RocketMQ 原理解析

斩秋

博客：http://blog.csdn.net/quhongwei_zhanqiu

目录

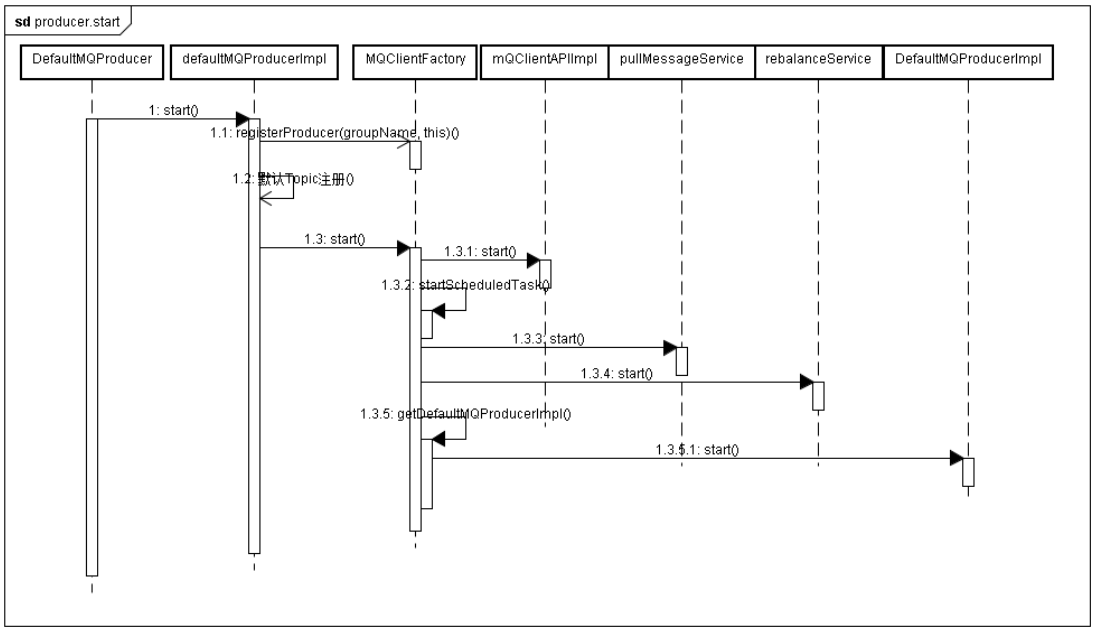
前言

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，

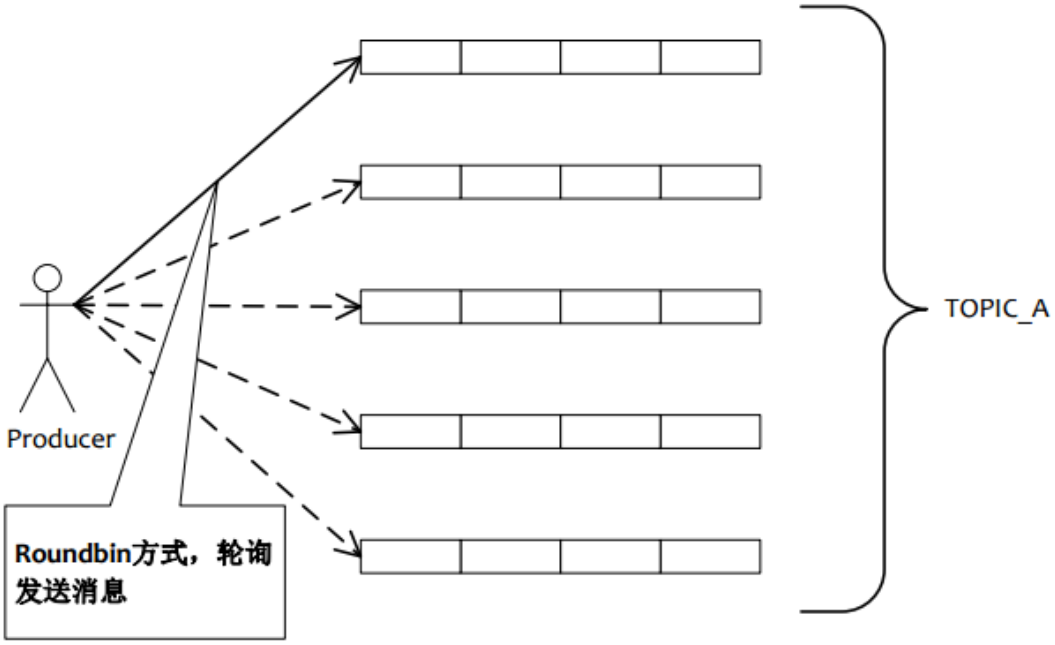
Producer 与 broker 间的心跳

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

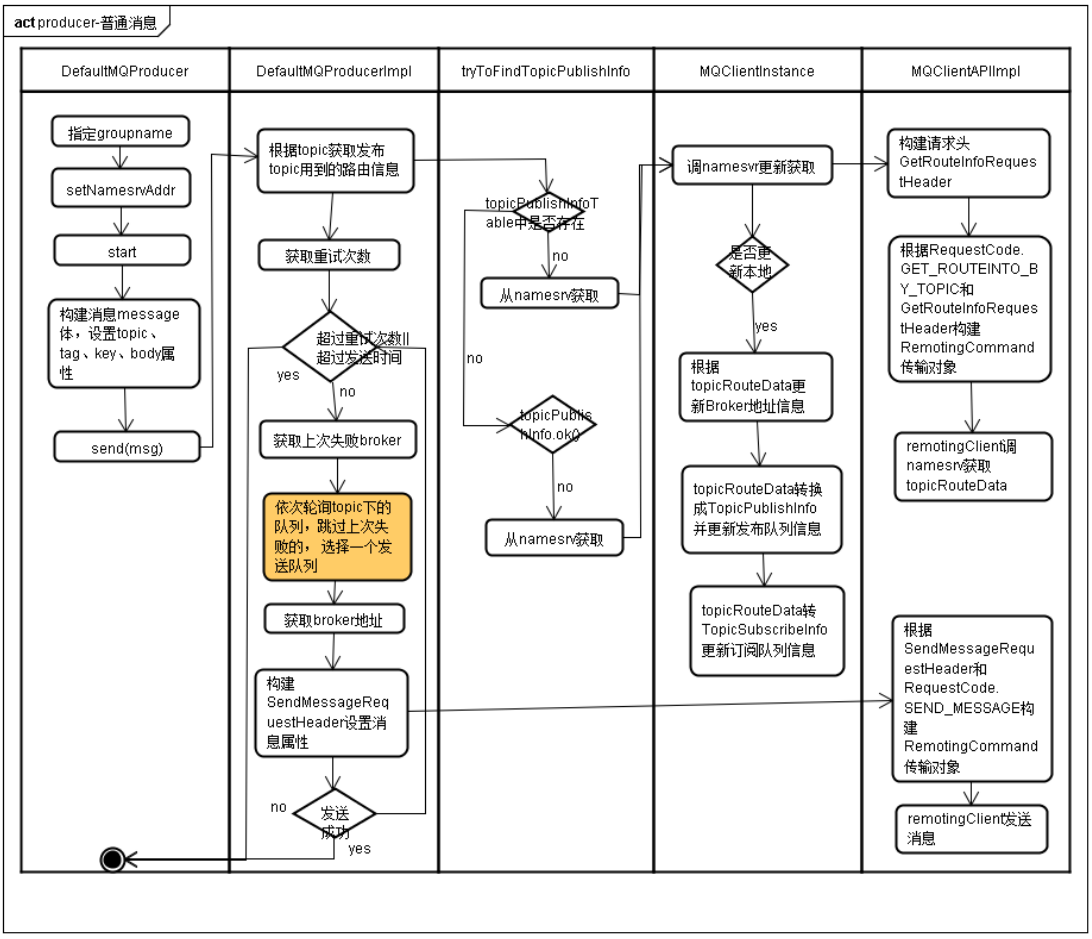
--brokerName //代表发送消息到达的 broker

--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

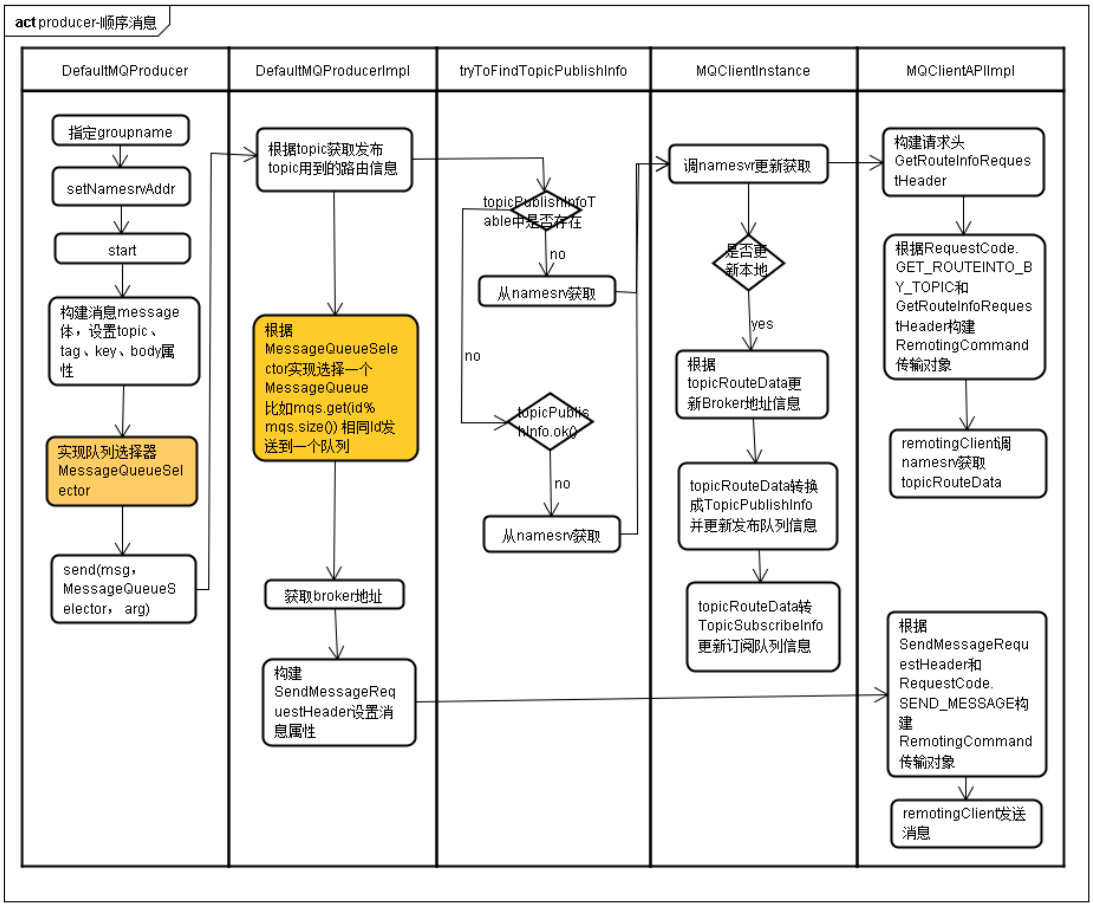
Rocketmq 能够保证消息严格顺序，但是 Rocketmq 需要 producer 保证顺序消息按顺序发送到同一个 queue 中， 比如购买流程(1)下单(2)支付(3)支付成功，这三个消息需要根据特定规则将这个三个消息按顺序发送到一个 queue

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

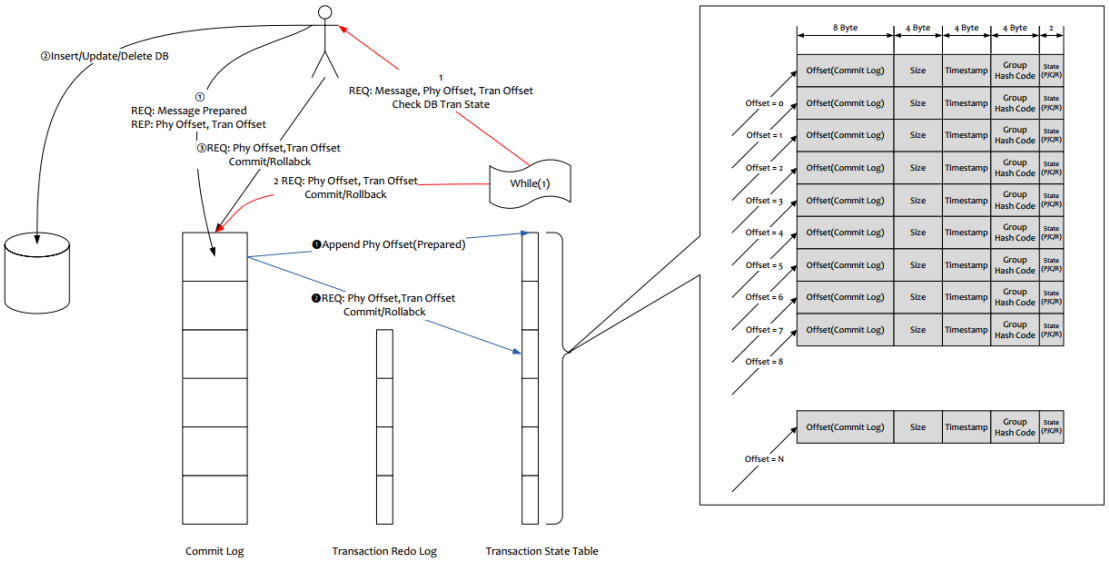
messageQueueList [orderId%messageQueueList.size()]

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

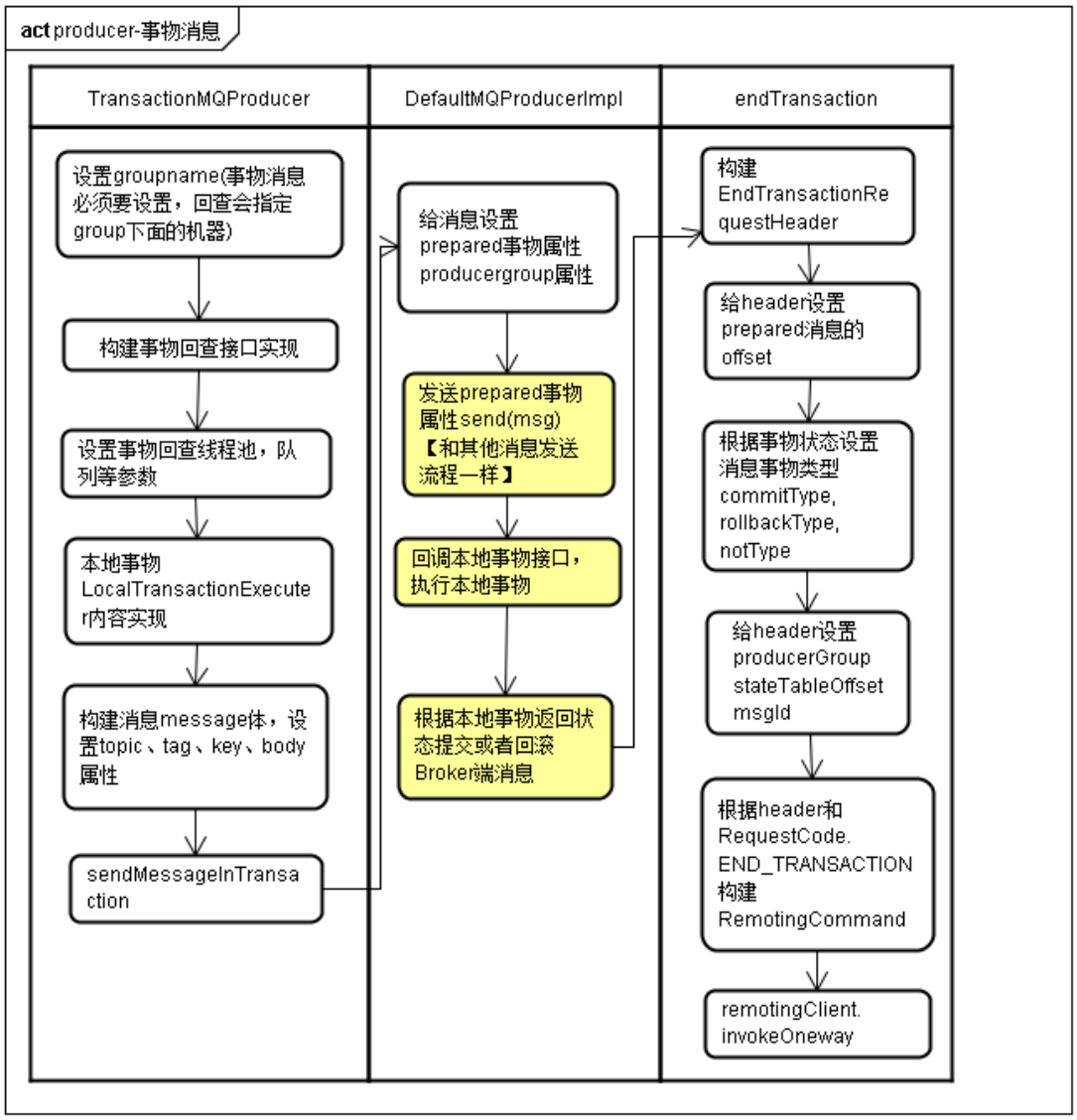
1) 一阶段，向 broker 发送一条 prepared 的消息，返回消息的 offset 即消息地址 commitLog 中消息偏移量。Prepared 状态消息不被消费

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

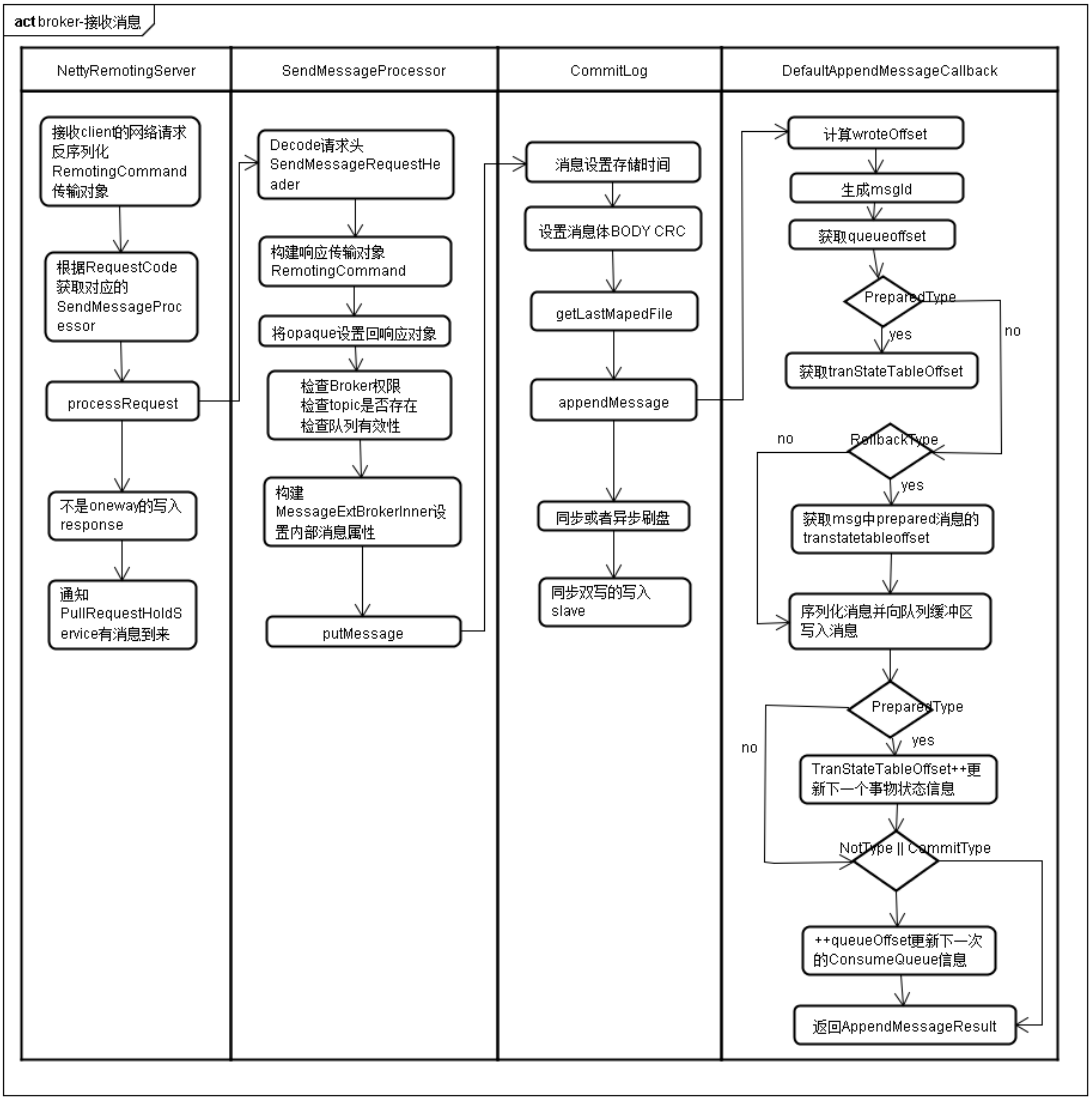
三：Broker 落地消息

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。

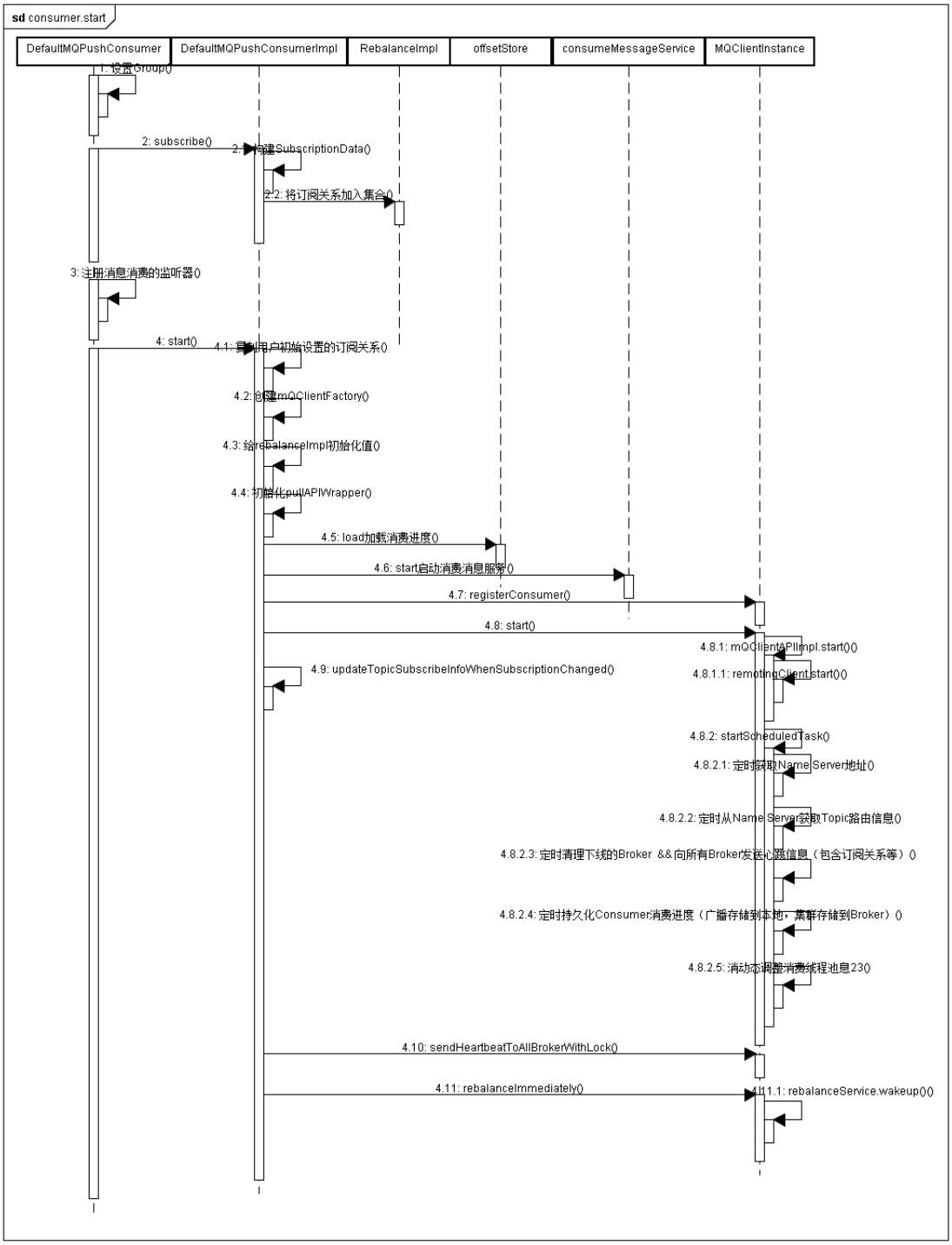
Consumer 消费拉取的消息的方式有两种

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

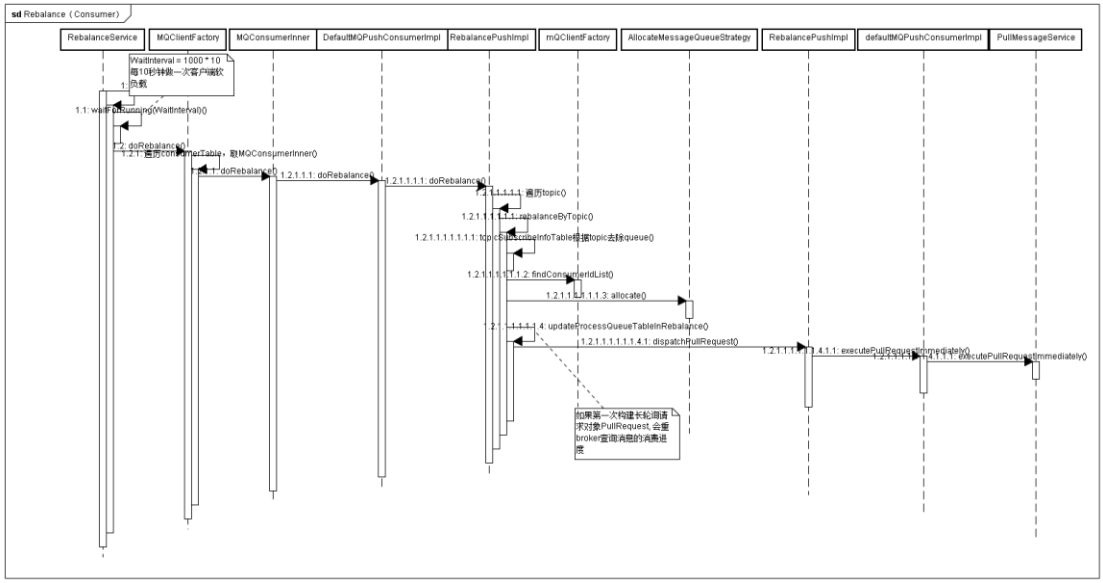
下面介绍默认以 push 方式为主， 因为绝大多数是由 push 消费方式来使用 rocketmq 的。

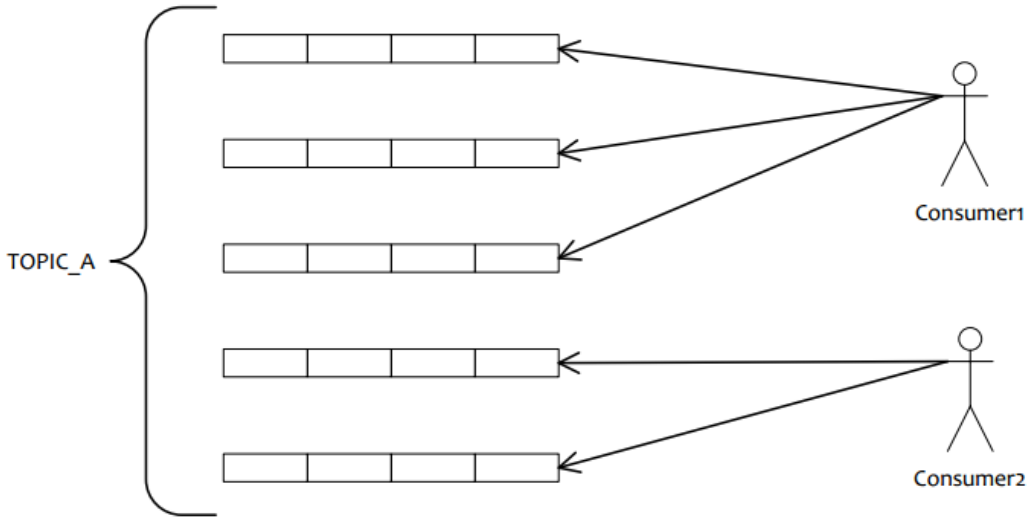
一：consumer 启动流程

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>

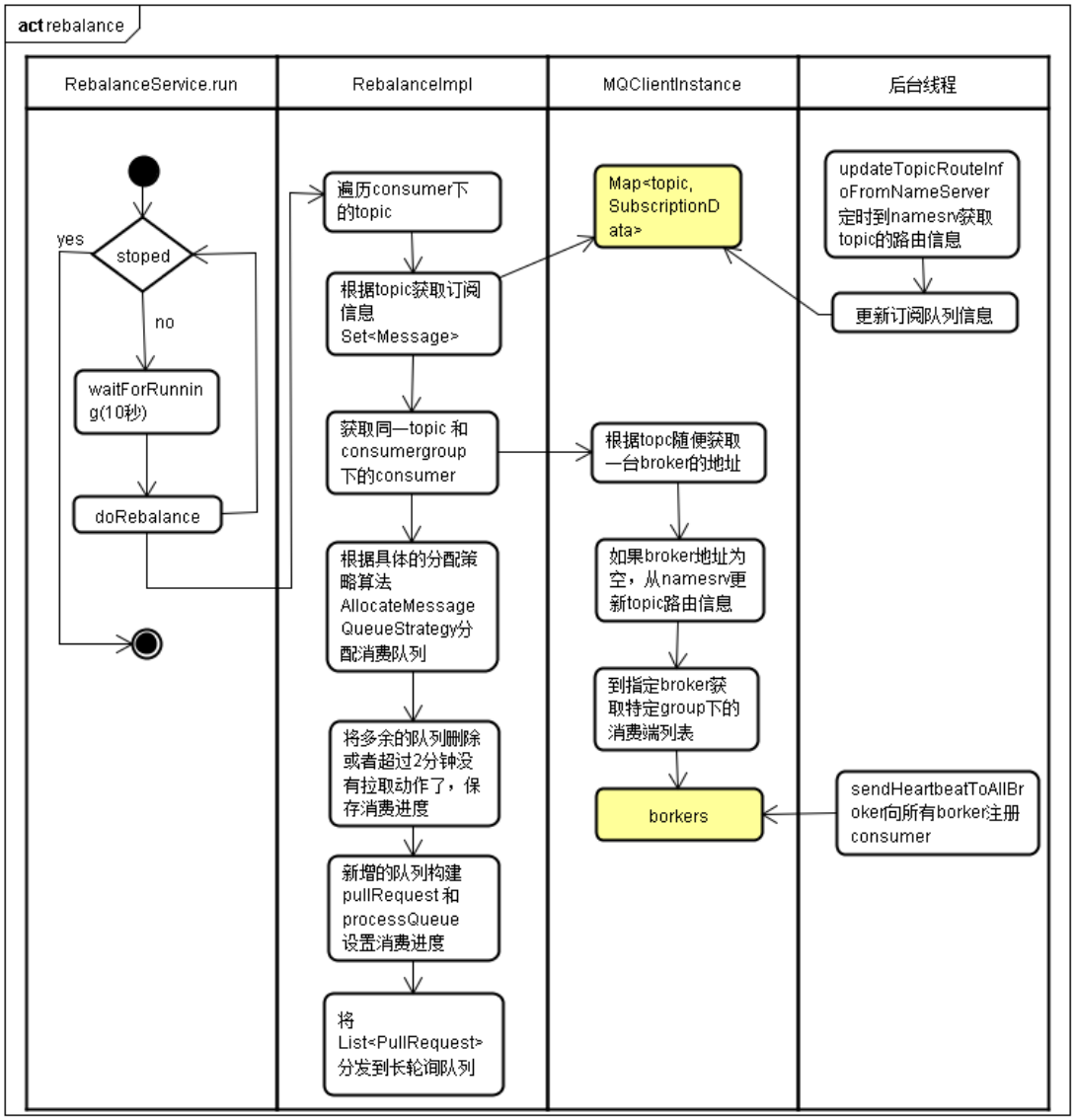
1) 比对 mqSet 将多余的队列删除， 当 broker 当机或者添加，会导致分配到 mqSet 变化，
   a) 将不在被本 consumer 消费的 messagequeue 的 ProcessQueue 删除， 其实是设置 ProcessQueue 的 droped 属性为 true
   b) 将超过两份中没有拉取动作 ProcessQueue 删除
      //TODO 为什么要删除掉，两分钟后来了消息怎么办?
      //
2) 添加新增队列， 比对 mqSet，给新增的 messagequeue
   构建长轮询对象 PullRequest 对象，会从 broker 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

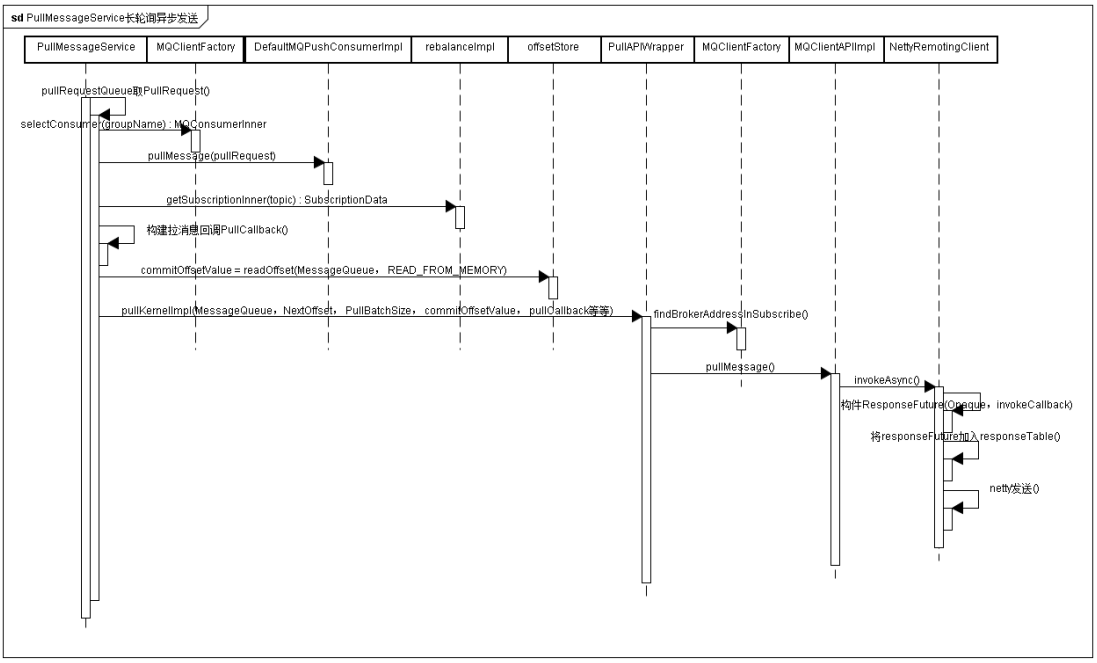
Rocketmq 的消息是由 consumer 端主动到 broker 拉取的, consumer 向 broker 发送拉消息请求， PullMessageService 服务通过一个线程将阻塞队列 LinkedBlockingQueue<PullRequest> 中的 PullRequest 到 broker 拉取消息

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

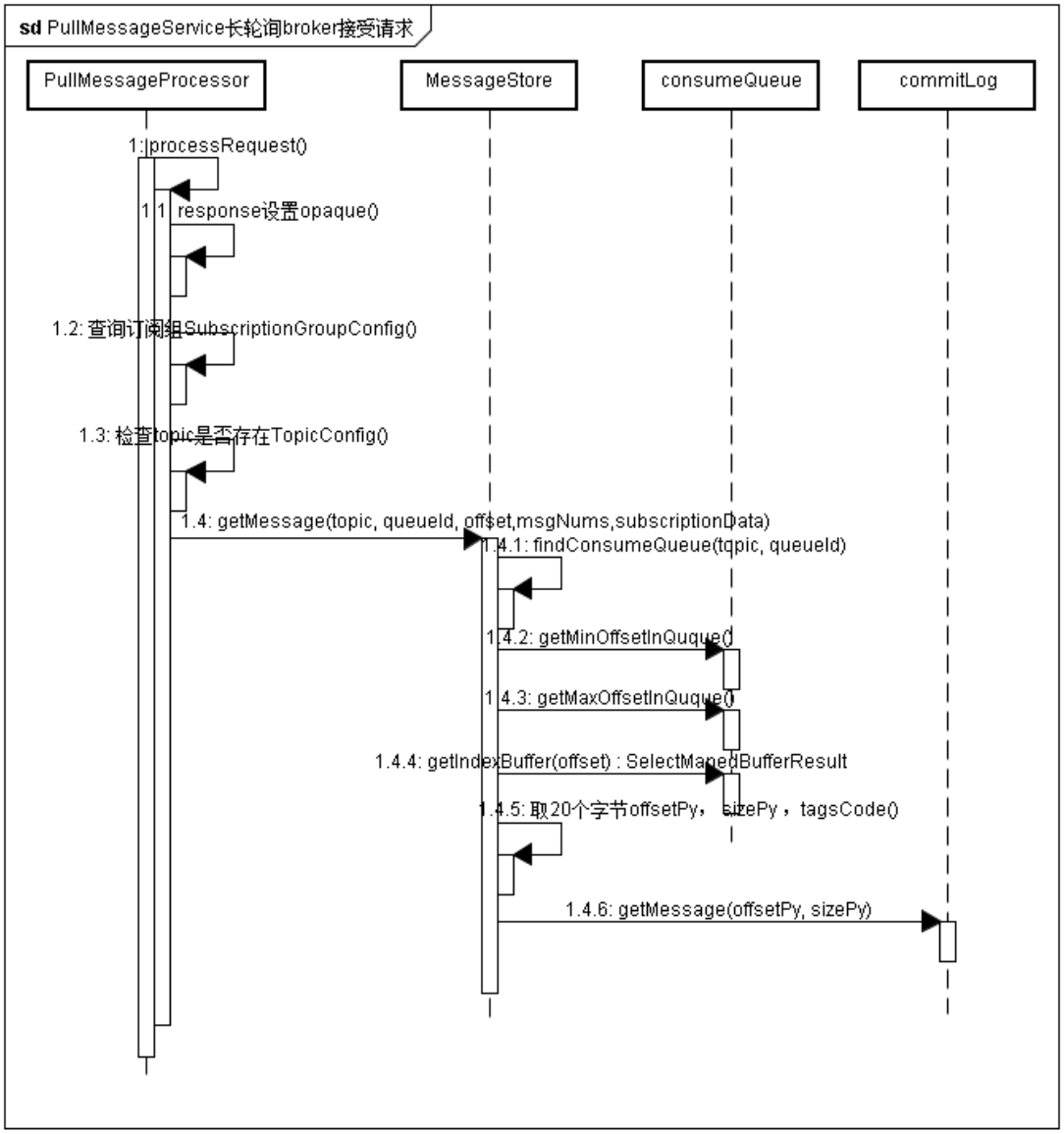
1. 获取 ProcessQueue 判读是否 drop 的， drop 为 true 返回
2. 给 ProcessQueue 设置拉消息时间戳
3. 流量控制，正在消费队列中消息（未被消费的）超过阀值，稍后在执行拉消息
4. 流量控制，正在消费队列中消息的跨度超过阀值（默认 2000），稍后在消费
5. 根据 topic 获取订阅关系
6. 构建拉消息回调对象 PullBack, 从 broker 拉取消息（异步拉取）返回结果是回调
7. 从内存中获取 commitOffsetValue //TODO 这个值跟 pullRequest.getNextOffset 区别
8. 构建 sysFlag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

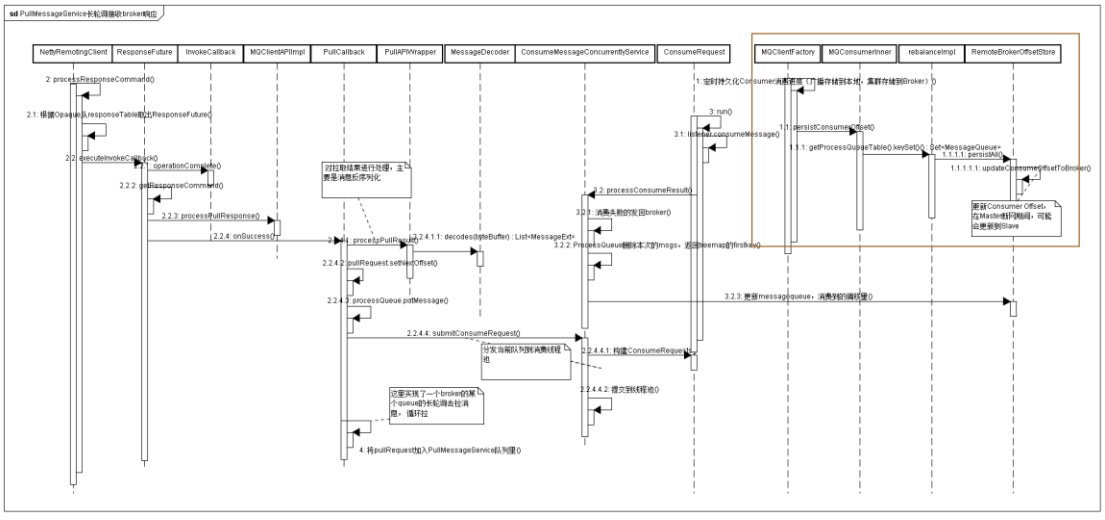
1. 向 broker 发送长轮询请求



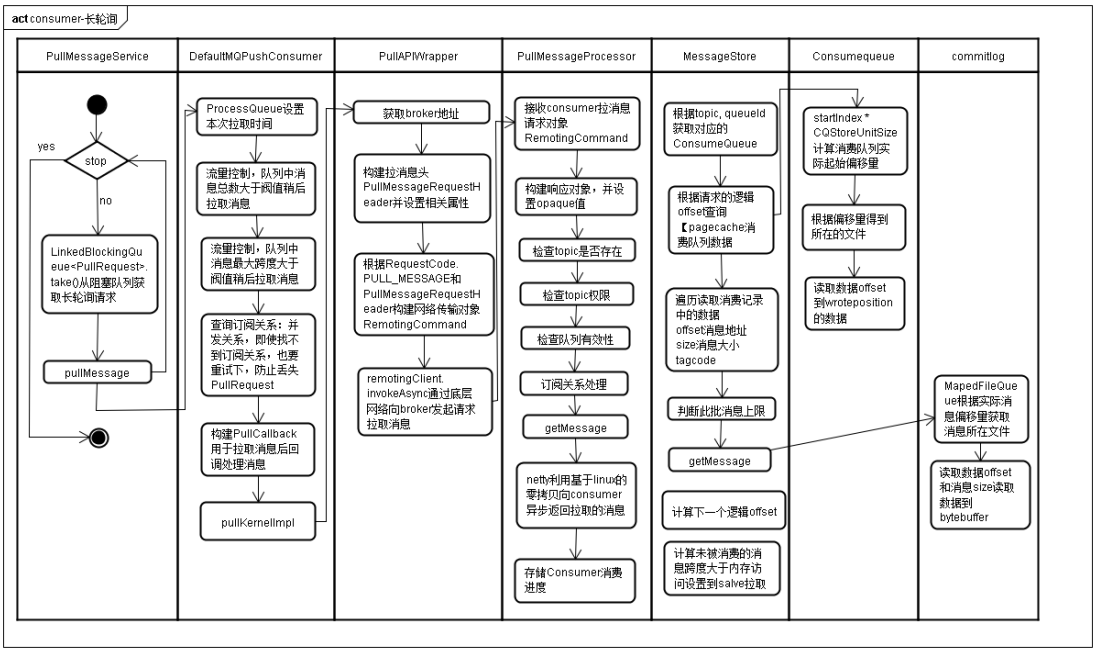
2. Broker 接收长轮询请求



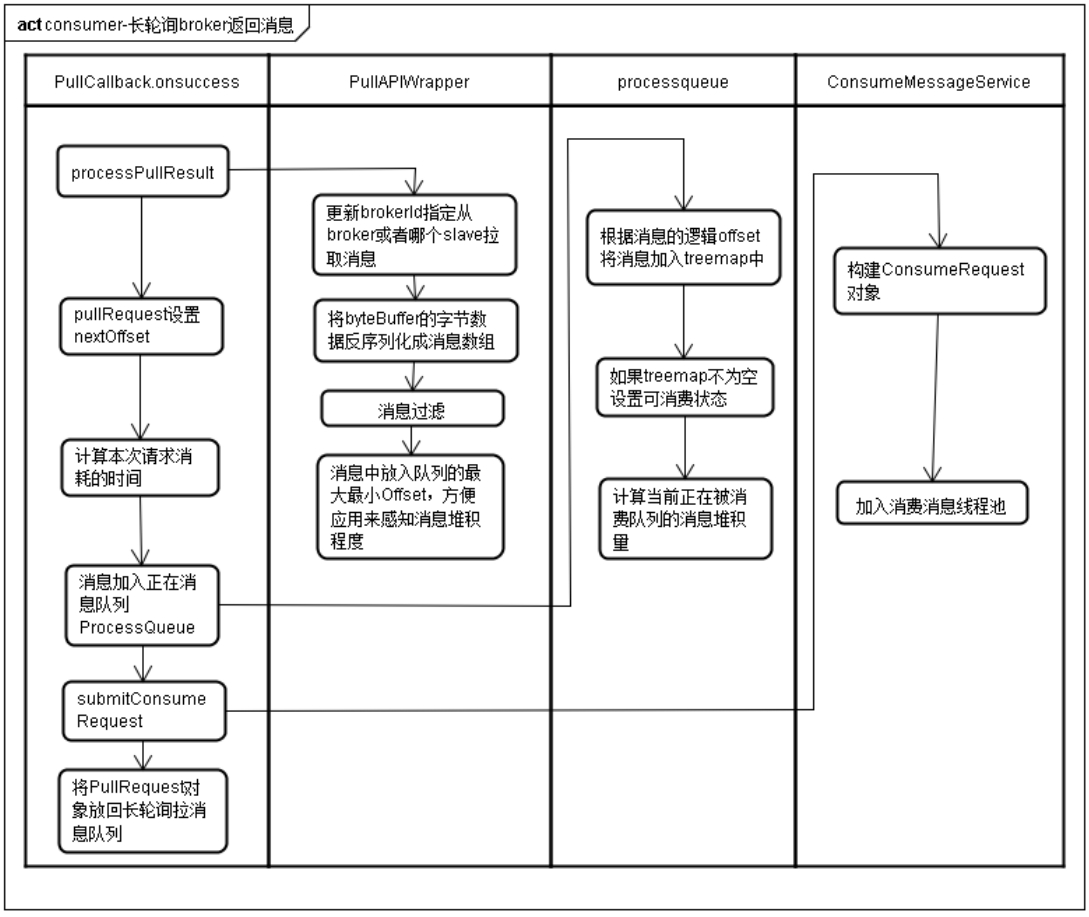
3. Consumer 接收 broker 响应



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

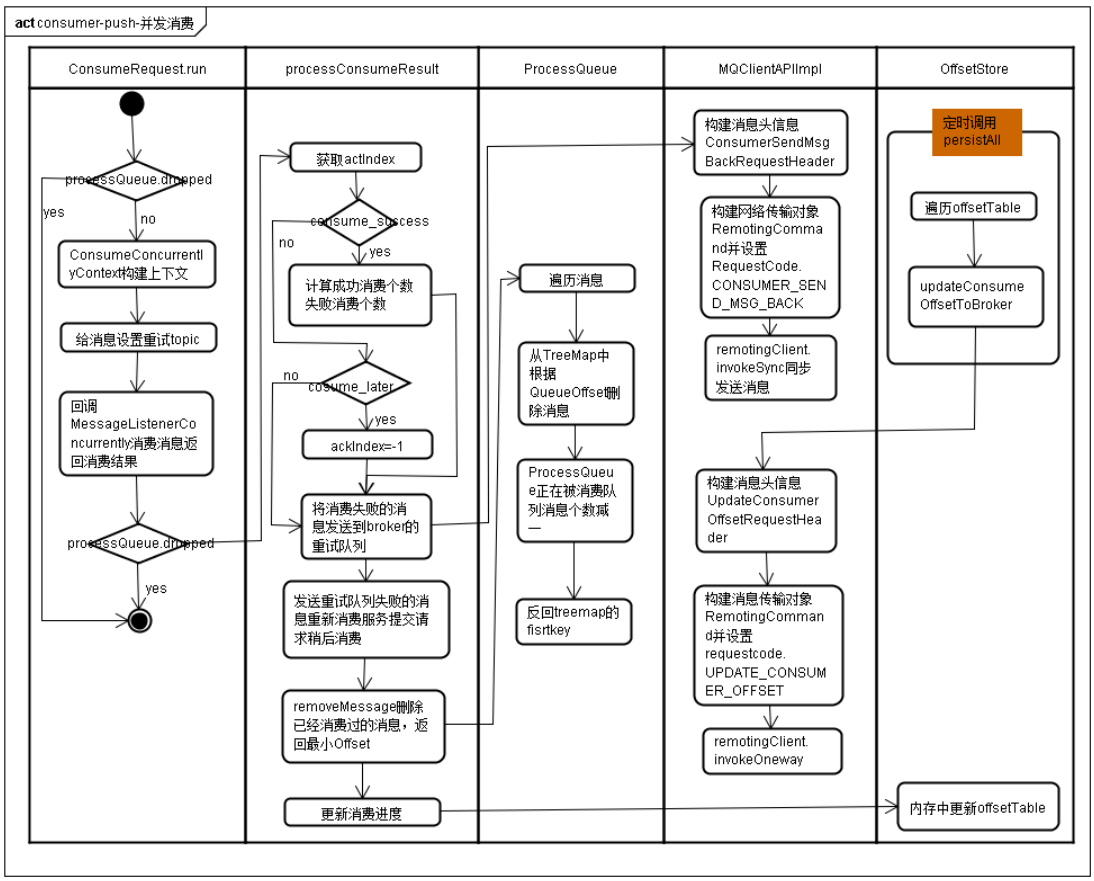
消费失败， ackIndex 设置为-1

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

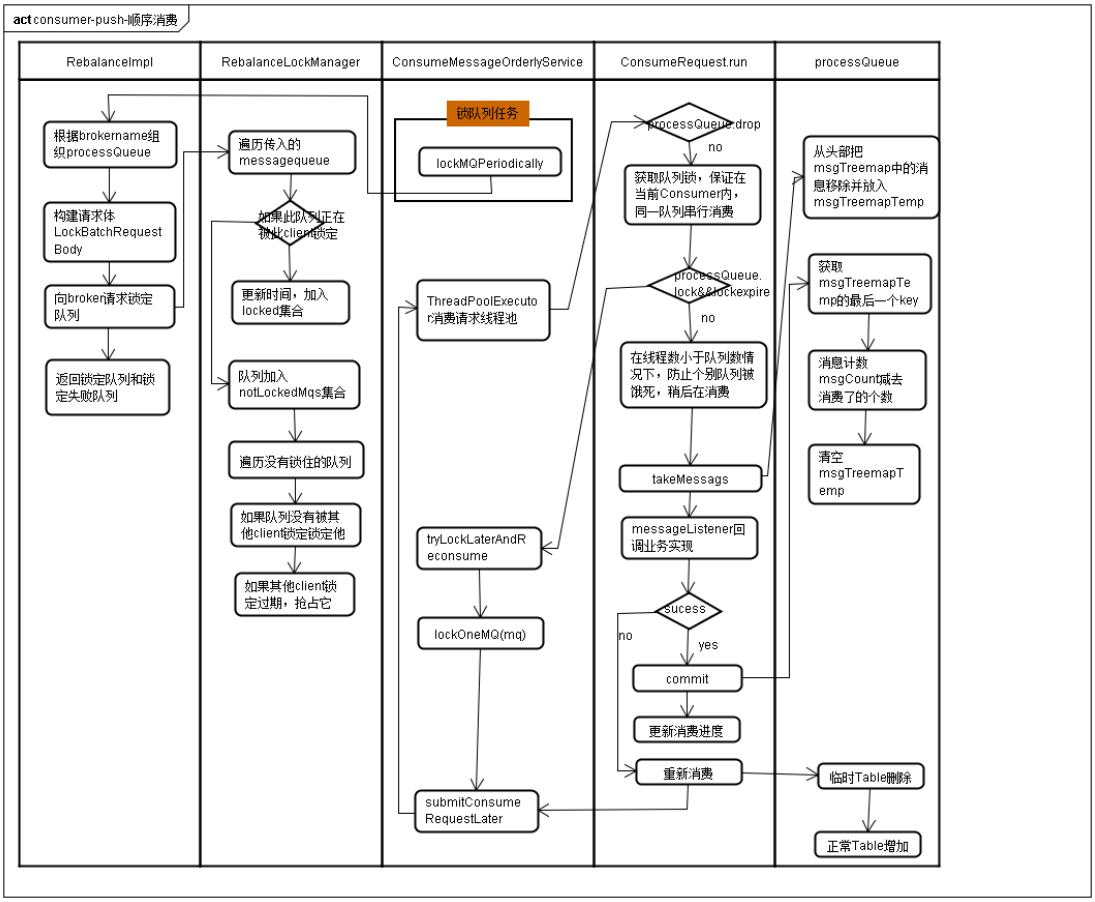
消息消息状态为 success： 调用 processQueue.commit 方法

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