ	java开发，简单，稳定，性能不如前面三个。小型系统用也ok，但是不推荐。推荐用互联网主流的。

ActiveMQ 的性能比如其他几个，适用于小型应用。RabbitMQ 对消息堆积的支持不太好，当消息大量堆积时，性能会急剧下降。KafKa 主要面向日志功能比较丰富，性能最高，当在在线业务场景中，每秒消息数量不是很多时，发送的时间延迟反而比较高，所以不适合在线业务场景。RocketMQ 对在线业务的响应时延做了很多优化，大多数情况下可以做到毫秒响应，每秒可以处理几十万消息。

Q14、为什么要使用MQ?

作用	描述
解耦	系统耦合度降低，没有强依赖关系
异步	不需要同步执行的远程调用可以有效提高响应时间
削峰	请求达到峰值后，后端service还可以保持固定消费速率消费，不会被压垮

可以异步发送消息，迅速返回给调用方，然后发下一批消息，提高系统的吞吐量。

解耦，把不同的模块的关系解耦了，在业务处理模块服务不可用时，接口模块还能继续下单。

削峰，当请求达到峰值后，可以把请求缓存在消息队列中，能保证后端的服务还能以比较平稳的速度消费，不会压垮应用和服务。

Q15、RocketMQ由哪些角色组成，每个角色作用和特点是什么?

角色	作用
Nameserver	无状态，动态列表；这也是和zookeeper的重要区别之一。zookeeper是有状态的。
Producer	消息生产者，负责发消息到Broker。
Broker	就是MQ本身，负责收发消息、持久化消息等。
Consumer	消息消费者，负责从Broker上拉取消息进行消费，消费完进行ack。

Q16、RocketMQ中的Topic和JMS的queue有什么区别？

queue就是来源于数据结构的FIFO队列。而Topic是个抽象的概念，每个Topic底层对应N个queue，而数据也真实存在queue上的。

Q17、RocketMQ Broker中的消息被消费后会立即删除吗？

不会，每条消息都会持久化到CommitLog中，每个Consumer连接到Broker后会维持消费进度信息，当有消息消费后只是当前Consumer的消费进度（CommitLog的offset）更新了。

> **持久化到 CommitLog，更新 offset**

消息会堆积吗？什么时候清理过期消息？

4.6版本默认48小时后会删除不再使用的CommitLog文件。检查文件的最后访问时间是否过期，过期则在凌晨4点删除。

Q18、RocketMQ消费模式有几种？

消费模型由Consumer决定，消费维度为Topic。

- 集群消费

> 1.一条消息只会被同Group中的一个Consumer消费
>
> 2.多个Group同时消费一个Topic时，每个Group都会有一个Consumer消费到数据

- 广播消费

> 消息将对一 个Consumer Group 下的各个 Consumer 实例都消费一遍。即即使这些 Consumer 属于同一个Consumer Group ，消息也会被 Consumer Group 中的每个 Consumer 都消费一次。

Q19、消费消息是push还是pull？

RocketMQ没有真正意义的push，都是pull，虽然有push类，但实际底层实现采用的是**长轮询机制**，即拉取方式

> broker端属性 longPollingEnable 标记是否开启长轮询。默认开启

为什么要主动拉取消息而不使用事件监听方式？

事件驱动方式是建立好长连接，由事件（发送数据）的方式来实时推送。

如果broker主动推送消息的话有可能push速度快，消费速度慢的情况，那么就会造成消息在consumer端堆积过多，同时又不能被其他consumer消费的情况。而pull的方式可以根据当前自身情况来pull，不会造成过多的压力而造成瓶颈。所以采取了pull的方式。

Q20、broker如何处理拉取请求的?

通过 Broker 端的长轮询的方式。

Broker收到拉取新消息请求后，如果队列里没有新消息，并不急于返回，而是每隔5s查看消息是否到达，如果到达了则到 commitLog 中取出消息返回给消费者，此时消费者是处于阻塞等待状态的，意思是被 Broker 挂起，如果连续检查3次都没有消息的话，此时 Broker 就返回一个空结果。

但这样最多只检查3次，每次消费者需要等待5s，实时性很差，所以 rocketmq 有另一个机制，就是如果在3次检查的过程中，有消息到达的话，就会立即复用当前连接，把消息发送给消费者。

Q21、RocketMQ如何做负载均衡?

通过Topic在多Broker中分布式存储实现。

producer端

发送端指定message queue发送消息到相应的broker，消息分布在多broker中，来达到写入时的负载均衡

consumer端

采用的是平均分配算法来进行负载均衡。

追问：当消费负载均衡consumer和queue不对等的时候会发生什么?

Consumer和queue会优先平均分配，如果Consumer少于queue的个数，则会存在部分Consumer消费多个queue的情况，如果Consumer等于queue的个数，那就是一个Consumer消费一个queue，如果Consumer个数大于queue的个数，那么会有部分Consumer空余出来，白白的浪费了。

Q22、消息重复消费

影响消息正常发送和消费的**重要原因是网络的不确定性。**

引起重复消费的原因

- ACK

正常情况下在consumer真正消费完消息后应该发送ack，通知broker该消息已正常消费，从queue中剔除

当ack因为网络原因无法发送到broker，broker会认为词条消息没有被消费，此后会开启消息重投机制把消息再次投递到consumer

- 消费模式

在CLUSTERING模式下，消息在broker中会保证相同group的consumer消费一次，但是针对不同group的consumer会推送多次

解决方案

- 数据库表

处理消息前，使用消息主键在表中带有约束的字段中insert

- Redis

分布式锁搞起来。

Q23、如何让RocketMQ保证消息的顺序消费

> 如果不需要保证消息顺序，为什么不需要?假如我有一个场景要保证消息的顺序，你们应该如何保证?

首先多个queue只能保证单个queue里的顺序，queue是典型的FIFO，天然顺序。多个queue同时消费是无法绝对保证消息的有序性的。所以总结如下：同一topic，同一个QUEUE，发消息的时候一个线程去发送消息，消费的时候 一个线程去消费一个queue里的消息。

怎么保证消息发到同一个queue?

相当 id 通过 hash 取模队列长度，或者 id 直接取模也行。

> **单个队列局部有序，全局无序。同个 id 取模队列长度发送，发送和接收都固定一个线程。**

Q24、RocketMQ如何保证消息不丢失

首先在如下三个部分都可能会出现丢失消息的情况：

- Producer端
- Broker端
- Consumer端

Producer端如何保证消息不丢失

- 采取send()同步发消息，发送结果是同步感知的。
- 发送失败后可以重试，设置重试次数。默认3次。
- 集群部署，比如发送失败了的原因可能是当前Broker宕机了，重试的时候会发送到其他Broker上。

> **send 同步发送，如果发送失败则进行重试，然后要集群部署，这样重试才能跳过可能已经挂机的Broker，选择其他 Broker**

Broker端如何保证消息不丢失

- 修改刷盘策略为同步刷盘。默认情况下是异步刷盘的。
- 集群部署，主从模式，高可用。

Consumer端如何保证消息不丢失

- 完全消费正常后在进行手动（程序执行）ack确认。
- 设置重试次数

Q25、rocketMQ的消息堆积如何处理

> 下游消费系统如果宕机了，导致几百万条消息在消息中间件里积压，此时怎么处理?
>
> 线上是否遇到过消息积压的生产故障?如果没遇到过，你考虑一下如何应对?

首先要找到是什么原因导致的消息堆积，是Producer太多了，Consumer太少了导致的还是说其他情况，总之先定位问题。然后看下消息消费速度是否正常，正常的话，可以通过上线更多consumer临时解决消息堆积问题。

追问：如果Consumer和Queue不对等，上线了多台也在短时间内无法消费完堆积的消息怎么办?

- 准备一个临时的topic
- queue的数量是堆积的几倍
- queue分布到多Broker中
- **上线一台Consumer做消息的搬运工，把原来Topic中的消息挪到新的Topic里**，不做业务逻辑处理，只是挪过去
- **上线N台Consumer同时消费临时Topic中的数据**
- 改bug
- 恢复原来的Consumer，继续消费之前的Topic

堆积时间过长消息超时了?

RocketMQ中的消息只会在commitLog被删除的时候才会消失，不会超时。也就是说未被消费的消息不会存在超时删除这情况。

堆积的消息会不会进死信队列?

不会，消息在消费失败后会进入重试队列（%RETRY%+ConsumerGroup），18次（默认18次，网上所有文章都说是16次，无一例外。但是我没搞懂为啥是16次，这不是18个时间吗？）才会进入死信队列（%DLQ%+ConsumerGroup）。

> **排查问题（BUG、生产者太多、消费者太少、都正常的话可以增加消费者）**

Q26、RocketMQ在分布式事务支持这块机制的底层原理?

分布式系统中的事务可以使用TCC（Try、Confirm、Cancel）、2pc来解决分布式系统中的消息原子性

RocketMQ 4.3+提供分布事务功能，通过 RocketMQ 事务消息能达到分布式事务的最终一致

RocketMQ实现方式：

Half Message：预处理消息，当broker收到此类消息后，会存储到RMQ_SYS_TRANS_HALF_TOPIC的消息消费队列中

检查事务状态：Broker会开启一个定时任务，消费RMQ_SYS_TRANS_HALF_TOPIC队列中的消息，每次执行任务会向消息发送者确认事务执行状态（提交、回滚、未知），如果是未知，Broker会定时去回调在重新检查。

超时：如果超过回查次数，默认回滚消息。

也就是他并未真正进入Topic的queue，而是用了临时queue来放所谓的half message，等提交事务后才会真正的将half message转移到topic下的queue。

Q27、动手实现一个分布式消息中间件，整体架构如何设计实现?

- 需要考虑能快速扩容、天然支持集群
- 持久化的姿势
- 高可用性
- 数据0丢失的考虑
- 服务端部署简单、client端使用简单

Q29、高吞吐量下如何优化生产者和消费者的性能?

开发

- 同一group下，多实例部署，并行消费
- 单个Consumer提高消费线程个数

- 批量消费（消息批量拉取、业务逻辑批量处理）（减少网络 IO）

> **多实例部署、消费线程、业务批量消费**

运维

- jvm调优
- 网卡调优
- 多线程与cpu调优
- Page Cache

Q30、RocketMQ 是如何保证数据的高容错性的?

- 在不开启容错的情况下，轮询队列进行发送，如果失败了，重试的时候过滤失败的Broker
- 如果开启了容错策略，会通过RocketMQ的预测机制来预测一个Broker是否可用
- 如果上次失败的Broker可用那么还是会选择该Broker的队列
- 如果上述情况失败，则随机选择一个进行发送
- 在发送消息的时候会记录一下调用的时间与是否报错，根据该时间去预测broker的可用时间

> **不开启容错时，轮询队列发送，失败则重试时跳过失败的 Broker**
>
> **开启容器，则会通过预测某个 Broker 是否可用**

Q31、任何一台Broker突然宕机了怎么办?

Broker主从架构以及多副本策略。Master收到消息后会同步给Slave，这样一条消息就不止一份了，Master宕机了还有slave中的消息可用，保证了MQ的可靠性和高可用性。而且Rocket MQ4.5.0开始就支持了Dlegder模式，基于raft的，做到了真正意义的HA。

Q32、 RocketMQ是如何发消息的?

消息发送时，首先初始化一些参数配置，比如消息的组名 GroupName 以及 NameServer 的地址等。然后启动Producer的start()方法，启动完成后调用send()方法进行发送。 start()方法内部会进行检查namesrv、生产者组名等参数验证，然后内部会获取一个通讯对象，底层主要是 netty 客户端，然后执行一些定时任务，比如与broker的心跳等，还会启动负载均衡服务等。

然后根据 Topic 获取对应的 Broker 的路由信息，先从本地的路由信息表 TopicPublishInfoTable 中查找，如果没有找到目标的 Topic，再发心跳包到 NameServer，查找该 Topic 的路由信息，拉取到本地再进行相应的更新。

获取到路由信息后，准备发送到目标 Broker，这时可能存在一个 Topic 对应多个 Broker 多个 Queue 的情况，所以需要进行负载均衡。常见的算法有两种，一种是正常情况下的队列轮询算法，一种是要顺序消费时，通过相同的某个关键字，比如可以是订单id，然后对消息队列个数求余来达到相同订单id发往同一个队列的目的。

选择好队列后准备发送，有三种发送方式，同步、异步、单向发送。同步就是发完后阻塞住等待结果，如果发送失败的话会自动重试，默认同步发送的次数是3次。异步就是发送完后通过回调接口来接收返回结果。单向发送则只发消息而不会关心返回结果，所以这种方式性能最高，但可靠性最差。

Q32、 Broker收到消息后怎么持久化的?

有两种方式：同步和异步。一般选择异步，同步效率低，但是更可靠。消息存储大致原理是：

消息首先发送给 broker，broker 接收后会写入一个 MapperFile 对象，该对象基于 MMAP 机制可以实现从虚拟内存中直接映射到磁盘文件，所以 MapperFile 对象相当于是对这一机制的封装，提供了一些 API 来支持这种实现。每个 MapperFile 对应于磁盘的一个 CommitLog 文件，多个 CommitLog 文件就是多个 MapperFile 构成了一个MappedFileQueue 对象，相当于文件夹，管理所有的文件。而每个 MapperFile 对应于内存中的一个 buffer 缓冲，所以消息首先写入 MapperFile 对象，也就是先写入内存buffer中，然后再刷新到磁盘中。如果是同步刷盘则立马刷新，如果是异步刷盘则默认500ms刷新一次。（预热、锁定内存、预分配）

Q33、 为什么可以去ZK?

zookeeper 可以在 broker leader 挂起时发起选举，选出新的 leader。而在RocketMQ中，不需要选举，Master/Slave的角色也是固定的。当一个Master挂了之后，你可以写到其他Master上，但不会说一个Slave切换成Master。4.6之后才有自动切换的功能。

Q34、 Kafka 和 RocketMQ 选型

1）topic 数量对性能的影响

当 topic 数量增多时，kafka 的性能下降很明显，而 rocketMQ 依然能保持很好的性能。因为 kakfa 的每个 topic 的每个分区都对应一个或多个日志文件，当这些文件需要刷盘时，多个文件就会造成磁盘 IO 的竞争，导致性能下降。 而 RocketMQ 所有的消息是保存在同一个物理文件中的，即使当超过1G会新建文件，文件的数量相比 kafka 也少很多，刷盘时的 IO 竞争会少很多。

消息系统要承担的业务比较多，光推送业务就需要很多个 topic，因为我们有安卓和IOS推送渠道，安卓有华为小米魅族等十多个渠道，每个渠道都会单独分配topic，而且渠道还有一些比如卸载回调的topic等。除了推送业务外，还有客户端上报的一些用户的注册、登录、卸载、注销等信息，这些都是用不同的 topic 来区分，以及发送过程中的一些事件码的上报，也是通过 topic 发到消费端统一处理的，所以在多个 topic 的情况下，选择 rocketmq 更好。

2）消息重试机制

rocketmq 支持根据设置的重试次数进行消息发送失败重试机制，如果消费失败了，会发回给broker存入延迟队列中，每个消费者在启动时默认订阅延迟队列，这样消费失败的消息在一段时候后又能够重新消费。而 kafka 本身不能支持发送失败的自动重试，但 Spring 的第三方工具实现了，实现方式就是把处理失败的消息再写到一个专门的topic，但是第三方的这个重试工具很少有测评，所以可能不可靠。

3）考虑到消息的可靠性保证

rocketmq 支持同步异步刷盘以及同步异步复制等策略，而 kafka 不支持同步刷盘，虽然可以通过调整参数使得 kafka 每发一条消息就刷盘一次，但这种 IO 开销太多，导致性能也急剧下降。而消息系统日常推送的消息中除了运营类的消息外，有时还包含一些重要的通知和活动信息，这些也需要比较强的可靠性和稳定性，所以这种情况下，选择 rocketmq 这种有更成熟的同步刷盘策略的更合适。

4）事务消息

RocketMQ支持事务消息，kafka 不支持。

5）消息查询、过滤、回溯

rocketmq 可以根据messageId查询消息信息、可以根据 tag 来过滤消息、可以根据时间点来回溯消息。但 kafka 都不直接支持。

如果某个消息没有被消费，rocketmq 支持根据某个业务 key 迅速地定位到消息的具体内容。

如果业务场景有需要重新消费历史消息时，那么 rocketmq 可以支持以时间为起点的消息重放功能，可以精确到毫秒。而Kafka理论上可以按照Offset来回溯消息，精度不高。

6）消费并行度

kafka的一个Partition同一时间只能被一个消费者消费，也就是有多少个Partition就最多有多少个线程同时消费。rocketmq消费者在并发模式下，每个 topic 的每个队列可以被多个消费者消费，所以rocketmq的并发度比Kafka高出一个量级。而消息系统由于 topic 很多，所以不同 topic 的队列的线程配置肯定也会比较多。

7）消息顺序性

无论是 kafka 还是 rocketmq 都只能保证正常情况下消息的局部有序而不能保证全局有序。比如同一个订单的三条消息，分别是下单、付款、完成，那么就可以通过订单 id 进行某种 hash 算法后，再对队列或者分区取模，这样就可以做到同个订单的消息发到同个队列，但同时还需要保证发送者和消费者同一时刻只有一个线程在使用这个分区或队列。

正常情况下，kafka 和 rocketmq 都可以通过这种方式来实现分区有序，但如果有一台 broker 宕机，那么通过分区或者队列数改变，消息就会发到其他队列中，而其他队列如果不是一个线程在消费，就有可能会发生乱序的问题，在 kafka 上就会有这种问题，但是 rocketmq 即使在这种情况下也能保证同一个时刻只有一个消费者的一个线程在消费，从而保证了有序，其原理是：

Rocker采用的是分段锁，它不是锁整个Broker而是锁里面的单个Queue。在创建消息拉取任务时，消息客户端向broker端申请锁定MessageQueue，使得一个MessageQueue同一个时刻只能被一个消费客户端消费。消息消费时，多线程针对同一个消息队列的消费先尝试使用synchronized申请独占锁，加锁成功才能进行消费，使得一个MessageQueue同一个时刻只能被一个消费客户端中一个线程消费。

Q35、RocketMQ 为什么快

【producer】

在一致性和可靠性要求不高的场景下，消息发送端可以设置单向和异步发送，很大程度提高了系统的性能和吞吐量。而且有高效的故障容错机制，保证消息发送失败时进行快速重发。

【broker】

顺序写

数据的顺序写，所有 topic 的数据顺序追加到 CommitLog，减少了磁头的寻道时间。

MMAP 和 预热

因为 rocketmq 使用 mmap 机制来将磁盘的地址映射到进程的虚拟地址中，使得程序可以直接将数据写入内存，然后再映射到磁盘中。而且 rocketmq 采用一种预热的机制，就是新建文件后会立马写入1G字节的0值，每个文件最大1G，这样的话就可以让系统提前为文件分配物理内存空间，防止在写入消息时发生缺页异常。同时预热后，又通过 mlock 系统调用锁住这部分内存，防止其被交换到交换区。

堆外内存

写消息时先写到堆外内存，纯内存操作非常快。读消息时是从pagecache中读，相当于实现了读写分离。

其他

当consumer消费很远的数据时，pagecache中肯定是没有缓存的，这时候rocketMq建议consumer去slave上读。支持异步刷盘。

【consumer】

长轮询

和 Broker 一同基于长轮询的机制，保证了消息消费的时效性。

负载均衡

消息消费队列在同一消费组不同消费者之间的负载均衡，其核心设计理念是在一个消息消费队列在同一时间只允许被同一消费组内的一个消费者消费，一个消息消费者能同时消费多个消息队列。

Q36、项目中推送的线程数

消费者

业务处理：推送300、查询10-30

消息推送：10-30

IOS 转发：200

参考资料

RocketMQ原理解析-NameServer

https://www.cnblogs.com/wxd0108/p/6055108.html

RocketMQ原理解析 - Producer
https://www.cnblogs.com/wxd0108/p/6041829.html?tt_from=weixin&utm_source=weixin&utm_medium=toutiao_ios&utm_campaign=client_share&wxshare_count=1&from=singlemessage&isappinstalled=0

RocketMQ源码-MappedFile介绍
https://www.jianshu.com/p/9bd672e1c5c1

rocketmq里面生产者发送一次消息以后commit-log做了哪些事情
https://www.cnblogs.com/notlate/p/12007242.html

RocketMQ原理解析 - Broker
https://www.cnblogs.com/wxd0108/p/6055004.html

RocketMQ原理解析 - Consumer
https://www.cnblogs.com/wxd0108/p/6054817.html

RocketMQ原理解析-Remoting
https://www.cnblogs.com/wxd0108/p/6055132.html

RocketMQ源码分析(六)之Broker存储MappedFile
https://blog.csdn.net/usagoole/article/details/100854529

浅谈消息队列之RocketMQ
https://www.cnblogs.com/eryun/p/12163775.html

otherCoding
https://www.cnblogs.com/milicool/p/11963024.html

快速弄明白RocketMQ的CommitLog、ConsumeQueue、indexFile、offset
https://www.cnblogs.com/zxporz/p/12336476.html

RocketMQ高性能原理(pushConsumer,CommitLog,ZeroCopy)
https://www.cnblogs.com/wlwl/p/10889850.html

RocketMQ入门到入土（一）新手也能看懂的原理和实战！
https://www.cnblogs.com/javazhiyin/p/13201315.html

庖丁解牛 | 图解 RocketMQ 核心原理
https://blog.csdn.net/liqiuman180688/article/details/88390667

十分钟入门RocketMQ
http://jm.taobao.org/2017/01/12/rocketmq-quick-start-in-10-minutes/

分布式开放消息系统(RocketMQ)的原理与实践
https://www.jianshu.com/p/468176c6bc1b

消息中间件—RocketMQ消息存储（一）
https://www.jianshu.com/p/b73fdd893f98

消息中间件—RocketMQ消息存储（二）
https://www.jianshu.com/p/6d0c118c17de

RocketMQ存储--映射文件预热【源码笔记】
https://www.jianshu.com/p/eddbd2371a7a

《浅入浅出》-RocketMQ
https://juejin.im/post/5de3c8026fb9a07194761641#heading-38