RocketMQ 原理解析

斩秋

目录

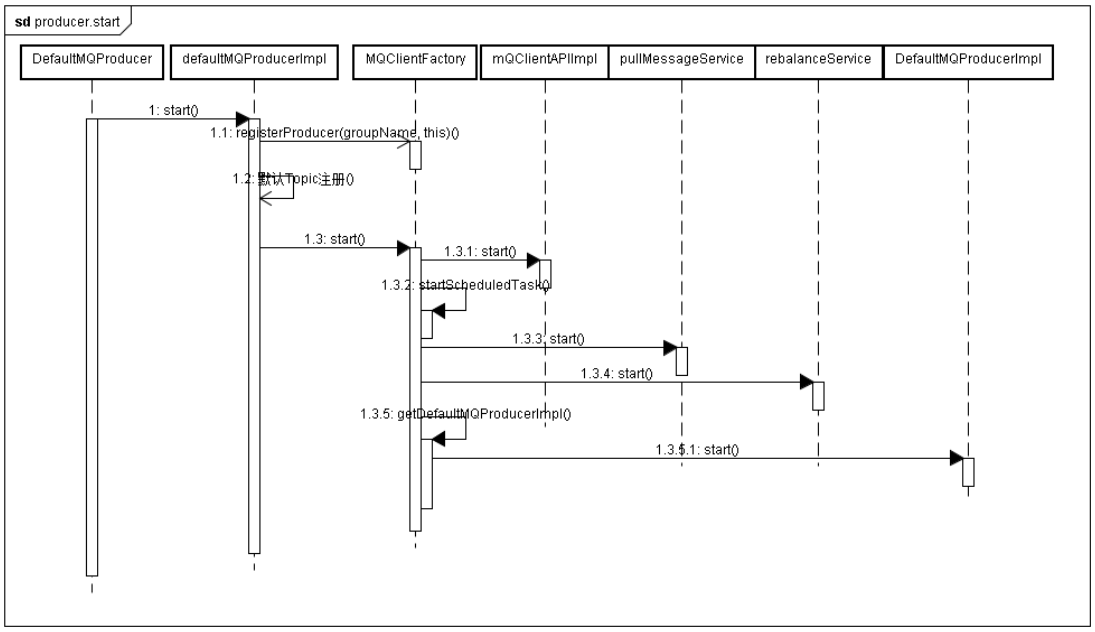
前言

此文档是从学习 rocketmq 源码过程中的笔记中整理出来的，由于时间及能力原因，理解有误之处还请谅解，希望对大家学习使用 rocketmq 有所帮助。

Rocketmq 是阿里基于开源思想做的一款产品，代码托管于 github 上，要想学好用好 rocketmq 请从 https://github.com/alibaba/RocketMQ 获取最权威的文档、问题解答、原理介绍等。

第一章： producer

一：Producer 启动流程



Producer 如何感知要发送消息的 broker 即 brokerAddrTable 中的值是怎么获得的，

1. 发送消息的时候指定会指定 topic，如果 producer 集合中没有会根据指定 topic 到 namesrv 获取 topic 发布信息 TopicPublishInfo，并放入本地集合
2. 定时从 namesrv 更新 topic 路由信息，

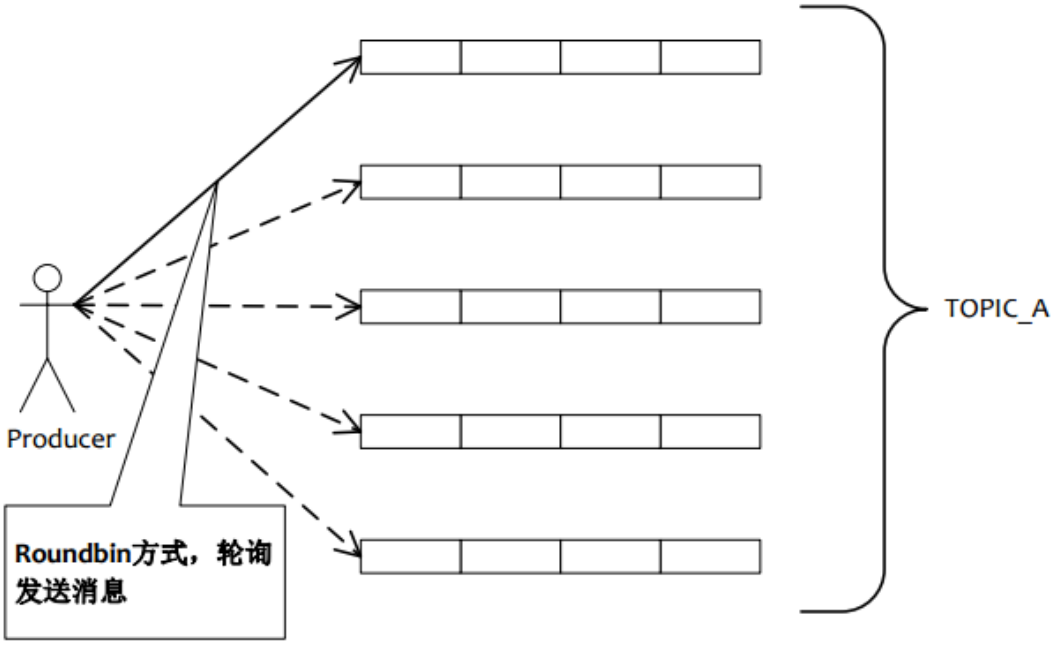
Producer 与 broker 间的心跳

Producer 定时发送心跳将 producer 信息（其实就是 procduer 的 group）定时发送到，brokerAddrTable 集合中列出的 broker 上去

Producer 发送消息只发送到 master 的 broker 机器，在通过 broker 的主从复制机制拷贝到 broker 的 slave 上去

二：Producer 如何发送消息

Producer 轮询某 topic 下的所有队列的方式来实现发送方的负载均衡



7-5 发送消息 Rebalance

1） Topic 下的所有队列如何理解：

比如 broker1, broker2, borker3 三台 broker 机器都配置了 Topic_A

Broker1 的队列为 queue0 , queue1

Broker2 的队列为 queue0, queue2, queue3,

Broker3 的队列为 queue0

当然一般情况下的 broker 的配置都是一样的

以上当 broker 启动的时候注册到 namesrv 的 Topic_A 队列为共 6 个分别为：

broker1_queue0, broker1_queue1,

broker2_queue0, broker2_queue1, broker2_queue2,

broker3_queue0,

2） Producer 如何实现轮询队列：

Producer 从 namesrv 获取的到 Topic_A 路由信息 TopicPublishInfo

--List<MessageQueue> messageQueueList //Topic_A 的所有的队列

--AtomicInteger sendWhichQueue //自增整型

方法 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

如果入参 lastBrokerName 不为空，代表上次选择的 queue 发送失败，这次选择应该避开同一个 queue

3） Producer 发消息系统重试:

发送失败后，重试几次 retryTimesWhenSendFailed = 2

发送消息超时 sendMsgTimeout = 3000

Producer 通过 selectOneMessageQueue 方法获取一个 MessagQueue 对象

--topic //Topic_A

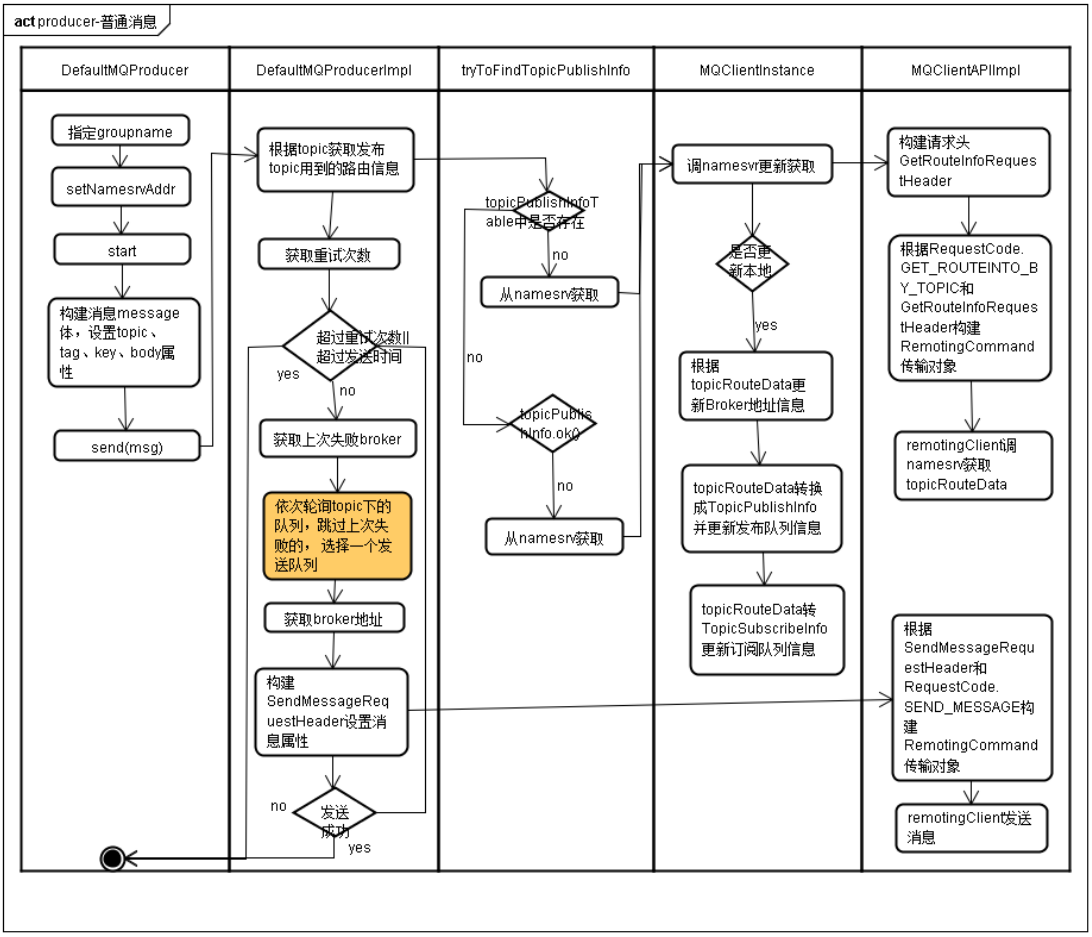
--brokerName //代表发送消息到达的 broker

--queueId //代表发送消息的在指定 broker 上指定 topic 下的队列编号

向指定 broker 的指定 topic 的指定 queue 发送消息

发送失败(1)重试次数不到两次(2)发送此条消息花费时间还没有到 3000(毫秒), 换个队列继续发送。

2.1 producer 发送普通消息



2.2 顺序消息发送

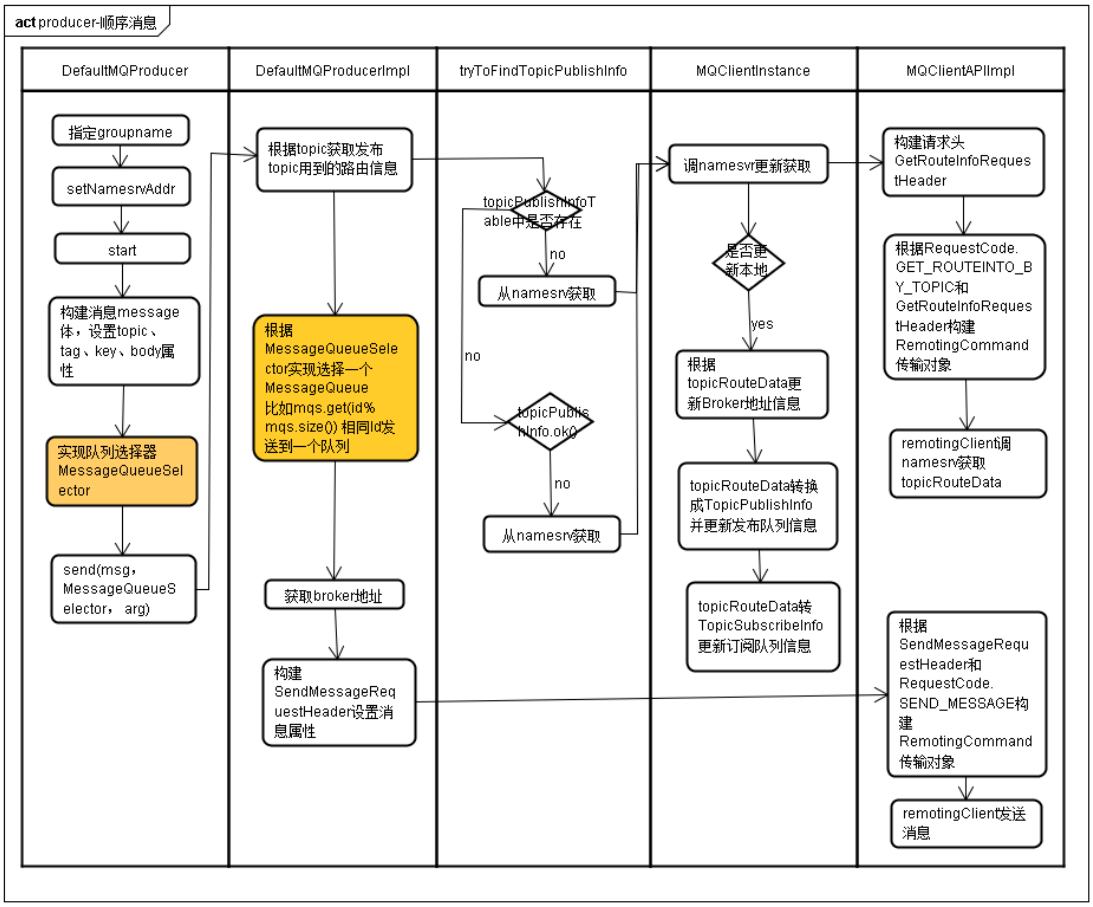
Rocketmq 能够保证消息严格顺序，但是 Rocketmq 需要 producer 保证顺序消息按顺序发送到同一个 queue 中， 比如购买流程(1)下单(2)支付(3)支付成功，这三个消息需要根据特定规则将这个三个消息按顺序发送到一个 queue

如何实现把顺序消息发送到同一个 queue:

一般消息是通过轮询所有队列发送的，顺序消息可以根据业务比如说订单号 orderId 相同的消息发送到同一个队列，或者同一用户 userId 发送到同一队列等等

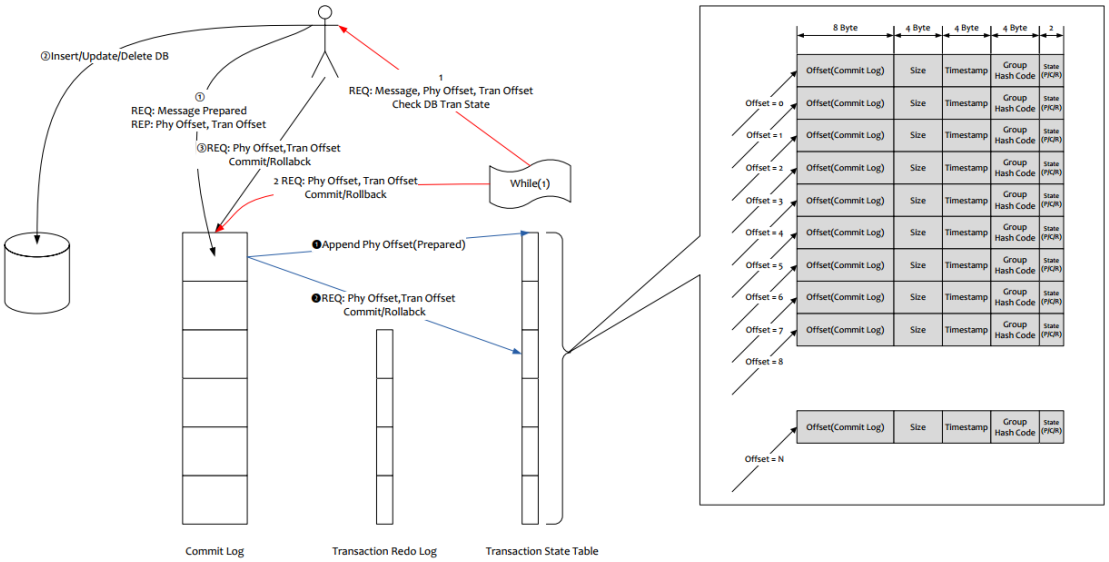
messageQueueList [orderId%messageQueueList.size()]

messageQueueList [userId%messageQueueList.size()]



2.3 分布式事物消息

先引入官方文档图：



分布式事物是基于二阶段提交的

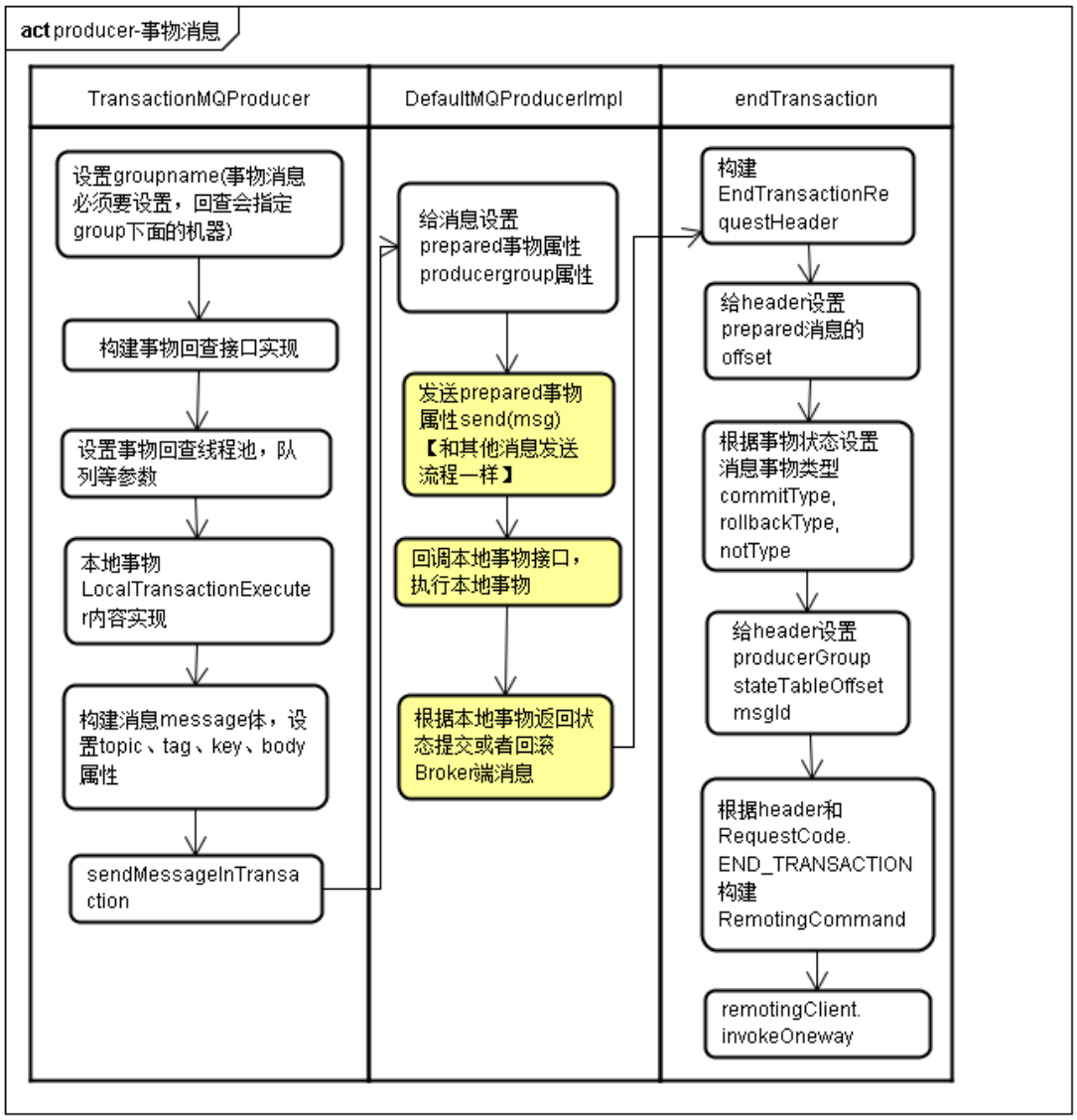
1) 一阶段，向 broker 发送一条 prepared 的消息，返回消息的 offset 即消息地址 commitLog 中消息偏移量。Prepared 状态消息不被消费

发送消息 ok，执行本地事物分支， 本地事物方法需要实现 rocketmq 的回调接口 2）2）

2) LocalTransactionExecuter ， 处 理 本 地 事 物 逻 辑 返 回 处 理 的 事 物 状 态 LocalTransactionState

3) 二阶段，处理完本地事物中业务得到事物状态， 根据 offset 查找到 commitLog 中的 prepared 消息，设置消息状态 commitType 或者 rollbackType ， 让后将信息添加到 commitLog 中， 其实二阶段生成了两条消息

事物消息发送


act producer-事物消息
TransactionMQProducer
DefaultMQProducerImpl
endTransaction
设置groupname(事物消息必须要设置，回查会指定group下面的机器)
构建事物回查接口实现
设置事物回查线程池，队列等参数
本地事物LocalTransactionExecuter内容实现
构建消息message体，设置topic、tag、key、body属性
sendMessageInTransaction
给消息设置prepared事物属性producergroup属性
发送prepared事物属性send(msg)【和其他消息发送流程一样】
回调本地事物接口，执行本地事物
根据本地事物返回状态提交或者回滚Broker端消息
构建EndTransactionRequestHeader
给header设置prepared消息的offset
根据事物状态设置消息事物类型commitType,rollbackType,notType
给header设置producerGroup stateTableOffset msgId
根据header和RequestCode.END_TRANSACTION构建RemotingCommand
remotingClient.invokeOneway

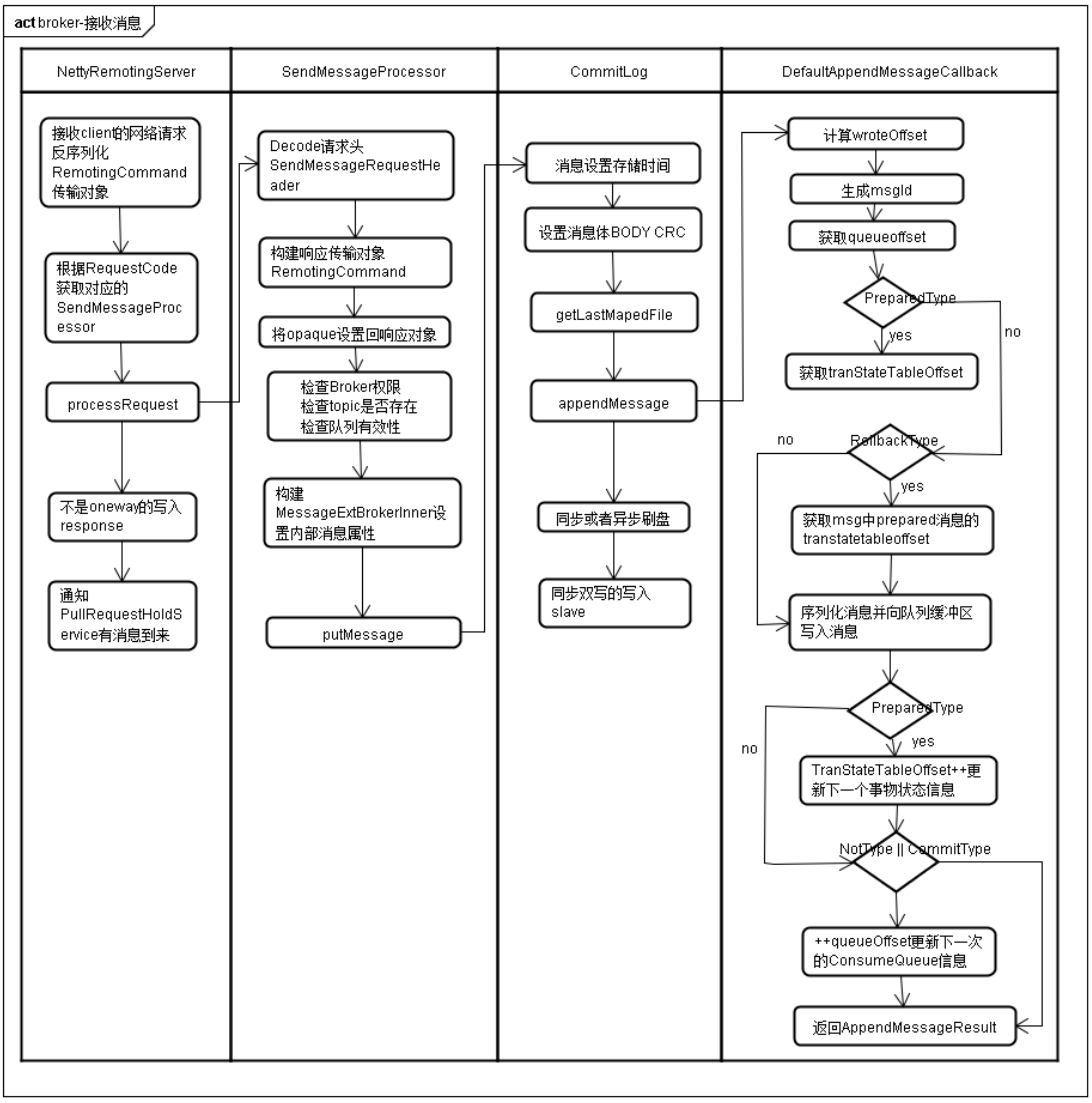
三：Broker 落地消息

2.1 普通消息落地

Broker 根据 producer 请求的 RequestCode.SEND_MESSAGE 选择对应的处理器 SendMessageProcessor

根据请求消息内容构建消息内部结构 MessageExtBrokerInner

调 DefaultMessageStore 加消息写入 commitlog



2.2 分布式事物消息落地

2.2.1 消息落地

commitLog 针对事物消息的处理，消息的第 20 位开始的八位记录是的消息在逻辑队列中的 queueoffset， 但是针对事物消息为 preparedType 和 rollbackType 的存储的是事物状态表的索引偏移量

2.2.2 分发事物消息：

分发消息位置信息到 ConsumeQueue：事物状态为 preparedType 和 rollbackType 的消息不会将请求分发到 ConsumeQueue 中去，即不处理， 所以不会被消息

更新 transaction stable table： 如果是 prepared 消息记，通过 TransactionStateService 服务将消息加到存储事务状态的表格 tranStateTable 的文件中；如果是 commitType 和 rollbackType 消息， 修改事物状态表格 tranStateTable 中的消息状态。

记录 Transaction Redo Log 日志： 记录了 commitLogOffset, msgSize, preapredTransactionOffset, storeTimestamp。

2.2.3 事物状态表

事物状态表是有 MapedFileQueue 将多个文件组成一个连续的队列，它的存储单元是定长为 24 个字节的数据，

tranStateTableOffset 可以认为是事物状态消息的个数，索引偏移量， 它的值是 tranStateTable.getMaxOffset() / TSStoreUnitSize

Offset(Commit Log)	Size	Timestamp	Group Hash Code	State (P/C/R)

2.2.4 事物回查

定时回查线程会定时扫描（默认每分钟）每个存储事务状态的表格文件， 遍历存储事务状态的表格记录

如果是已经提交或者回滚的消息调过过，

如果是 prepared 状态的如果消息小于事务回查至少间隔时间（默认是一分钟）跳出终止遍历

调 transactionCheckExecuter.gotocheck 方法向 producer 回查事物状态，

根据 group 随机选择一台 producer

查询消息，根据 commitLogOffset 和 msgSize 到 commitlog 查找消息

向 Producder 发起请求,请求 code 类型为 CHECK_TRANSACTION_STATE，producer 的 DefaultMQProducerImpl. checkTransactionState()方法来处理 broker 定时回调的请求， 这里构建一个 Runnable 任务异步执行 producer 注册的回调接口，处理回调，在调 endTransactionOneway 向 broker 发送请求更新事物消息的最终状态

无 Prepared 消息，且遍历完，则终止扫描这个文件的定时任务

2.2.5 事物消息的 load&recover

TransactionStateService.load ()事物状态服务加载， 加载只是建立文件映射

redoLog 队列恢复，加载本地 redoLog 文件

tranStateTable 事物状态表， 加载本地 tranStateTable 文件

recover:

正常恢复：

利用 tranRedoLog 文件的 recover

利用 tranStateTable 文件重建事物状态表

异常恢复：

先按照正常流程恢复 Tran Redo Log

commitLog 异常恢复，commitLog 根据 checkpoint 时间点重新生成 redolog， 重新分发消息 DispatchRequest,

分发消息到位置信息到 ConsumeQueue

更新 Transaction State Table

记录 Transaction Redo Log

删除事物状态表 tranStateTable

通过 RedoLog 全量恢复 StateTable

重头扫描 RedoLog， 过滤出所有 prepared 状态的消息， 将 commit 或者 rollback 的消息对应的 prepared 消息删除

重建 StateTable， 将上面过滤出的 prepared 消息，添加到事物状态表文件中

这个事物状态表 transstable 的作用是定期(1 分钟)将状态为 prepared 事物回查 producer 端 redolog 这个队列其实标记消费到哪了， 事物状态的恢复根本上是有 commitlog 来做的

第二章 consumer

有别于其他消息中间件由 broker 做负载均衡并主动向 consumer 投递消息，RocketMq 是基于拉模式拉取消息，consumer 做负载均衡并通过长轮询向 broker 拉消息。

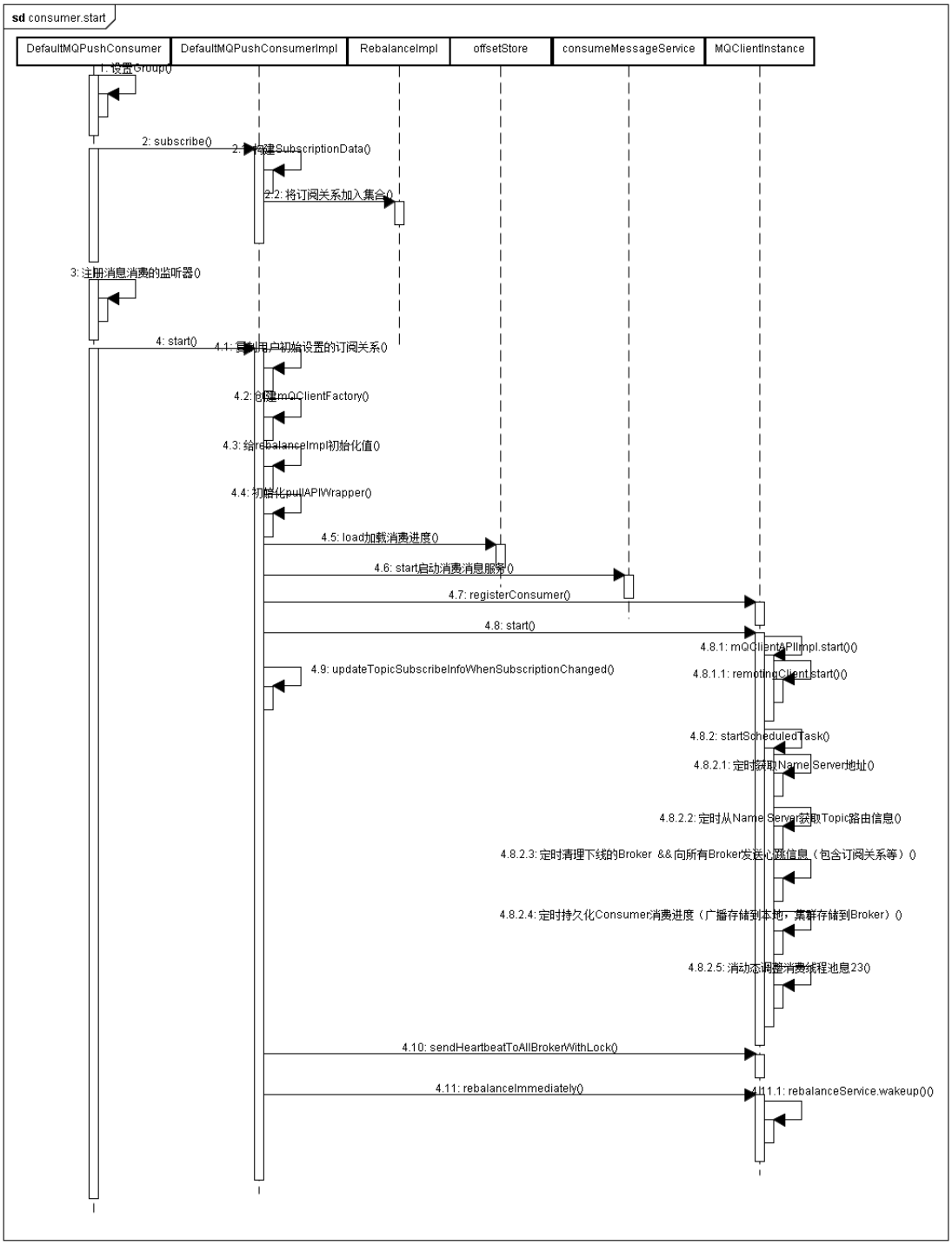
Consumer 消费拉取的消息的方式有两种

1. Push 方式：rocketmq 已经提供了很全面的实现， consumer 通过长轮询拉取消息后回调 MessageListener 接口实现完成消费， 应用系统只要 MessageListener 完成业务逻辑即可
2. Pull 方式：完全由业务系统去控制，定时拉取消息，指定队列消费等等， 当然这里需要业务系统去根据自己的业务需求去实现

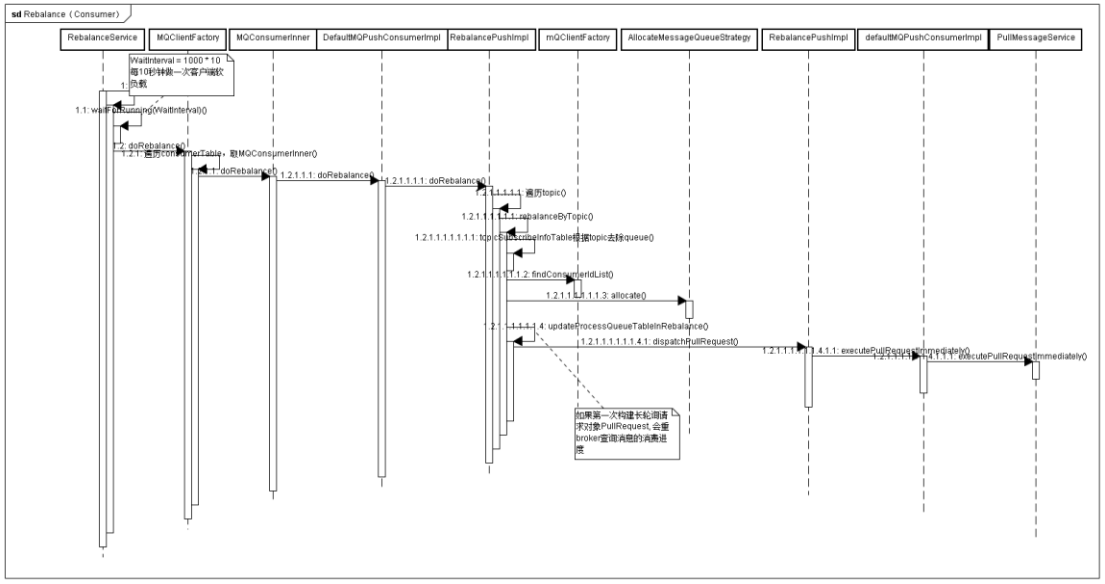
下面介绍默认以 push 方式为主， 因为绝大多数是由 push 消费方式来使用 rocketmq 的。

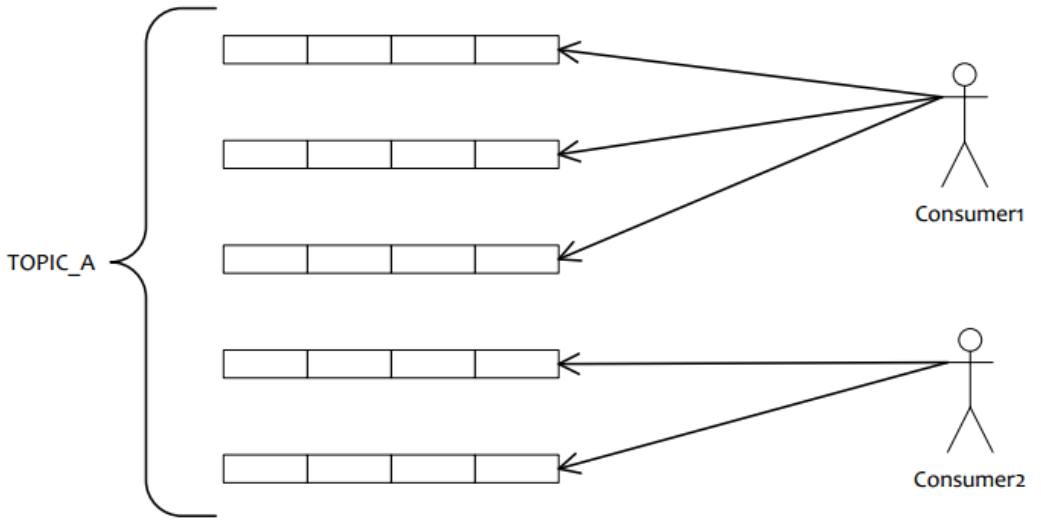
一：consumer 启动流程

指定 group
订阅 topic
注册消息监听处理器，当消息到来时消费消息
消费端 Start
    复制订阅关系
    初始化 rebalance 变量
    构建 offsetStore 消费进度存储对象
    启动消费消息服务
    向 mqClientFactory 注册本消费者
    启动 client 端远程通信
    启动定时任务
        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）
        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
        动态调整消费线程池
    启动拉消息服务 PullMessageService
    启动消费端负载均衡服务 RebalanceService
    从 namesrv 更新 topic 路由信息
    向所有 broker 发送心跳信息，（包括订阅关系）
    唤醒 Rebalance 服务线程


sd consumer.start
DefaultMQPushConsumer
DefaultMQPushConsumerImpl
RebalanceImpl
offsetStore
consumeMessageService
MQClientInstance
1: 设置Group()
2: subscribe()
2.1: 构建SubscriptionData()
2.2: 将订阅关系加入集合()
3: 注册消息消费的监听器()
4: start()
4.1: 复制用户初始设置的订阅关系()
4.2: 创建mQClientFactory()
4.3: 给rebalanceImpl初始化值()
4.4: 初始化pullAPIWrapper()
4.5: load加载消费进度()
4.6: start启动消费消息服务()
4.7: registerConsumer()
4.8: start()
4.8.1: mQClientAPIImpl.start()()
4.8.1.1: remotingClient.start()()
4.9: updateTopicSubscribeInfoWhenSubscriptionChanged()
4.8.2: startScheduledTask()
4.8.2.1: 定时获取NameServer地址()
4.8.2.2: 定时从NameServer获取Topic路由信息()
4.8.2.3: 定时清理下线的Broker &&向所有Broker发送心跳信息（包含订阅关系等）()
4.8.2.4: 定时持久化Consumer消费进度（广播存储到本地，集群存储到Broker）()
4.8.2.5: 消动态调整消费线程池息23()
4.10: sendHeartbeatToAllBrokerWithLock()
4.11: rebalanceImmediately()
4.11.1: rebalanceService.wakeup()()

二：消费端负载均衡





消费端会通过 RebalanceService 线程，10 秒钟做一次基于 topic 下的所有队列负载

消费端遍历自己的所有 topic，依次调 rebalanceByTopic

根据 topic 获取此 topic 下的所有 queue

选择一台 broker 获取基于 group 的所有消费端（有心跳向所有 broker 注册客户端信息）

选择队列分配策略实例 AllocateMessageQueueStrategy 执行分配算法,获取队列集合 Set<MessageQueue> mqSet

1） 平均分配算法，其实是类似于分页的算法

将所有 queue 排好序类似于记录

将所有消费端 consumer 排好序，相当于页数

然后获取当前 consumer 所在页面应该分配到的 queue

2） 按照配置来分配队列， 也就是说在 consumer 启动的时候指定了 queue

3） 按照机房来配置队列

Consumer 启动的时候会指定在哪些机房的消息

获取指定机房的 queue

然后在执行如 1）平均算法

根据分配队列的结果更新 ProccessQueueTable<MessageQueue, ProcessQueue>

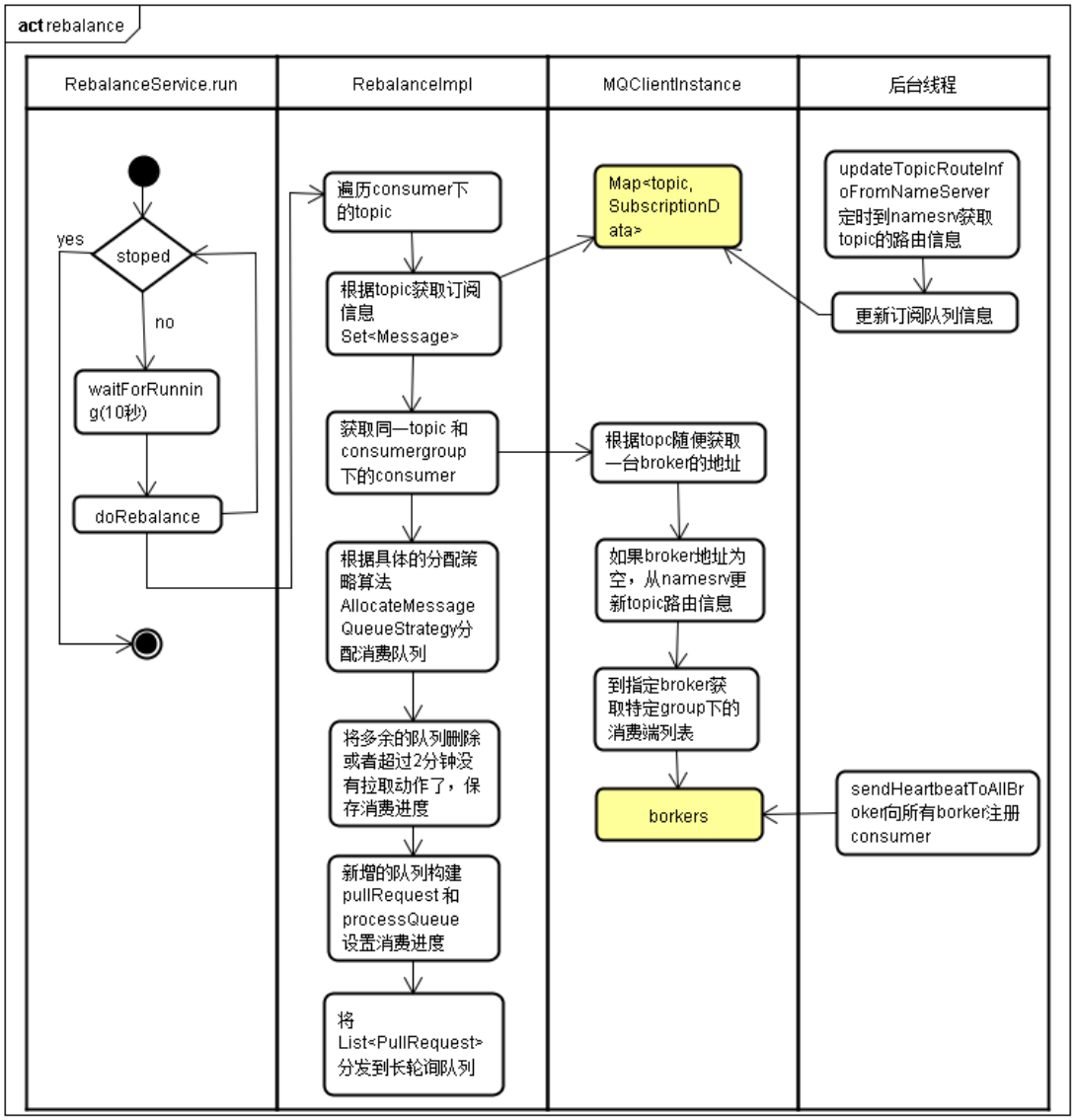
1) 比对 mqSet 将多余的队列删除， 当 broker 当机或者添加，会导致分配到 mqSet 变化，
   a) 将不在被本 consumer 消费的 messagequeue 的 ProcessQueue 删除， 其实是设置 ProcessQueue 的 droped 属性为 true
   b) 将超过两份中没有拉取动作 ProcessQueue 删除
      //TODO 为什么要删除掉，两分钟后来了消息怎么办?
      //
2) 添加新增队列， 比对 mqSet，给新增的 messagequeue
   构建长轮询对象 PullRequest 对象，会从 broker 获取消费的进度
   构建这个队列的 ProcessQueue
   将 PullRequest 对象派发到长轮询拉消息服务（单线程异步拉取）

注：ProcessQueue 正在被消费的队列，

(1) 长轮询拉取到消息都会先存储到 ProcessQueue 的 TreeMap<Long, MessageExt> 集合中，消费调后会删除掉，用来控制 consumer 消息堆积，
TreeMap<Long, MessageExt> key 是消息在此 ConsumeQueue 队列中索引

(2) 对于顺序消息消费 处理
locked 属性:当 consumer 端向 broker 申请锁队列成功后设置 true，只有被锁定的 processqueue 才能被执行消费
rollback: 将消费在 msgTreeMapTemp 中的消息，放回 msgTreeMap 重新消费
commit: 将临时表 msgTreeMapTemp 数据清空，代表消费完成，放回最大偏移值

(3) 这里是个 TreeMap，对 key 即消息的 offset 进行排序，这个样可以使得消息进行顺序消费


act rebalance
RebalanceService.run
RebalanceImpl
MQClientInstance
后台线程
yes
stoped
no
waitForRunning(10秒)
doRebalance
遍历consumer下的topic
根据topic获取订阅信息 Set<Message>
获取同一topic 和 consumergroup 下的consumer
根据具体的分配策略算法 AllocateMessageQueueStrategy分配消费队列
将多余的队列删除或者超过2分钟没有拉取动作了，保存消费进度
新增的队列构建 pullRequest 和 processQueue 设置消费进度
将 List<PullRequest> 分发到长轮询队列
Map<topic, SubscriptionData>
根据topic随便获取一台broker的地址
如果broker地址为空，从namesrv更新topic路由信息
到指定broker获取特定group下的消费端列表
borkers
updateTopicRouteInfoFromNameServer 定时到namesrv获取topic的路由信息
更新订阅队列信息
sendHeartbeatToAllBroker向所有borker注册consumer

三： 长轮询

Rocketmq 的消息是由 consumer 端主动到 broker 拉取的, consumer 向 broker 发送拉消息请求， PullMessageService 服务通过一个线程将阻塞队列 LinkedBlockingQueue<PullRequest> 中的 PullRequest 到 broker 拉取消息

DefaultMQPushConsumerImpl 的 pullMessage(pullRequest)方法执行向 broker 拉消息动作

1. 获取 ProcessQueue 判读是否 drop 的， drop 为 true 返回
2. 给 ProcessQueue 设置拉消息时间戳
3. 流量控制，正在消费队列中消息（未被消费的）超过阀值，稍后在执行拉消息
4. 流量控制，正在消费队列中消息的跨度超过阀值（默认 2000），稍后在消费
5. 根据 topic 获取订阅关系
6. 构建拉消息回调对象 PullBack, 从 broker 拉取消息（异步拉取）返回结果是回调
7. 从内存中获取 commitOffsetValue //TODO 这个值跟 pullRequest.getNextOffset 区别
8. 构建 sysFlag pull 接口用到的 flag
9. 调底层通信层向 broker 发送拉消息请求

   如果 master 压力过大，会建议去 slave 拉取消息

   如果是到 broker 拉取消息清楚实时提交标记位，因为 slave 不允许实时提交消费进度，可以定时提交

   //TODO 关于 master 拉消息实时提交指的是什么？
10. 拉到消息后回调 PullCallback

    处理 broker 返回结果 pullResult

    更新从哪个 broker（master 还是 slave）拉取消息

    反序列化消息

    消息过滤

    消息中放入队列最大最小 offset, 方便应用来感知消息堆积度

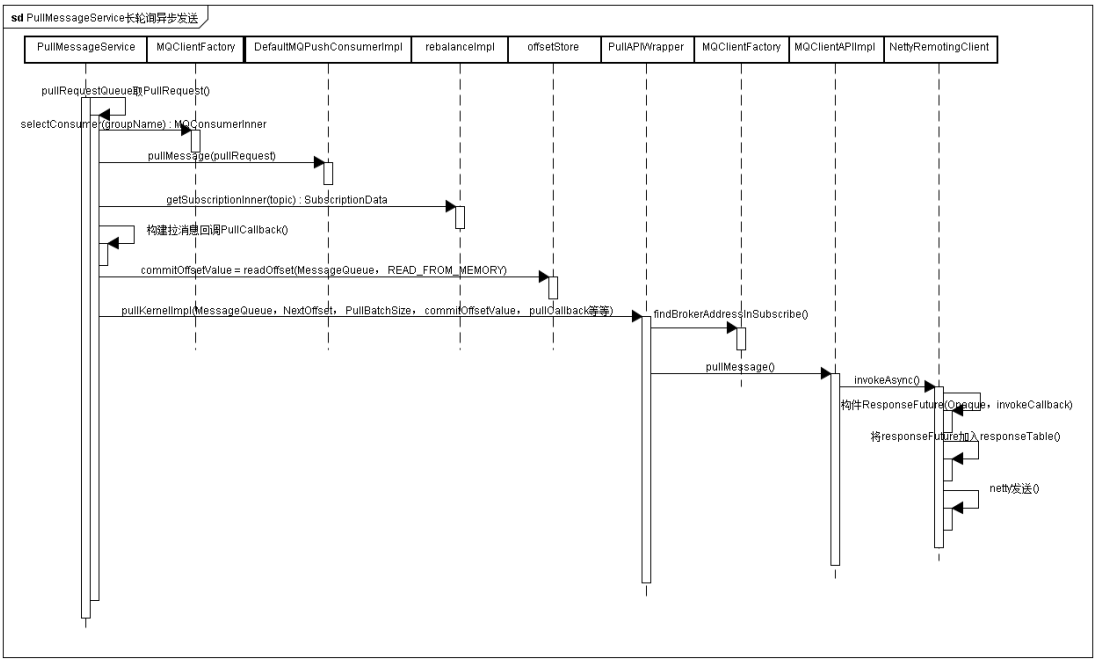
    将消息加入正在处理队列 ProcessQueue

    将消息提交到消费消息服务 ConsumeMessageService

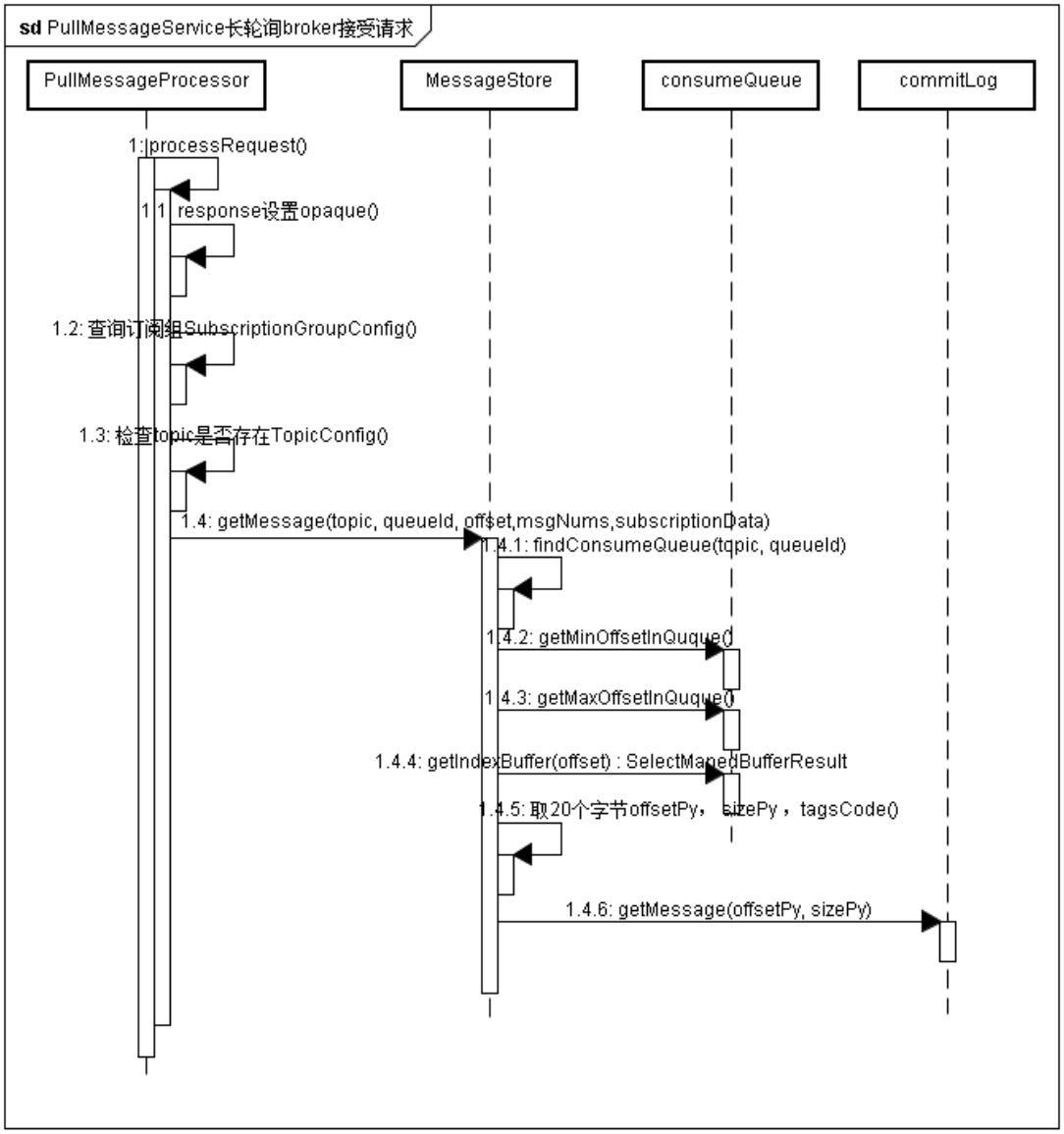
    流控处理， 如果 pullInterval 参数大于 0 （拉消息间隔，如果为了降低拉取速度，可以设置大于 0 的值）， 延迟再执行拉消息， 如果 pullInterval 为 0 立刻在执行拉消息动作

序列图

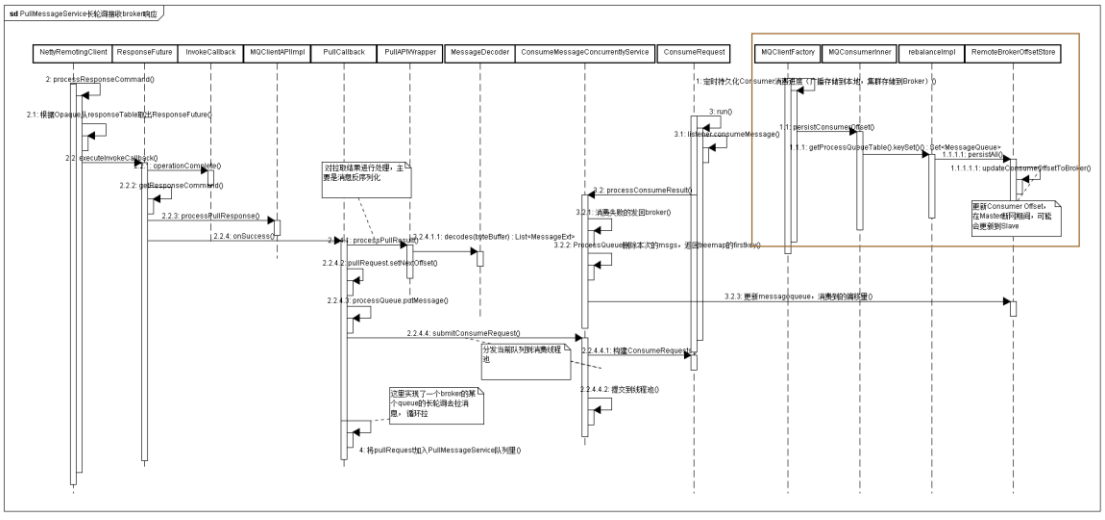
1. 向 broker 发送长轮询请求



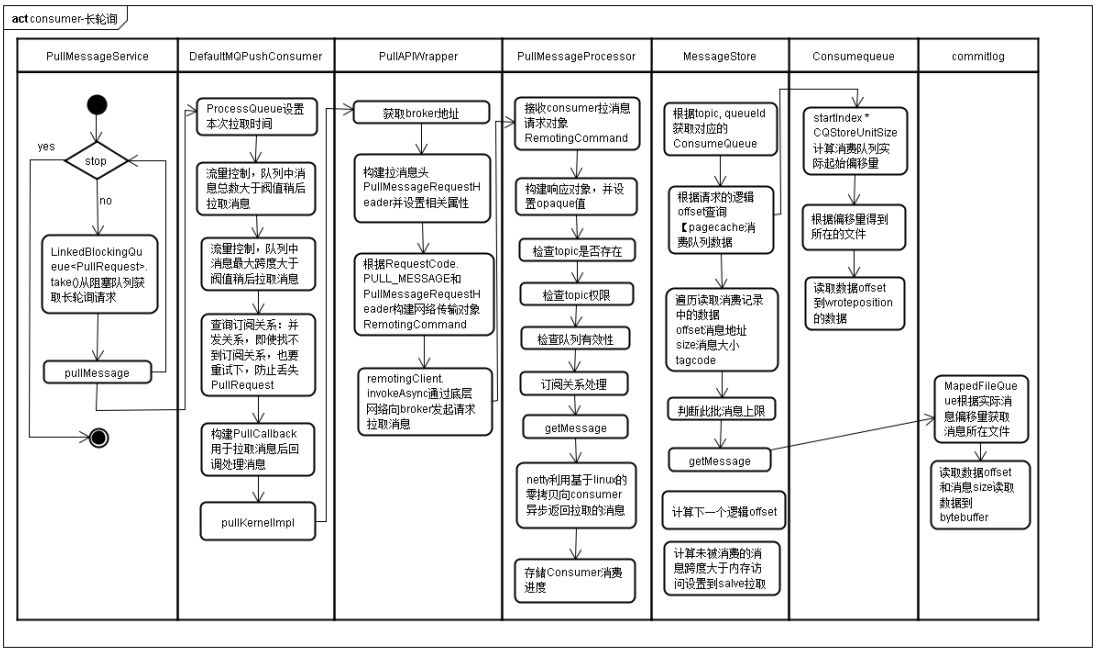
2. Broker 接收长轮询请求



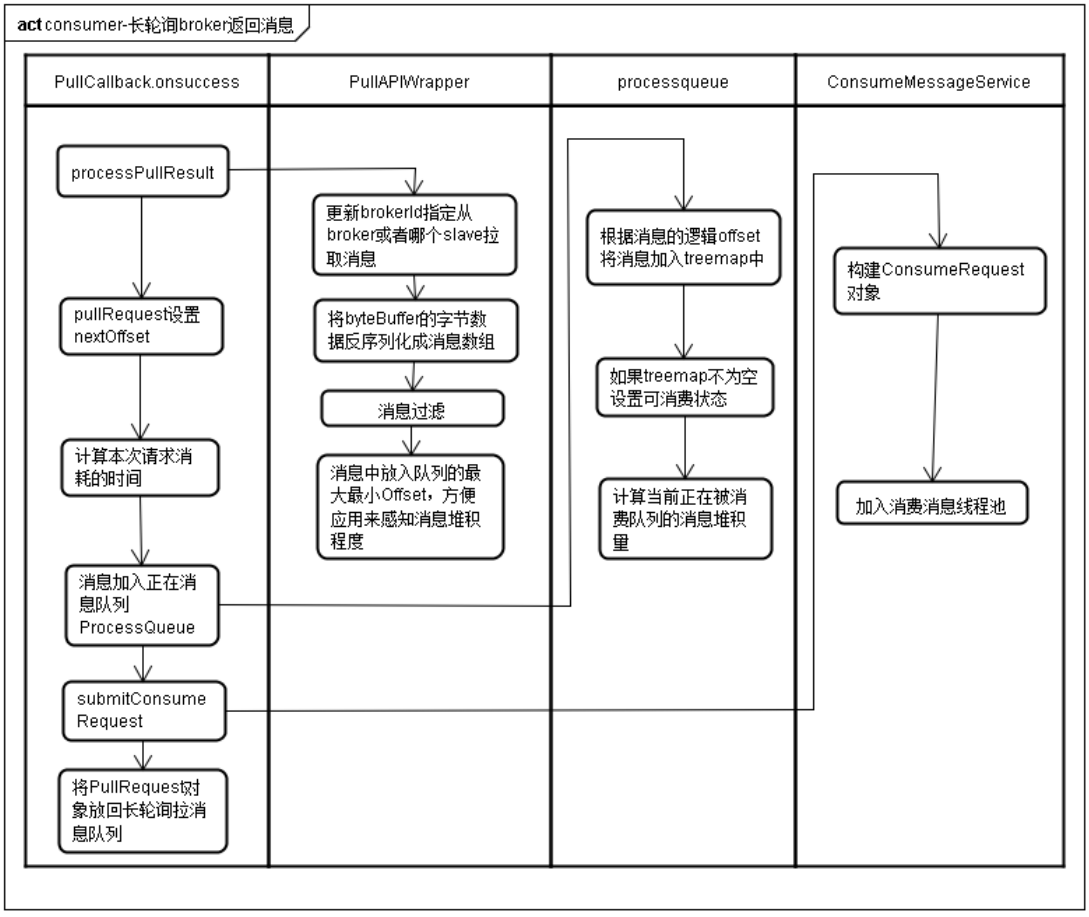
3. Consumer 接收 broker 响应



长轮询活动图：



一张图画不下，再来一张



四：push 消息—并发消费消息

通过长轮询拉取到消息后会提交到消息服务 ConsumeMessageConcurrentlyService，ConsumeMessageConcurrentlyServic 的 submitConsumeRequest 方法构建 ConsumeRequest 任务提交到线程池。

长轮询向 broker 拉取消息是批量拉取的，默认设置批量的值为 pullBatchSize = 32，可配置

消费端 consumer 构建一个消费消息任务 ConsumeRequest 消费一批消息的个数是可配置的 consumeMessageBatchMaxSize = 1， 默认批量个数为一个

ConsumeRequest 任务 run 方法执行

判断 proccessQueue 是否被 droped 的， 废弃直接返回，不在消费消息

构建并行消费上下文

给消息设置消费失败时候的 retry topic，当消息发送失败的时候发送到 topic 为%RETRY%groupname 的队列中

调 MessageListenerConcurrently 监听器的 consumeMessage 方法消费消息，返回消费结果

如果 ProcessQueue 的 droped 为 true，不处理结果，不更新 offset， 但其实这里消费端是消费了消息的，这种情况感觉有被重复消费的风险

处理消费结果

消费成功，对于批次消费消息，返回消费成功并不代表所有消息都消费成功，但是消费消息的时候一旦遇到消费消息失败直接放回，根据 ackIndex 来标记成功消费到哪里了

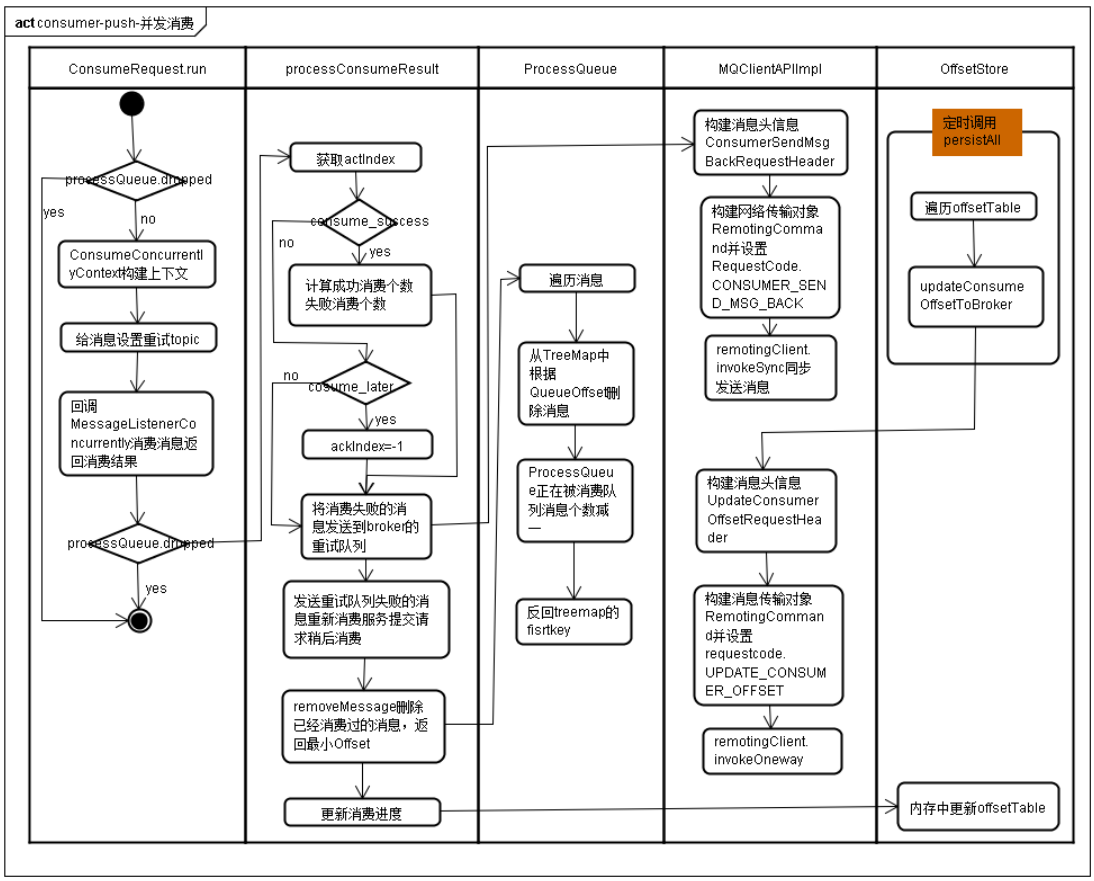
消费失败， ackIndex 设置为-1

广播模式发送失败的消息丢弃， 广播模式对于失败重试代价过高，对整个集群性能会有较大影响，失败重试功能交由应用处理

集群模式， 将消费失败的消息一条条的发送到 broker 的重试队列中去，如果此时还有发送到重试队列发送失败的消息，那就在 cosumer 的本地线程定时 5 秒钟以后重试重新消费消息， 在走一次上面的消费流程。

删除正在消费的队列 processQueue 中本次消费的消息，放回消费进度

更新消费进度， 这里的更新只是一个内存 offsetTable 的更新，后面有定时任务定时更新到 broker 上去


act consumer-push-并发消费
ConsumeRequest.run
processConsumeResult
ProcessQueue
MQClientAPIImpl
OffsetStore
processQueue.dropped
yes
no
ConsumeConcurrentlyContext构建上下文
给消息设置重试topic
回调MessageListenerConcurrently消费消息返回消费结果
processQueue.dropped
yes
获取actIndex
consume_success
no
yes
计算成功消费个数失败消费个数
consume_later
no
yes
ackIndex=-1
将消费失败的消息发送到broker的重试队列
发送重试队列失败的消息重新消费服务提交请求稍后消费
removeMessage删除已经消费过的消息，返回最小Offset
更新消费进度
遍历消息
从TreeMap中根据QueueOffset删除消息
ProcessQueue正在被消费队列消息个数减一
反回treemap的fisrtkey
构建消息头信息ConsumerSendMsgBackRequestHeader
构建网络传输对象RemotingCommand并设置RequestCode.CONSUMER_SEND_MSG_BACK
remotingClient.invokeSync同步发送消息
构建消息头信息UpdateConsumerOffsetRequestHeader
构建消息传输对象RemotingCommand并设置requestcode.UPDATE_CONSUMER_OFFSET
remotingClient.invokeOneway
定时调用persistAll
遍历offsetTable
updateConsumeOffsetToBroker
内存中更新offsetTable

五：push 消费-顺序消费消息

顺序消费服务 ConsumeMessageConcurrentlyService 构建的时候

构建一个线程池来接收消费请求 ConsumeRequest

构建一个单线程的本地线程, 用来稍后定时重新消费 ConsumeRequest, 用来执行定时周期性（一秒）钟锁队列任务

周期性锁队列 lockMQPeriodically

获取正在消费队列列表 ProcessQueueTable 所有 MesssageQueue， 构建根据 broker 归类成 MessageQueue 集合 Map<brokername, Set<MessageQueue>>

遍历 Map<brokername, Set<MessageQueue>>的 brokername, 获取 broker 的 master 机器地址，将 brokerName 的 Set<MessageQueue>发送到 broker 请求锁定这些队列。在 broker 端锁定队列，其实就是在 broker 的 queue 中标记一下消费端，表示这个 queue 被某个 client 锁定。 Broker 会返回成功锁定队列的集合， 根据成功锁定的 MessageQueue,设置对应的正在处理队列 ProccessQueue 的 locked 属性为 true 没有锁定设置为 false

通过长轮询拉取到消息后会提交到消息服务 ConsumeMessageOrderlyService，ConsumeMessageOrderlyService 的 submitConsumeRequest 方法构建 ConsumeRequest 任务提交到线程池。ConsumeRequest 是由 ProcessQueue 和 Messagequeue 组成。

ConsumeRequest 任务的 run 方法

判断 proccessQueue 是否被 droped 的， 废弃直接返回，不在消费消息

每个 messagequeue 都会生成一个队列锁来保证在当前 consumer 内，同一个队列串行消费，

判断 processQueue 的 lock 属性是否为 true，lock 属性是否过期，如果为 false 或者过期，放到本地线程稍后锁定在消费。 如果 lock 为 true 且没有过期，开始消费消息

计算任务执行的时间如果大于一分钟且线程数小于队列数情况下，将 processqueue，messagequeue 重新构建 ConsumeRequest 加到线程池 10ms 后在消费，这样防止个别队列被饿死

获取客户端的消费批次个数，默认一批次为一条

从 proccessqueue 获取批次消息，processqueue.takeMessags(batchSize)，从 msgTreeMap 中移除消息放到临时 map 中 msgTreeMapTemp， 这个临时 map 用来回滚消息和 commit 消息来实现事物消费

调回调接口消费消息，返回状态对象 ConsumeOrderlyStatus

根据消费状态，处理结果

1） 非事物方式，自动提交

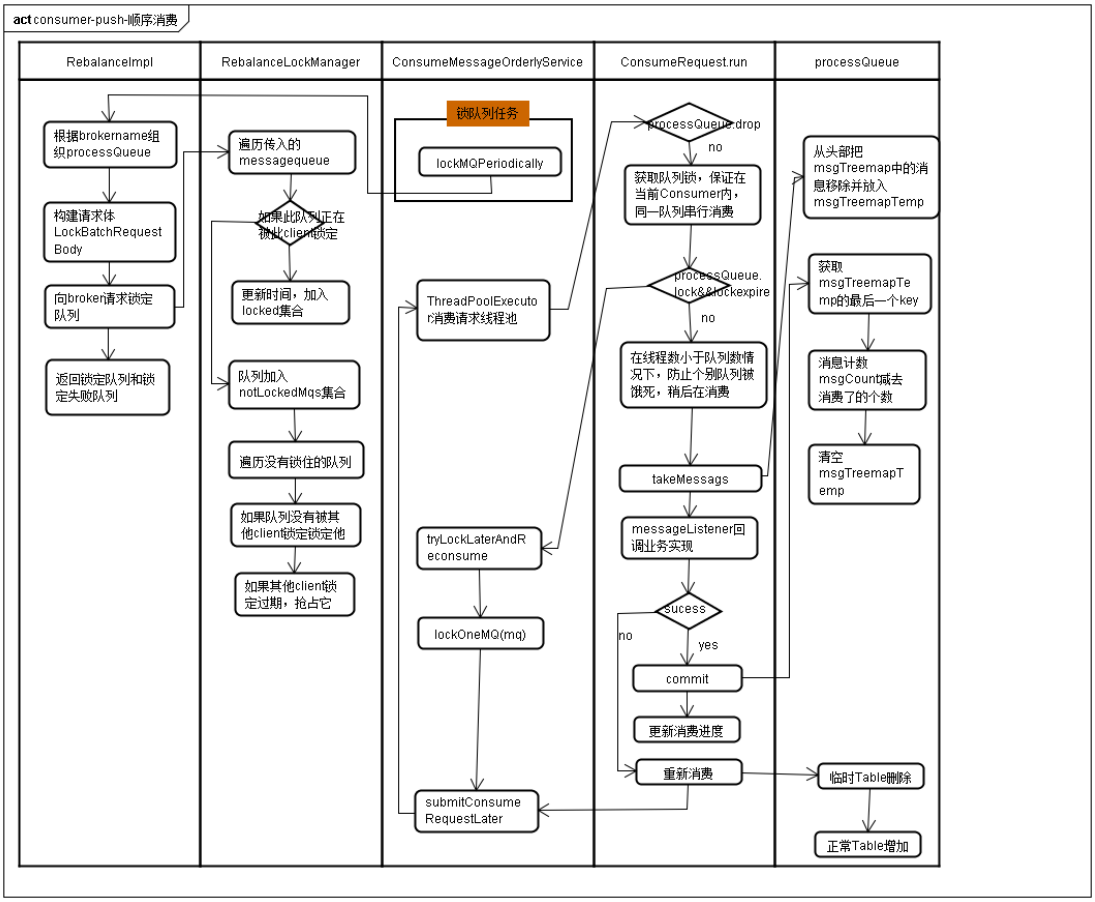
消息消息状态为 success： 调用 processQueue.commit 方法

获取 msgTreeMapTemp 的最后一个 key，表示提交的 offset

清空 msgTreeMapTemp 的消息，已经成功消费

2） 事物提交，由用户来控制提交回滚（精卫专用）

更新消费进度， 这里的更新只是一个内存 offsetTable 的更新，后面有定时任务定时更新到 broker 上去


act consumer-push-顺序消费
RebalanceImpl
RebalanceLockManager
ConsumeMessageOrderlyService
ConsumeRequest.run
processQueue
根据brokername组织processQueue
构建请求体LockBatchRequestBody
向broker请求锁定队列
返回锁定队列和锁定失败队列
遍历传入的messagequeue
如果此队列正在被此client锁定
更新时间，加入locked集合
队列加入notLockedMqs集合
遍历没有锁住的队列
如果队列没有被其他client锁定锁定他
如果其他client锁定过期，抢占它
锁队列任务
lockMQPeriodically
ThreadPoolExecutor消费请求线程池
tryLockLaterAndReconsume
lockOneMQ(mq)
submitConsumeRequestLater
processQueue.drop
no
获取队列锁，保证在当前Consumer内，同一队列串行消费
processQueue.lock&&!lockexpire
no
在线程数小于队列数情况下，防止个别队列被饿死，稍后在消费
takeMessags
messageListener回调业务实现
sucess
no
yes
commit
更新消费进度
重新消费
从头部把msgTreemap中的消息移除并放入msgTreemapTemp
获取msgTreemapTemp的最后一个key
消息计数msgCount减去消费了的个数
清空msgTreemapTemp
临时Table删除
正常Table增加

六：pull 消息消费

消费者主动拉取消息消费，客户端通过类 DefaultMQPullConsumer

客户端可以指定特定 MessageQueue

也可以通过 DefaultMQPullConsumer. fetchMessageQueuesInBalance(topic) 获取消费的队列

业务自己获取消费队列，自己到 broker 拉取消息，以及自己更新消费进度

因为内部实现跟 push 方式类似就不在啰嗦，用法也请求看示例代码去

七：shutdown

DefaultMQPushConsumerImpl 关闭消费端

关闭消费线程

将分配到的 Set<MessageQueue>的消费进度保存到 broker

利用 DefaultMQPushConsumerImpl 获取 ProcessQueueTable<MessageQueue, ProcessQueue>的 keyset 的 messagequeue 去获取

RemoteBrokerOffsetStore.offsetTable<MessageQueue, AutomicLong>Map 中的消费进度，

offsetTable 中的 messagequeue 的值，在 update 的时候如果没有对应的 Messagequeue 会构建， 但是也会 rebalance 的时候将没有分配到的 messagequeue 删除

rebalance 会将 offsettable 中没有分配到 messagequeue 删除， 但是在从 offsettable 删除之前会将 offset 保存到 broker

Unregiser 客户端

pullMessageService 关闭

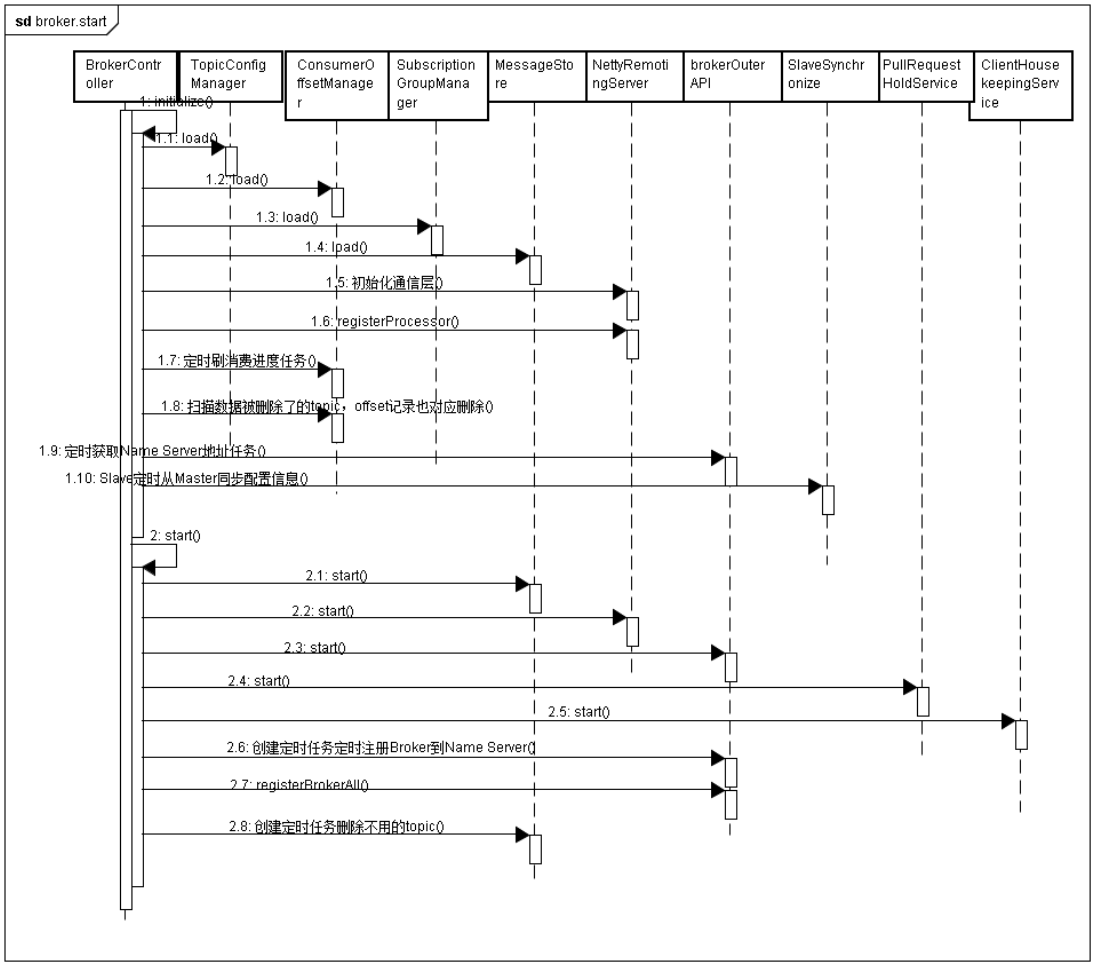
scheduledExecutorService 关闭，关闭一些客户端的起的定时任务

mqClientApi 关闭

rebalanceService 关闭

第三章： broker

一：brker 的启动



Broker 向 namesrv 注册

1. 获取 namesrv 的地址列表（是乱序的）
2. 遍历向每个 namesrv 注册 topic 的配置信息 topicconfig

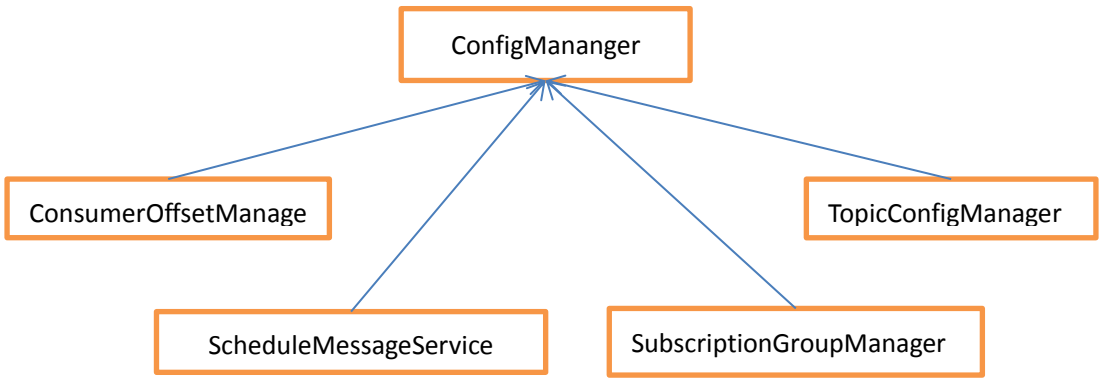
Topic 在 broker 文件上的存储 json 格式

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

Namesrv 接收 Broker 注册的 topic 信息， namesrv 只存内存，但是 broker 有任务定时推送

1. 接收数据向 RouteInfoManager 注册。

Broker 初始化加载本地配置，配置信息是以 json 格式存储在本地， rocketmq 强依赖 fastjson 作转换， RocketMq 通过 ConfigMananger 来管理配置加载以及持久化



1. 加载 topic 配置${user.home}/store/config/topics.json

```
{
    "dataVersion":{
        "counter":2,
        "timestatmp":1393729865073
    },
    "topicConfigTable":{
        //根据 consumer 的 group 生成的重试 topic
        "%RETRY% group_name":{
            "perm":6,
            "readQueueNums":1,
            "topicFilterType":"SINGLE_TAG",
            "topicName":"%RETRY% group_name",
            "writeQueueNums":1
        },
        "TopicTest":{
            "perm":6,   // 100 读权限 , 10 写权限     6 是 110 读写权限
            "readQueueNums":8,
            "topicFilterType":"SINGLE_TAG",
            "topicName":"TopicTest",
            "writeQueueNums":8
        }
    }
}
```

2. 加载消费进度偏移量 ${user.home}/store/config/consumerOffset.json

```
{
    "offsetTable":{
```

```
            "%RETRY% group_name@ group_name":{
                0:0 //重试队列消费进度为零
            },
            "TopicTest@ group_name":{
                0:23,1:23,2:22,3:22,4:21,5:18,6:18,7:18
                //分组名 group_name 消费 topic 为 TopicTest 的进度为：
                //  队列 queue=0    消费进度 23
                //  队列  queue=2    消费进度为 22    等等…
            }
        }
    }
```

3. 加载消费者订阅关系 ${user.home}/store/config/subscriptionGroup.json

```
{
    "dataVersion":{
        "counter":1,
        "timestatmp":1393641744664
    },
    " group_name":{
            "brokerId":0,    //0 代表这台 broker 机器为 master，若要设为 slave 值大于 0
            "consumeBroadcastEnable":true,
            "consumeEnable":true,
            "consumeFromMinEnable":true,
            "groupName":" group_name",
            "retryMaxTimes":5,
            "retryQueueNums":1,
            "whichBrokerWhenConsumeSlowly":1
        }
    }
}
```

二：消息存储

Rocketmq 的消息的存储是由 consume queue 和 commitLog 配合完成的

1） 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 都有一个对应的 consumequeue 文件。

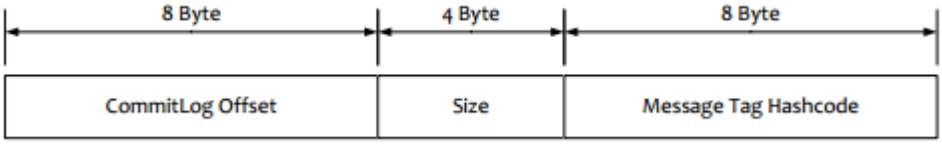
文件地址：${user.home} \store\consumequeue\${topicName}\${queueId}\${fileName}

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 文件实际偏移量

（2） size 就是指消息大小

（3） 消息 tag 的哈希值



ConsumeQueue 文件组织：

(1) topic queueId 来组织的，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.

(2) 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

(3) 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

(4) 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
```

2） 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

3） MapedFile 是 PageCache 文件封装，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

以下 io 对象构建了物理文件映射内存的对象

FileChannel fileChannel = new RandomAccessFile(file, “rw”).getChannel();

MappedByteBuffer mappedByteBuffer=fileChannel.map(READE_WRITE,0,fileSize);

构建 mapedFile 对象需要两个参数

fileSize: 映射的物理文件的大小

commitLog 每个文件的大小默认 1G = 1024*1024*1024

ConsumeQueue 每个文件默认存 30W 条 = 300000 * CQStoreUnitSize(每条大小)

filename: filename 文件名称但不仅仅是名称还表示文件记录的初始偏移量， 文件名其

实是个 long 类型的值

4） 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

```
|-- commitlog
|    |-- 00000003384434229248
|    |-- 00000003385507971072
|    `-- 00000003386581712896
```

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

5) DefaultMessageStore 消息存储层实现

(1) putMessage 添加消息委托给 commitLog.putMessage(msg)，主要流程：

<1> 从 mapedFileQueue 获取最新的映射文件

<2>向 mapedFile 中添加一条消息记录

<3> 构建 DispatchRequest 对象，添加到分发索引服务 DispatchMessageService 线程中去

<4>唤醒异步刷盘线程

<5> 向发送方返回结果

(2)DispatchMessageService

<1>分发消息位置到 ConsumeQueue

<2>分发到 IndexService 建立索引

三：load&recover

Broker 启动的时候需要加载一系列的配置，启动一系列的任务，主要分布在 BrokerController 的 initialize()和 start()方法中

1. 加载 topic 配置
2. 加载消费进度 consumer offset
3. 加载消费者订阅关系 consumer subscription
4. 加载本地消息 messageStore.load()
   a) Load 定时进度
   b) Load commit log
      commitLog 其实调用存储消费队列 mapedFileQueue.load()方法来加载的。
      遍历出${user.home} \store\${commitlog}目录下所有 commitLog 文件，按文件名（文件名就是文件的初始偏移量）升序排一下， 每个文件构建一个 MapedFile 对象, 在 MapedFileQueue 中用集合 list 把这些 MapedFile 文件组成一个逻辑上连续的队列
   c) Load consume Queue
      遍历${user.home} \store\consumequeue 下的所有文件夹（每个 topic 就是一个文件夹）
      遍历${user.home} \store\consumequeue\${topic}下的所有文件夹（每个 queueId 就是一个文件夹）
      遍历${user.home} \store\consumequeue\${topic}\${queueId}下所有文件，根据 topic，queueId， 文件来构建 ConsueQueue 对象
      DefaultMessageStore 中存储结构 Map<topic, Map<queueId, CosnueQueue>>

      每个 Consumequeue 利用 MapedFileQueue 把 mapedFile 组成一个逻辑上连续的队列
   d) 加载事物模块

   e) 加载存储检查点
      加载${user.home} \store\checkpoint 这个文件存储了 3 个 long 类型的值来记录存储模型最终一致的时间点，这个 3 个 long 的值为
      physicMsgTimestamp 为 commitLog 最后刷盘的时间
      logicMsgTimestamp 为 consumeQueue 最终刷盘的时间
      indexMsgTimestamp 为索引最终刷盘时间

      checkpoint 作用是当异常恢复时需要根据 checkpoint 点来恢复消息

   f) 加载索引服务 indexService

   g) recover 尝试数据恢复
      判断是否是正常恢复，系统启动的启动存储服务(DefaultMessageStore)的时候会创建一个临时文件 abort, 当系统正常关闭的时候会把这个文件删掉 ，这个类似在 linux 下打开 vi 编辑器生成那个临时文件， 所有当这个 abort 文件存在，系统认为是异常恢复

1） 先按照正常流程恢复 Consume Queue

为什么说先正常恢复，那么异常恢复在哪呢？ 当 broker 是异常启动时候，在异常恢复 commitLog 时会重新构建请到 DispatchMessageService 服务，来重新生成 ConsumeQueue 数据，索引以及事物消息的 redolog

什么是恢复 ConsumeQueue，前面不是有步骤 load 了 ConsumeQueue 吗，为什么还要恢复?

前面 load 步骤创建了 MapedFile 对象建立了文件的内存映射，但是数据是否正确，现在文件写到哪了(wrotePosition), Flush 到了什么位置(committedPosition)? 恢复数据来帮我解决这些问题。

每个 ConsumeQueue 的 mapedFiles 集合中，从倒数第三个文件开始恢复(为什么只恢复倒数三个文件，也许只是个经验值吧)，因为 consumequeue 的存储单元是 20 字节的定长数据，所以是依次分别取了

Offset long 类型存储了 commitLog 的数据偏移量

Size int 类型存储了在 commitLog 上消息大小

tagcode tag 的哈希值

目前 rocketmq 判断存储的 consumequeue 数据是否有效的方式为判断 offset >= 0 && size > 0

如果数据有效读取下 20 个字节判断是否有效

如果数据无效跳出循环，记录此时有效数据的偏移量 processOffset

如果读到文件尾，读取下一个文件

proccessOffset 是有效数据的偏移量，获取这个值的作用什么?

(1) proccessOffset 后面的数据属于脏数据，后面的文件要删除掉

(2) 设置 proccessOffset 所在文件 MapedFile 的 wrotePosition 和 commitedPosition 值， 值为 proccessOffset%mapedFileSize

2） 正常恢复 commitLog 文件

步骤跟流程恢复 Consume Queue

判断消息有效， 根据消息的存储格式读取消息到 DispatchRequest 对象， 获取消息大小值 msgSize

大于 0 正常数据

等于-1 文件读取错误 恢复结束

等于 0 读到文件末尾

3） 异常数据恢复，OS CRASH 或者 JVM CRASH 或者机器掉电

当${user.home} \store\abort 文件存在，代表异常恢复

读取${user.home} \store\checkpoint 获取最终一致的时间点
判断最终一致的点所在的文件是哪个
从最新的 mapedFile 开始，获取存储的一条消息在 broker 的生成时间，大于 checkpoint 时间点的放弃找前一个文件，小于等于 checkpoint 时间点的说明 checkpoint 在此 mapedfile 文件中
从 checkpoint 所在 mapedFile 开始恢复数据，它的整体过程跟正常恢复 commitlog 类似，最重要的区别在于
（1）读取消息后派送到分发消息服务 DispatchMessageService 中， 来重建 ConsumeQueue 以及索引
（2）根据恢复的物理 offset，清除 ConsumeQueue 多余的数据

4） 恢复 TopicQueueTable=Map<topic-queueid,offset>
（1） 恢复写入消息时，消费记录队列的 offset
（2） 恢复每个队列的最小 offset

5. 初始化通信层
6. 初始化线程池
7. 注册 broker 端处理器用来接收 client 请求后选择处理器处理
8. 启动每天凌晨 00:00:00 统计消费量任务
9. 启动定时刷消费进度任务
10. 启动扫描数据被删除了的 topic，offset 记录也对应删除任务
11. 如果 namesrv 地址不是指定的，而是从静态服务器取的， 启动定时向静态服务器获取 namesrv 地址的任务
12. 如果 broker 是 master， 启动任务打印 slave 落后 master 没有同步的 bytes
如果 broker 是 slave，启动任务定时到 mastser 同步配置信息

四：HA & master slave

在 broker 启动的时候 BrokerController 如果是 slave， 配置了 master 地址更新，没有配置所有 broker 会想 namesrv 注册，从 namesrv 获取 haServerAddr，然后更新到 HAClient

当 HAClient 的 MasterAddress 不为空的时候（因为 broker master 和 slave 都构建了 HAClient）会主动连接 master 获取 SocketChannel

Master 监听 Slave 请求的端口，默认为服务端口+1
接收 slave 上传的 offset long 类型
int pos = this.byteBufferRead.position() - (this.byteBufferRead.position() % 8) //没有理解意图
long readOffset = this.byteBufferRead.getLong(pos - 8);
this.processPostion = pos;

主从复制 从哪里开始复制：如果请求时 0 ，从最后一个文件开始复制

Slave 启动的时候 brokerController 开启定时任务定时拷贝 master 的配置信息
SlaveSynchronize 类代表 slave 从 master 同步信息（非消息）
syncTopicConfig 同步 topic 的配置信息
syncConsumerOffset 同步消费进度
syncDelayOffset 同步定时进度
syncSubcriptionGroupConfig 同步订阅组配 7F6E

HaService 类实现了 HA 服务，负责同步双写，异步复制功能，这个类 master 和 slave 的 broker 都会实例化，
Master 通过 AcceptSocketService 监听 slave 的连接，每个 master slave 连接都会构建一个 HAConnection 对象搭建他们之间的桥梁， 对于一个 master 多 slave 部署结构的会有多个 HAConnection 实例，

Master 构建 HAConnection 时会构建向 slave 写入数据服务线程对象 WriteSocketService 对象和读取 Slave 反馈服务线程对象 ReadSocketService

4.1 WriteSocketService

向 slave 同步 commitLog 数据线程，

slaveRequestOffset 是每次 slave 同步完数据都会向 master 发送一个 ack 表示下次同步的数据的 offset。

如果 slave 是第一次启动的话 slaveRequestOffset=0， master 会从最近那个 commitLog 文件开始同步。（如果要把 master 上的所有 commitLog 文件同步到 slave 的话，把 masterOffset 值赋为 minOffset）

```
// 第一次传输，需要计算从哪里开始
// Slave如果本地没有数据，请求的Offset为0，那么master则从物理文件最后一个文件开始传送数据
if (-1 == this.nextTransferFromWhere) {
    if (0 == HAConnection.this.slaveRequestOffset) {
        long masterOffset =
                HAConnection.this.haService.getDefaultMessageStore().getCommitLog()
                        .getMaxOffset();
        masterOffset =
                masterOffset
                        - (masterOffset % HAConnection.this.haService
                            .getDefaultMessageStore().getMessageStoreConfig()
                            .getMapedFileSizeCommitLog());

        if (masterOffset < 0) {
            masterOffset = 0;
        }

        this.nextTransferFromWhere = masterOffset;
    }
```

向 socket 写入同步数据： 传输数据协议 <Phy Offset> <Body Size> <Body Data>

4.2 ReadSocketService

读取 slave 通过 HAClient 向 master 返回同步 commitLog 的物理偏移量 phyOffset 值
通知前端线程，如果是同步复制的话通知是否复制成功

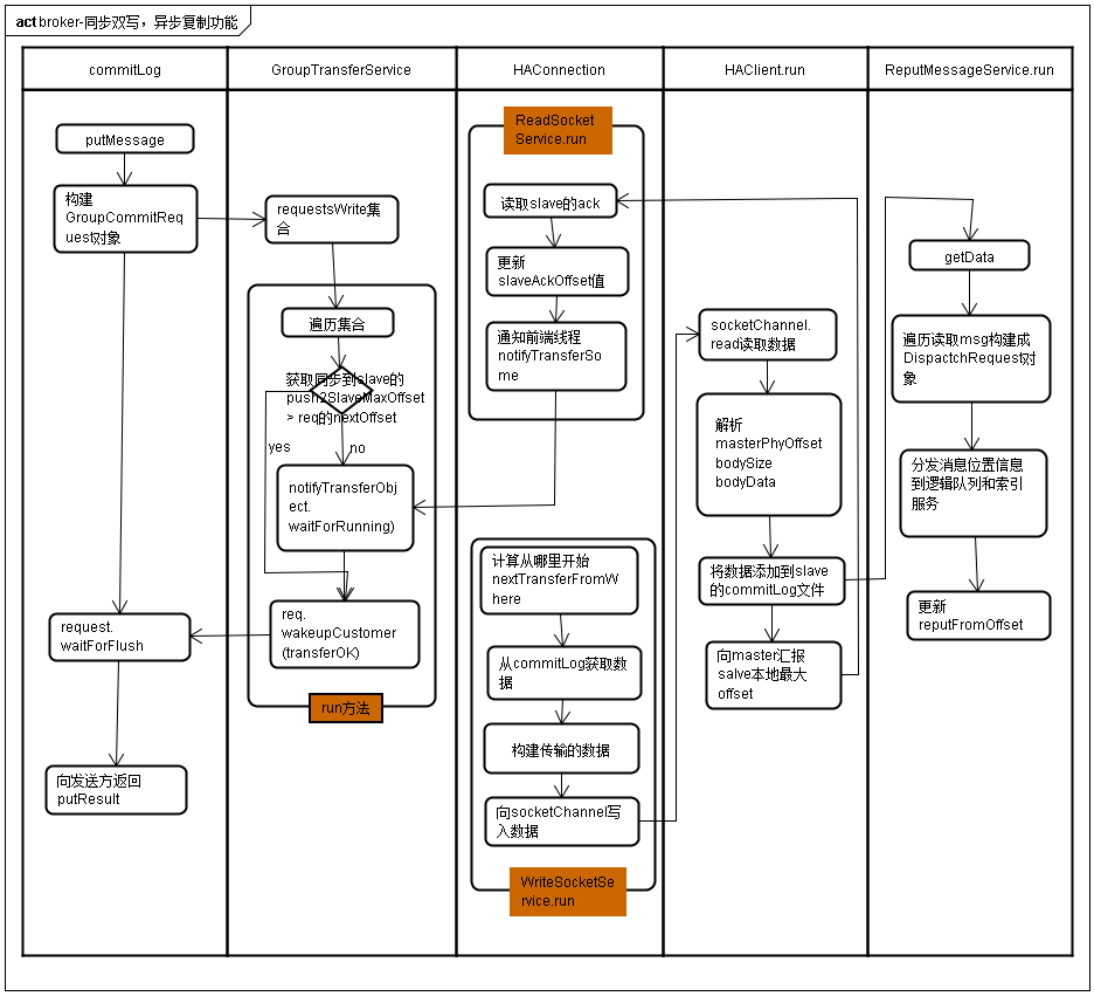
Slave 通过 HAClient 建立与 master 的连接，
来定时汇报 slave 最大物理 offset，默认 5 秒汇报一次也代表了跟 master 之间的心跳检测
读取 master 向 slave 写入 commitlog 的数据， master 向 slave 写入数据的格式是

MasterPhyOffset	bodysize	bodyData

Slave 初始化 DefaultMessageStore 时候会构建 ReputMessageService 服务线程并在启动存储服务的 start 方法中被启动
ReputMessageService 的作用是 slave 从物理队列（由 commitlog 文件构成的 MapedFileQueue）加载数据，并分发到各个逻辑队列

HA 同步复制， 当 msg 写入 master 的 commitlog 文件后，判断 maser 的角色如果是同步双写 SYNC_MASTER， 等待 master 同步到 slave 在返回结果

4.3 HA 异步复制


act broker-同步双写，异步复制功能
commitLog
GroupTransferService
HAConnection
HAClient.run
ReputMessageService.run
putMessage
构建GroupCommitRequest对象
requestsWrite集合
遍历集合
获取同步到slave的push2SlaveMaxOffset > req的nextOffset
yes
no
notifyTransferObject.waitForRunning)
req.wakeupCustomer(transferOK)
run方法
request.waitForFlush
向发送方返回putResult
ReadSocketService.run
读取slave的ack
更新slaveAckOffset值
通知前端线程notifyTransferSome
计算从哪里开始nextTransferFromWhere
从commitLog获取数据
构建传输的数据
向socketChannel写入数据
WriteSocketService.run
socketChannel.read读取数据
解析masterPhyOffset bodySize bodyData
将数据添加到slave的commitLog文件
向master汇报salve本地最大offset
getData
遍历读取msg构建成DispactchRequest对象
分发消息位置信息到逻辑队列和索引服务
更新reputFromOffset

五：刷盘策略

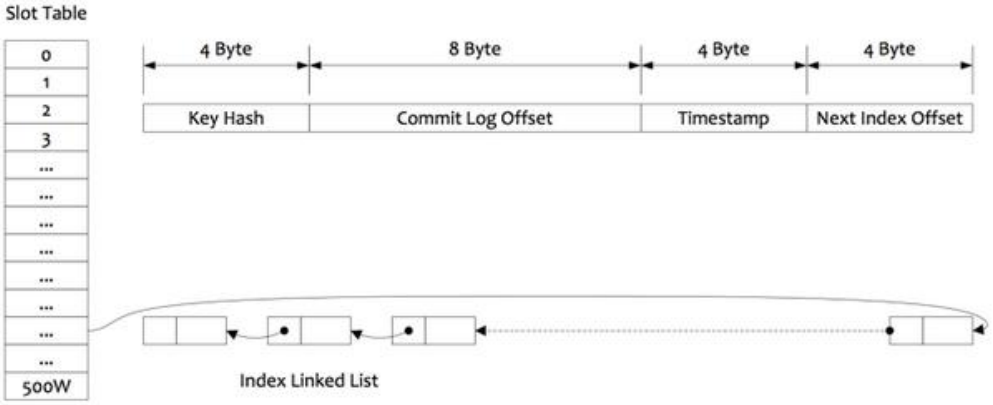
刷盘， 可以通过 setFlushDiskType 来指定刷盘策略，
同步，producer 发送消息到 broker 保证消息持久化到磁盘在返回

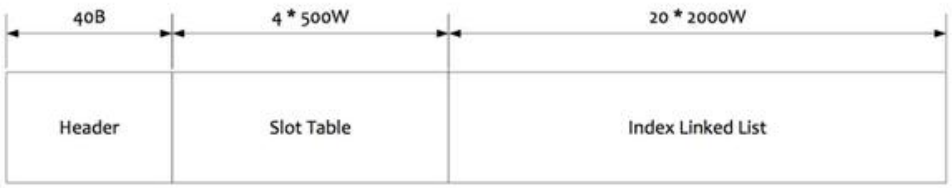
异步, 通过 FlushRealTimeService 服务异步实时刷盘
1 秒钟刷一次，至少刷 4 页

六：索引服务

6.1 索引结构

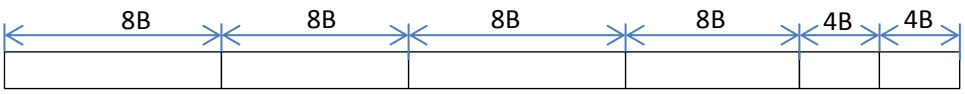
IndexFile 存储具体消息索引的文件，文件的内容结构如图：





索引文件由索引文件头 IndexHeader, 槽位 Slot 和消息的索引内容三部分构成

IndexHeader: 索引文件头信息 40 个字节的数据组成



beginTimestamp	8 位 long 类型，索引文件构建第一个索引的消息落在 broker 的时间
endTimestamp	8 位 long 类型，索引文件构建最后一个索引消息落 broker 时间
beginPhyOffset	8 位 long 类型，索引文件构建第一个索引的消息 commitLog 偏移量
endPhyOffset	8 位 long 类型，索引文件构建最后一个索引消息 commitLog 偏移量
hashSlotCount	4 位 int 类型，构建索引占用的槽位数(这个值貌似没有具体作用)
indexCount	4 位 int 类型，索引文件中构建的索引个数

槽位 slot, 默认每个文件配置的 slot 个数为 500 万个， 每个 slot 是 4 位的 int 类型数据
计算消息的对应的 slotPos=Math.abs(keyHash)%hashSlotNum
消息在 IndexFile 中的偏移量 absSlotPos = IndexHeader.INDEX_HEADER_SIZE + slotPos * HASH_SLOT_SIZE
Slot 存储的值为消息个数索引

消息的索引内容是 20 位定长内容的数据

4 位 int 值， 存储的是 key 的 hash 值
8 位 long 值 存储的是消息在 commitlog 的物理偏移量 phyOffset
4 位 int 值 存储了当前消息跟索引文件中第一个消息在 broker 落地的时间差
4 位 int 值 如果存在 hash 冲突，存储的是上一个消息的索引地址

6.2：索引服务 IndexService 线程

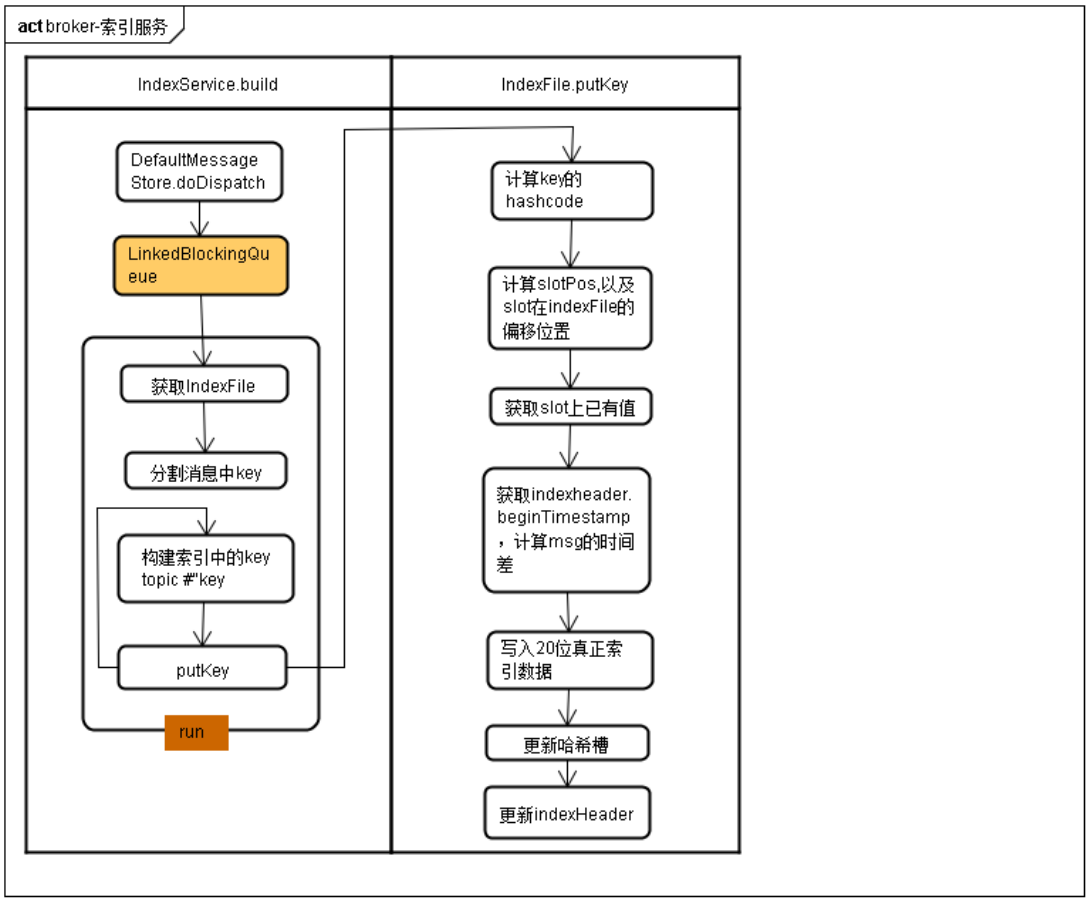
1. 索引配置：hashSlotNum 哈希槽位个数、indexNum 存储索引的最大个数、storePath 索引文件 indexFile 存储的路径
2. Load broker 启动的时候加载本地 IndexFile,

如果是异常启动删除之后 storeCheckPoint 文件， 因为 commitLog 根据 storeCheckPoint 会重建之后的索引文件，

3. Run 方法，任务从阻塞队列中获取请求构建索引
4. queryOffset 根据 topic key 时间跨度来查询消息
   倒叙遍历所有索引文件
   每一个 indexfile 存储了第一个消息和最后一个消息的存储时间，根据传入时间范围来判断索引是否落在此索引文件

6.3：构建索引服务

分发消息索引服务将消息位置分发到 ConsumeQueue 中后，加入 IndexService 的 LinkedBlockingQueue 队列中， IndexService 通过任务向队列中获取请求来构建索引
剔除 commitType 或者 rollbackType 消息，因为这两种消息都有对应的 preparedType 的消息
构建索引 key(topic + "#" + key)
根据 key 的 hashcode 计算槽位，即跟槽位最大值取余数
计算槽位在 indexfile 的具体偏移量位置
根据槽位偏移量获取存储的上一个索引
计算消息跟文件头存储开始时间的时间差
根据消息头记录的存储消息个数计算消息索引存储的集体偏移量位置
写入真正的索引，内容参考上面索引内容格式
将槽位中的更新为此消息索引
更新索引头文件信息



6.4：Broker 与 client（comsumer ， producer）之间的心跳，

一： Broker 接收 client 心跳 ClientManageProcessor 处理 client 的心跳请求

1. 构建 ClientChannelInfo 对象
   1） 持有 channel 对象，表示与客户端的连接通道
   2） ClientID 表示客户端
   .....
2. 每次心跳会更新 ClientChannelInfo 的时间戳，来表示 client 还活着
3. 注册或者更新 consumer 的订阅关系（是以 group 为单位来组织的，group 下可能有多个订阅关系）
4. 注册 producer， 其实就是发送 producer 的 group（这个在事物消息中才有点作用）

二：ClientHouseKeepingService 线程定时清除不活动的连接

1） ProducerManager.scanNotActiveChannel 默认两分钟 producer 没有发送心跳清除

2） ConsumerManager.scanNotActiveChannel 默认两份中 Consumer 没有发送心跳清除

6.5：Broker 与 namesrv 之间的心跳

1） namesrv 接收 borker 心跳 DefaultRequestProcessor 的 REGISTER_BROKE 事件处理，

（1） 注册 broker 的 topic 信息

（2） 构建或者更新 BrokerLiveInfo 的时间戳

2） NamesrvController 初始化时启动线程定时调用 RouteInfoManger 的 scanNotActiveBroker 方法来定时不活动的 broker（默认两分钟没有向 namesrv 发送心跳更新时间戳的）

第四章： NameServer

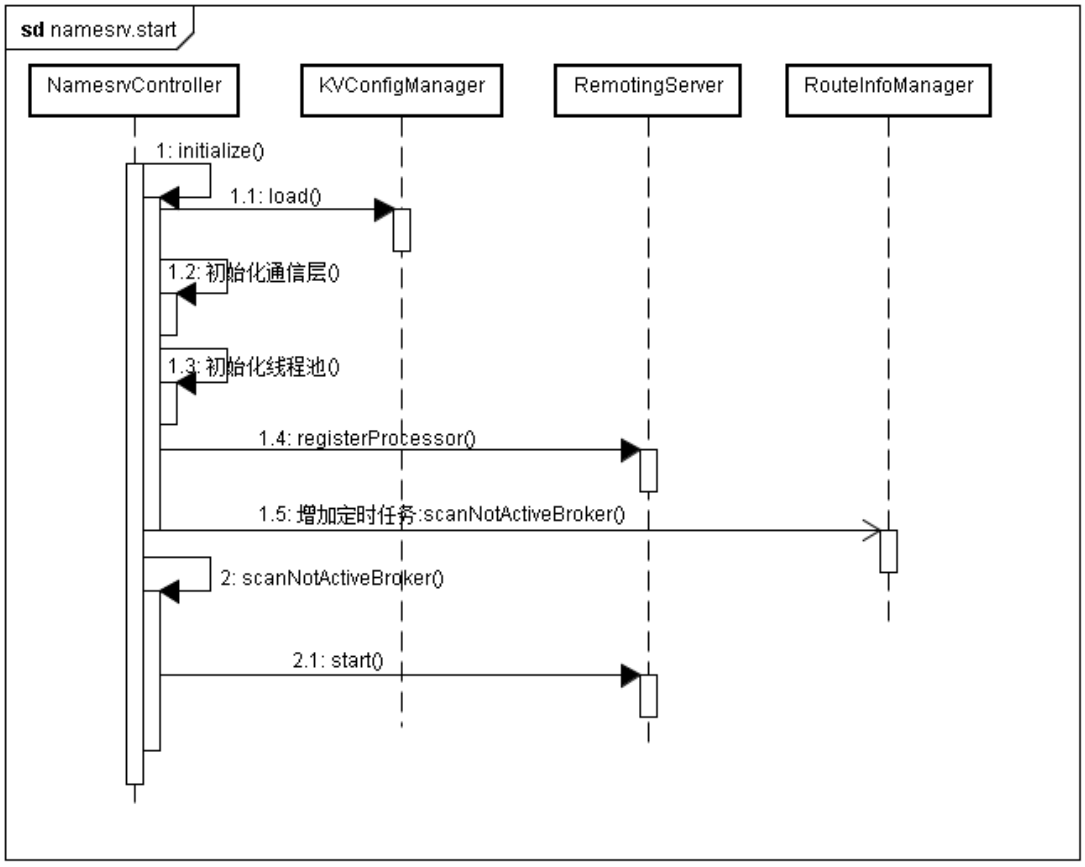
Namesrv 名称服务，是没有状态可集群横向扩展。

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

一：Namesrv 功能：

接收 broker 的请求注册 broker 路由信息（包括 master 和 slave）
接收 client 的请求根据某个 topic 获取所有到 broker 的路由信息

二：Namesrv 启动流程：



三：RouteInfoManager

路由信息 RouteInfoManager 类的管理
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

Map<brokerName, brokerData>
brokerData 由 brokerName 和它的 broker ids 和 address
id 表示是 master 还是 slave
id = 0 为 master 大于 0 为 slave

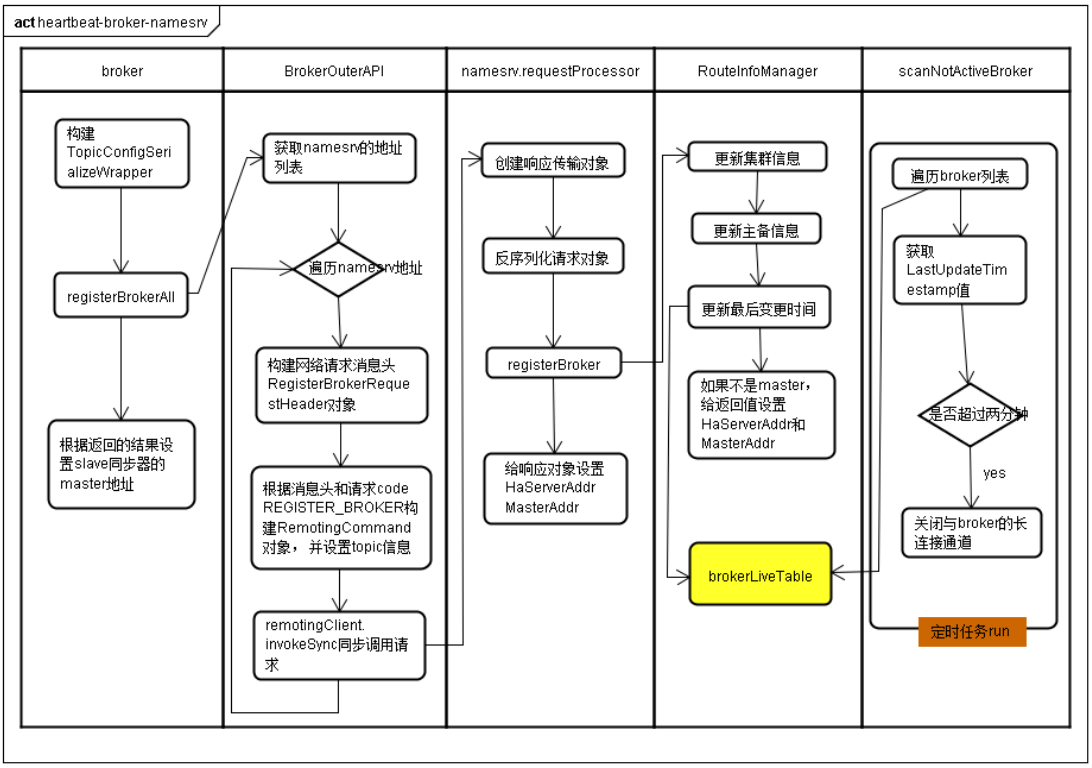
Map<topic, List<queueData>>
queueData 由 brokerName， 读队列数，写队列数，已经权限值

Map<clusterName, Set<brokerName>> 将 broker 按照集群分组

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
Broker 定时向 namesrv 注册并更新 BrokerLiveInfo 的时间戳

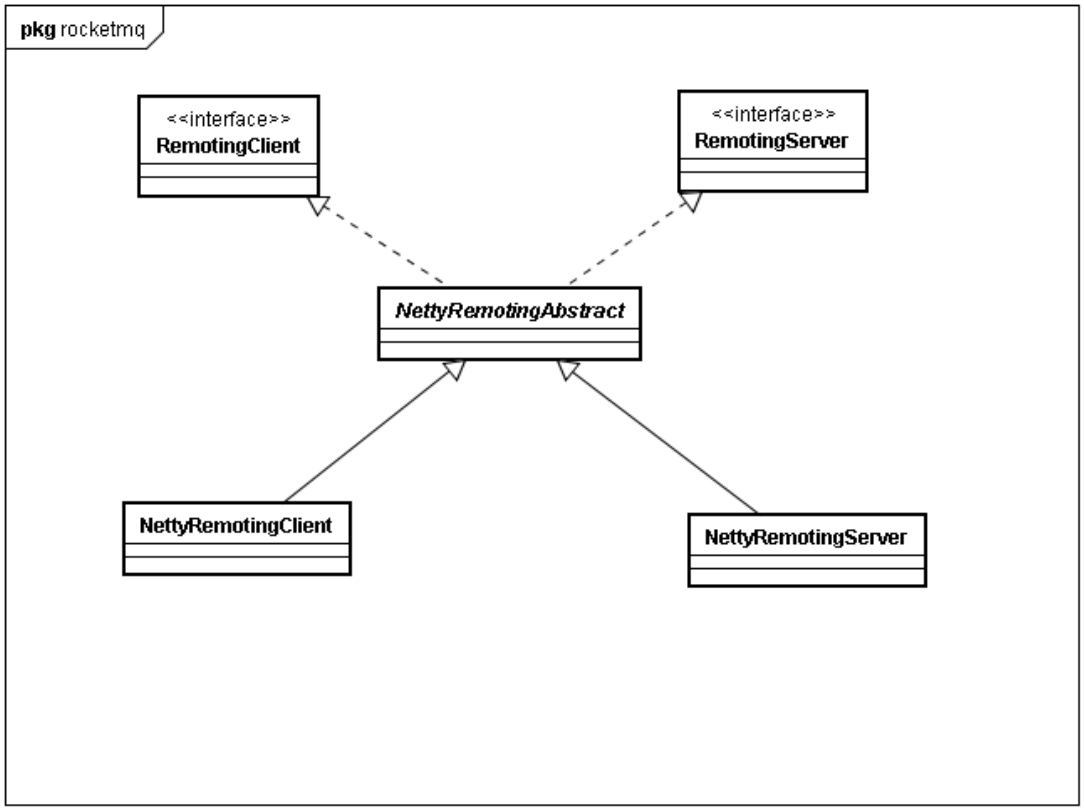
四：Namesrv 与 broker 间的心跳：

1. Broker 启动的时候会启动定时任务，每隔十秒钟会向所有 namesrv 发送心跳请求，同时也是注册 topic 信息到 namesrv
2. namesrv 接收 borker 心跳 DefaultRequestProcessor 的 REGISTER_BROKE 事件处理，
   （3） 注册 broker 的 topic 信息
   （4） 构建或者更新 BrokerLiveInfo 的时间戳
3. NamesrvController 初始化时启动线程定时调用 RouteInfoManger 的 scanNotActiveBroker 方法来定时清理不活动的 broker（默认两分钟没有向 namesrv 发送心跳更新 BrokerLiveInfo 时间戳的），比较 BrokerLiveInfo 的时间戳，如果过期关闭 channel 连接



第五章 Remoting 通信层：

Rocketmq 的通信层是基于通信框架 netty 4.0.21.Final 之上做了简单的协议封装，是强依赖。



一： NettyRemotingAbstract Server 与 Client 公用抽象类

ResponseFuture 模式:

invokeSyncImpl 和 invokeAsyncImpl 都使用了

请求方会 new 一个 ResponseFuture 对象缓存起来 ConcurrentHashMap<Integer /* opaque */, ResponseFuture>，并且设置 opaque 值

Broker 接收请求将 opaque 直接把这个值设置回响应对象，客户端接收到这个响应，通过 opaque 从缓存查找对应的 ResponseFuture 对象

1. invokeSyncImpl 同步调用实现

构建 ResponseFuture，设置 opaque 值

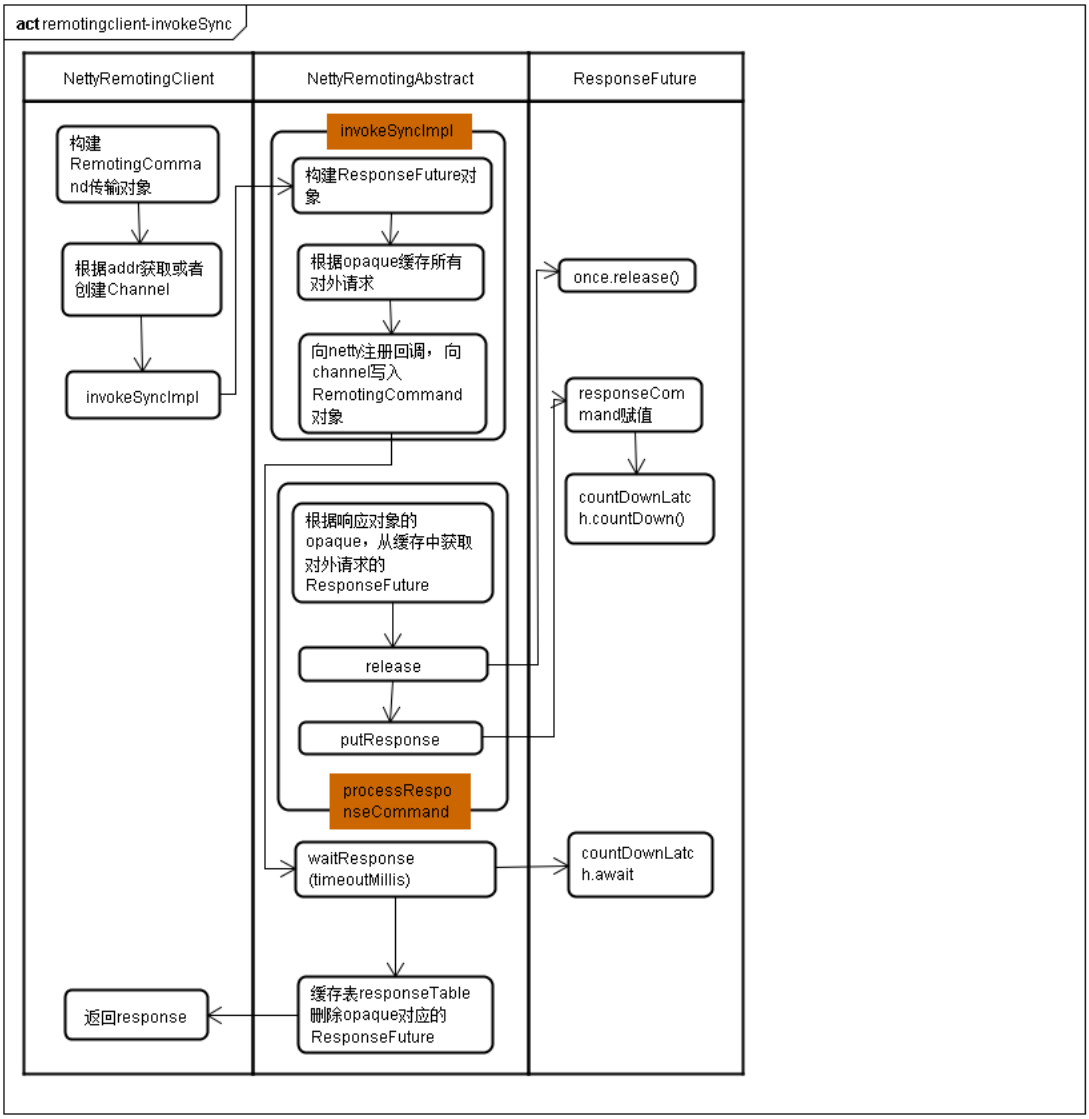
netty 发送请送并且设置监听器回调响应

发送成功设置 ResponseFuture 发送成功，退出监听器

发送失败设置 ResponseFuture 发送失败，并且从缓存中移除 ResponseFuture（没有响应过来，就用不到缓存中的 ResponseFuturel）

responseFuture.waitResponse(timeoutMillis)获取响应

发送成功，没有响应对象说明超时



2. invokeAsyncImpl 异步调用实现

异步一般链路耗时比较长， 为了防止本地缓存的 netty 请求过多， 使用信号量控制上限默认 2048 个

获取是否可以处理请求

构建一次释放对象

构建 responseFuture 对象，设置 opaque， callback， once，超时时间等值，并放入缓存集合

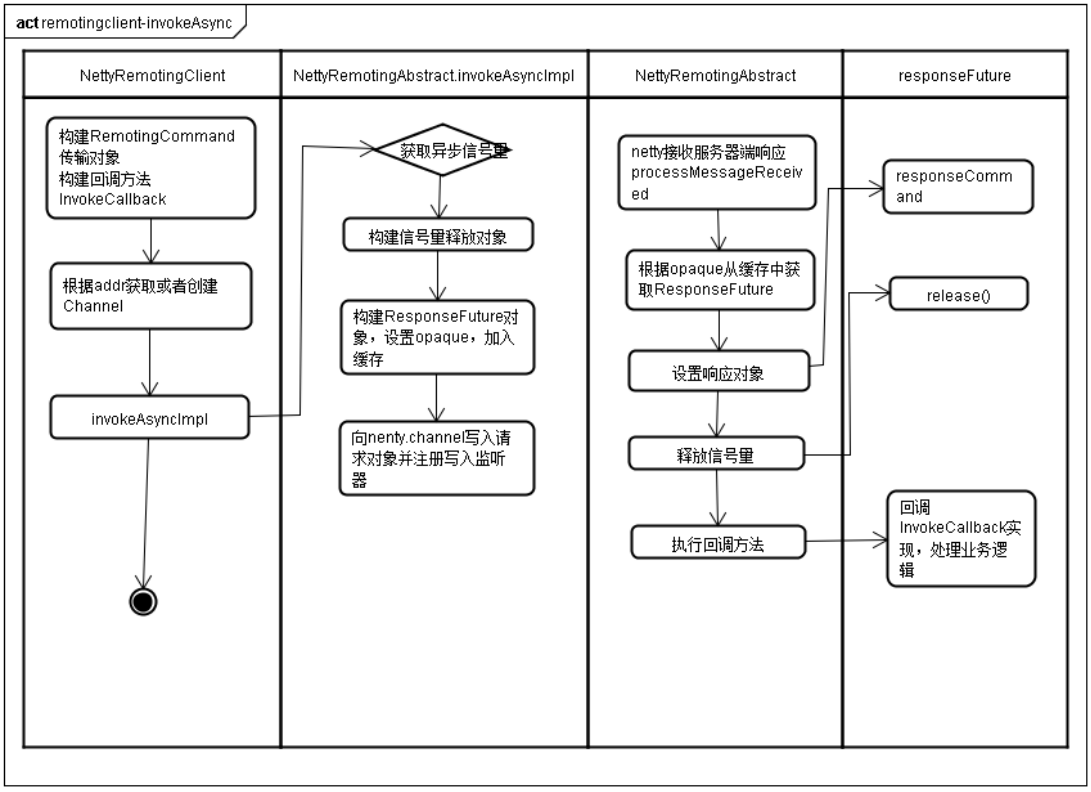
通过 netty 发送请求，设置 listener,

发送成功 responseFuture.setSendRequestOK(true);

发送失败 responseFuture.setSendRequestOK(false), 信号量通过 once 释放， 删除缓存

Netty 接收 server 端响应，根据 opaque 从缓存获取 responseFuture，调用回调方法即接

口 InvokeCallback 实现



3. invokeOnewayImpl 单向请求

标记 onewayRpc
用信号量控制并发的数 //这是我对在这里用新号量控制的理解

4 scanResponseTable

由定时任务启动， 定时查看超时的缓存请求， 有 callback 的执行 callback， 让后从缓存中移除再释放请求

5 processRequestCommand 接收请求处理

根据请求 code 查找对应的处理器线程池 pair， 没有用默认的
有处理器处理请求返回 RemotingCommand 对象的响应 response
若不是 onewayRpc 给 response 设置 opaque
标记响应类型
通过 netty 写入响应

6 processResponseCommand 接收响应处理

当 client 向 server 发送请求的时候，server 处理后向 client 反馈处理结果。

根据 RemotingCommand 的 opaque,从缓存中取出对应的 ResponseFuture

ResponseFuture 设置响应对象 RemotingCommand

responseFuture 释放信号量

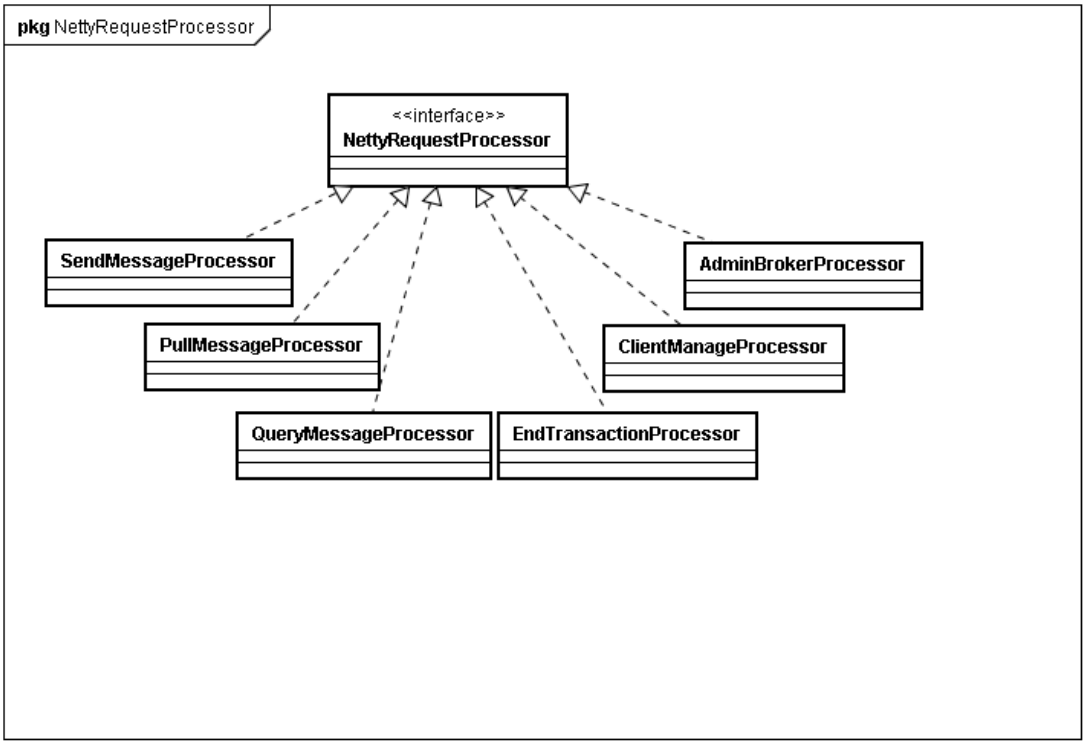
有 callback 的执行 callback（通过线程池）， 没有的 putResponse（这个方法同步调用使用，来 countDownLatch， 因为调用线程在等待呢）

二：NettyRemotingServer Remoting 服务端实现

broker 启动初始化 NettyRemotingServer ，
向 netty 注册 handler
NettyEncoder 协议编码器，将 RemotingCommand 转换为字节，给 netty 传输
NettyDecoder 协议解码器， 将 netty 接收的输入流，转换成 RemotingCommand
NettyConnetManageHandler 处理 register，unregiter， active， inactive， exception
NettyServerHandler netty 处理请求的业

向 NettyRemotingServer 注册业务处理器
server 接收 client 请求根据 RequestCode 选择具体的处理器 RequestProcessor，就是利用 RequestCode 进行策略的选择
server(broker, namesrv)在启动的时候会把 RequestCode 与对应的 RequestProcessor 和处理线程池注册到 NettyRemotingServer 中去，代码类似如下：

```
remotingServer.registerProcessor(RequestCode.SEND_MESSAGE, SendMessageProcessor, sendMessageExecutor)
```



三：NettyRemotingClient

向 netty 注册 handler

NettyEncoder 协议编码器，将 RemotingCommand 转换为字节，给 netty 传输
NettyDecoder 协议解码器， 将 netty 接收的输入流，转换成 RemotingCommand
NettyConnetManageHandler 处理 register，unregiter， active， inactive， exception
NettyClientHandler netty 处理请求的业

Client 与通信层的交互封装了 MQClientAPIImpl 统一处理，在 MQClientAPIImpl 构造的时候注册了 ClientRemotingProcessor 来处理 server 的请求

四：底层传输协议

RocketMq 服务器与客户端通过传递 RemotingCommand 来交互，通过 NettyDecoder，NettyEncoder 对 RemotingCommand 进行协议的编码与解码

协议格式 <length> <header length> <header data> <body data>
1 2 3 4
协议分4部分，含义分别如下
1、大端4个字节整数，等于2、3、4长度总和
2、大端4个字节整数，等于3的长度
3、使用json序列化数据
4、应用自定义二进制序列化数据

Header字段名	类型	Request	Response
code	整数	请求操作代码，请求接收方根据不同的代码做不同的操作	应答结果代码，0 表示成功，非 0 表示各种错误代码
language	字符串	请求发起方实现语言，默认 JAVA	应答接收方实现语言
version	整数	请求发起方程序版本	应答接收方程序版本
opaque	整数	请求发起方在同一连接上不同的请求标识代码，多线程连接复用使用	应答方不做修改，直接返回
flag	整数	通信层的标志位	通信层的标志位
remark	字符串	传输自定义文本信息	错误详细描述信息
extFields	HashMap<String,String>	请求自定义字段	应答自定义字段

Header 部分数据是通过 FastJson 序列化数据

```
{
  "code": 0,
  "language": "JAVA",
  "version": 0,
  "opaque": 0,
  "flag": 1,
  "remark": "hello, I am respponse /127.0.0.1:27603",
  "extFields": {
    "count": "0",
    "messageTitle": "HelloMessageTitle"
  }
}
```

请求自定义字段都会实现 CommandCustomHeader 接口， 在 RemotingCommand 序列化之前会将 CommandCustomHeader 的字段拷贝到 Header 的 extFields 中去， 让后在整体通过 Fastjson 序列化

Netty 通过 NettyEncoder、NettyDecoder 自定义实现将 RemotingCommand 转换成 byte[]
NettyEncoder:

```
public void encode(ChannelHandlerContext ctx, RemotingCommand remotingCommand, ByteBuf out
        throws Exception {
    try {
        ByteBuffer header = remotingCommand.encodeHeader();
        out.writeBytes(header);
        byte[] body = remotingCommand.getBody();
        if (body != null) {
            out.writeBytes(body);
        }
    }
```

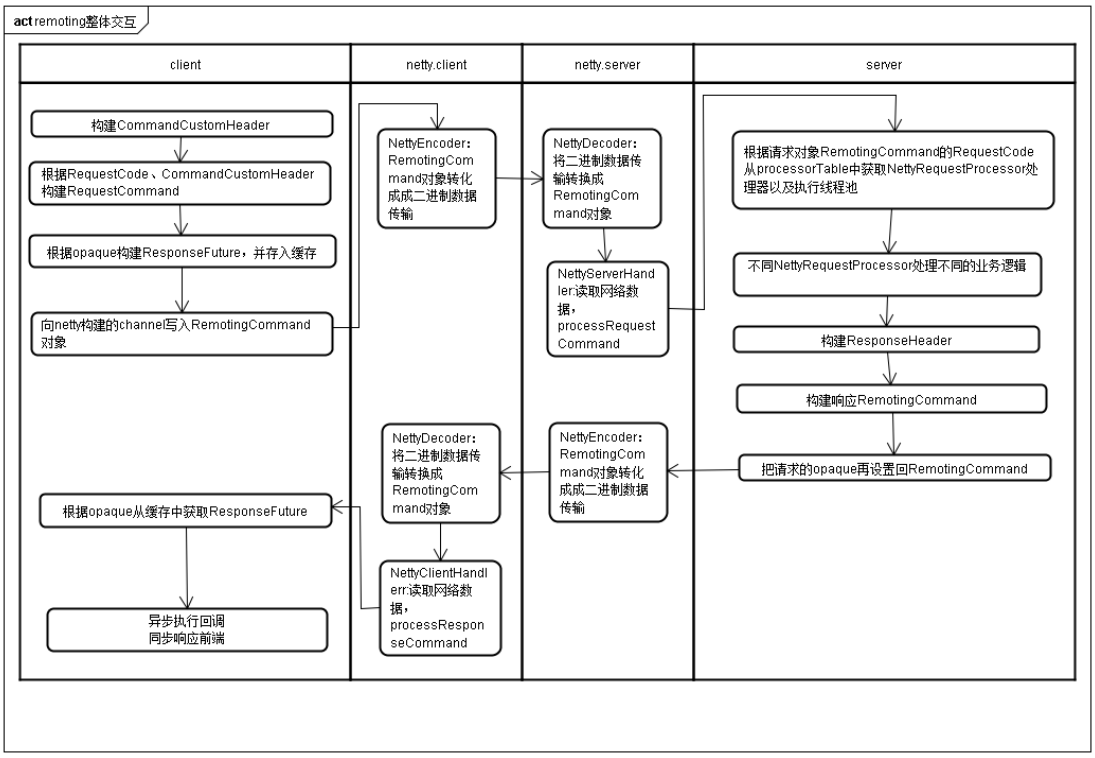
NettyDecoder

```
public Object decode(ChannelHandlerContext ctx, ByteBuf in) throws Exception {
    ByteBuf frame = null;
    try {
        frame = (ByteBuf) super.decode(ctx, in);
        if (null == frame) {
            return null;
        }

        ByteBuffer byteBuffer = frame.nioBuffer();

        return RemotingCommand.decode(byteBuffer);
    }
```

五：通信层的整体交互


act remoting整体交互
client
netty.client
netty.server
server
构建CommandCustomHeader
根据RequestCode、CommandCustomHeader构建RequestCommand
根据opaque构建ResponseFuture，并存入缓存
向netty构建的channel写入RemotingCommand对象
NettyEncoder：RemotingCommand对象转化成成二进制数据传输
NettyDecoder：将二进制数据传输转换成RemotingCommand对象
NettyServerHandler:读取网络数据，processRequestCommand
根据请求对象RemotingCommand的RequestCode从processorTable中获取NettyRequestProcessor处理器以及执行线程池
不同NettyRequestProcessor处理不同的业务逻辑
构建ResponseHeader
构建响应RemotingCommand
把请求的opaque再设置回RemotingCommand
NettyEncoder：RemotingCommand对象转化成成二进制数据传输
NettyDecoder：将二进制数据传输转换成RemotingCommand对象
NettyClientHandler:读取网络数据，processResponseCommand
根据opaque从缓存中获取ResponseFuture
异步执行回调
同步响应前端