

RocketMQ(讲师： Old Jia)

课程目标

1. 了解消息中间件背景知识、使用场景、发展等
2. 掌握RocketMQ消息中间件的架构、模型和使用（开发、安装、集群部署、运维、监控等）
3. 掌握消息的可靠性、幂等性、顺序消息、延迟消息、事务消息等进阶的知识，以及大规模生产环境中的使用经验，轻松应对各种复杂的业务场景
4. 掌握顶级开源消息中间件核心源码，理解其背后的架构设计思想以及在高性能存储系统、网络编程等方面的技巧（会涉及网络通信、操作系统等底层知识）
5. 理解主流消息中间件的优缺点，具备技术选型能力
6. 让你无论是在日后的工作还是面试求职中遇到消息中间件相关问题都能轻松应对

第一部分 RocketMQ架构与实战

RocketMQ是阿里巴巴中间件团队自研的一款高性能、高吞吐量、低延迟、高可用、高可靠（具备金融级稳定性）的分布式消息中间件，开源后并于2016年捐赠给Apache社区孵化，目前已经成为了Apache顶级项目。当前在国内被广泛的使用，包括互联网、电商、金融、企业服务等领域，包括：字节跳动、滴滴、微众银行等知名的互联网公司。

<https://github.com/apache/rocketmq>

1.1 RocketMQ的前世今生

RocketMQ在阿里内部叫做Metaq（最早名为Metamorphosis，中文意思“变形记”，是作家卡夫卡的中篇小说代表作，可见是为了致敬Kafka）。

RocketMQ是Metaq 3.0之后开源的版本。

Metaq在阿里巴巴集团内部、蚂蚁金服、菜鸟等各业务中被广泛使用，接入了上万个应用系统中。并平稳支撑了历年的双十一大促（万亿级的消息），在性能、稳定性、可靠性等方面表现出色，在整个阿里技术体系和大中台战略中发挥着举足轻重的作用。

Metaq最早源于Kafka，早期借鉴了Kafka很多优秀的设计。但是由于Kafka是Scala语言编写而阿里系主要使用Java，且无法满足阿里的电商、金融业务场景，所以誓嘉（花名）团队用Java重新造轮子，并做了大量的改造和优化。

在此之前，淘宝有一款消息中间件名为Notify，目前已经逐步被Metaq所取代。

第一代的Notify主要使用了推模型，解决了事务消息；第二代的MetaQ主要使用了拉模型，解决了顺序消息和海量堆积的问题。相比起Kafka使用的Scala语言编写，RabbitMQ 使用Erlang语言编写，基于Java的RocketMQ开源后更容易被广泛的研究，以及其他大厂定制开发。



1.2 RocketMQ的使用场景

- 应用解耦

系统的耦合性越高，容错性就越低。以电商应用为例，用户创建订单后，如果耦合调用库存系统、物流系统、支付系统，任何一个子系统出了故障或者因为升级等原因暂时不可用，都会造成下单操作异常，影响用户使用体验。

- 流量削峰

应用系统如果遇到系统请求流量的瞬间猛增，有可能会将系统压垮。有了消息队列可以将大量请求缓存起来，分散到很长一段时间处理，这样可以大大提高系统的稳定性和用户体验。

举例：业务系统正常时段的QPS如果是1000，流量最高峰是10000，为了应对流量高峰配置高性能的服务器显然不划算，这时可以使用消息队列对峰值流量削峰

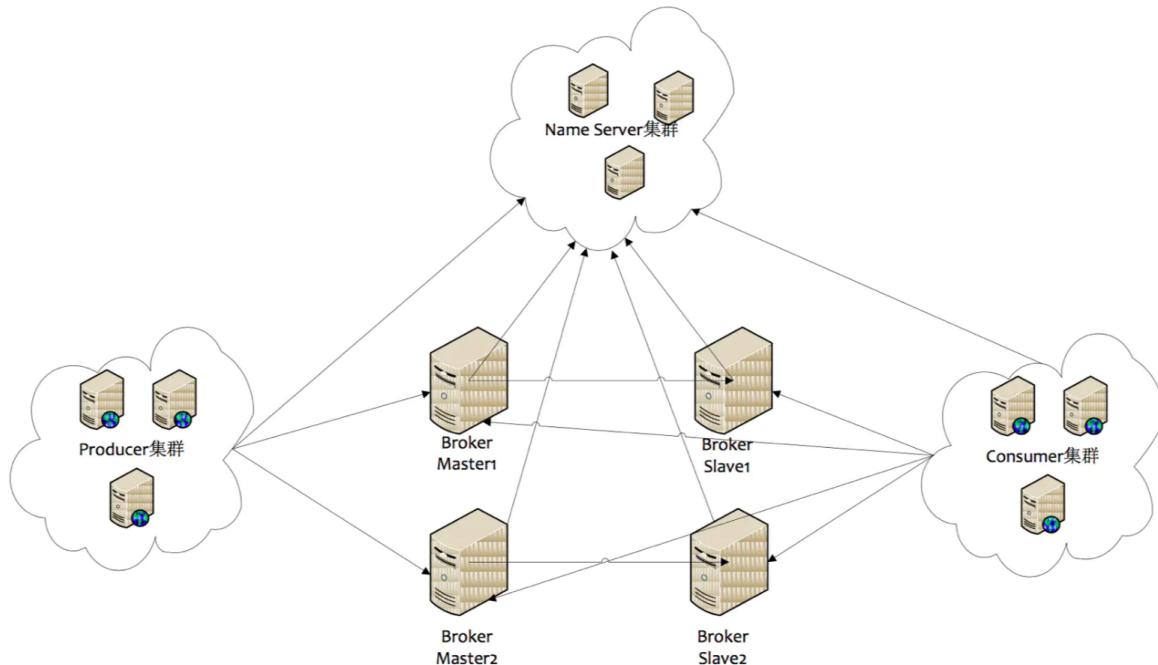
- 数据分发

通过消息队列可以让数据在多个系统之间进行流通。数据的产生方不需要关心谁来使用数据，只需要将数据发送到消息队列，数据使用方直接在消息队列中直接获取数据即可

1.3 RocketMQ 部署架构

RocketMQ的角色介绍

- Producer: 消息的发送者；举例：发信者
- Consumer: 消息接收者；举例：收信者
- Broker: 暂存和传输消息；举例：邮局
- NameServer: 管理Broker；举例：各个邮局的管理机构
- Topic: 区分消息的种类；一个发送者可以发送消息给一个或者多个Topic；一个消息的接收者可以订阅一个或者多个Topic消息
- Message Queue: 相当于Topic的分区；用于并行发送和接收消息



- NameServer是一个几乎无状态节点，可集群部署，节点之间无任何信息同步。
- Broker部署相对复杂，Broker分为Master与Slave，一个Master可以对应多个Slave，但是一个Slave只能对应一个Master，Master与Slave 的对应关系通过指定相同的BrokerName，**不同的BrokerId来定义**，BrokerId为0表示Master，非0表示Slave。Master也可以部署多个。每个Broker与NameServer集群中的所有节点建立长连接，定时注册Topic信息到所有NameServer。注意：当前RocketMQ版本在部署架构上支持一Master多Slave，但**只有BrokerId=1的从服务器才会参与消息的读负载**。
- Producer与NameServer集群中的其中一个节点（随机选择）建立长连接，定期从NameServer获取Topic路由信息，并向提供Topic服务的Master建立长连接，且定时向Master发送心跳。Producer完全无状态，可集群部署。
- Consumer与NameServer集群中的其中一个节点（随机选择）建立长连接，定期从NameServer获取Topic路由信息，并向提供Topic服务的Master、Slave建立长连接，且定时向Master、Slave发送心跳。Consumer既可以从Master订阅消息，也可以从Slave订阅消息，消费者在向Master拉取消息时，Master服务器会根据拉取偏移量与最大偏移量的距离（判断是否读老消息，产生读I/O），以及从服务器是否可读等因素建议下一次是从Master还是Slave拉取。

执行流程：

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

1.4 RocketMQ特性

1 订阅与发布

消息的发布是指某个生产者向某个topic发送消息；消息的订阅是指某个消费者关注了某个topic中带有某些tag的消息。

2 消息顺序

消息有序指的是一类消息消费时，能按照发送的顺序来消费。例如：一个订单产生了三条消息分别是订单创建、订单付款、订单完成。消费时要按照这个顺序消费才能有意义，但是同时订单之间是可以并行消费的。RocketMQ可以严格的保证消息有序。

3 消息过滤

RocketMQ的消费者可以根据Tag进行消息过滤，也支持自定义属性过滤。消息过滤目前是在Broker端实现的，优点是减少了对于Consumer无用消息的网络传输，缺点是增加了Broker的负担、而且实现相对复杂。

4 消息可靠性

RocketMQ支持消息的高可靠，影响消息可靠性的几种情况： 1)Broker非正常关闭 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。注：RocketMQ从3.0版本开始支持同步双写。

5 至少一次

至少一次(At least Once)指每个消息必须投递一次。Consumer先Pull消息到本地，消费完成后，才向服务器返回ack，如果没有消费一定不会ack消息，所以RocketMQ可以很好的支持此特性。

6 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。

7 事务消息

RocketMQ事务消息（Transactional Message）是指应用本地事务和发送消息操作可以被定义到全局事务中，要么同时成功，要么同时失败。

RocketMQ的事务消息提供类似 X/Open XA 的分布事务功能，通过事务消息能达到分布式事务的最终一致性。

8 定时消息

定时消息（延迟队列）是指消息发送到broker后，不会立即被消费，等待特定时间投递给真正的topic。

broker有配置项messageDelayLevel，默认值为“1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m 30m 1h 2h”，18个level。

messageDelayLevel是broker的属性，不属于某个topic。发消息时，设置delayLevel等级即可：msg.setDelayLevel(level)。level有以下三种情况：

- level == 0, 消息为非延迟消息
- 1<=level<=maxLevel, 消息延迟特定时间，例如level==1, 延迟1s
- level > maxLevel, 则level== maxLevel, 例如level==20, 延迟2h

定时消息会暂存在名为SCHEDULE_TOPIC_XXXX的topic中，并根据delayTimeLevel存入特定的queue, queueId = delayTimeLevel - 1, 即一个queue只存相同延迟的消息，保证具有相同发送延迟的消息能够顺序消费。broker会调度地消费SCHEDULE_TOPIC_XXXX，将消息写入真实的topic。

需要注意的是，定时消息会在第一次写入和调度写入真实topic时都会计数，因此发送数量、tps都会变高。

9 消息重试

Consumer消费消息失败后，要提供一种重试机制，令消息再消费一次。Consumer消费消息失败通常可以认为有以下几种情况：

1)由于消息本身的原因，例如反序列化失败，消息数据本身无法处理（例如话费充值，当前消息的手机号被注销，无法充值）等。这种错误通常需要跳过这条消息，再消费其它消息，而这条失败的消息即使立刻重试消费，99%也不成功，所以最好提供一种定时重试机制，即过10秒后再重试。

2)由于依赖的下游应用服务不可用，例如db连接不可用，外系统网络不可达等。遇到这种错误，即使跳过当前失败的消息，消费其他消息同样也会报错。这种情况建议应用sleep 30s，再消费下一条消息，这样可以减轻Broker重试消息的压力。

10 消息重投

生产者在发送消息时：

- 同步消息失败会重投
- 异步消息有重试
- oneway没有任何保证。

消息重投保证消息尽可能发送成功、不丢失，但可能会造成消息重复，**消息重复**在RocketMQ中是**无法避免**的问题。消息重复在一般情况下不会发生，当出现消息量大、网络抖动，消息重复就会是大概率事件。另外，生产者主动重发、consumer负载变化也会导致重复消息。

如下方法可以设置消息重试策略：

1)retryTimesWhenSendFailed：同步发送失败重投次数，默认为2，因此生产者会最多尝试发送retryTimesWhenSendFailed + 1次。不会选择上次失败的broker，尝试向其他broker发送，最大程度保证消息不丢失。超过重投次数，抛异常，由客户端保证消息不丢失。当出现RemotingException、MQClientException和部分MQBrokerException时会重投。

2)retryTimesWhenSendAsyncFailed：异步发送失败重试次数，异步重试不会选择其他broker，仅在同一个broker上做重试，不保证消息不丢。

3)retryAnotherBrokerWhenNotStoreOK：消息刷盘（主或备）超时或slave不可用（返回状态非SEND_OK），是否尝试发送到其他broker，默认false。十分重要的消息可以开启。

11 流量控制

生产者流控，因为broker处理能力达到瓶颈；消费者流控，因为消费能力达到瓶颈。

1) 生产者流控：

- commitLog文件被锁时间超过osPageCacheBusyTimeOutMills时，参数默认为1000ms，发生流控。
- 如果开启transientStorePoolEnable = true，且broker为异步刷盘的主机，且transientStorePool中资源不足，拒绝当前send请求，发生流控。
- broker每隔10ms检查send请求队列头部请求的等待时间，如果超过waitTimeMillsInSendQueue，默认200ms，拒绝当前send请求，发生流控。
- broker通过拒绝send 请求方式实现流量控制。

注意，**生产者流控，不会尝试消息重投。**

2) 消费者流控：

- 消费者本地缓存消息数超过pullThresholdForQueue时，默认1000。
- 消费者本地缓存消息大小超过pullThresholdSizeForQueue时，默认100MB。
- 消费者本地缓存消息跨度超过consumeConcurrentlyMaxSpan时，默认2000。
- 消费者流控的结果是降低拉取频率。

12 死信队列

死信队列用于处理无法被正常消费的消息。

当一条消息初次消费失败，消息队列会自动进行消息重试；

达到最大重试次数后，若消费依然失败，则表明消费者在正常情况下无法正确地消费该消息，

此时，消息队列不会立刻将消息丢弃，而是将其发送到该消费者对应的特殊队列中。

RocketMQ将这种正常情况下无法被消费的消息称为死信消息（Dead-Letter Message），

将存储死信消息的特殊队列称为死信队列（Dead-Letter Queue）。

在RocketMQ中，可以通过使用console控制台对死信队列中的消息进行重发来使得消费者实例再次进行消费。

1.5 消费模式Push or Pull

RocketMQ消息订阅有两种模式，一种是Push模式（MQPushConsumer），即MQServer主动向消费端推送；另外一种是Pull模式（MQPullConsumer），即消费端在需要时，主动到MQ Server拉取。但在具体实现时，Push和Pull模式**本质都是采用消费端主动拉取的方式**，即consumer轮询从broker拉取消息。

- Push模式特点：

好处就是实时性高。不好处在于消费端的处理能力有限，当瞬间推送很多消息给消费端时，容易造成消费端的消息积压，严重时会压垮客户端。

- Pull模式

好处就是主动权掌握在消费端自己手中，根据自己的处理能力量力而行。缺点就是如何控制Pull的频率。定时间隔太久担心影响时效性，间隔太短担心做太多“无用功”浪费资源。比较折中的办法就是长轮询。

- Push模式与Pull模式的区别：

Push方式里，consumer把长轮询的动作封装了，并注册MessageListener监听器，取到消息后，唤醒MessageListener的consumeMessage()来消费，对用户而言，感觉消息是被推送过来的。

Pull方式里，取消息的过程需要用户自己主动调用，首先通过打算消费的Topic拿到MessageQueue的集合，遍历MessageQueue集合，然后针对每个MessageQueue批量取消息，一次取完后，记录该队列下一次要取的开始offset，直到取完了，再换另一个MessageQueue。

RocketMQ使用长轮询机制来模拟Push效果，算是兼顾了二者的优点。

1.6 RocketMQ中的角色及相关术语

1)消息模型 (Message Model)

RocketMQ主要由 Producer、Broker、Consumer 三部分组成，其中Producer 负责生产消息，Consumer 负责消费消息，Broker 负责存储消息。Broker 在实际部署过程中对应一台服务器，每个 Broker 可以存储多个Topic的消息，每个Topic的消息也可以分片存储于不同的 Broker。Message Queue 用于存储消息的物理地址，每个Topic中的消息地址存储于多个 Message Queue 中。ConsumerGroup 由多个Consumer 实例构成。

2)Producer

消息生产者，负责产生消息，一般由业务系统负责产生消息。

3)Consumer

消息消费者，负责消费消息，一般是后台系统负责异步消费。

4)PushConsumer

Consumer消费的一种类型，该模式下Broker收到数据后会主动推送给消费端。应用通常向 Consumer对象注册一个Listener接口，一旦收到消息，Consumer对象立刻回调Listener接口方法。该消费模式一般实时性较高。

5)PullConsumer

Consumer消费的一种类型，应用通常主动调用Consumer的拉消息方法从Broker服务器拉消息、主动权由应用控制。一旦获取了批量消息，应用就会启动消费过程。

6)ProducerGroup

同一类Producer的集合，这类Producer发送同一类消息且发送逻辑一致。如果发送的是事务消息且原始生产者在发送之后崩溃，则Broker服务器会联系同一生产者组的其他生产者实例以提交或回溯消费。

7)ConsumerGroup

同一类Consumer的集合，这类Consumer通常消费同一类消息且消费逻辑一致。消费者组使得在消息消费方面，实现负载均衡和容错的目标变得非常容易。要注意的是，消费者组的消费者实例**必须订阅完全相同的Topic**。RocketMQ 支持两种消息模式：集群消费（Clustering）和广播消费（Broadcasting）。

8)Broker

消息中转角色，负责存储消息，转发消息，一般也称为 Server。在 JMS 规范中称为 Provider。

9)广播消费

一条消息被多个 Consumer 消费，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，广播消费中的 Consumer Group 概念可以认为在消息划分方面无意义。

在 CORBA Notification 规范中，消费方式都属于广播消费。

在 JMS 规范中，相当于 JMS Topic（publish/subscribe）模型

10)集群消费

一个 Consumer Group 中的 Consumer 实例平均分摊消费消息。例如某个 Topic 有 9 条消息，其中一个 Consumer Group 有 3 个实例(可能是 3 个进程，或者3台机器)，那举每个实例只消费其中的 3 条消息。

11)顺序消息

消费消息的顺序要同发送消息的顺序一致，在RocketMQ 中主要指的是局部顺序，即一类消息为满足顺序性，必须Producer单线程顺序发送，且发送到同一个队列，这样Consumer 就可以按照 Producer发送的顺序去消费消息。

12)普通顺序消息

顺序消息的一种，正常情况下可以保证完全的顺序消息，但是一旦发生通信异常，Broker 重启，由于队列总数发生变化，哈希取模后定位的队列会变化，产生短暂的消息顺序不一致。如果业务能容忍在集群异常情况(如某个Broker 宕机或者重启)下，消息短暂的乱序，使用普通顺序方式比较合适。

13)严格顺序消息

顺序消息的一种，无论正常异常情况都能保证顺序，但是牺牲了分布式 Failover特性，即Broker集群中只要有一台机器不可用，则整个集群都不可用，服务可用性大大降低。如果服务器部署为同步双写模式，此缺陷可通过备机自动切换为主避免，不过仍然会存在几分钟的服务不可用。(依赖同步双写，主备自动切换，自动切换功能目前还未实现)

目前已知的应用只有数据库 binlog 同步强依赖严格顺序消息，其他应用绝大部分都可以容忍短暂乱序，推荐使用普通的顺序消息。

14)Message Queue

在 RocketMQ 中，所有消息队列都是持久化的，长度无限的数据结构，所谓长度无限是指队列中的每个存储单元都是定长，访问其中的存储单元使用**Offset**来访问，offset 为 java long 类型，64 位，理论上在 100 年内不会溢出，所以认为是长度无限，另外队列中只保存最近几天的数据，之前的数据会按照过期时间来删除。也可以认为Message Queue是一个长度无限的数组，offset 就是下标。

15)标签 (Tag)

为消息设置的标志，用于同一主题下区分不同类型的消息。来自同一业务单元的消息，可以根据不同业务目的在同一主题下设置不同标签。标签能够有效地保持代码的清晰度和连贯性，并优化 RocketMQ 提供的查询系统。消费者可以根据 Tag 实现对不同子主题的不同消费逻辑，实现更好的扩展性。

1.7 RocketMQ环境搭建

1.软件准备:

RocketMQ最新版本：4.5.1

[下载地址](#)

2.环境要求

- JDK 11.0.5
- Linux64位系统(CentOS Linux release 7.7.1908)
- 源码安装需要安装Maven 3.2.x
- 4G+ free

3.安装及启动

1. 下载rocketmq

```
1 #下载
2 wget https://archive.apache.org/dist/rocketmq/4.5.1/rocketmq-all-
4.5.1-bin-release.zip
```

2. 修改脚本

bin/runserver.sh
bin/runbroker.sh
bin/tools.sh

nameserver:

```
1#!/bin/sh
2
3# Licensed to the Apache Software Foundation (ASF) under one or more
4# contributor license agreements. See the NOTICE file distributed with
5# this work for additional information regarding copyright ownership.
6# The ASF licenses this file to You under the Apache License, Version 2.0
7# (the "License"); you may not use this file except in compliance with
8# the License. You may obtain a copy of the License at
9#
10#     http://www.apache.org/licenses/LICENSE-2.0
11#
12# Unless required by applicable law or agreed to in writing, software
13# distributed under the License is distributed on an "AS IS" BASIS,
14# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
15# See the License for the specific language governing permissions and
16# limitations under the License.
17
18#####
19# Java Environment Setting
20#####
21error_exit()
22{
```

```

23 echo "ERROR: $1 !!" 
24 exit 1
25 }
26
27 [ ! -e "$JAVA_HOME/bin/java" ] && JAVA_HOME=$HOME/jdk/java
28 [ ! -e "$JAVA_HOME/bin/java" ] && JAVA_HOME=/usr/java
29 [ ! -e "$JAVA_HOME/bin/java" ] && error_exit "Please set the JAVA_HOME
variable in your environment, we need java(x64)!"
30
31 export JAVA_HOME
32 export JAVA="$JAVA_HOME/bin/java"
33 export BASE_DIR=$(dirname $0)..
34 export
35 CLASSPATH=.:${BASE_DIR}/conf:${JAVA_HOME}/jre/lib/ext:${BASE_DIR}/lib/*
36 #=====
37 =====
38 # JVM Configuration
39 #=====
40 =====
41 JAVA_OPT="${JAVA_OPT} -server -Xms256m -Xmx256m -Xmn128m -
XX:MetaspaceSize=64m -XX:MaxMetaspaceSize=160m"
42 JAVA_OPT="${JAVA_OPT} -XX:CMSInitiatingOccupancyFraction=70 -
XX:+CMSParallelRemarkEnabled -XX:SoftRefLRUPolicyMSPerMB=0 -
XX:+CMSClassUnloadingEnabled -xx:SurvivorRatio=8"
43 JAVA_OPT="${JAVA_OPT} -verbose:gc -Xlog:gc:/dev/shm/rmq_srv_gc.log -
XX:+PrintGCDetails"
44 JAVA_OPT="${JAVA_OPT} -XX:-OmitStackTraceInFastThrow"
45 JAVA_OPT="${JAVA_OPT} -XX:-UseLargePages"
46 # JAVA_OPT="${JAVA_OPT} -
Djava.ext.dirs=${JAVA_HOME}/jre/lib/ext:${BASE_DIR}/lib"
47 #JAVA_OPT="${JAVA_OPT} -Xdebug -
Xrunjdwp:transport=dt_socket,address=9555,server=y,suspend=n"
48 JAVA_OPT="${JAVA_OPT} ${JAVA_OPT_EXT}"
49 JAVA_OPT="${JAVA_OPT} -cp ${CLASSPATH}"
50
51 $JAVA ${JAVA_OPT} $@

```

```

1 vim bin/runserver.sh
2 删除
3 UseCMSCompactAtFullCollection
4 UseParNewGC
5 UseConcMarkSweepGC
6 修改内存:
7 JAVA_OPT="${JAVA_OPT} -server -Xms256m -Xmx256m -Xmn128m -
XX:MetaspaceSize=64mm -XX:MaxMetaspaceSize=160mm"
8 -Xlogg修改为-Xlog:gc

```

broker:

```

1 #!/bin/sh
2
3 # Licensed to the Apache Software Foundation (ASF) under one or more
4 # contributor license agreements. See the NOTICE file distributed with
5 # this work for additional information regarding copyright ownership.
6 # The ASF licenses this file to You under the Apache License, Version 2.0

```

```
7 # (the "License"); you may not use this file except in compliance with
8 # the License. You may obtain a copy of the License at
9 #
10 #     http://www.apache.org/licenses/LICENSE-2.0
11 #
12 # Unless required by applicable law or agreed to in writing, software
13 # distributed under the License is distributed on an "AS IS" BASIS,
14 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
15 # See the License for the specific language governing permissions and
16 # limitations under the License.
17
18 #=====
19 # Java Environment Setting
20 #=====
21 =====
22 error_exit ()
23 {
24     echo "ERROR: $1 !!"
25     exit 1
26 }
27 [ ! -e "$JAVA_HOME/bin/java" ] && JAVA_HOME=$HOME/jdk/java
28 [ ! -e "$JAVA_HOME/bin/java" ] && JAVA_HOME=/usr/java
29 [ ! -e "$JAVA_HOME/bin/java" ] && error_exit "Please set the JAVA_HOME
variable in your environment, we need java(x64)!"
30
31 export JAVA_HOME
32 export JAVA="$JAVA_HOME/bin/java"
33 export BASE_DIR=$(dirname $0)..
34 export
35 CLASSPATH=.:${JAVA_HOME}/jre/lib/ext:${BASE_DIR}/lib/*:${BASE_DIR}/conf:${CL
ASSPATH}
36 #=====
37 # JVM Configuration
38 #=====
39 JAVA_OPT="${JAVA_OPT} -server -Xms256m -Xmx256m -Xmn128m"
40 JAVA_OPT="${JAVA_OPT} -XX:+UseG1GC -XX:G1HeapRegionSize=16m -
XX:G1ReservePercent=25 -XX:InitiatingHeapOccupancyPercent=30 -
XX:SoftRefLRUPolicyMSPerMB=0"
41 JAVA_OPT="${JAVA_OPT} -verbose:gc -xloggc:/dev/shm/mq_gc_%p.log -
XX:+PrintGCDetails"
42 JAVA_OPT="${JAVA_OPT} -XX:-OmitStackTraceInFastThrow"
43 JAVA_OPT="${JAVA_OPT} -XX:+AlwaysPreTouch"
44 JAVA_OPT="${JAVA_OPT} -XX:MaxDirectMemorySize=15g"
45 JAVA_OPT="${JAVA_OPT} -XX:-UseLargePages -XX:-UseBiasedLocking"
46 #JAVA_OPT="${JAVA_OPT} -Xdebug -
Xrunjdwp:transport=dt_socket,address=9555,server=y,suspend=n"
47 JAVA_OPT="${JAVA_OPT} ${JAVA_OPT_EXT}"
48 JAVA_OPT="${JAVA_OPT} -cp ${CLASSPATH}"
49
50 numactl --interleave=all pwd > /dev/null 2>&1
51 if [ $? -eq 0 ]
52 then
53     if [ -z "$RMQ_NUMA_NODE" ] ; then
```

```

54         numactl --interleave=all $JAVA ${JAVA_OPT} @@
55     else
56         numactl --cpunodebind=$RMQ_NUMA_NODE --membind=$RMQ_NUMA_NODE $JAVA
57             ${JAVA_OPT} @@
58     fi
59 else
60     $JAVA ${JAVA_OPT} --add-exports=java.base/jdk.internal.ref=ALL-UNNAMED
61 @@
62 fi

```

```

58 else
59     $JAVA ${JAVA_OPT} --add-exports=java.base/jdk.internal.ref=ALL-UNNAMED @@
60 fi

```

```

1 vim bin/runbroker.sh
2
3 删除:
4 PrintGCDateStamps
5 PrintGCApplicationStoppedTime
6 PrintAdaptiveSizePolicy
7 UseGCLogFileRotation
8 NumberOfGCLogFile=5
9 GCLogFile=30m

```

tools:

```

1#!/bin/sh
2
3# Licensed to the Apache Software Foundation (ASF) under one or more
4# contributor license agreements. See the NOTICE file distributed with
5# this work for additional information regarding copyright ownership.
6# The ASF licenses this file to You under the Apache License, Version 2.0
7# (the "License"); you may not use this file except in compliance with
8# the License. You may obtain a copy of the License at
9#
10#      http://www.apache.org/licenses/LICENSE-2.0
11#
12# Unless required by applicable law or agreed to in writing, software
13# distributed under the License is distributed on an "AS IS" BASIS,
14# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
15# See the License for the specific language governing permissions and
16# limitations under the License.
17
18#####
19# Java Environment Setting
20#####
21error_exit ()
22{
23    echo "ERROR: $1 !"
24    exit 1
25}
26
27[ ! -e "$JAVA_HOME/bin/java" ] && JAVA_HOME=$HOME/jdk/java
28[ ! -e "$JAVA_HOME/bin/java" ] && JAVA_HOME=/usr/java

```

```
29 [ ! -e "$JAVA_HOME/bin/java" ] && error_exit "Please set the JAVA_HOME  
30 variable in your environment, we need java(x64)!"  
31  
32 export JAVA_HOME  
33 export JAVA="$JAVA_HOME/bin/java"  
34 export BASE_DIR=$(dirname $0)/..  
35 # export CLASSPATH=.:${BASE_DIR}/conf:${CLASSPATH}  
36 export  
37 CLASSPATH=.:${JAVA_HOME}/jre/lib/ext:${BASE_DIR}/lib/*:${BASE_DIR}/conf:${CL  
ASSPATH}  
38 #=====  
39 # JVM Configuration  
40 #=====  
41 JAVA_OPT="${JAVA_OPT} -server -Xms256m -Xmx256m -Xmn256m -XX:PermSize=128m  
-XX:MaxPermSize=128m"  
42 # JAVA_OPT="${JAVA_OPT} -  
Djava.ext.dirs=${BASE_DIR}/lib:${JAVA_HOME}/jre/lib/ext"  
43 JAVA_OPT="${JAVA_OPT} -cp ${CLASSPATH}"  
44  
45 $JAVA ${JAVA_OPT} $@
```

```
1 vim bin/tools.sh  
2  
3 # 删除 JAVA_OPT="${JAVA_OPT} -  
Djava.ext.dirs=${BASE_DIR}/lib:${JAVA_HOME}/jre/lib/ext"
```

3. 启动NameServer

```
1 # 1.启动NameServer  
2 mqnamesrv  
3 # 2.查看启动日志  
4 tail -f ~/logs/rocketmq/logs/namesrv.log
```

4. 启动Broker

```
1 # 1.启动Broker  
2 mqbroker -n localhost:9876  
3 # 2.查看启动日志  
4 tail -f ~/logs/rocketmq/logs/broker.log
```

1.8 RocketMQ环境测试

1. 发送消息

```
1 # 1.设置环境变量  
2 export NAMESRV_ADDR=localhost:9876  
3 # 2.使用安装包的Demo发送消息  
4 sh bin/tools.sh org.apache.rocketmq.example.quickstart.Producer
```

2. 接收消息

```
1 # 1. 设置环境变量  
2 export NAMESRV_ADDR=localhost:9876  
3 # 2. 接收消息  
4 sh bin/tools.sh org.apache.rocketmq.example.quickstart.Consumer
```

3. 关闭RocketMQ

```
1 # 1. 关闭NameServer  
2 mqshutdown namesrv  
3 # 2. 关闭Broker  
4 mqshutdown broker
```

1.9 RocketMQ相关API使用

具体参考代码，学员需熟悉各种用法，API等

DefaultMQProducer 生产者的默认实现

生产消息分同步发送和异步发送

DefaultMQConsumer：消费者的默认实现

消息的拉取和消息的推送

MyProducer.java

```
1 package com.lagou.rocket.demo.producer;  
2  
3 import org.apache.rocketmq.client.exception.MQBrokerException;  
4 import org.apache.rocketmq.client.exception.MQClientException;  
5 import org.apache.rocketmq.client.producer.DefaultMQProducer;  
6 import org.apache.rocketmq.client.producer.SendResult;  
7 import org.apache.rocketmq.common.message.Message;  
8 import org.apache.rocketmq.remoting.common.RemotingHelper;  
9 import org.apache.rocketmq.remoting.exception.RemotingException;  
10  
11 import java.io.UnsupportedEncodingException;  
12  
13 public class MyProducer {  
14  
15     public static void main(String[] args) throws  
UnsupportedEncodingException, InterruptedException, RemotingException,  
MQClientException, MQBrokerException {  
16         // 在实例化生产者的同时，指定了生产组名称  
17         DefaultMQProducer producer = new  
DefaultMQProducer("myproducer_grp_01");  
18  
19         // 指定NameServer的地址  
20         producer.setNamesrvAddr("node1:9876");  
21  
22         // 对生产者进行初始化，然后就可以使用了  
23         producer.start();
```

```
24
25     // 创建消息，第一个参数是主题名称，第二个参数是消息内容
26     Message message = new Message(
27         "tp_demo_01",
28         "hello lagou 01".getBytes(RemotingHelper.DEFAULT_CHARSET)
29     );
30     // 发送消息
31     final SendResult result = producer.send(message);
32     System.out.println(result);
33
34     // 关闭生产者
35     producer.shutdown();
36 }
37
38
39 }
40 }
```

MyAsyncProducer.java

```
1 package com.lagou.rocket.demo.producer;
2
3 import org.apache.rocketmq.client.exception.MQClientException;
4 import org.apache.rocketmq.client.producer.DefaultMQProducer;
5 import org.apache.rocketmq.client.producer.SendCallback;
6 import org.apache.rocketmq.client.producer.SendResult;
7 import org.apache.rocketmq.common.message.Message;
8 import org.apache.rocketmq.remoting.exception.RemotingException;
9
10 import java.io.UnsupportedEncodingException;
11
12 public class MyAsyncProducer {
13     public static void main(String[] args) throws MQClientException,
14     UnsupportedEncodingException, RemotingException, InterruptedException {
15         // 实例化生产者，并指定生产组名称
16         DefaultMQProducer producer = new
17         DefaultMQProducer("producer_grp_01");
18
19         // 指定nameserver的地址
20         producer.setNamesrvAddr("node1:9876");
21
22         // 初始化生产者
23         producer.start();
24
25         for (int i = 0; i < 100; i++) {
26
27             Message message = new Message(
28                 "tp_demo_02",
29                 ("hello lagou " + i).getBytes("utf-8")
30             );
31
32             // 消息的异步发送
33             producer.send(message, new SendCallback() {
34                 @Override
35                 public void onSuccess(SendResult sendResult) {
```

```

34             System.out.println("发送成功：" + sendResult);
35         }
36
37         @Override
38         public void onException(Throwable throwable) {
39             System.out.println("发送失败：" +
throwable.getMessage());
40         }
41     });
42 }
43
44 // 由于是异步发送消息，上面循环结束之后，消息可能还没收到broker的响应
45 // 如果不sleep一会儿，就报错
46 Thread.sleep(10_000);
47
48 // 关闭生产者
49 producer.shutdown();
50 }
51
52 }
```

MyPullConsumer.java

```

1 package com.lagou.rocket.demo.consumer;
2
3 import org.apache.rocketmq.client.consumer.DefaultMQPullConsumer;
4 import org.apache.rocketmq.client.consumer.PullResult;
5 import org.apache.rocketmq.client.exception.MQBrokerException;
6 import org.apache.rocketmq.client.exception.MQClientException;
7 import org.apache.rocketmq.common.message.MessageExt;
8 import org.apache.rocketmq.common.message.MessageQueue;
9 import org.apache.rocketmq.remoting.exception.RemotingException;
10
11 import java.io.UnsupportedEncodingException;
12 import java.util.List;
13 import java.util.Set;
14
15 /**
16  * 拉取消息的消费者
17 */
18 public class MyPullConsumer {
19     public static void main(String[] args) throws MQClientException,
RemotingException, InterruptedException, MQBrokerException,
UnsupportedEncodingException {
20         // 拉取消息的消费者实例化，同时指定消费组名称
21         DefaultMQPullConsumer consumer = new
DefaultMQPullConsumer("consumer_grp_01");
22         // 设置nameserver的地址
23         consumer.setNamesrvAddr("node1:9876");
24
25         // 对消费者进行初始化，然后就可以使用了
26         consumer.start();
27
28         // 获取指定主题的消息队列集合
29 }
```

```

29         final Set<MessageQueue> messageQueues =
30         consumer.fetchSubscribeMessageQueues("tp_demo_01");
31
32         // 遍历该主题的各个消息队列，进行消费
33         for (MessageQueue messageQueue : messageQueues) {
34             // 第一个参数是MessageQueue对象，代表了当前主题的一个消息队列
35             // 第二个参数是一个表达式，对接收的消息按照tag进行过滤
36             // 支持"tag1 || tag2 || tag3"或者 "*"类型的写法；null或者""表示不对
37             // 消息进行tag过滤
38             // 第三个参数是消息的偏移量，从这里开始消费
39             // 第四个参数表示每次最多拉取多少条消息
40             final PullResult result = consumer.pull(messageQueue, "*", 0,
41             10);
42             // 打印消息队列的信息
43             System.out.println("message*****queue*****" + messageQueue);
44             // 获取从指定消息队列中拉取到的消息
45             final List<MessageExt> msgFoundList = result.getMsgFoundList();
46             if (msgFoundList == null) continue;
47             for (MessageExt messageExt : msgFoundList) {
48                 System.out.println(messageExt);
49                 System.out.println(new String(messageExt.getBody(), "utf-
50                 8"));
51             }
52         }
53     }
54 }
```

MyPushConsumer.java

```

1 package com.taobao.rocket.demo.consumer;
2
3 import org.apache.rocketmq.client.consumer.DefaultMQPushConsumer;
4 import
5 org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlyContext;
6 import
7 org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlyStatus;
8 import
9 org.apache.rocketmq.client.consumer.listener.MessageListenerConcurrently;
10 import org.apache.rocketmq.client.exception.MQClientException;
11 import org.apache.rocketmq.common.message.MessageExt;
12 import org.apache.rocketmq.common.message.MessageQueue;
13
14 /**
15 * 推送消息的消费
16 */
17 public class MyPushConsumer {
18     public static void main(String[] args) throws MQClientException,
19     InterruptedException {
```

```

19 // 实例化推送消息消费者的对象，同时指定消费组名称
20 DefaultMQPushConsumer consumer = new
21 DefaultMQPushConsumer("consumer_grp_02");
22
23 // 指定nameserver的地址
24 consumer.setNamesrvAddr("node1:9876");
25
26 // 订阅主题
27 consumer.subscribe("tp_demo_02", "*");
28
29 // 添加消息监听器，一旦有消息推送过来，就进行消费
30 consumer.setMessageListener(new MessageListenerConcurrently() {
31     @Override
32     public ConsumeConcurrentlyStatus
33     consumeMessage(List<MessageExt> msgs, ConsumeConcurrentlyContext context) {
34
35         final MessageQueue messageQueue =
36         context.getMessageQueue();
37         System.out.println(messageQueue);
38
39         for (MessageExt msg : msgs) {
40             try {
41                 System.out.println(new String(msg.getBody(), "utf-
42                     8"));
43             } catch (UnsupportedEncodingException e) {
44                 e.printStackTrace();
45             }
46
47             // 消息消费成功
48             return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
49             // 消息消费失败
50             // return ConsumeConcurrentlyStatus.RECONSUME_LATER;
51         }
52
53         // 初始化消费者，之后开始消费消息
54         consumer.start();
55
56         // 此处只是示例，生产中除非运维关掉，否则不应停掉，长服务
57         // Thread.sleep(30_000);
58         // 关闭消费者
59         // consumer.shutdown();
60     }
61 }
```

1.10 RocketMQ和Spring的整合

下载[rocketmq-spring](#)项目

将rocketmq-spring安装到本地仓库

```
1 | mvn install -Dmaven.skip.test=true
```

1.10.1 消息生产者

1) 添加依赖

```
1 <parent>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-parent</artifactId>
4   <version>2.0.1.RELEASE</version>
5 </parent>
6
7 <properties>
8   <rocketmq-spring-boot-starter-version>2.0.3</rocketmq-spring-boot-
9   starter-version>
10 </properties>
11
12 <dependencies>
13   <dependency>
14     <groupId>org.apache.rocketmq</groupId>
15     <artifactId>rocketmq-spring-boot-starter</artifactId>
16     <version>${rocketmq-spring-boot-starter-version}</version>
17   </dependency>
18   <dependency>
19     <groupId>org.projectlombok</groupId>
20     <artifactId>lombok</artifactId>
21     <version>1.18.6</version>
22   </dependency>
23   <dependency>
24     <groupId>org.springframework.boot</groupId>
25     <artifactId>spring-boot-starter-test</artifactId>
26     <scope>test</scope>
27   </dependency>
28 </dependencies>
```

2) 配置文件

```
1 # application.properties
2 rocketmq.name-server=192.168.80.121:9876;192.168.80.122:9876
3 rocketmq.producer.group=my-group
```

3) 启动类

```
1 @SpringBootApplication
2 public class MQProducerApplication {
3   public static void main(String[] args) {
4     SpringApplication.run(MQSpringBootApplication.class);
5   }
6 }
```

4) 测试类

```
1 @RunWith(SpringRunner.class)
2 @SpringBootTest(classes = {MQSpringBootApplication.class})
3 public class ProducerTest {
4
5     @Autowired
6     private RocketMQTemplate rocketMQTemplate;
7
8     @Test
9     public void test1(){
10         rocketMQTemplate.convertAndSend("springboot-mq", "hello springboot
11         rocketmq");
12     }
13 }
```

1.10.2 消息消费者

1) 添加依赖

同消息生产者

2) 配置文件

同消息生产者

3) 启动类

```
1 @SpringBootApplication
2 public class MQConsumerApplication {
3     public static void main(String[] args) {
4         SpringApplication.run(MQSpringBootApplication.class);
5     }
6 }
```

4) 消息监听器

```
1 @Slf4j
2 @Component
3 @RocketMQMessageListener(topic = "springboot-mq", consumerGroup =
4     "springboot-mq-consumer-1")
5 public class Consumer implements RocketMQListener<String> {
6
7     @Override
8     public void onMessage(String message) {
9         log.info("Receive message: "+message);
10    }
11 }
```

第二部分 RocketMQ高级特性及原理

2.1 消费发送

生产者向消息队列里写入消息，不同的业务场景需要生产者采用不同的写入策略。比如同步发送、异步发送、Oneway发送、延迟发送、发送事务消息等。默认使用的是DefaultMQProducer类，发送消息要经过五个步骤：

- 1) 设置Producer的GroupName。
- 2) 设置InstanceName，当一个JVM需要启动多个Producer的时候，通过设置不同的InstanceName来区分，不设置的话系统使用默认名称“DEFAULT”。
- 3) 设置发送失败重试次数，当网络出现异常的时候，这个次数影响消息的重复投递次数。想保证不丢消息，可以设置多重试几次。
- 4) 设置NameServer地址
- 5) 组装消息并发送。

消息发生返回状态 (SendResult#SendStatus) 有如下四种：

FLUSH_DISK_TIMEOUT

FLUSH_SLAVE_TIMEOUT

SLAVE_NOT_AVAILABLE

SEND_OK

不同状态在不同的刷盘策略和同步策略的配置下含义是不同的：

1. **FLUSH_DISK_TIMEOUT**：表示没有在规定时间内完成刷盘（需要Broker的刷盘策略被设置成SYNC_FLUSH才会报这个错误）。
2. **FLUSH_SLAVE_TIMEOUT**：表示在主备方式下，并且Broker被设置成SYNC_MASTER方式，没有在设定时间内完成主从同步。
3. **SLAVE_NOT_AVAILABLE**：这个状态产生的场景和FLUSH_SLAVE_TIMEOUT类似，表示在主备方式下，并且Broker被设置成SYNC_MASTER，但是没有找到被配置成Slave的Broker。
4. **SEND_OK**：表示发送成功，发送成功的具体含义，比如消息是否已经被存储到磁盘？消息是否被同步到了Slave上？消息在Slave上是否被写入磁盘？需要结合所配置的刷盘策略、主从策略来定。这个状态还可以简单理解为，没有发生上面列出的三个问题状态就是SEND_OK。

写一个高质量的生产者程序，重点在于对发送结果的处理，要充分考虑各种异常，写清对应的处理逻辑。

提升写入的性能

发送一条消息出去要经过三步

1. 客户端发送请求到服务器。
2. 服务器处理该请求。
3. 服务器向客户端返回应答

一次消息的发送耗时是上述三个步骤的总和。

在一些对速度要求高，但是可靠性要求不高的场景下，比如日志收集类应用，可以采用**Oneway方式发送**

Oneway方式只发送请求不等待应答，即将数据写入客户端的Socket缓冲区就返回，不等待对方返回结果。

用这种方式发送消息的耗时可以缩短到**微妙级**。

另一种提高发送速度的方法是**增加Producer的并发量，使用多个Producer同时发送**，

我们不用担心多Producer同时写会降低消息写磁盘的效率，RocketMQ引入了一个并发窗口，在窗口内消息可以并发地写入DirectMem中，然后异步地将连续一段无空洞的数据刷入文件系统当中。

顺序写CommitLog可让RocketMQ无论在HDD还是SSD磁盘情况下都能保持较高的写入性能。

目前在阿里内部经过调优的服务器上，写入性能达到90万+的TPS，我们可以参考这个数据进行系统优化。

在Linux操作系统层级进行调优，推荐使用EXT4文件系统，IO调度算法使用deadline算法。

2.2 消息消费

简单总结消费的几个要点：

1. 消息消费方式 (Pull和Push)
2. 消息消费的模式 (广播模式和集群模式)
3. 流量控制 (可以结合sentinel来实现，后面单独讲)
4. 并发线程数设置
5. 消息的过滤 (Tag、Key) TagA || TagB || TagC * null

当Consumer的处理速度跟不上消息的产生速度，会造成越来越多的消息积压，这个时候首先查看消费逻辑本身有没有优化空间，除此之外还有三种方法可以提高Consumer的处理能力。

1. 提高消费并行度

在同一个**ConsumerGroup**下 (Clustering方式)，可以通过增加**Consumer实例**的数量来提高并行度。

通过加机器，或者在已有机器中启动多个Consumer进程都可以增加Consumer实例数。

注意：总的Consumer数量不要超过Topic下**Read Queue**数量，超过的Consumer实例接收不到消息。

此外，通过提高单个Consumer实例中的**并行处理的线程数**，可以在同一个Consumer内增加并行度来提高吞吐量（设置方法是修改consumeThreadMin和consumeThreadMax）。

2. 以批量方式进行消费

某些业务场景下，多条消息同时处理的时间会大大小于逐个处理的时间总和，比如消费消息中涉及update某个数据库，一次update10条的时间会大大小于十次update1条数据的时间。

可以通过批量方式消费来提高消费的吞吐量。实现方法是设置Consumer的consumeMessageBatchMaxSize这个参数，默认是1，如果设置为N，在消息多的时候每次收到的是一个长度为N的消息链表。

3. 检测延时情况，跳过非重要消息

Consumer在消费的过程中，如果发现由于某种原因发生严重的消息堆积，短时间无法消除堆积，这个时候可以选择丢弃不重要的消息，使Consumer尽快追上Producer的进度。

2.3 消息存储

2.3.1 存储介质

- 关系型数据库DB

Apache下开源的另外一款MQ—ActiveMQ（默认采用的KahaDB做消息存储）可选用JDBC的方式来做消息持久化，通过简单的xml配置信息即可实现JDBC消息存储。由于，普通关系型数据库（如Mysql）在单表数据量达到千万级别的情况下，其IO读写性能往往会出现瓶颈。在可靠性方面，该种方案非常依赖DB，如果一旦DB出现故障，则MQ的消息就无法落盘存储会导致线上故障



- 文件系统

目前业界较为常用的几款产品（RocketMQ/Kafka/RabbitMQ）均采用的是消息刷盘至所部署虚拟机/物理机的文件系统来做持久化（刷盘一般可以分为异步刷盘和同步刷盘两种模式）。消息刷盘为消息存储提供了一种高效率、高可靠性和高性能的数据持久化方式。除非部署MQ机器本身或是本地磁盘挂了，否则一般是不会出现无法持久化的故障问题。



2.3.2 性能对比

文件系统>关系型数据库DB

2.3.3 消息的存储和发送

1) 消息存储

目前的高性能磁盘，顺序写速度可以达到600MB/s，超过了一般网卡的传输速度。

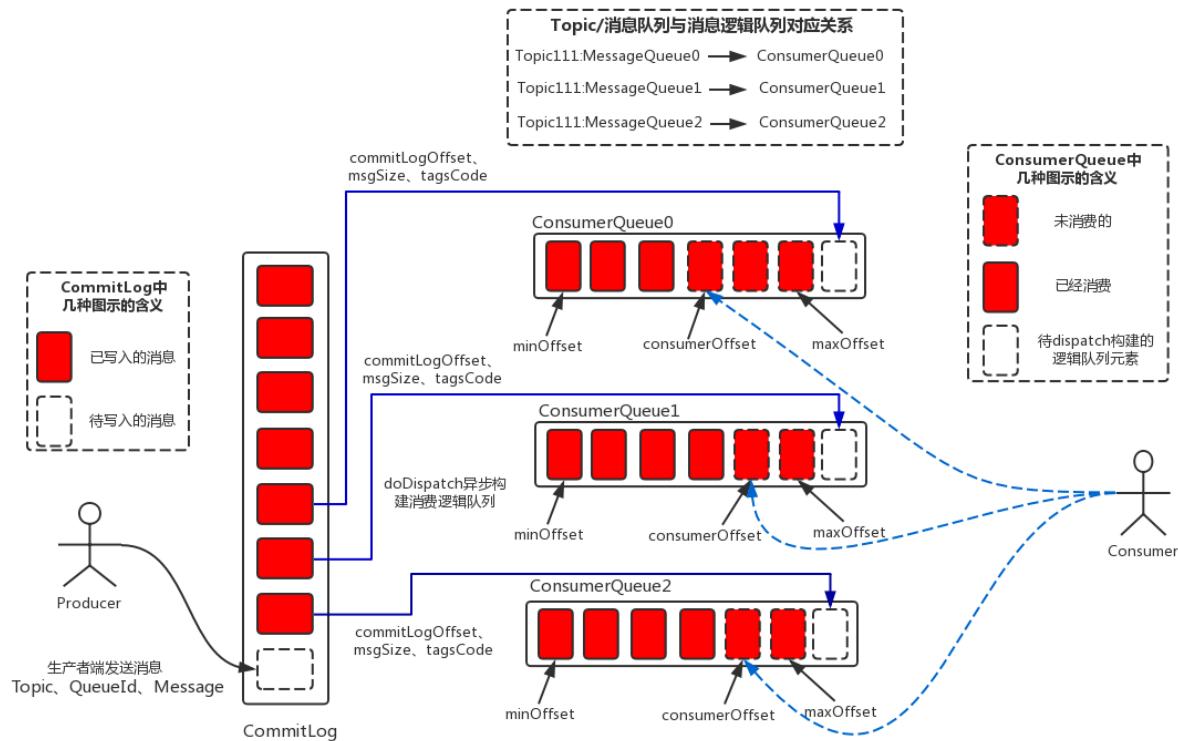
但是磁盘随机写的速度只有大概100KB/s，和顺序写的性能相差6000倍！

因为有如此巨大的速度差别，好的消息队列系统会比普通的消息队列系统速度快多个数量级。

RocketMQ的消息用顺序写，保证了消息存储的速度。

2) 存储结构

RocketMQ消息的存储是由ConsumeQueue和CommitLog配合完成的，消息真正的物理存储文件是CommitLog，ConsumeQueue是消息的逻辑队列，**类似数据库的索引文件**，存储的是**指向物理存储的地址**。每个Topic下的每个Message Queue都有一个对应的ConsumeQueue文件。



- 消息存储架构图中主要有下面三个跟消息存储相关的文件构成。

(1) CommitLog：消息主体以及元数据的存储主体，存储Producer端写入的消息主体内容,消息内容不是定长的。单个文件大小默认1G，文件名长度为20位，左边补零，剩余为起始偏移量，比如00000000000000000000代表了第一个文件，起始偏移量为0，文件大小为 $1G=1073741824$ ；当第一个文件写满了，第二个文件为0000000001073741824，起始偏移量为1073741824，以此类推。消息主要是顺序写入日志文件，当文件满了，写入下一个文件；

```
[root@node1 commitlog]# pwd
/root/store/commitlog
[root@node1 commitlog]# ls
00000000000000000000
[root@node1 commitlog]#
```

(2) ConsumeQueue：消息消费队列，引入的目的主要是**提高消息消费的性能**

RocketMQ是基于主题topic的订阅模式，消息消费是针对主题进行

如果要遍历commitlog文件根据topic检索消息是非常低效。

Consumer即可根据ConsumeQueue来查找待消费的消息。

其中，ConsumeQueue（逻辑消费队列）作为消费消息的索引：

1. 保存了指定Topic下的队列消息在CommitLog中的起始物理偏移量offset
2. 消息大小size
3. 消息Tag的HashCode值。

consumequeue文件可以看成是基于topic的commitlog索引文件，故consumequeue文件夹的组织方式如下：

topic/queue/file三层组织结构

具体存储路径为: \$HOME/store/consumequeue/{topic}/{queueId}/{fileName}。

```
[root@node1 ~]# cd store/
[root@node1 store]# ls
checkpoint  commitlog  config  consumequeue  index  lock
[root@node1 store]# cd consumequeue/
[root@node1 consumequeue]# ls
RMQ_SYS_TRACE_TOPIC  springboot-mq  TopicTest  tp_demo_01
[root@node1 consumequeue]# cd tp_demo_01/
[root@node1 tp_demo_01]# ls
0  1  3
[root@node1 tp_demo_01]# cd 0/
[root@node1 0]# ls
000000000000000000000000
[root@node1 0]# pwd
/root/store/consumequeue/tp_demo_01/0
[root@node1 0]#
```

consumequeue文件采取定长设计，**每个条目共20个字节**，分别为：

1. 8字节的commitlog物理偏移量
2. 4字节的消息长度
3. 8字节tag hashcode

单个文件由30W个条目组成，可以像数组一样随机访问每一个条目

每个ConsumeQueue**文件大小约5.72M**；

(3) IndexFile: IndexFile (索引文件) 提供了一种**可以通过key或时间区间来查询消息的方法**。

1. Index文件的存储位置是: \$HOME/store/index/\${fileName}
2. 文件名fileName是以创建时的时间戳命名的
3. 固定的单个IndexFile文件大小约为400M
4. 一个IndexFile可以保存 2000W个索引
5. IndexFile的底层存储设计为在文件系统中实现**HashMap**结构，故rocketmq的索引文件其底层实现为**hash索引**。

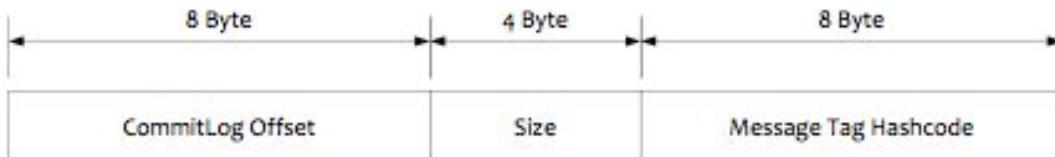
```
[root@node1 store]# pwd
/root/store
[root@node1 store]# ls
checkpoint  commitlog  config  consumequeue  index  lock
[root@node1 store]# cd index/
[root@node1 index]# ls
20200812120656631
[root@node1 index]# pwd
/root/store/index
[root@node1 index]# █
```

2.4 过滤消息

RocketMQ分布式消息队列的消息过滤方式有别于其它MQ中间件，是在**Consumer端订阅消息时再做消息过滤的**。

RocketMQ这么做是在于其Producer端写入消息和Consumer端订阅消息采用**分离存储的机制**来实现的，Consumer端订阅消息是需要通过**ConsumeQueue**这个消息消费的逻辑队列拿到一个索引，然后再从**CommitLog**里面读取真正的消息实体内容，所以说到底也是还绕不开其存储结构。

其ConsumeQueue的存储结构如下，可以看到其中有**8个字节**存储的**Message Tag的哈希值**，基于Tag的消息过滤正式基于这个字段值的。



主要支持如下2种的过滤方式

(1) Tag过滤方式：Consumer端在**订阅消息**时除了指定Topic还可以指定TAG，如果一个消息有多个TAG，可以用||分隔。

1. Consumer端会将这个订阅请求构建成一个 SubscriptionData，发送一个Pull消息的请求给 Broker端。
2. Broker端从RocketMQ的文件存储层—Store读取数据之前，会用这些数据先构建一个 MessageFilter，然后传给Store。
3. Store从 ConsumeQueue读取到一条记录后，会用它记录的消息tag hash值去做过滤。
4. 在服务端只是根据hashcode进行判断，无法精确对tag原始字符串进行过滤，在消息消费端拉取到消息后，还需要对消息的原始tag字符串进行比对，如果不同，则丢弃该消息，不进行消息消费。

(2) SQL92的过滤方式：

仅对push的消费者起作用。

Tag方式虽然效率高，但是支持的过滤逻辑比较简单。

SQL表达式可以更加灵活的支持复杂过滤逻辑，这种方式的大致做法和上面的Tag过滤方式一样，只是在Store层的具体过滤过程不太一样

真正的 SQL expression 的构建和执行由rocketmq-filter模块负责的。

每次过滤都去执行SQL表达式会影响效率，所以RocketMQ使用了BloomFilter避免了每次都去执行。

SQL92的表达式上下文为消息的属性。

```
WARNING: All illegal access operations will be denied in a future release
Exception in thread "main" org.apache.rocketmq.client.exception.MQClientException: CODE: 1 DESC: The broker does not support consumer to filter message by SQL92
For more information, please visit the url, http://rocketmq.apache.org/docs/faq/
at org.apache.rocketmq.client.impl.MQClientAPIImpl.checkClientInBroker(MQClientAPIImpl.java:2102)
at org.apache.rocketmq.client.factory.MQClientInstance.checkClientInBroker(MQClientInstance.java:455)
```

conf/broker.conf

```
flushDiskType = ASYNC_FLUSH
# 启用SQL92
enablePropertyFilter=true
```

```
[root@node1 conf]# mqbroker -p | grep enablePropertyFilter  
2020-08-13 20\:\:44\:\:36 INFO main - enablePropertyFilter=false  
[root@node1 conf]# mqbroker -c /opt/rocket/conf/broker.conf -p | grep enablePropertyFilter  
2020-08-13 20\:\:45\:\:02 INFO main - enablePropertyFilter=true  
[root@node1 conf]#
```

首先需要开启支持SQL92的特性，然后重启broker：

```
1 | mqbroker -n localhost:9876 -c /opt/rocket/conf/broker.conf
```

RocketMQ仅定义了几种基本的语法，用户可以扩展：

1. 数字比较： >, >=, <, <=, BETWEEN, =
2. 字符串比较： =, <>, IN; IS NULL或者IS NOT NULL;
3. 逻辑比较： AND, OR, NOT;
4. Constant types are: 数字如： 123, 3.1415; 字符串如： 'abc'， 必须是单引号引起来 NULL, 特殊常量 布尔型如： TRUE or FALSE;

```
message = new Message( topic: "tp_demo_01", tags: "tag" + (i % 3), ("hello lagou " + i).getBytes());  
  
message.putUserProperty("keya", "valuea" + i);  
message.putUserProperty( name: "keya", value: "valuea");  
  
//  
    consumer.subscribe("tp_demo_01", MessageSelector.bySql("TAGS in ('tag1', 'tag2')"));  
    consumer.subscribe( topic: "tp_demo_01", MessageSelector.bySql("keya = 'valuea'"));  
  
g=0, properties={keya=valuea, MIN_OFFSET=0, MAX_OFFSET=97, CONSUME_START_TIME=1597323667431, UNIQ_KEY=0A4E00A745D078308DB142479DA80026, WAIT=true, TAGS=tag2}, t  
ag=0, properties={keya=valuea, MIN_OFFSET=0, MAX_OFFSET=97, CONSUME_START_TIME=1597323667431, UNIQ_KEY=0A4E00A745D078308DB142479DA80028, WAIT=true, TAGS=tag0}, t
```

(3) Filter Server方式。这是一种比SQL表达式更灵活的过滤方式，允许用户自定义Java函数，根据Java函数的逻辑对消息进行过滤。

要使用Filter Server，首先要在启动Broker前在配置文件里加上 filterServer-Nums=3 这样的配置，Broker在启动的时候，就会在本机启动3个Filter Server进程。Filter Server类似一个RocketMQ的Consumer进程，它从本机Broker获取消息，然后根据用户上传过来的Java函数进行过滤，过滤后的消息再传给远端的Consumer。

这种方式会占用很多Broker机器的CPU资源，要根据实际情况谨慎使用。上传的java代码也要经过检查，不能有申请大内存、创建线程等这样的操作，否则容易造成Broker服务器宕机。

2.5 零拷贝原理

2.5.1 PageCache

- 由内存中的物理page组成，其内容对应磁盘上的block。
- page cache的大小是动态变化的。
- backing store: cache缓存的存储设备
- 一个page通常包含多个block, 而block不一定是连续的。

2.5.1.1 读Cache

- 当内核发起一个读请求时, 先会检查请求的数据是否缓存到了page cache中。
 - 如果有, 那么直接从内存中读取, 不需要访问磁盘, 此即 cache hit(缓存命中)
 - 如果没有, 就必须从磁盘中读取数据, 然后内核将读取的数据再缓存到cache中, 如此后续的读请求就可以命中缓存了。
- page可以只缓存一个文件的部分内容, 而不需要把整个文件都缓存进来。

2.5.1.2 写Cache

- 当内核发起一个写请求时, 也是直接往cache中写入, 后备存储中的内容不会直接更新。
- 内核会将被写入的page标记为dirty, 并将其加入到dirty list中。
- 内核会周期性地将dirty list中的page写回到磁盘上, 从而使磁盘上的数据和内存中缓存的数据一致。

2.5.1.3 cache回收

- Page cache的另一个重要工作是释放page, 从而释放内存空间。
- cache回收的任务是选择合适的page释放
 - 如果page是dirty的, 需要将page写回到磁盘中再释放。

2.5.2 cache和buffer的区别

1. Cache: 缓存区, 是高速缓存, 是位于CPU和主内存之间的容量较小但速度很快的存储器, 因为CPU的速度远远高于主内存的速度, CPU从内存中读取数据需等待很长的时间, 而 Cache 保存着CPU刚用过的数据或循环使用的部分数据, 这时从Cache中读取数据会更快, 减少了CPU等待的时间, 提高了系统的性能。

Cache并不是缓存文件的, 而是缓存块的(块是I/O读写最小的单元); Cache一般会用在I/O请求上, 如果多个进程要访问某个文件, 可以把此文件读入Cache中, 这样下一个进程获取CPU控制权并访问此文件直接从Cache读取, 提高系统性能。

2. Buffer: 缓冲区, 用于存储速度不同步的设备或优先级不同的设备之间传输数据; 通过buffer可以减少进程间通信需要等待的时间, 当存储速度快的设备与存储速度慢的设备进行通信时, 存储慢的数据先把数据存放到buffer, 达到一定程度存储快的设备再读取buffer的数据, 在此期间存储快的设备CPU可以干其他的事情。

Buffer: 一般是在写入磁盘的, 例如: 某个进程要求多个字段被读入, 当所有要求的字段被读入之前已经读入的字段会先放到buffer中。

2.5.3 HeapByteBuffer和DirectByteBuffer

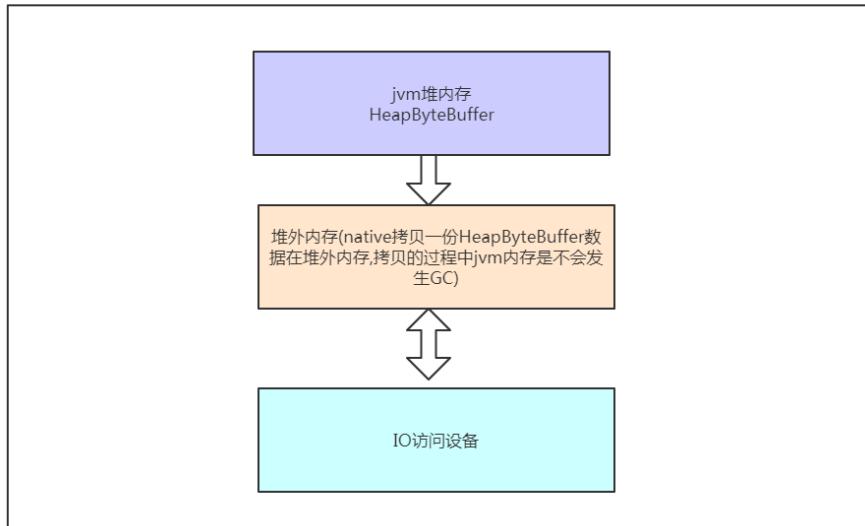
HeapByteBuffer, 是在jvm堆上面一个buffer, 底层的本质是一个数组, 用类封装维护了很多的索引 (limit/position/capacity等) 。

DirectByteBuffer, 底层的数据是维护在操作系统的内存中, 而不是jvm里, DirectByteBuffer里维护了一个引用address指向数据, 进而操作数据。

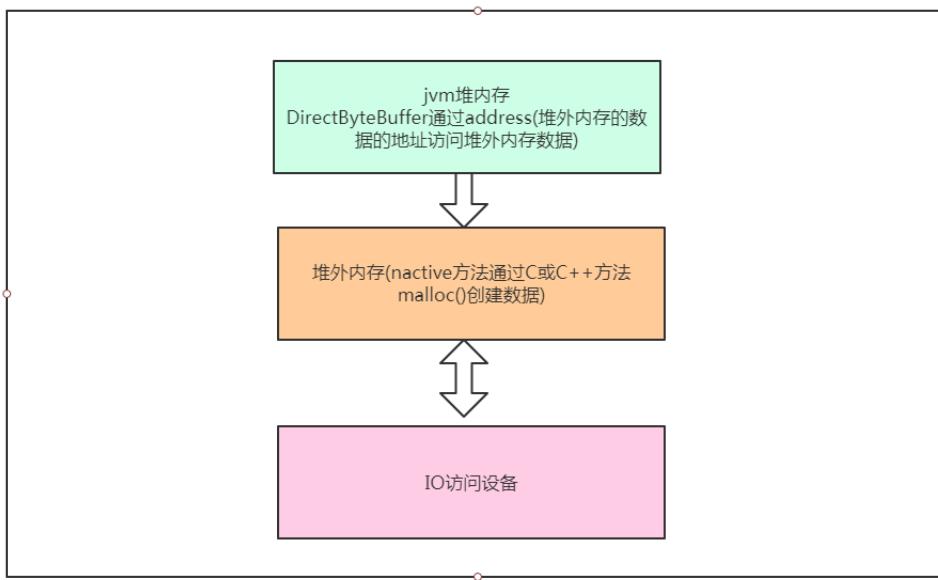
HeapByteBuffer优点: 内容维护在jvm里, 把内容写进buffer里速度快; 更容易回收

DirectByteBuffer优点：跟外设（IO设备）打交道时会快很多，因为外设读取jvm堆里的数据时，不是直接读取的，而是把jvm里的数据读到一个内存块里，再在这个块里读取的，如果使用DirectByteBuffer，则可以省去这一步，实现zero copy（零拷贝）

外设之所以要把jvm堆里的数据copy出来再操作，不是因为操作系统不能直接操作jvm内存，而是因为jvm在进行gc（垃圾回收）时，会对数据进行移动，一旦出现这种问题，外设就会出现数据错乱的情况。



所有的通过allocate方法创建的buffer都是HeapByteBuffer.



堆外内存实现零拷贝

1. 前者分配在JVM堆上（`ByteBuffer.allocate()`），后者分配在操作系统物理内存上（`ByteBuffer.allocateDirect()`，JVM使用C库中的`malloc()`方法分配堆外内存）；
2. DirectByteBuffer可以减少JVM GC压力，当然，堆中依然保存对象引用，fullgc发生时也会回收直接内存，也可以通过`System.gc()`主动通知JVM回收，或者通过`Cleaner.clean()`主动清理。`Cleaner.create()`方法需要传入一个DirectByteBuffer对象和一个`Deallocator`（一个堆外内存回收线程）。GC发生时发现堆中的DirectByteBuffer对象没有强引用了，则调用`Deallocator.run()`方法回收直接内存，并释放堆中DirectByteBuffer的对象引用；
3. 底层I/O操作需要连续的内存（JVM堆内存容易发生GC和对象移动），所以在执行write操作时需要将HeapByteBuffer数据拷贝到一个临时的（操作系统用户态）内存空间中，会多一次额外拷贝。而DirectByteBuffer则可以省去这个拷贝动作，这是Java层面的“零拷贝”技术，在netty中广泛使用；

4. MappedByteBuffer底层使用了操作系统的mmap机制，FileChannel#map()方法就会返回MappedByteBuffer。DirectByteBuffer虽然实现了MappedByteBuffer，不过DirectByteBuffer默认并没有直接使用mmap机制。

2.5.4 缓冲IO和直接IO

2.5.4.1 缓存IO

缓存I/O又被称作标准I/O，大多数文件系统的默认I/O操作都是缓存I/O。在Linux的缓存I/O机制中，数据先从磁盘复制到内核空间的缓冲区，然后从内核空间缓冲区复制到应用程序的地址空间。

读操作：操作系统检查内核的缓冲区有没有需要的数据，如果已经缓存了，那么就直接从缓存中返回；否则从磁盘中读取，然后缓存在操作系统的缓存中。

写操作：将数据从用户空间复制到内核空间的缓存中。这时对用户程序来说写操作就已经完成，至于什么时候再写到磁盘中由操作系统决定，除非显示地调用了sync同步命令。

缓存I/O的优点：

1. 在一定程度上分离了内核空间和用户空间，保护系统本身的运行安全；
2. 可以减少读盘的次数，从而提高性能。

缓存I/O的缺点：

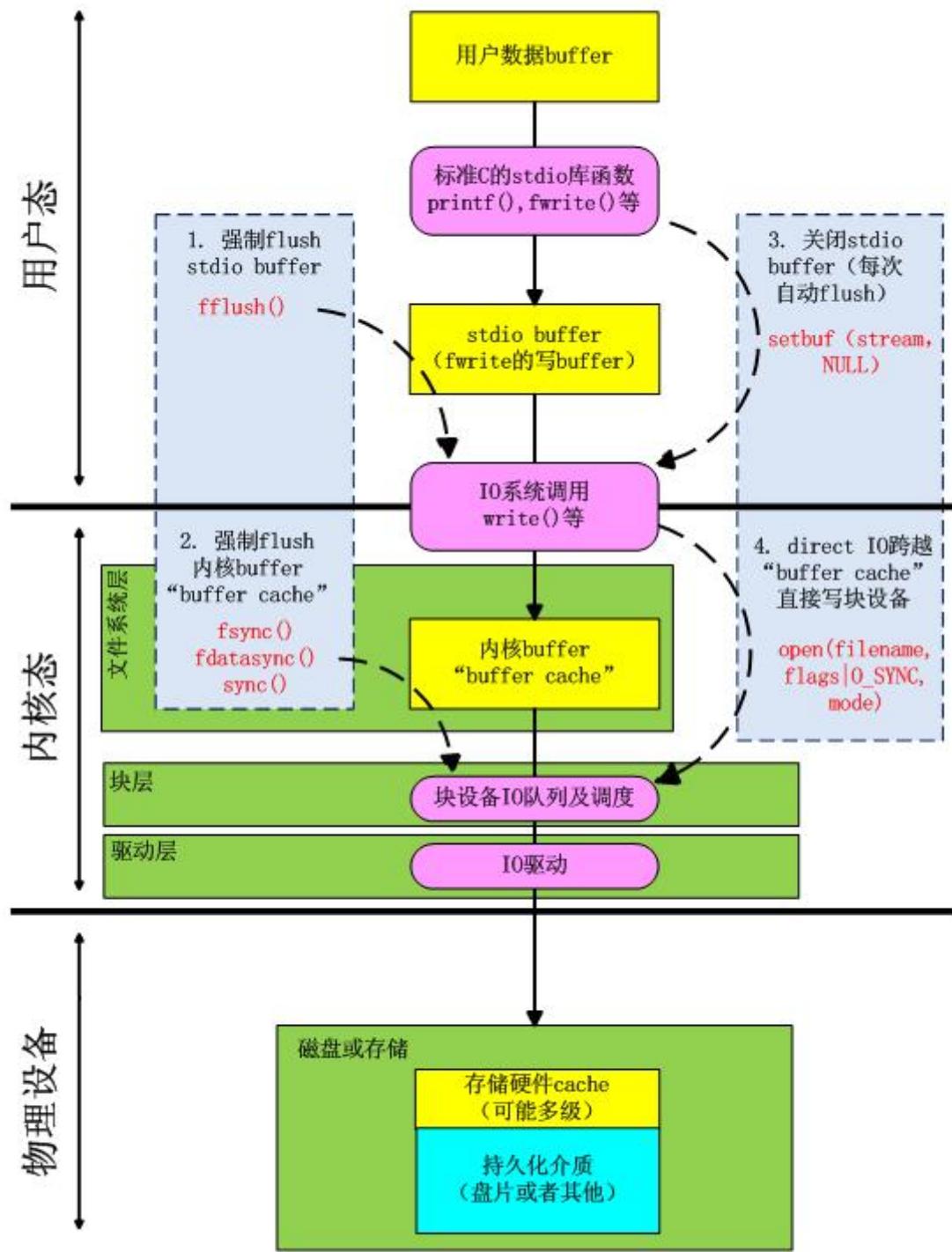
1. 在缓存 I/O 机制中，DMA 方式可以将数据直接从磁盘读到页缓存中，或者将数据从页缓存直接写回到磁盘上，而不能直接在应用程序地址空间和磁盘之间进行数据传输。数据在传输过程中就需要在**应用程序地址空间（用户空间）和缓存（内核空间）之间进行多次数据拷贝操作**，这些数据拷贝操作所带来的CPU以及内存开销是非常大的。

2.5.4.2 直接IO

直接IO就是应用程序直接访问磁盘数据，而不经过内核缓冲区，这样做的目的是减少一次从内核缓冲区到用户程序缓存的数据复制。比如说数据库管理系统这类应用，它们更倾向于选择它们自己的缓存机制，因为数据库管理系统往往比操作系统更了解数据库中存放的数据，数据库管理系统可以提供一种更加有效的缓存机制来提高数据库中数据的存取性能。

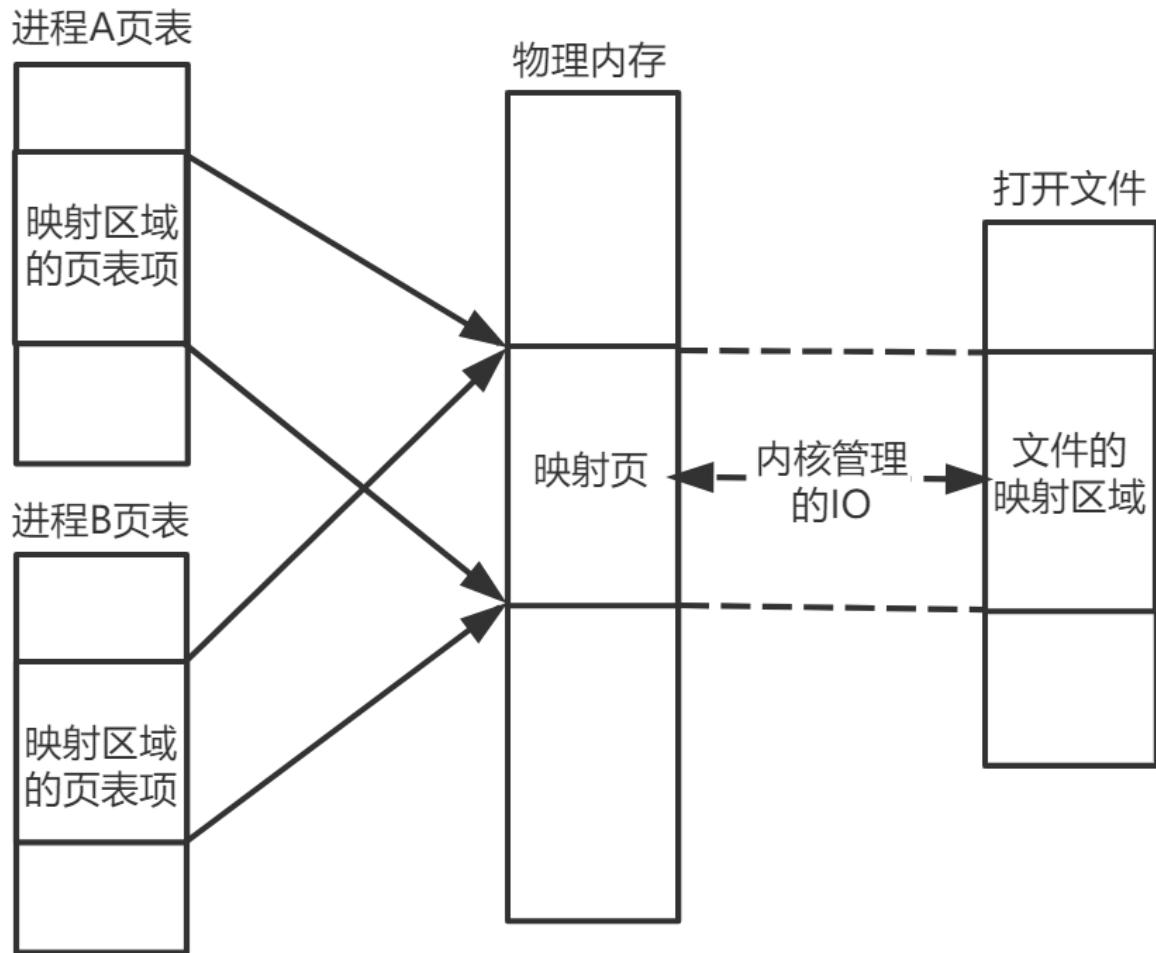
直接IO的缺点：**如果访问的数据不在应用程序缓存中，那么每次数据都会直接从磁盘加载**，这种直接加载会非常缓慢。通常直接IO与异步IO结合使用，会得到比较好的性能。

下图分析了写场景下的DirectIO和BufferIO：



2.5.5 内存映射文件 (Mmap)

在LINUX中我们可以使用mmap用来在进程虚拟内存地址空间中分配地址空间，创建和物理内存的映射关系。



映射关系可以分为两种

1. 文件映射 磁盘文件映射进程的虚拟地址空间，使用文件内容初始化物理内存。
2. 匿名映射 初始化全为0的内存空间。

而对于映射关系是否共享又分为

1. 私有映射(MAP_PRIVATE) 多进程间数据共享，修改不反应到磁盘实际文件，是一个copy-on-write (写时复制) 的映射方式。
2. 共享映射(MAP_SHARED) 多进程间数据共享，修改反应到磁盘实际文件中。

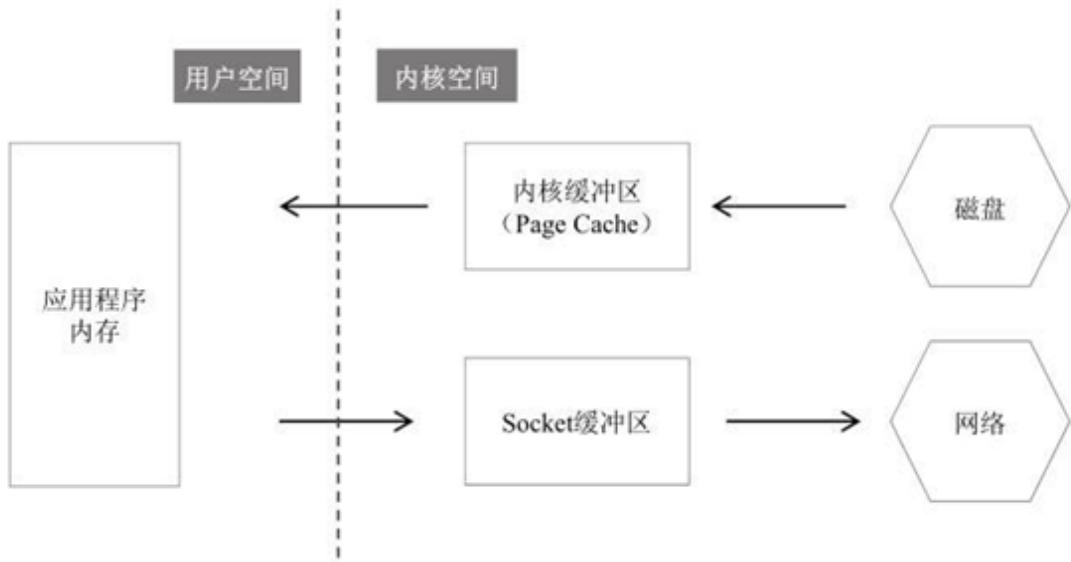
因此总结起来有4种组合

1. 私有文件映射 多个进程使用同样的物理内存页进行初始化，但是各个进程对内存文件的修改不会共享，也不会反应到物理文件中
2. 私有匿名映射 mmap会创建一个新的映射，各个进程不共享，这种使用主要用于分配内存 (malloc分配大内存会调用mmap)。例如开辟新进程时，会为每个进程分配虚拟的地址空间，这些虚拟地址映射的物理内存空间各个进程间读的时候共享，写的时候会copy-on-write。
3. 共享文件映射 多个进程通过虚拟内存技术共享同样的物理内存空间，对内存文件的修改会反应到实际物理文件中，他也是进程间通信(IPC)的一种机制。
4. 共享匿名映射 这种机制在进行fork的时候不会采用写时复制，父子进程完全共享同样的物理内存页，这也就实现了父子进程通信(IPC)。

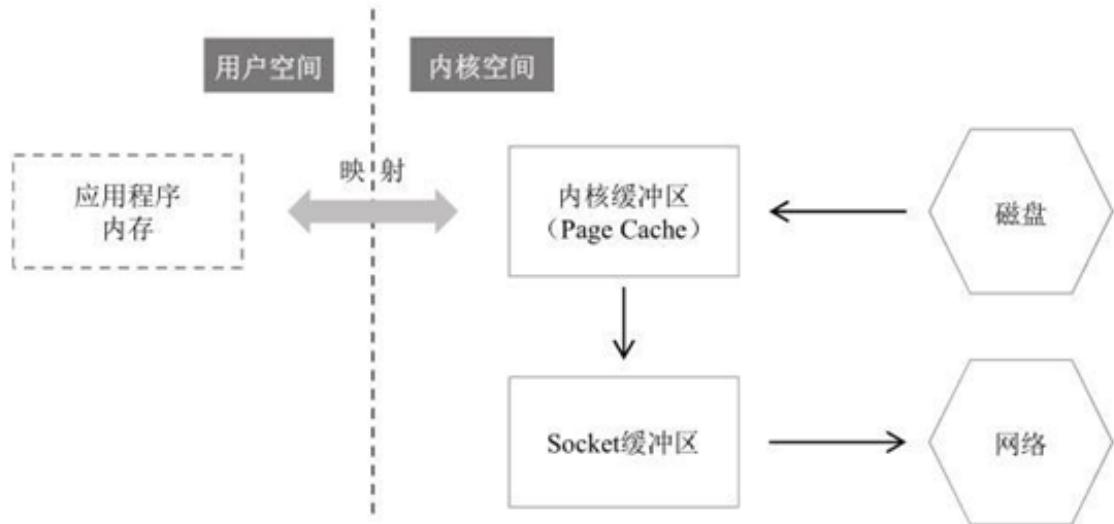
mmap只是在虚拟内存分配了地址空间，只有在第一次访问虚拟内存的时候才分配物理内存。

在mmap之后，并没有在将文件内容加载到物理页上，只在在虚拟内存中分配了地址空间。当进程在访问这段地址时，通过查找页表，发现虚拟内存对应的页没有在物理内存中缓存，则产生“缺页”，由内核的缺页异常处理程序处理，将文件对应内容，以页为单位(4096)加载到物理内存，注意是只加载缺页，但也会受操作系统一些调度策略影响，加载的比所需的多。

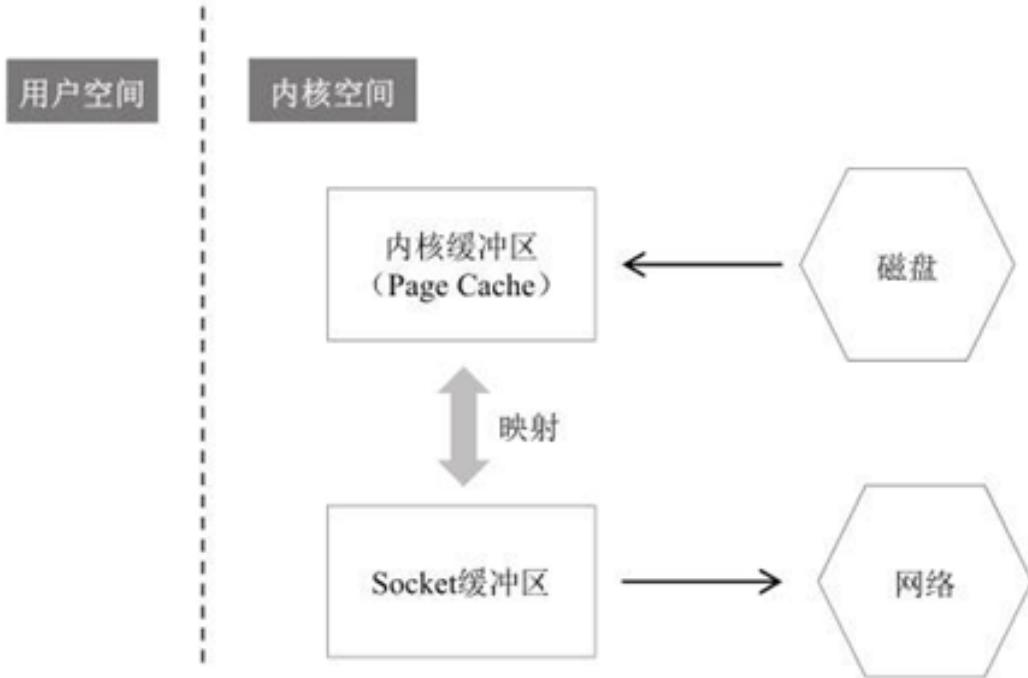
2.5.6 直接内存读取并发送文件的过程



2.5.7 Mmap读取并发送文件的过程



2.5.8 Sendfile零拷贝读取并发送文件的过程



零拷贝 (zero copy) 小结

1. 虽然叫零拷贝，实际上sendfile有2次数据拷贝的。第1次是从磁盘拷贝到内核缓冲区，第二次是从内核缓冲区拷贝到网卡（协议引擎）。如果网卡支持 SG-DMA (The Scatter-Gather Direct Memory Access) 技术，就无需从PageCache拷贝至 Socket 缓冲区；
- 2.之所以叫零拷贝，是从内存角度来看的，数据在内存中没有发生过拷贝，只是在内存和I/O设备之间传输。很多时候我们认为sendfile才是零拷贝，mmap严格来说不算；
3. Linux中的API为sendfile、mmap，Java中的API为FileChannel.transferTo()、FileChannel.map()等；
4. Netty、Kafka(sendfile)、Rocketmq (mmap) 、Nginx等高性能中间件中，都有大量利用操作系统零拷贝特性。

2.6 同步复制和异步复制

如果一个Broker组有Master和Slave，消息需要从Master复制到Slave 上，有同步和异步两种复制方式。

1) 同步复制

同步复制方式是等Master和Slave均写成功后才反馈给客户端写成功状态；

在同步复制方式下，如果Master出故障，Slave上有全部的备份数据，容易恢复，但是同步复制会增大数据写入延迟，降低系统吞吐量。

2) 异步复制

异步复制方式是只要Master写成功即可反馈给客户端写成功状态。

在异步复制方式下，系统拥有较低的延迟和较高的吞吐量，但是如果Master出了故障，有些数据因为没有被写入Slave，有可能会丢失；

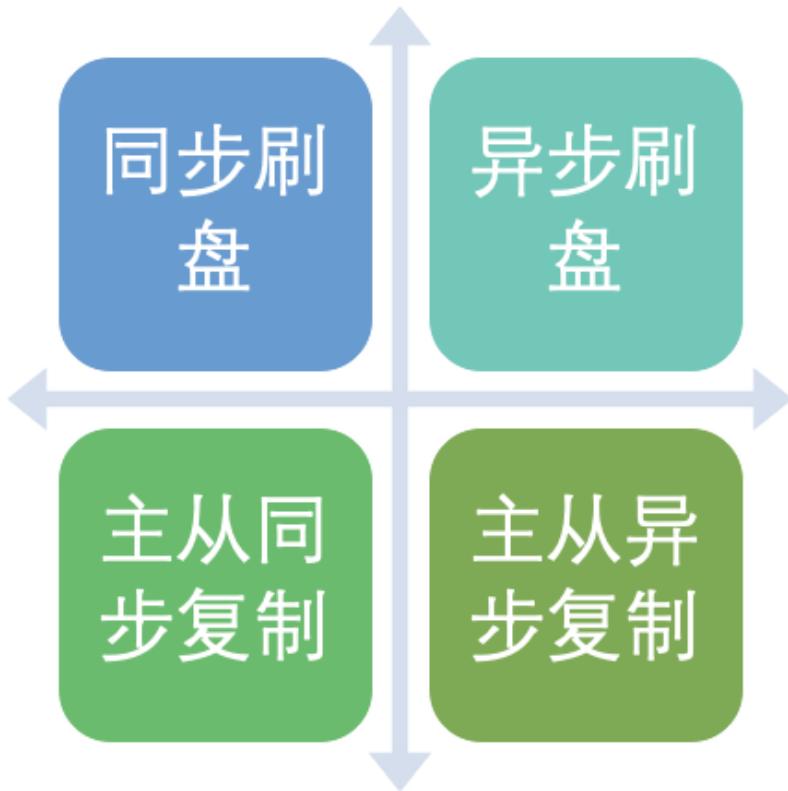
3) 配置

同步复制和异步复制是通过Broker配置文件里的brokerRole参数进行设置的，这个参数可以被设置成ASYNC_MASTER、 SYNC_MASTER、 SLAVE三个值中的一个。

/opt/rocket/conf/broker.conf 文件：Broker的配置文件

参数名	默认值	说明
listenPort	10911	接受客户端连接的监听端口
namesrvAddr	null	nameServer 地址
brokerIP1	网卡的 InetAddress	当前 broker 监听的 IP
brokerIP2	跟 brokerIP1 一样	存在主从 broker 时，如果在 broker 主节点上配置了 brokerIP2 属性， broker 从节点会连接主节点配置的 brokerIP2 进行同步
brokerName	null	broker 的名称
brokerClusterName	DefaultCluster	本 broker 所属的 Cluster 名称
brokerId	0	broker id, 0 表示 master, 其他的正整数表示 slave
storePathCommitLog	\$HOME/store/commitlog/	存储 commit log 的路径
storePathConsumerQueue	\$HOME/store/consumequeue/	存储 consume queue 的路径
mapedFileSizeCommitLog	1024 * 1024 * 1024(1G)	commit log 的映射文件大小
deleteWhen	04	在每天的什么时间删除已经超过文件保留时间的 commit log
fileReserverdTime	72	以小时计算的文件保留时间
brokerRole	ASYNC_MASTER	SYNC_MASTER或者ASYNC_MASTER或者SLAVE SYNC_MASTER表示当前broker是一个同步复制的Master ASYNC_MASTER表示当前broker是一个异步复制的Master SLAVE表示当前broker是一个Slave。
flushDiskType	ASYNC_FLUSH	SYNC_FLUSH/ASYNC_FLUSH SYNC_FLUSH 模式下的 broker 保证在收到确认生产者之前将消息刷盘。 ASYNC_FLUSH 模式下的 broker 则利用刷盘一组消息的模式，可以取得更好的性能。

4) 总结



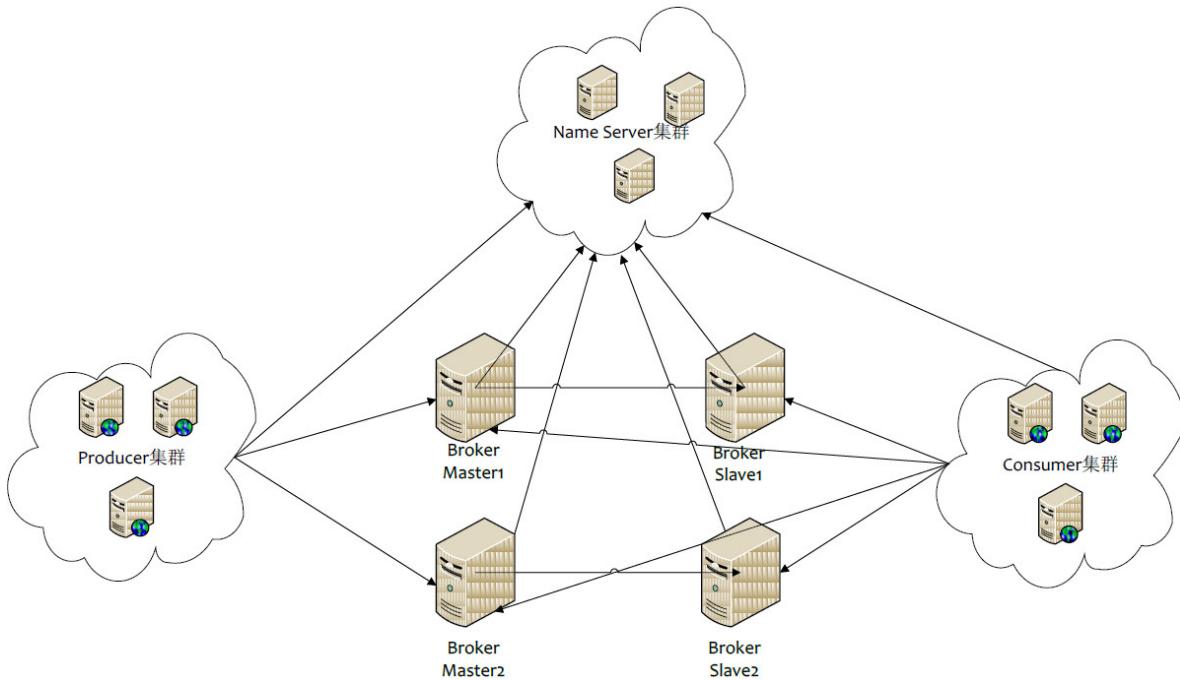
实际应用中要结合业务场景，合理设置刷盘方式和主从复制方式，尤其是SYNC_FLUSH方式，由于频繁地触发磁盘写动作，会明显降低性能。通常情况下，应该把Master和Slave配置成ASYNC_FLUSH的刷盘方式，主从之间配置成SYNC_MASTER的复制方式，这样即使有一台机器出故障，仍然能保证数据不丢，是个不错的选择。

2.7 高可用机制

RocketMQ分布式集群是通过Master和Slave的配合达到高可用性的。

Master和Slave的区别：

1. 在Broker的配置文件中，参数brokerId的值为0表明这个Broker是Master，
2. 大于0表明这个Broker是Slave，
3. brokerRole参数也说明这个Broker是Master还是Slave。
(SYNC_MASTER/ASYNC_MASTER/SALVE)
4. Master角色的Broker支持读和写，Slave角色的Broker仅支持读。
5. Consumer可以连接Master角色的Broker，也可以连接Slave角色的Broker来读取消息。



2.7.1 消息消费高可用

在Consumer的配置文件中，并不需要设置是从Master读还是从Slave 读，当Master不可用或者繁忙的时候，Consumer会被**自动切换**到从Slave 读。

有了自动切换Consumer这种机制，当一个Master角色的机器出现故障后，Consumer仍然可以从Slave读取消息，不影响Consumer程序。

这就达到了消费端的高可用性。

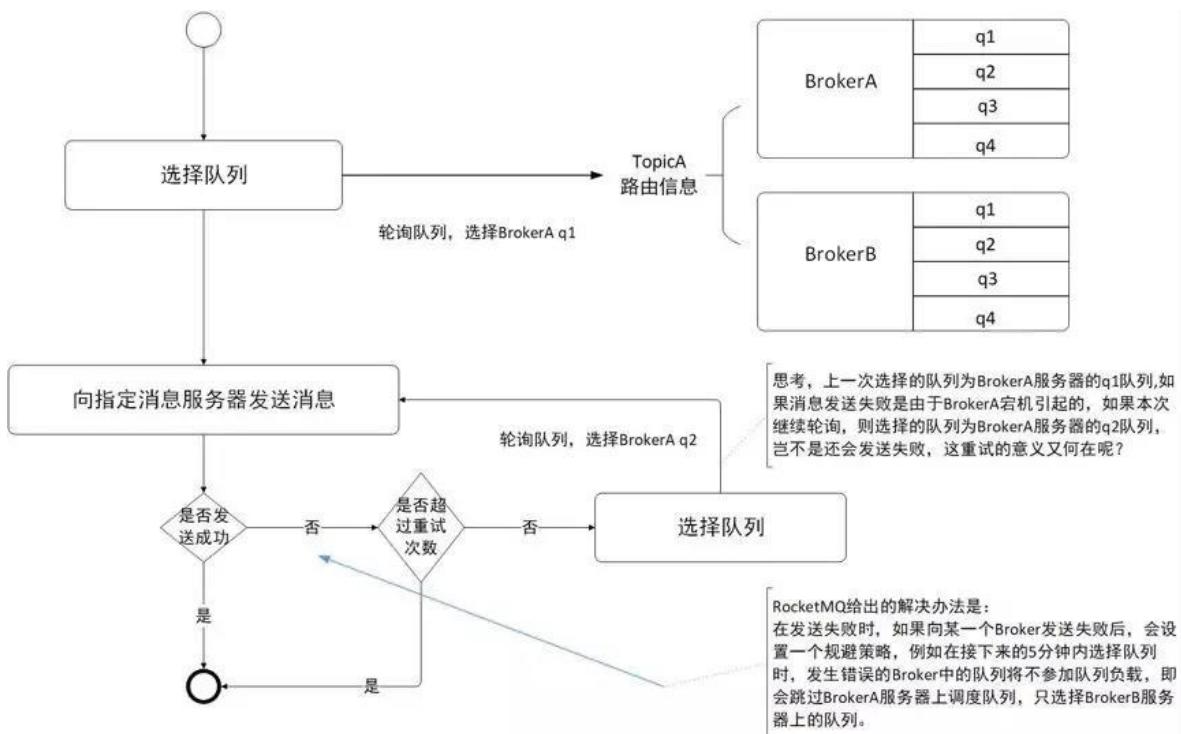
2.7.2 消息发送高可用

如何达到发送端的高可用性呢？

在创建Topic的时候，把Topic的多个Message Queue创建在多个Broker组上（相同Broker名称，不同brokerId的机器组成一个Broker组），这样既可以在性能方面具有扩展性，也可以降低主节点故障对整体上带来的影响，而且当一个Broker组的Master不可用后，其他组的Master仍然可用，Producer仍然可以发送消息的。

RocketMQ目前还不支持把Slave自动转成Master，如果机器资源不足，需要把Slave转成Master。

1. 手动停止Slave角色的Broker。
2. 更改配置文件。
3. 用新的配置文件启动Broker。



这种早期方式在大多数场景下都可以很好的工作，但也面临一些问题。

比如，在需要保证消息严格顺序的场景下，由于在主题层面无法保证严格顺序，所以必须指定队列来发送消息，对于任何一个队列，它一定是落在一组特定的主从节点上，如果这个主节点宕机，其他的主节点是无法替代这个主节点的，否则就无法保证严格顺序。

在这种复制模式下，严格顺序和高可用只能选择一个。

RocketMQ 在 2018 年底迎来了一次重大的更新，引入 Dledger，增加了一种全新的复制方式。

RocketMQ 引入 Dledger，使用新的复制方式，可以很好地解决这个问题。

Dledger 在写入消息的时候，要求至少消息复制到半数以上的节点之后，才给客户端返回写入成功，并且它是支持通过选举来动态切换主节点的。

举例：

假如有3个节点，当主节点宕机的时候，2 个从节点会通过投票选出一个新的主节点来继续提供服务，相比主从的复制模式，解决了可用性的问题。

由于消息要至少复制到 2 个节点上才会返回写入成功，即使主节点宕机了，也至少有一个节点上的消息是和主节点一样的。

Dledger在选举时，总会把数据和主节点一样的从节点选为新的主节点，这样就保证了数据的一致性，既不会丢消息，还可以保证严格顺序。

存在问题：

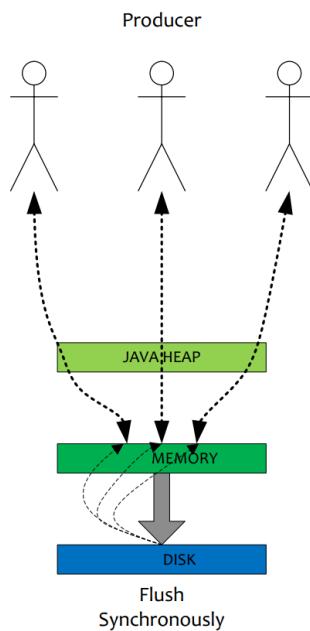
当然，Dledger的复制方式也不是完美的，依然存在一些不足：

1. 比如，选举过程中不能提供服务。
2. 最少需要 3 个节点才能保证数据一致性，3 节点时，只能保证 1 个节点宕机时可用，如果 2 个节点同时宕机，即使还有 1 个节点存活也无法提供服务，资源的利用率比较低。
3. 另外，由于至少要复制到半数以上的节点才返回写入成功，性能上也不如主从异步复制的方式快。

2.8 刷盘机制

RocketMQ 的所有消息都是持久化的，先写入系统 PageCache，然后刷盘，可以保证内存与磁盘都有一份数据，访问时，直接从内存读取。消息在通过 Producer 写入 RocketMQ 的时候，有两种写磁盘方式，分布式同步刷盘和异步刷盘。

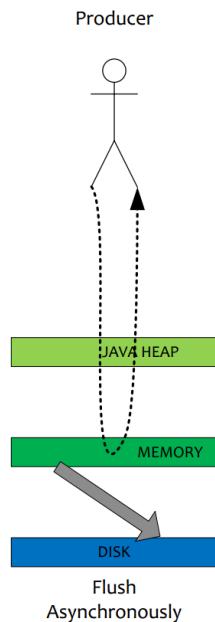
2.8.1 同步刷盘



同步刷盘与异步刷盘的唯一区别是异步刷盘写完 PageCache 直接返回，而同步刷盘需要等待刷盘完成才返回，同步刷盘流程如下：

- (1). 写入 PageCache 后，线程等待，通知刷盘线程刷盘。
- (2). 刷盘线程刷盘后，唤醒前端等待线程，可能是一批线程。
- (3). 前端等待线程向用户返回成功

2.8.2 异步刷盘



在有 RAID 卡，SAS 15000 转磁盘测试顺序写文件，速度可以达到 300M 每秒左右，而线上的网卡一般都为千兆网卡，写磁盘速度明显快于数据网络入口速度，那么是否可以做到写完内存就向用户返回，由后台线程刷盘呢？

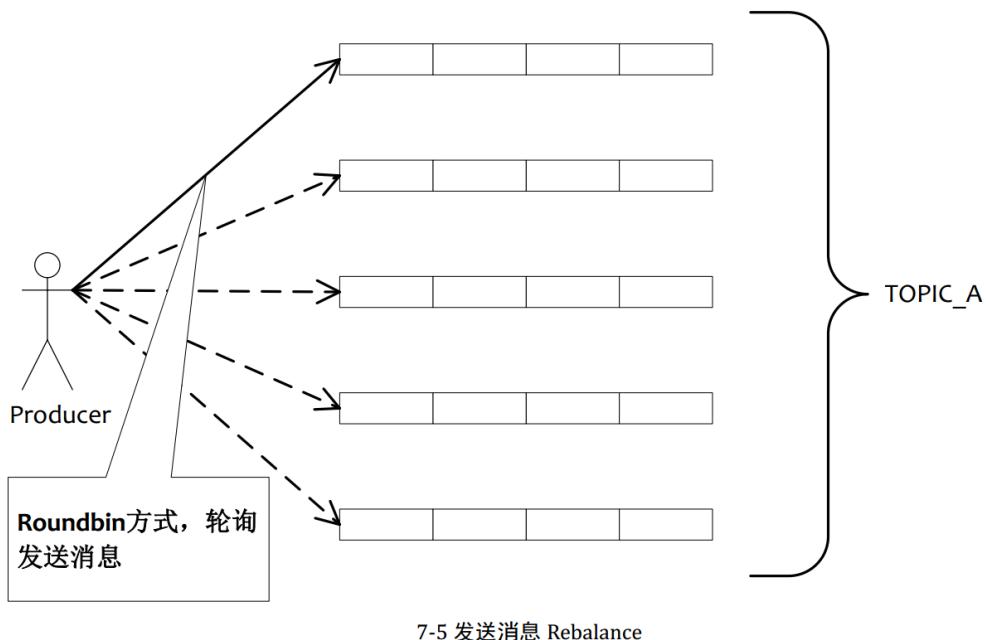
1. 由于磁盘速度大于网卡速度，那么刷盘的进度肯定可以跟上消息的写入速度。
2. 万一由于此时系统压力过大，可能堆积消息，除了写入 IO，还有读取 IO，万一出现磁盘读取落后情况，会不会导致系统内存溢出，答案是否定的，原因如下：
 - 写入消息到 PageCache 时，如果内存不足，则尝试丢弃干净的 PAGE，腾出内存供新消息使用，策略是 LRU 方式。
 - 如果干净页不足，此时写入 PageCache 会被阻塞，系统尝试刷盘部分数据，大约每次尝试 32 个 PAGE，来找出更多干净 PAGE。

综上，内存溢出的情况不会出现。

2.9 负载均衡

RocketMQ 中的负载均衡都在 Client 端完成，具体来说的话，主要可以分为 Producer 端发送消息时候的负载均衡和 Consumer 端订阅消息的负载均衡。

2.9.1 Producer 的负载均衡



7-5 发送消息 Rebalance

如图所示，5 个队列可以部署在一台机器上，也可以分别部署在 5 台不同的机器上，发送消息通过轮询队列的方式发送，每个队列接收平均的消息量。通过增加机器，可以水平扩展队列容量。另外也可以自定义方式选择发往哪个队列。

```
1 # 创建主题  
2 [root@node1 ~]# mqadmin updateTopic -n localhost:9876 -t tp_demo_02 -w 6 -b  
localhost:10911
```

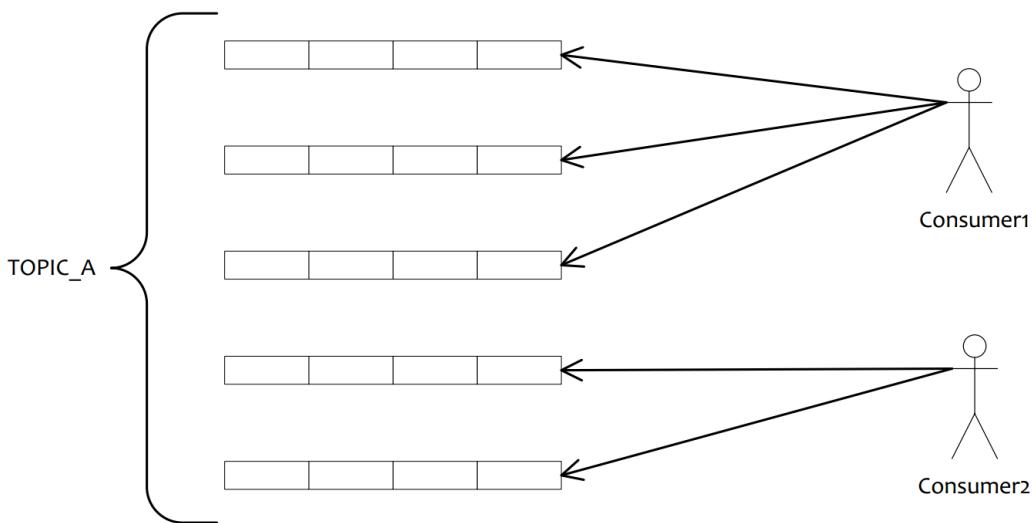
```
1 DefaultMQProducer producer = new DefaultMQProducer("producer_grp_02");  
2  
3 producer.setNamesrvAddr("node1:9876");
```

```

4 producer.start();
5
6
7 Message message = new Message();
8 message.setTopic("tp_demo_02");
9 message.setBody("hello lagou".getBytes());
10 // 指定MQ
11 SendResult result = producer.send(message,
12         new MessageQueue("tp_demo_06", "node1", 5),
13         1_000
14 );
15
16 System.out.println(result.getSendStatus());
17
18 producer.shutdown();

```

2.9.2 Consumer的负载均衡



如图所示，如果有 5 个队列，2 个 consumer，那么第一个 Consumer 消费 3 个队列，第二个 consumer 消费 2 个队列。这样即可达到平均消费的目的，可以水平扩展 Consumer 来提高消费能力。但是 Consumer 数量要小于等于队列数量，如果 Consumer 超过队列数量，那么多余的 Consumer 将不能消费消息。

在RocketMQ中，Consumer端的两种消费模式（Push/Pull）底层都是基于拉模式来获取消息的，而在Push模式只是对pull模式的一种封装，其本质实现为消息拉取线程在从服务器拉取到一批消息后，然后提交到消息消费线程池后，又“马不停蹄”的继续向服务器再次尝试拉取消息。

如果未拉取到消息，则延迟一下又继续拉取。

在两种基于拉模式的消费方式（Push/Pull）中，均需要Consumer端在知道从Broker端的哪一个消息队列中去获取消息。

因此，有必要在Consumer端来做负载均衡，即Broker端中多个MessageQueue分配给同一个ConsumerGroup中的哪些Consumer消费。

要做负载均衡，必须知道一些全局信息，也就是一个ConsumerGroup里到底有多少个Consumer。

知道了全局信息，才可以根据某种算法来分配，比如简单地平均分到各个Consumer。

在RocketMQ中，负载均衡或者消息分配是在Consumer端代码中完成的，**Consumer**从Broker处获得全局信息，然后自己做负载均衡，只处理分给自己的那部分消息。

Pull Consumer可以看到所有的Message Queue，而且从哪个Message Queue读取消息，读消息时的Offset都由使用者控制，使用者可以实现任何特殊方式的负载均衡。

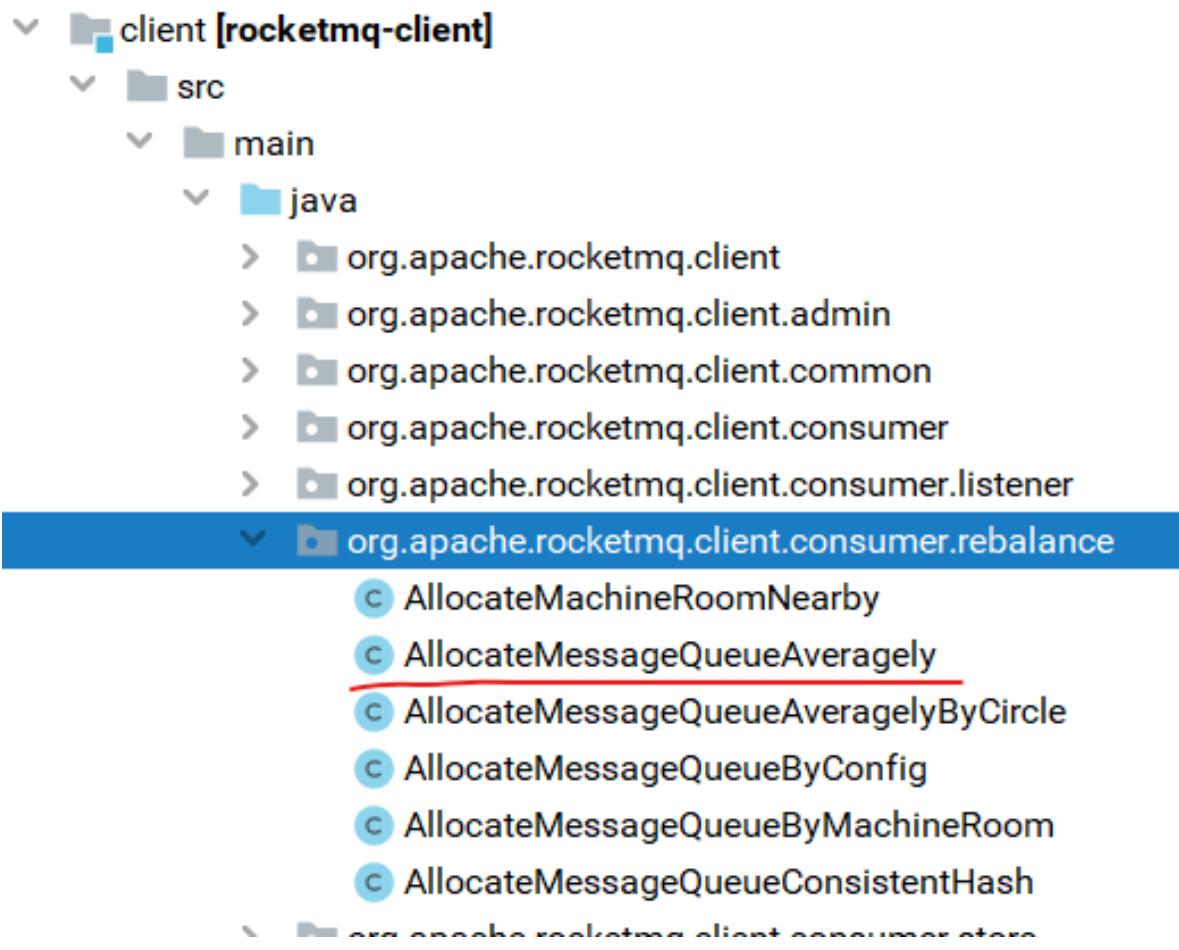
DefaultMQPullConsumer有两个辅助方法可以帮助实现负载均衡，一个是registerMessageQueueListener函数，一个是MQPullConsumerScheduleService（使用这个Class类似使用DefaultMQPushConsumer，但是它把Pull消息的主动性留给了使用者）

```
1 public class MyConsumer {
2     public static void main(String[] args) throws MQClientException,
3     RemotingException, InterruptedException, MQBrokerException {
4
5         DefaultMQPullConsumer consumer = new
6 DefaultMQPullConsumer("consumer_pull_grp_01");
7
8         consumer.setNamesrvAddr("node1:9876");
9
10        consumer.start();
11
12        Set<MessageQueue> messageQueues =
13 consumer.fetchSubscribeMessageQueues("tp_demo_01");
14        for (MessageQueue messageQueue : messageQueues) {
15            // 指定从哪个MQ拉取数据
16            PullResult result = consumer.pull(messageQueue, "*", 0L, 10);
17
18            List<MessageExt> msgFoundList = result.getMsgFoundList();
19            for (MessageExt messageExt : msgFoundList) {
20                System.out.println(messageExt);
21            }
22        }
23        consumer.shutdown();
24    }
}
```

DefaultMQPushConsumer的负载均衡过程不需要使用者操心，客户端程序会自动处理，每个DefaultMQPushConsumer启动后，会马上会触发一个**doRebalance动作**；而且在同一个ConsumerGroup里加入新的DefaultMQPush-Consumer时，各个Consumer都会被触发**doRebalance**动作。

负载均衡的**分配粒度只到Message Queue**，把Topic下的所有Message Queue分配到不同的Consumer中

如下图所示，具体的负载均衡算法有几种，**默认用的是AllocateMessageQueueAveragely**。



我们可以设置负载均衡的算法：

```
1 DefaultMQPushConsumer consumer = new
DefaultMQPushConsumer("consumer_push_grp_01");
2
3 consumer.setNamesrvAddr("node1:9876");
4 // 设置负载均衡算法
5 consumer.setAllocateMessageQueueStrategy(new
AllocateMessageQueueAveragely());
6
7 consumer.setMessageListener(new MessageListenerConcurrently() {
8     @Override
9     public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs,
10        ConsumeConcurrentlyContext context) {
11         // todo 处理接收到的消息
12         return null;
13     }
14 });
15 consumer.start();
```

以AllocateMessageQueueAveragely策略为例，如果创建Topic的时候，把Message Queue数设为3，当Consumer数量为2的时候，有一个Consumer需要处理Topic三分之二的消息，另一个处理三分之一的消息；当Consumer数量为4的时候，有一个Consumer无法收到消息，其他3个Consumer各处理Topic三分之一的消息。

可见Message Queue数量设置过小不利于做负载均衡，通常情况下，应把一个Topic的Message Queue数设置为16。

1、Consumer端的心跳包发送

在Consumer启动后，它就会通过**定时任务**不断地向RocketMQ集群中的所有Broker实例发送心跳包（其中包含了消息消费分组名称、订阅关系集合、消息通信模式和客户端id的值等信息）。Broker端在收到Consumer的心跳消息后，会将它维护在ConsumerManager的本地缓存变量—`consumerTable`，同时并将封装后的客户端网络通道信息保存在本地缓存变量—`channelInfoTable`中，为之后做Consumer端的负载均衡提供可以依据的元数据信息。

2、Consumer端实现负载均衡的核心类—RebalanceImpl

在Consumer实例的启动流程中启动MQClientInstance实例的部分，会完成负载均衡服务线程—`RebalanceService`的启动（每隔20s执行一次）。

The screenshot shows four tabs of Java code in a code editor:

- `DefaultMQPullConsumer.java`: Line 148 shows `this.rebalanceService = new RebalanceService(mqClientFactory: this);`.
- `MQClientInstance.java`: Lines 240-244 show `// Start pull service
this.pullMessageService.start();
// Start rebalance service
// 启动再平衡服务
this.rebalanceService.start();`.
- `ServiceThread.java`: Lines 45-51 show `public void start() {
 Log.info(var1: "Try to start service thread:{} started:{} lastThread:{}
 if (!started.compareAndSet(expectedValue: false, newValue: true)) {
 return;
 }
 stopped = false;
 this.thread = new Thread(target: this, getServiceName());
 this.thread.setDaemon(isDaemon);
 this.thread.start();`.
- `RebalanceService.java`: This tab is currently not visible.

```
23
24     public class RebalanceService extends ServiceThread {
25         private static long waitInterval =
26             Long.parseLong(System.getProperty(
27                 "rocketmq.client.rebalance.waitInterval", def: "20000"));
28         private final InternalLogger log = ClientLogger.getLog();
29         private final MQClientInstance mqClientFactory;
30
31         public RebalanceService(MQClientInstance mqClientFactory) { this.mqClientFactory = mqClientFactory; }
32
33         @Override
34         public void run() {
35             log.info(var1: this.getServiceName() + " service started");
36
37             while (!this.isStopped()) {
38                 this.waitForRunning(waitInterval);
39                 this.mqClientFactory.doRebalance();
40             }
41
42             log.info(var1: this.getServiceName() + " service end");
43         }
44
45         @Override
46         public String getServiceName() {
47             return RebalanceService.class.getSimpleName();
48         }
49     }
50
51 }
```

通过查看源码可以发现，RebalanceService线程的run()方法最终调用的是RebalanceImpl类的rebalanceByTopic()方法，该方法是实现Consumer端负载均衡的核心。

```
978     public void doRebalance() {
979         for (Map.Entry<String, MQConsumerInner> entry : this.consumerTable.entrySet()) {
980             MQConsumerInner impl = entry.getValue();
981             if (impl != null) {
982                 try {
983                     impl.doRebalance();
984                 } catch (Throwable e) {
985                     log.error("doRebalance error", e);
986                 }
987             }
988         }
989     }
```

```
1004
1005
1006     @Override
1007     public void doRebalance() {
1008         if (!this.pause) {
1009             this.rebalanceImpl.doRebalance(this.isConsumeOrderly());
1010         }
1011     }
1012 }
```

```
219     public void doRebalance(final boolean isOrder) {
220         Map<String, SubscriptionData> subTable = this.getSubscriptionInner();
221         if (subTable != null) {
222             for (final Map.Entry<String, SubscriptionData> entry : subTable.entrySet()) {
223                 final String topic = entry.getKey();
224                 try {
225                     this.rebalanceByTopic(topic, isOrder);
226                 } catch (Throwable e) {
227                     if (!topic.startsWith(MixAll.RETRY_GROUP_TOPIC_PREFIX)) {
228                         Log.warn("rebalanceByTopic Exception", e);
229                     }
230                 }
231             }
232         }
233         this.truncateMessageQueueNotMyTopic();
234     }
235 }
```

这里，rebalanceByTopic()方法根据消费者通信类型为“广播模式”还是“集群模式”做不同的逻辑处理。

对于集群模式：

```
1 case CLUSTERING: {
2     Set<MessageQueue> mqSet = this.topicSubscribeInfoTable.get(topic);
3     List<String> cidAll = this.mQClientFactory.findConsumerIdList(topic,
4 consumerGroup);
5     if (null == mqSet) {
6         if (!topic.startsWith(MixAll.RETRY_GROUP_TOPIC_PREFIX)) {
7             log.warn("doRebalance, {}, but the topic[{}] not exist.", consumerGroup, topic);
8         }
9     }
10    if (null == cidAll) {
11        log.warn("doRebalance, {} {}, get consumer id list failed", consumerGroup, topic);
12    }
13
14    if (mqSet != null && cidAll != null) {
15        List<MessageQueue> mqAll = new ArrayList<MessageQueue>();
16        mqAll.addAll(mqSet);
17
18        // 对MQ进行排序
19        Collections.sort(mqAll);
20        // 对消费者ID进行排序
21        Collections.sort(cidAll);
22
23        AllocateMessageQueueStrategy strategy =
24        this.allocateMessageQueueStrategy;
25
26        List<MessageQueue> allocateResult = null;
27        try {
28            // 计算当前消费者应该分配的MQ集合
29            allocateResult = strategy.allocate(
30                // 当前消费者所属的消费组
31                this.consumerGroup,
32                // 当前消费者ID
33            );
34        } catch (Exception e) {
35            log.error("doRebalance error, consumerGroup:{}, topic:{}, error:{}", consumerGroup, topic, e);
36        }
37    }
38}
```

```

32         this.mQClientFactory.getClientId(),
33         // MQ集合
34         mqAll,
35         // 消费组中消费者ID集合
36         cidAll);
37     } catch (Throwable e) {
38         log.error("AllocateMessageQueueStrategy.allocate Exception.",
39             allocateMessageQueueStrategyName={}, strategy.getName(),
40             e);
41     }
42     return;
43
44     Set<MessageQueue> allocateResultSet = new HashSet<MessageQueue>();
45     if (allocateResult != null) {
46         allocateResultSet.addAll(allocateResult);
47     }
48
49     boolean changed = this.updateProcessQueueTableInRebalance(topic,
50     allocateResultSet, isorder);
51     if (changed) {
52         log.info(
53             "rebalanced result changed.
54             allocateMessageQueueStrategyName={}, group={}, topic={}, clientId={},
55             mqAllSize={}, cidAllSize={}, rebalanceResultsSize={}, rebalanceResultSet=
56             {}",
57             strategy.getName(), consumerGroup, topic,
58             this.mQClientFactory.getClientId(), mqSet.size(), cidAll.size(),
59             allocateResultSet.size(), allocateResultSet);
60         this.messageQueueChanged(topic, mqSet, allocateResultSet);
61     }
62 }
63 break;
64 }
```

默认的负载均衡算法：

```

95     private static final long BROKER_SUSPEND_MAX_TIME_MILLIS = 1000 * 15;
96     private static final long CONSUMER_TIMEOUT_MILLIS_WHEN_SUSPEND = 1000 * 30;
97     private final InternalLogger log = ClientLogger.getLog();
98     private final DefaultMQPushConsumer defaultMQPushConsumer;
99     private final RebalanceImpl rebalanceImpl = new RebalancePushImpl( defaultMQPushConsumerImpl: this);
100    private final ArrayList<FilterMessageHook> filterMessageHookList = new ArrayList<>();

38
39     public RebalancePushImpl(DefaultMQPushConsumerImpl defaultMQPushConsumerImpl) {
40         this( consumerGroup: null, messageModel: null, allocateMessageQueueStrategy: null, mQClientFactory: null, defaultMQPushConsumerImpl );
41     }

42
43     public RebalancePushImpl(String consumerGroup, MessageModel messageModel,
44         AllocateMessageQueueStrategy allocateMessageQueueStrategy,
45         MQClientInstance mQClientFactory, DefaultMQPushConsumerImpl defaultMQPushConsumerImpl) {
46         super(consumerGroup, messageModel, allocateMessageQueueStrategy, mQClientFactory);
47         this.defaultMQPushConsumerImpl = defaultMQPushConsumerImpl;
48     }

```

DefaultMQPushConsumerImpl.java

```

56     protected MQClientInstance mQClientFactory;
57
58     public RebalanceImpl(String consumerGroup, MessageModel messageModel,
59                           AllocateMessageQueueStrategy allocateMessageQueueStrategy,
60                           MQClientInstance mQClientFactory) {
61         this.consumerGroup = consumerGroup;
62         this.messageModel = messageModel;
63         this.allocateMessageQueueStrategy = allocateMessageQueueStrategy;
64         this.mQClientFactory = mQClientFactory;
65     }

```

DefaultMQPushConsumerImpl.java

```

567     public synchronized void start() throws MQClientException {
568         switch (this.serviceState) {
569             case CREATE_JUST:
570                 log.info("the consumer [{}], start beginning. messageModel={}, isUnitMode={}", this.defaultMQPushConsumer.getCorrelationId(), this.defaultMQPushConsumer.getMessageModel(), this.defaultMQPushConsumer.isUnitMode());
571                 this.serviceState = ServiceState.START_FAILED;
572
573                 this.checkConfig();
574
575                 this.copySubscription();
576
577                 if (this.defaultMQPushConsumer.getMessageModel() == MessageModel.CLUSTERING) {
578                     this.defaultMQPushConsumer.changeInstanceIdToPID();
579                 }
580
581
582                 this.mQClientFactory = MQClientManager.getInstance().getAndCreateMQClientInstance(this.defaultMQPushConsumer, this);
583
584                 this.rebalanceImpl.setConsumerGroup(this.defaultMQPushConsumer.getConsumerGroup());
585                 this.rebalanceImpl.setMessageModel(this.defaultMQPushConsumer.getMessageModel());
586                 this.rebalanceImpl.setAllocateMessageQueueStrategy(this.defaultMQPushConsumer.getAllocateMessageQueueStrategy());
587                 this.rebalanceImpl.setMQClientFactory(this.mQClientFactory);
588

```

DefaultMQPushConsumer.java

```

268
269     /**
270      * Constructor specifying consumer group.
271      *
272      * @param consumerGroup Consumer group.
273      */
274     public DefaultMQPushConsumer(final String consumerGroup) {
275         this(namespace: null, consumerGroup, rpcHook: null, new AllocateMessageQueueAveragely());
276     }
277

```

AllocateMessageQueueAveragely是默认的MQ分配对象。

算法：

AllocateMessageQueueAveragely.java

```

32     /**
33      * 计算当前消费者应该消费哪些MQ的消息。
34      *
35      * @param consumerGroup 当前消费组
36      * @param currentCID 当前消费者ID
37      * @param mqAll 当前主题包含的MQ集合
38      * @param cidAll 当前消费组中包含的消费者ID
39      *
40      * @return 当前消费者应该消费的MQ集合
41      */
42     @Override
43     public List<MessageQueue> allocate(String consumerGroup, String currentCID, List<MessageQueue> mqAll,
44                                         List<String> cidAll) {
45         if (currentCID == null || currentCID.length() < 1) {

```

```

1 // 获取当前消费者在cidAll集合中的下标
2 int index = cidAll.indexOf(currentCID);
3 // mqAll对cidAll大小取模
4 int mod = mqAll.size() % cidAll.size();
5 // 计算每个消费者应该分配到的mq数量
6 // 如果mq个数小于等于消费者个数，每个消费者最多分配一个mq
7 // 如果mq个数大于消费者个数，
8 int averagesize =
9     // 如果mq个数小于等于消费组中消费者个数
10    mqAll.size() <= cidAll.size() ?
11        // 平均数就是1
12        1
13        :
14        // 否则，看mod和index大小
15        (
16            mod > 0 && index < mod ?
17                // 如果余数大于0并且当前消费者下标小于余数，则当前消费者应该消费平
18                // 均数个mq+1
19                mqAll.size() / cidAll.size() + 1
20                :
21                // 如果余数大于0并且当前消费者下标大于等于余数，则当前消费者应该消
22                // 费平均数个mq
23                mqAll.size() / cidAll.size()
24        );
25 // 计算当前消费者消费mq的起始位置
26 int startIndex = (mod > 0 && index < mod)
27     ?
28         index * averagesize
29         :
30         index * averagesize + mod;
31 // 计算当前消费者消费mq的跨度，即当前消费者分几个MQ
32 int range = Math.min(averagesize, mqAll.size() - startIndex);
33 // 分配MQ，放到result集合中返回
34 for (int i = 0; i < range; i++) {
35     result.add(mqAll.get((startIndex + i) % mqAll.size()));
36 }
37 return result;

```

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。

2.10 消息重试

2.10.1 顺序消息的重试

对于顺序消息，当消费者消费消息失败后，消息队列 RocketMQ 会自动不断进行消息重试（每次间隔时间为 1 秒），这时，应用会出现消息消费被阻塞的情况。因此，在使用顺序消息时，务必保证应用能够及时监控并处理消费失败的情况，避免阻塞现象的发生。

```

1 DefaultMQPushConsumer consumer = new
2 DefaultMQPushConsumer("consumer_grp_04_01");
3 consumer.setNamesrvAddr("node1:9876");
4

```

```
5 consumer.setConsumeMessageBatchMaxSize(1);
6 consumer.setConsumeThreadMin(1);
7 consumer.setConsumeThreadMax(1);
8
9 // 消息订阅
10 consumer.subscribe("tp_demo_04", "*");
11
12 // 并发消费
13 // consumer.setMessageListener(new MessageListenerConcurrently() {
14 //     @Override
15 //     public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt>
16 // msgs, ConsumeConcurrentlyContext context) {
17 //         return null;
18 //     }
19 // });
20
21 // 顺序消费
22 consumer.setMessageListener(new MessageListenerOrderly() {
23     @Override
24     public ConsumeOrderlyStatus consumeMessage(List<MessageExt> msgs,
25     ConsumeOrderlyContext context) {
26
27         for (MessageExt msg : msgs) {
28             System.out.println(msg.getMsgId() + "\t" + msg.getQueueId() +
29             "\t" + new String(msg.getBody()));
30         }
31
32     }
33 });
34 consumer.start();
```

2.10.2 无序消息的重试

对于无序消息（普通、定时、延时、事务消息），当消费者消费消息失败时，您可以通过设置返回状态达到消息重试的结果。

无序消息的重试**只针对集群消费方式生效**；广播方式不提供失败重试特性，即消费失败后，失败消息不再重试，继续消费新的消息。

1) 重试次数

消息队列 RocketMQ 默认允许每条消息最多重试 16 次，每次重试的间隔时间如下：

第几次重试	与上次重试的间隔时间	第几次重试	与上次重试的间隔时间
1	10 秒	9	7 分钟
2	30 秒	10	8 分钟
3	1 分钟	11	9 分钟
4	2 分钟	12	10 分钟
5	3 分钟	13	20 分钟
6	4 分钟	14	30 分钟
7	5 分钟	15	1 小时
8	6 分钟	16	2 小时

如果消息重试 16 次后仍然失败，消息将不再投递。如果严格按照上述重试时间间隔计算，某条消息在一直消费失败的前提下，将会在接下来的 4 小时 46 分钟之内进行 16 次重试，超过这个时间范围消息将不再重试投递。

注意：一条消息无论重试多少次，这些重试消息的 Message ID 不会改变。

2) 配置方式

消费失败后，重试配置方式

集群消费方式下，消息消费失败后期望消息重试，需要在消息监听器接口的实现中明确进行配置（三种方式任选一种）：

- 返回 ConsumeConcurrentlyStatus.RECONSUME_LATER; (推荐)
- 返回 Null
- 抛出异常

```

1 public class MyConcurrentlyMessageListener implements
2     MessageListenerConcurrently {
3     @Override
4     public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs,
5         ConsumeConcurrentlyContext context) {
6
7         //处理消息
8         doConsumeMessage(msgs);
9         //方式1：返回 ConsumeConcurrentlyStatus.RECONSUME_LATER，消息将重试
10        return ConsumeConcurrentlyStatus.RECONSUME_LATER;
11        //方式2：返回 null，消息将重试
12        return null;
13        //方式3：直接抛出异常， 消息将重试
14        throw new RuntimeException("Consumer Message exceotion");
15    }
16 }
```

消费失败后，不重试配置方式

集群消费方式下，消息失败后期望消息不重试，需要捕获消费逻辑中可能抛出的异常，最终返回 ConsumeConcurrentlyStatus.CONSUME_SUCCESS，此后这条消息将不会再重试。

```
1 public class MyConcurrentlyMessageListener implements
2     MessageListenerConcurrently {
3         @Override
4         public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs,
5             ConsumeConcurrentlyContext context) {
6
7             try {
8                 doConsumeMessage(msgs);
9             } catch (Throwable e) {
10                 //捕获消费逻辑中的所有异常，并返回
11                 ConsumeConcurrentlyStatus.CONSUME_SUCCESS
12                 return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
13             }
14         }
15 }
```

自定义消息最大重试次数

消息队列 RocketMQ 允许 Consumer 启动的时候设置最大重试次数，重试时间间隔将按照如下策略：

- 最大重试次数小于等于 16 次，则重试时间间隔同上表描述。
- 最大重试次数大于 16 次，超过 16 次的重试时间间隔均为每次 2 小时。

```
1 DefaultMQPushConsumer consumer = new
2     DefaultMQPushConsumer("consumer_grp_04_01");
3 // 设置重新消费的次数
4 // 共16个级别，大于16的一律按照2小时重试
5 consumer.setMaxReconsumeTimes(20);
```

注意：

- 消息最大重试次数的设置对**相同 Group ID 下的所有 Consumer 实例有效**。
- 如果只对相同 Group ID 下两个 Consumer 实例中的其中一个设置了 MaxReconsumeTimes，那么该配置对两个 Consumer 实例均生效。
- 配置采用覆盖的方式生效，即**最后启动的 Consumer 实例会覆盖之前的启动实例的配置**

获取消息重试次数

消费者收到消息后，可按照如下方式获取消息的重试次数：

```
1 public class MyConcurrentlyMessageListener implements
2     MessageListenerConcurrently {
3         @Override
4         public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs,
5             ConsumeConcurrentlyContext context) {
6
7             for (MessageExt msg : msgs) {
8                 System.out.println(msg.getReconsumeTimes());
9             }
10            doConsumeMessage(msgs);
11            return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
12        }
13    }
```

2.11 死信队列

RocketMQ中消息重试超过一定次数后（默认16次）就会被放到死信队列中，在消息队列RocketMQ 中，这种正常情况下无法被消费的消息称为死信消息（Dead-Letter Message），存储死信消息的特殊队列称为死信队列（Dead-Letter Queue）。可以在控制台Topic列表中看到“DLQ”相关的Topic，默认命名是：

%RETRY%消费组名称（重试Topic）

%DLQ%消费组名称（死信Topic）

死信队列也可以被订阅和消费，并且也会过期

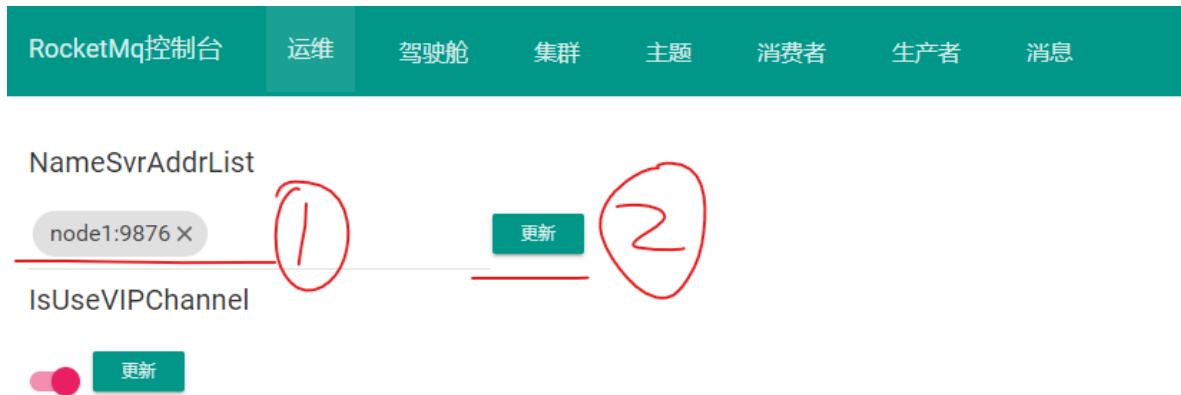
可视化工具：rocketmq-console下载地址：

<https://github.com/apache/rocketmq-externals/archive/rocketmq-console-1.0.0.zip>

使用jdk8：

```
1 # 编译打包
2 mvn clean package -DskipTests
3 # 运行工具
4 java -jar target/rocketmq-console-1.0.0.jar
```

页面设置NameSrv地址即可。如果不生效，就直接修改项目的application.properties中的namesrv地址选项的值。



2.11.1 死信特性

死信消息具有以下特性

- 不会再被消费者正常消费。
- 有效期与正常消息相同，均为3天，3天后会被自动删除。因此，请在死信消息产生后的3天内及时处理。

死信队列具有以下特性：

- 一个死信队列对应一个Group ID，而不是对应单个消费者实例。
- 如果一个Group ID未产生死信消息，消息队列RocketMQ不会为其创建相应的死信队列。
- 一个死信队列包含了对应Group ID产生的所有死信消息，不论该消息属于哪个Topic。

2.11.2 查看死信信息

1.在控制台查询出现死信队列的主题信息

2.在消息界面根据主题查询死信消息

3.选择重新发送消息

一条消息进入死信队列，意味着某些因素导致消费者无法正常消费该消息，因此，通常需要您对其进行特殊处理。排查可疑因素并解决问题后，可以在消息队列 RocketMQ 控制台重新发送该消息，让消费者重新消费一次。

2.12 延迟消息

定时消息（延迟队列）是指消息发送到broker后，不会立即被消费，等待特定时间投递给真正的topic。broker有配置项messageDelayLevel，默认值为“1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m 30m 1h 2h”，18个level。可以配置自定义messageDelayLevel。注意，messageDelayLevel是broker的属性，不属于某个topic。发消息时，设置delayLevel等级即可：msg.setDelayLevel(level)。level有以下三种情况：

- level == 0，消息为非延迟消息
- 1 <= level <= maxLevel，消息延迟特定时间，例如level==1，延迟1s
- level > maxLevel，则level== maxLevel，例如level==20，延迟2h

定时消息会暂存在名为SCHEDULE_TOPIC_XXXX的topic中，并根据delayTimeLevel存入特定的queue，queueId = delayTimeLevel - 1，即一个queue只存相同延迟的消息，保证具有相同发送延迟的消息能够顺序消费。broker会调度地消费SCHEDULE_TOPIC_XXXX，将消息写入真实的topic。

需要注意的是，定时消息会在第一次写入和调度写入真实topic时都会计数，因此发送数量、tps都会变高。

查看SCHEDULE_TOPIC_XXXX主题信息:

```
[root@node1 SCHEDULE_TOPIC_XXXX]# pwd  
/root/store/consumequeue/SCHEDULE_TOPIC_XXXX  
[root@node1 SCHEDULE_TOPIC_XXXX]# ls  
0 1 10 11 12 13 14 15 16 17 2 3 4 5 6 7 8 9  
[root@node1 SCHEDULE_TOPIC_XXXX]#
```

生产者:

```
1 public class MyProducer {  
2     public static void main(String[] args) throws MQClientException,  
3     RemotingException, InterruptedException, MQBrokerException {  
4         DefaultMQProducer producer = new  
5         DefaultMQProducer("producer_grp_06_01");  
6  
7         producer.setNamesrvAddr("node1:9876");  
8  
9         producer.start();  
10  
11         Message message = null;  
12  
13         for (int i = 0; i < 20; i++) {  
14             // 1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m 30m 1h 2h  
15             message = new Message("tp_demo_06", ("hello lagou - " +  
16             i).getBytes());  
17             // 设置延迟级别, 0表示不延迟, 大于18的总是延迟2h  
18             message.setDelayTimeLevel(i);  
19             producer.send(message);  
20         }  
21         producer.shutdown();  
22     }  
23 }
```

消费者:

```
1 public class MyConsumer {  
2     public static void main(String[] args) throws MQClientException {  
3         DefaultMQPushConsumer consumer = new  
4         DefaultMQPushConsumer("consumer_grp_06_01");  
5         consumer.setNamesrvAddr("node1:9876");  
6  
7         consumer.subscribe("tp_demo_06", "*");  
8  
9         consumer.setMessageListener(new MessageListenerConcurrently() {  
10             @Override  
11             public ConsumeConcurrentlyStatus  
12             consumeMessage(List<MessageExt> msgs, ConsumeConcurrentlyContext context) {
```

```

11         System.out.println(system.currentTimeMillis() / 1000);
12         for (MessageExt msg : msgs) {
13             System.out.println(
14                 msg.getTopic() + "\t"
15                 + msg.getQueueId() + "\t"
16                 + msg.getMsgId() + "\t"
17                 + msg.getDelayTimeLevel() + "\t"
18                 + new String(msg.getBody())
19             );
20         }
21     }
22     return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
23 }
24 });
25
26 consumer.start();
27
28 }
29 }
```

2.13 顺序消息

顺序消息是指消息的消费顺序和产生顺序相同，在有些业务逻辑下，必须保证顺序。比如订单的生成、付款、发货，这3个消息必须按顺序处理才行。

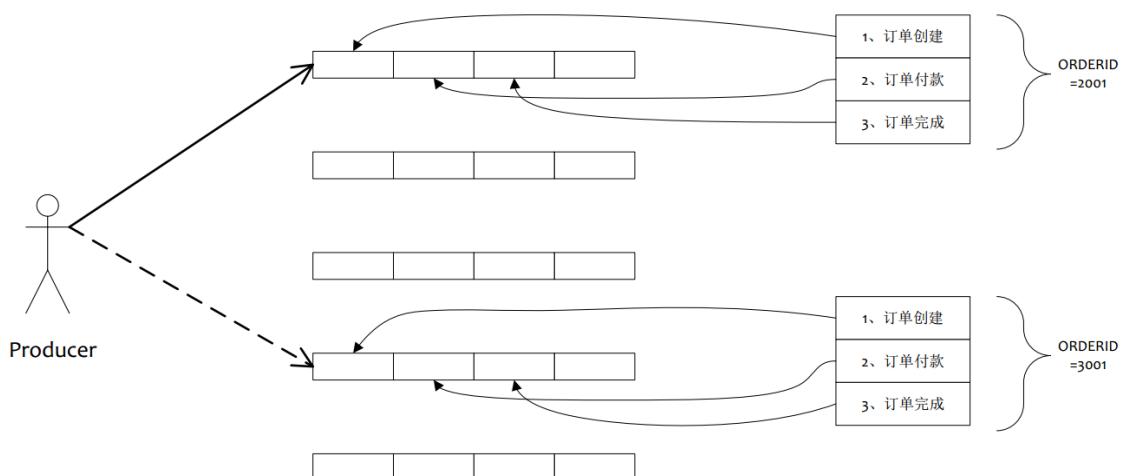
顺序消息分为全局顺序消息和部分顺序消息：

1. 全局顺序消息指某个Topic下的所有消息都要保证顺序；
2. 部分顺序消息只要保证每一组消息被顺序消费即可，比如上面订单消息的例子，只要保证同一个订单ID的三个消息能按顺序消费即可。

在多数的业务场景中实际上只需要局部有序就可以了。

RocketMQ在默认情况下不保证顺序，比如创建一个Topic，默认八个写队列，八个读队列。这时候一条消息可能被写入任意一个队列里；在数据的读取过程中，可能有多个Consumer，每个Consumer也可能启动多个线程并行处理，所以消息被哪个Consumer消费，被消费的顺序和写入的顺序是否一致是不确定的。

要保证全局顺序消息，需要先把Topic的读写队列数设置为一，然后Producer和Consumer的并发设置也要是一。简单来说，为了保证整个Topic的全局消息有序，只能消除所有的并发处理，各部分都设置成单线程处理。



原理如上图所示：

要保证部分消息有序，需要发送端和消费端配合处理。在发送端，要做到把同一业务ID的消息发送到同一个Message Queue；在消费过程中，要做到从同一个Message Queue读取的消息不被并发处理，这样才能达到部分有序。消费端通过使用MessageListenerOrderly类来解决单Message Queue的消息被并发处理的问题。

Consumer使用MessageListenerOrderly的时候，下面四个Consumer的设置依旧可以使用：

1. setConsumeThreadMin
2. setConsumeThreadMax
3. setPullBatchSize
4. setConsumeMessageBatchMaxSize。

前两个参数设置Consumer的线程数；

PullBatchSize指的是一次从Broker的一个Message Queue获取消息的最大数量，默认值是32；

ConsumeMessageBatchMaxSize指的是这个Consumer的Executor（也就是调用MessageListener处理的地方）一次传入的消息数（List<MessageExt>msgs这个链表的最大长度），默认值是1。

上述四个参数可以使用，说明MessageListenerOrderly并不是简单地禁止并发处理。在MessageListenerOrderly的实现中，为每个Consumer Queue加个锁，消费每个消息前，需要先获得这个消息对应的Consumer Queue所对应的锁，这样保证了同一时间，同一个Consumer Queue的消息不被并发消费，但不同Consumer Queue的消息可以并发处理。

部分有序：

顺序消息的生产和消费：

```
1 # 创建主题，8写8读
2 [root@node1 ~]# mqadmin updateTopic -b node1:10911 -n localhost:9876 -r 8 -t
tp_demo_07 -w 8
3 # 删除主题的操作：
4 [root@node1 ~]# mqadmin deleteTopic -c DefaultCluster deleteTopic -n
localhost:9876 -t tp_demo_07
5 # 主题描述
6 [root@node1 ~]# mqadmin topicStatus -n localhost:9876 -t tp_demo_07
```

OrderProducer.java

```
1 package com.lagou.rocket.demo.producer;
2
3 import org.apache.rocketmq.client.exception.MQBrokerException;
4 import org.apache.rocketmq.client.exception.MQClientException;
5 import org.apache.rocketmq.client.producer.DefaultMQProducer;
6 import org.apache.rocketmq.common.message.Message;
7 import org.apache.rocketmq.common.message.MessageQueue;
8 import org.apache.rocketmq.remoting.exception.RemotingException;
9
10 import java.util.List;
11
12 public class OrderProducer {
```

```

13     public static void main(String[] args) throws MQClientException,
14         RemotingException, InterruptedException, MQBrokerException {
15         DefaultMQProducer producer = new
16             DefaultMQProducer("producer_grp_07_01");
17             producer.setNamesrvAddr("node1:9876");
18
19             producer.start();
20
21             Message message = null;
22
23             List<MessageQueue> queues =
24                 producer.fetchPublishMessageQueues("tp_demo_07");
25             System.err.println(queues.size());
26             MessageQueue queue = null;
27             for (int i = 0; i < 100; i++) {
28                 queue = queues.get(i % 8);
29                 message = new Message("tp_demo_07", ("hello lagou - order
30                     create" + i).getBytes());
31                 producer.send(message, queue);
32                 message = new Message("tp_demo_07", ("hello lagou - order
33                     payed" + i).getBytes());
34                 producer.send(message, queue);
35                 message = new Message("tp_demo_07", ("hello lagou - order ship"
36                     + i).getBytes());
37                 producer.send(message, queue);
38             }
39
40             producer.shutdown();
41         }
42     }

```

OrderConsumer.java

```

1 package com.lagou.rocket.demo.consumer;
2
3 import org.apache.rocketmq.client.consumer.DefaultMQPullConsumer;
4 import org.apache.rocketmq.client.consumer.PullResult;
5 import org.apache.rocketmq.client.exception.MQBrokerException;
6 import org.apache.rocketmq.client.exception.MQClientException;
7 import org.apache.rocketmq.common.message.MessageExt;
8 import org.apache.rocketmq.common.message.MessageQueue;
9 import org.apache.rocketmq.remoting.exception.RemotingException;
10
11 import java.util.List;
12 import java.util.Set;
13
14 public class OrderConsumer {
15     public static void main(String[] args) throws MQClientException,
16         RemotingException, InterruptedException, MQBrokerException {
17         DefaultMQPullConsumer consumer = new
18             DefaultMQPullConsumer("consumer_grp_07_01");
19             consumer.setNamesrvAddr("node1:9876");
20             consumer.start();
21     }
22 }

```

```

20     Set<MessageQueue> messageQueues =
21 consumer.fetchSubscribeMessageQueues("tp_demo_07");
22     System.out.println(messageQueues.size());
23
24     for (MessageQueue messageQueue : messageQueues) {
25         Long nextBeginOffset = 0;
26         System.out.println("=====");
27         do {
28             PullResult pullResult = consumer.pull(messageQueue, "*", nextBeginOffset, 1);
29             if (pullResult == null || pullResult.getMsgFoundList() == null) break;
30             nextBeginOffset = pullResult.getNextBeginOffset();
31
32             List<MessageExt> msgFoundList =
33             pullResult.getMsgFoundList();
34             System.out.println(messageQueue.getQueueId() + "\t" +
35             msgFoundList.size());
36             for (MessageExt messageExt : msgFoundList) {
37                 System.out.println(
38                     messageExt.getTopic() + "\t" +
39                     messageExt.getQueueId() + "\t" +
40                     messageExt.getMsgId() + "\t" +
41                     new String(messageExt.getBody())
42                 );
43             }
44         } while (true);
45     }
46 }
```

全局有序:

顺序消息的生产和消费:

```

1 # 创建主题, 8写8读
2 [root@node1 ~]# mqadmin updateTopic -b node1:10911 -n localhost:9876 -r 1 -t
3 tp_demo_07_01 -w 1
4 # 删除主题的操作:
5 [root@node1 ~]# mqadmin deleteTopic -c DefaultCluster deleteTopic -n
localhost:9876 -t tp_demo_07_01
6 # 主题描述
7 [root@node1 ~]# mqadmin topicStatus -n localhost:9876 -t tp_demo_07_01
```

GlobalOrderProducuer.java

```

1 package com.lagou.rocket.demo.producer;
2
3 import org.apache.rocketmq.client.exception.MQBrokerException;
4 import org.apache.rocketmq.client.exception.MQClientException;
5 import org.apache.rocketmq.client.producer.DefaultMQProducer;
```

```

6 import org.apache.rocketmq.common.message.Message;
7 import org.apache.rocketmq.remoting.exception.RemotingException;
8
9 public class GlobalOrderProducer {
10     public static void main(String[] args) throws MQClientException,
11     RemotingException, InterruptedException, MQBrokerException {
12         DefaultMQProducer producer = new
13         DefaultMQProducer("producer_grp_07_02");
14         producer.setNamesrvAddr("node1:9876");
15
16         producer.start();
17
18         Message message = null;
19
20         for (int i = 0; i < 100; i++) {
21             message = new Message("tp_demo_07_01", ("hello lagou" +
22             i).getBytes());
23             producer.send(message);
24         }
25     }

```

GlobalOrderConsumer.java

```

1 package com.lagou.rocket.demo.consumer;
2
3 import org.apache.rocketmq.client.consumer.DefaultMQPushConsumer;
4 import org.apache.rocketmq.client.consumer.listener.ConsumeOrderlyContext;
5 import org.apache.rocketmq.client.consumer.listener.ConsumeOrderlyStatus;
6 import org.apache.rocketmq.client.consumer.listener.MessageListenerOrderly;
7 import org.apache.rocketmq.client.exception.MQClientException;
8 import org.apache.rocketmq.common.message.MessageExt;
9
10 import java.util.List;
11
12 public class GlobalOrderConsumer {
13     public static void main(String[] args) throws MQClientException {
14         DefaultMQPushConsumer consumer = new
15         DefaultMQPushConsumer("consumer_grp_07_03");
16         consumer.setNamesrvAddr("node1:9876");
17         consumer.subscribe("tp_demo_07_01", "*");
18
19         consumer.setConsumeThreadMin(1);
20         consumer.setConsumeThreadMax(1);
21         consumer.setPullBatchSize(1);
22         consumer.setConsumeMessageBatchMaxsize(1);
23
24         consumer.setMessageListener(new MessageListenerOrderly() {
25             @Override
26             public ConsumeOrderlyStatus consumeMessage(List<MessageExt>
27             msgs, ConsumeOrderlyContext context) {
28                 for (MessageExt msg : msgs) {
29                     System.out.println(new String(msg.getBody()));
30                 }
31             }
32         });
33     }

```

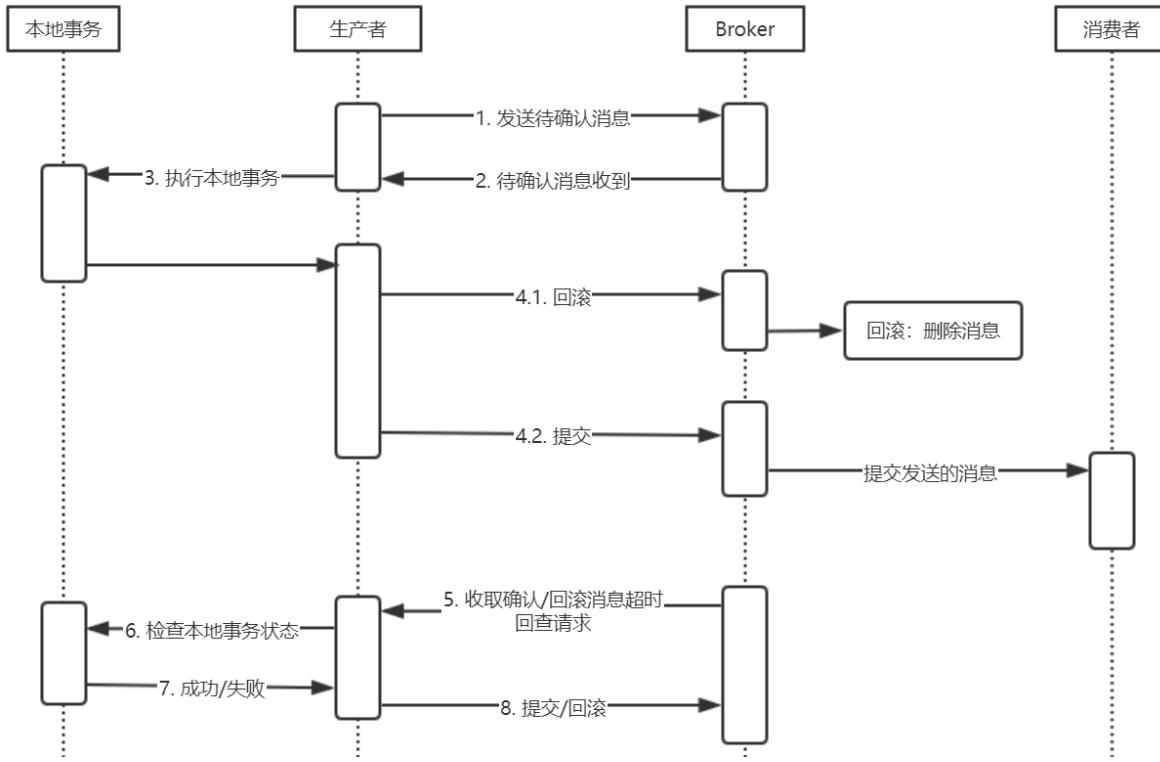
```
28     }
29         return ConsumeOrderlyStatus.SUCCESS;
30     }
31 }
32 consumer.start();
33 }
34 }
```

2.14 事务消息

RocketMQ的事务消息，是指发送消息事件和其他事件需要同时成功或同时失败。比如银行转账，A银行的某账户要转一万元到B银行的某账户。A银行发送“B银行账户增加一万元”这个消息，要和“从A银行账户扣除一万元”这个操作同时成功或者同时失败。

RocketMQ采用两阶段提交的方式实现事务消息，TransactionMQProducer处理上面情况的流程是，先发一个“准备从B银行账户增加一万元”的消息，发送成功后做从A银行账户扣除一万元的操作，根据操作结果是否成功，确定之前的“准备从B银行账户增加一万元”的消息是做commit还是rollback，具体流程如下：

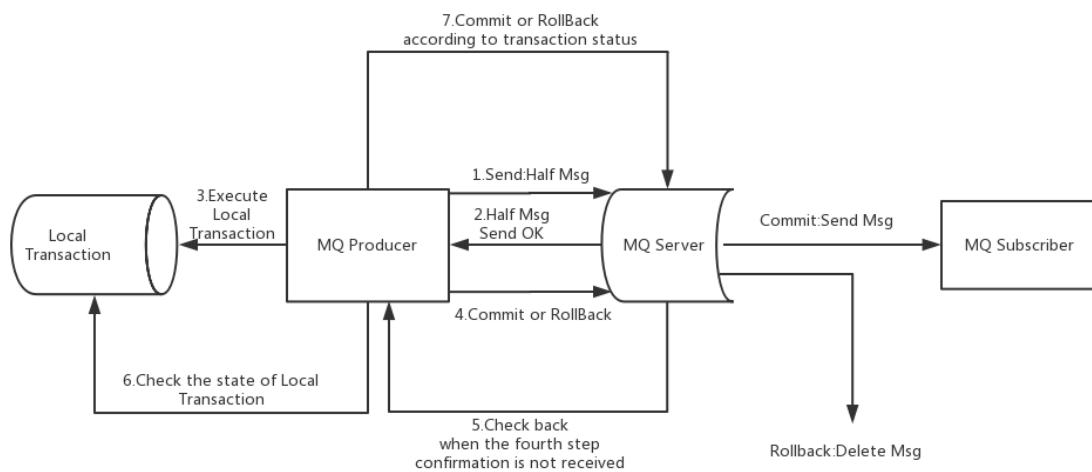
- 1) 发送方向RocketMQ发送“待确认”消息。
- 2) RocketMQ将收到的“待确认”消息持久化成功后，向发送方回复消息已经发送成功，此时第一阶段消息发送完成。
- 3) 发送方开始执行本地事件逻辑。
- 4) 发送方根据本地事件执行结果向RocketMQ发送二次确认（Commit或是Rollback）消息，RocketMQ收到Commit状态则将第一阶段消息标记为可投递，订阅方将能够收到该消息；收到Rollback状态则删除第一阶段的消息，订阅方接收不到该消息。
- 5) 如果出现异常情况，步骤4) 提交的二次确认最终未到达RocketMQ，服务器在经过固定时间段后将对“待确认”消息发起回查请求。
- 6) 发送方收到消息回查请求后（如果发送一阶段消息的Producer不能工作，回查请求将被发送到和Producer在同一个Group里的其他Producer），通过检查对应消息的本地事件执行结果返回Commit或Rollback状态。
- 7) RocketMQ收到回查请求后，按照步骤4) 的逻辑处理。



上面的逻辑似乎很好地实现了事务消息功能，它也是RocketMQ之前的版本实现事务消息的逻辑。但是因为RocketMQ依赖将数据顺序写到磁盘这个特征来提高性能，步骤4) 却需要更改第一阶段消息的状态，这样会造成磁盘Catch的脏页过多，降低系统的性能。所以RocketMQ在4.x的版本中将这部分功能去除。系统中的一些上层Class都还在，用户可以根据实际需求实现自己的事务功能。

客户端有三个类来支持用户实现事务消息，第一个类是LocalTransaction-Executer，用来实例化步骤3) 的逻辑，根据情况返回LocalTransactionState.ROLLBACK_MESSAGE或者LocalTransactionState.COMMIT_MESSAGE状态。第二个类是TransactionMQProducer，它的用法和DefaultMQProducer类似，要通过它启动一个Producer并发消息，但是比DefaultMQProducer多设置本地事务处理函数和回查状态函数。第三个类是TransactionCheckListener，实现步骤5) 中MQ服务器的回查请求，返回LocalTransactionState.ROLLBACK_MESSAGE或者

或者LocalTransactionState.COMMIT_MESSAGE



2.12.1 RocketMQ事务消息流程概要

上图说明了事务消息的大致方案，其中分为两个流程：正常事务消息的发送及提交、事务消息的补偿流程。

1. 事务消息发送及提交：

- (1) 发送消息 (half消息)。
- (2) 服务端响应消息写入结果。
- (3) 根据发送结果执行本地事务 (如果写入失败, 此时half消息对业务不可见, 本地逻辑不执行)。
- (4) 根据本地事务状态执行Commit或者Rollback (Commit操作生成消息索引, 消息对消费者可见)

2. 补偿流程:

- (1) 对没有Commit/Rollback的事务消息 (pending状态的消息), 从服务端发起一次“回查”
- (2) Producer收到回查消息, 检查回查消息对应的本地事务的状态
- (3) 根据本地事务状态, 重新Commit或者Rollback

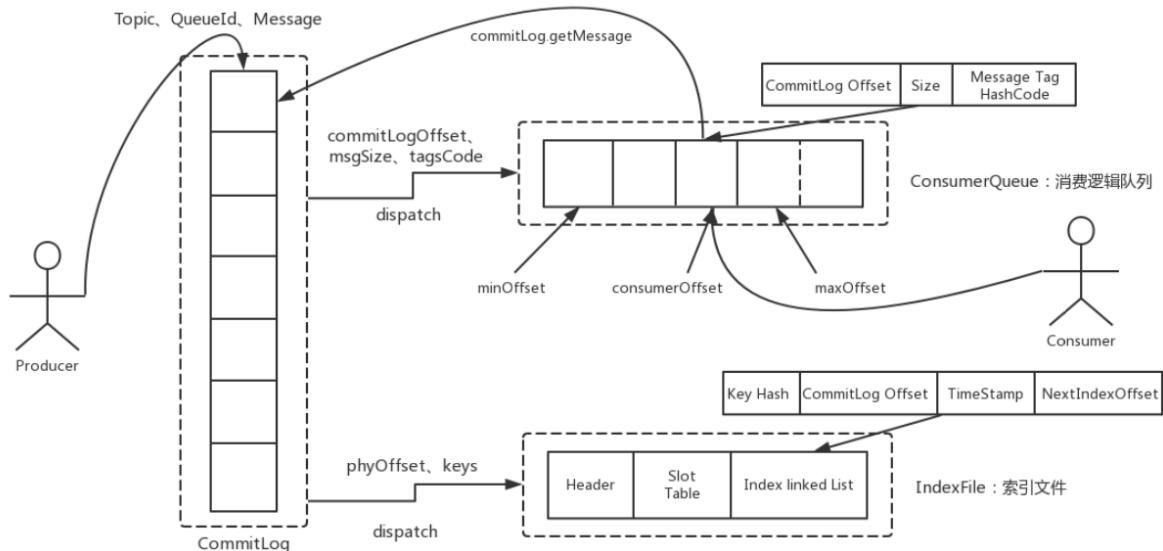
其中, 补偿阶段用于解决消息Commit或者Rollback发生超时或者失败的情况。

2.12.2 RocketMQ事务消息设计

1. 事务消息在一阶段对用户不可见

在RocketMQ事务消息的主要流程中, 一阶段的消息如何对用户不可见。其中, 事务消息相对普通消息最大的特点就是一阶段发送的消息对用户是不可见的。那么, 如何做到写入消息但是对用户不可见呢? RocketMQ事务消息的做法是: 如果消息是half消息, 将备份原消息的主题与消息消费队列, 然后改变主题为RMQ_SYS_TRANS_HALF_TOPIC。由于消费组未订阅该主题, 故消费端无法消费half类型的消息。然后二阶段会显示执行提交或者回滚half消息 (逻辑删除)。当然, 为了防止二阶段操作失败, RocketMQ会开启一个定时任务, 从Topic为RMQ_SYS_TRANS_HALF_TOPIC中拉取消息进行消费, 根据生产者组获取一个服务提供者发送回查事务状态请求, 根据事务状态来决定是提交或回滚消息。

在RocketMQ中, 消息在服务端的存储结构如下, 每条消息都会有对应的索引信息, Consumer通过ConsumeQueue这个二级索引来读取消息实体内容, 其流程如下:



RocketMQ的具体实现策略是: 写入的如果事务消息, 对消息的Topic和Queue等属性进行替换, 同时将原来的Topic和Queue信息存储到消息的属性中, 正因为消息主题被替换, 故消息并不会转发到该原主题的消息消费队列, 消费者无法感知消息的存在, 不会消费。其实改变消息主题是RocketMQ的常用“套路”, 回想一下延时消息的实现机制。RMQ_SYS_TRANS_HALF_TOPIC

2. Commit和Rollback操作以及Op消息的引入

在完成一阶段写入一条对用户不可见的消息后，二阶段如果是Commit操作，则需要让消息对用户可见；如果是Rollback则需要撤销一阶段的消息。先说Rollback的情况。对于Rollback，本身一阶段的消息对用户是不可见的，其实不需要真正撤销消息（实际上RocketMQ也无法去真正的删除一条消息，因为是顺序写文件的）。但是区别于这条消息没有确定状态（Pending状态，事务悬而未决），需要一个操作来标识这条消息的最终状态。RocketMQ事务消息方案中引入了Op消息的概念，用Op消息标识事务消息已经确定的状态（Commit或者Rollback）。如果一条事务消息没有对应的Op消息，说明这个事务的状态还无法确定（可能是二阶段失败了）。引入Op消息后，事务消息无论是Commit或者Rollback都会记录一个Op操作。Commit相对于Rollback只是在写入Op消息前创建Half消息的索引。

3.Op消息的存储和对应关系

RocketMQ将Op消息写入到全局一个特定的Topic中通过源码中的方法—TransactionalMessageUtil.buildOpTopic()；这个Topic是一个内部的Topic（像Half消息的Topic一样），不会被用户消费。Op消息的内容为对应的Half消息的存储的Offset，这样通过Op消息能索引到Half消息进行后续的回查操作。

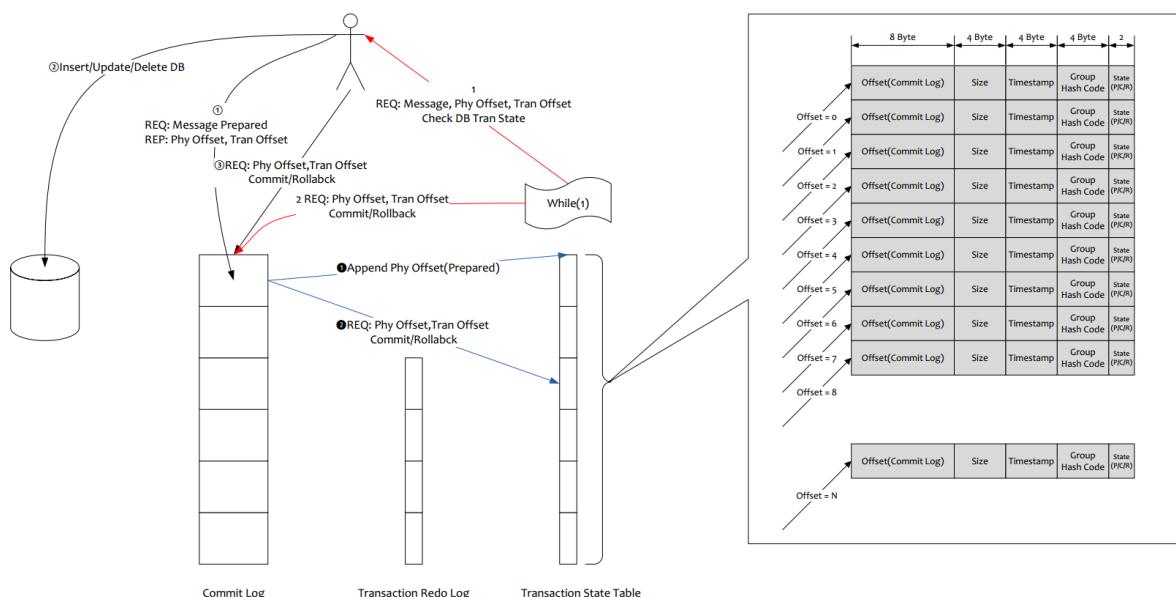
4.Half消息的索引构建

在执行二阶段Commit操作时，需要构建出Half消息的索引。一阶段的Half消息由于是写到一个特殊的Topic，所以二阶段构建索引时需要读取出Half消息，并将Topic和Queue替换成真正的目标的Topic和Queue，之后通过一次普通消息的写入操作来生成一条对用户可见的消息。所以RocketMQ事务消息二阶段其实是利用了一阶段存储的消息的内容，在二阶段时恢复出一条完整的普通消息，然后走一遍消息写入流程。

5.如何处理二阶段失败的消息？

如果在RocketMQ事务消息的二阶段过程中失败了，例如在做Commit操作时，出现网络问题导致Commit失败，那么需要通过一定的策略使这条消息最终被Commit。RocketMQ采用了一种补偿机制，称为“回查”。Broker端对未确定状态的消息发起回查，将消息发送到对应的Producer端（同一个Group的Producer），由Producer根据消息来检查本地事务的状态，进而执行Commit或者Rollback。Broker端通过对比Half消息和Op消息进行事务消息的回查并且推进CheckPoint（记录那些事务消息的状态是确定的）。

值得注意的是，rocketmq并不会无休止的信息事务状态回查，**默认回查15次**，如果15次回查还是无法得知事务状态，rocketmq默认回滚该消息。



事务消息：

TxProducer.java

```
1 package com.lagou.rocket.demo.producer;
2
3 import org.apache.rocketmq.client.exception.MQClientException;
4 import org.apache.rocketmq.client.producer.LocalTransactionState;
5 import org.apache.rocketmq.client.producer.TransactionListener;
6 import org.apache.rocketmq.client.producer.TransactionMQProducer;
7 import org.apache.rocketmq.common.message.Message;
8 import org.apache.rocketmq.common.message.MessageExt;
9
10 public class TxProducer {
11     public static void main(String[] args) throws MQClientException {
12         TransactionListener listener = new TransactionListener() {
13             @Override
14             public LocalTransactionState executeLocalTransaction(Message
15                         msg, Object arg) {
16                 // 当发送事务消息prepare(half)成功后, 调用该方法执行本地事务
17                 System.out.println("执行本地事务, 参数为: " + arg);
18
19                 try {
20                     Thread.sleep(100000);
21                 } catch (InterruptedException e) {
22                     e.printStackTrace();
23                 }
24                 // return LocalTransactionState.ROLLBACK_MESSAGE;
25                 return LocalTransactionState.COMMIT_MESSAGE;
26             }
27
28             @Override
29             public LocalTransactionState checkLocalTransaction(MessageExt
30                         msg) {
31                 // 如果没有收到生产者发送的Half Message的响应, broker发送请求到生产
32                 // 者回查生产者本地事务的状态
33                 // 该方法用于获取本地事务执行的状态。
34                 System.out.println("检查本地事务的状态: " + msg);
35                 return LocalTransactionState.COMMIT_MESSAGE;
36             }
37
38             TransactionMQProducer producer = new
39             TransactionMQProducer("tx_producer_grp_08");
40             producer.setTransactionListener(listener);
41             producer.setNamesrvAddr("node1:9876");
42
43             producer.start();
44
45             Message message = null;
46
47             message = new Message("tp_demo_08", "hello lagou - tx".getBytes());
48             producer.sendMessageInTransaction(message, "
49             {"name":"zhangsan"}");
50         }
51     }
52 }
```

TxConsumer.java

```
1 package com.lagou.rocket.demo.consumer;
2
3 import org.apache.rocketmq.client.consumer.DefaultMQPushConsumer;
4 import
5 org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlyContext;
6 import
7 org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlyStatus;
8 import
9 org.apache.rocketmq.client.consumer.listener.MessageListenerConcurrently;
10 import org.apache.rocketmq.client.exception.MQClientException;
11 import org.apache.rocketmq.common.message.MessageExt;
12
13 import java.util.List;
14
15 public class TxConsumer {
16     public static void main(String[] args) throws MQClientException {
17         DefaultMQPushConsumer consumer = new
18 DefaultMQPushConsumer("txconsumer_grp_08_01");
19
20         consumer.setNamesrvAddr("node1:9876");
21         consumer.subscribe("tp_demo_08", "*");
22
23         consumer.setMessageListener(new MessageListenerConcurrently() {
24             @Override
25             public ConsumeConcurrentlyStatus
26 consumeMessage(List<MessageExt> msgs, ConsumeConcurrentlyContext context) {
27                 for (MessageExt msg : msgs) {
28                     System.out.println(new String(msg.getBody()));
29                 }
30
31                 return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
32             }
33         });
34
35         consumer.start();
36     }
37 }
```

2.15 消息查询

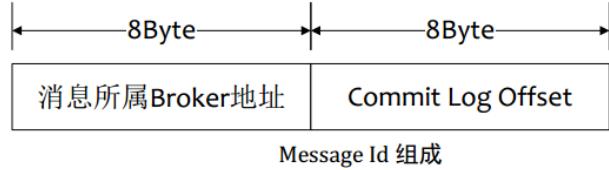
区别于消息消费：先尝后买

尝就是消息查询

买：消息的消费

RocketMQ支持按照下面两种维度（“按照Message Id查询消息”、“按照Message Key查询消息”）进行消息查询。

2.15.1 按照MessageId查询消息

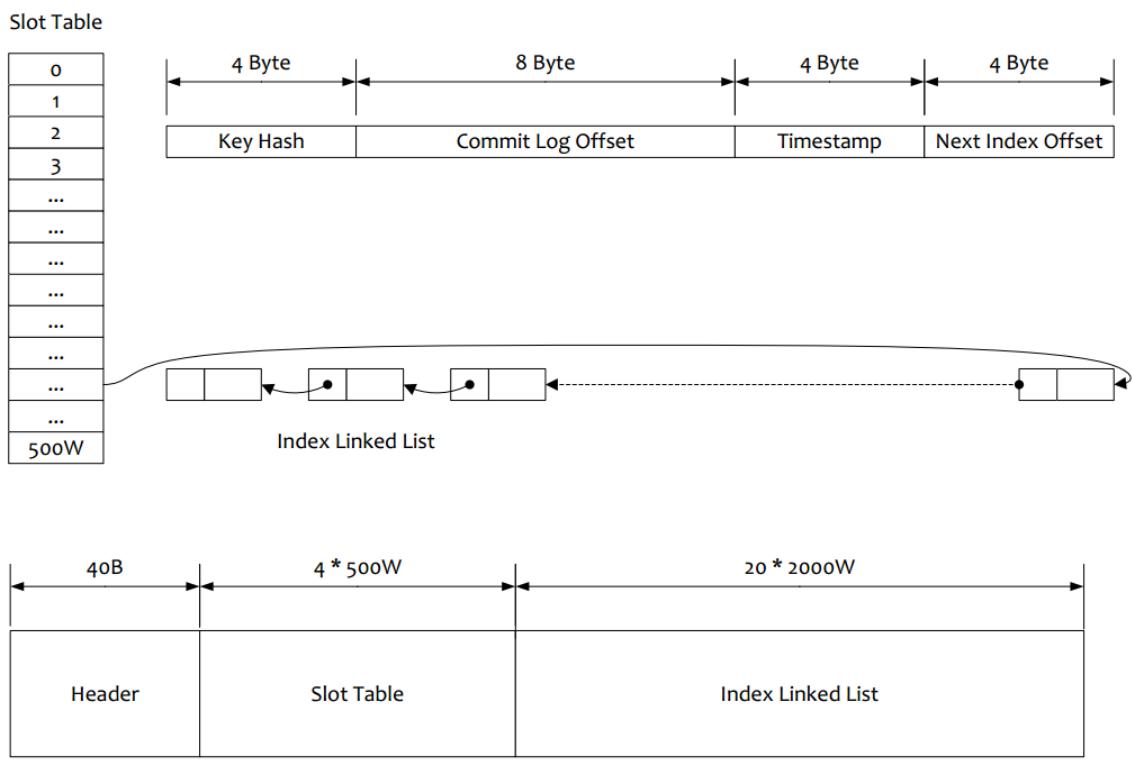


MsgId 总共 16 字节，包含消息存储主机地址 (ip/port)，消息 Commit Log offset。从 MsgId 中解析出 Broker 的地址和 Commit Log 的偏移地址，然后按照存储格式所在位置将消息 buffer 解析成一个完整的消息。

在RocketMQ中具体做法是：Client端从MessageId中解析出Broker的地址（IP地址和端口）和Commit Log的偏移地址后封装成一个RPC请求后，通过Remoting通信层发送（业务请求码：VIEW_MESSAGE_BY_ID）。Broker使用QueryMessageProcessor，使用请求中的commitLog offset 和 size 去 commitLog 中找到真正的记录并解析成一个完整的消息返回。

2.15.2 按照Message Key查询消息

“按照Message Key查询消息”，主要是基于RocketMQ的IndexFile索引文件来实现的。RocketMQ的索引文件逻辑结构，类似JDK中HashMap的实现。索引文件的具体结构如下：



1.根据查询的 key 的 hashCode%slotNum 得到具体的槽的位置 (slotNum 是一个索引文件里面包含的最大槽的数目，例如图中所示 slotNum=5000000) 。

2.根据 slotValue (slot 位置对应的值) 查找到索引项列表的最后一项 (倒序排列，slotValue 总是指向最新的一个索引项) 。

3.遍历索引项列表返回查询时间范围内的结果集 (默认一次最大返回的 32 条记录)

4.Hash 冲突；

第一种，key 的 hash 值不同但模数相同，此时查询的时候会再比较一次 key 的 hash 值 (每个索引项保存了 key 的 hash 值)，过滤掉 hash 值不相等的项。

第二种，hash 值相等但 key 不等，出于性能的考虑冲突的检测放到客户端处理（key 的原始值是存储在消息文件中的，避免对数据文件的解析），客户端比较一次消息体的 key 是否相同。

5. 存储；为了节省空间索引项中存储的时间是时间差值（存储时间-开始时间，开始时间存储在索引文件头中），整个索引文件是定长的，结构也是固定的。

API的使用：

```
1 package com.taobao.rocket.demo.query;
2
3 import org.apache.rocketmq.client.consumer.DefaultMQPullConsumer;
4 import org.apache.rocketmq.client.exception.MQBrokerException;
5 import org.apache.rocketmq.client.exception.MQClientException;
6 import org.apache.rocketmq.common.message.MessageExt;
7 import org.apache.rocketmq.remoting.exception.RemotingException;
8
9 public class QueryingMessageDemo {
10     public static void main(String[] args) throws InterruptedException,
11             RemotingException, MQClientException, MQBrokerException {
12         DefaultMQPullConsumer consumer = new
13             DefaultMQPullConsumer("consumer_grp_09_01");
14         consumer.setNamesrvAddr("node1:9876");
15         consumer.start();
16         MessageExt message = consumer.viewMessage("tp_demo_08",
17             "0A4E00A7178878308DB150A780BB0000");
18         System.out.println(message);
19         System.out.println(message.getMsgId());
20         consumer.shutdown();
21     }
22 }
```

2.16 消息优先级

有些场景，需要应用程序处理几种类型的消息，不同消息的优先级不同。RocketMQ是个先入先出的队列，不支持消息级别或者Topic级别的优先级。业务中简单的优先级需求，可以通过间接的方式解决，下面列举三种优先级相关需求的具体处理方法。

第一种

多个不同的消息类型使用同一个topic时，由于某一个种消息流量非常大，导致其他类型的消息无法及时消费，造成不公平，所以把流量大的类型消息在一个单独的 Topic，其他类型消息在另外一个 Topic，应用程序创建两个 Consumer，分别订阅不同的 Topic，这样就可以了。

第二种

情况和第一种情况类似，但是不用创建大量的 Topic。举个实际应用场景：一个订单处理系统，接收从 100 家快递门店过来的请求，把这些请求通过 Producer 写入 RocketMQ；订单处理程序通过 Consumer 从队列里读取消息并处理，每天最多处理 1 万单。如果这 100 个快递门店中某几个门店订单量大增，比如门店一接了个大客户，一个上午就发出 2 万单消息请求，这样其他的 99 家门店可能被迫等待门店一的 2 万单处理完，也就是两天后订单才能被处理，显然很不公平。

这时可以创建一个 Topic，设置 Topic 的 MessageQueue 数量超过 100 个，Producer 根据订单的门店号，把每个门店的订单写入一个 MessageQueue。DefaultMQPushConsumer 默认是采用循环的方式逐个读取一个 Topic 的所有 MessageQueue，这样如果某家门店订单量大增，这家门店对应的 MessageQueue 消息数增多，等待时间增长，但不会造成其他家门店等待时间增长。

DefaultMQPushConsumer 默认的 pullBatchSize 是 32，也就是每次从某个 MessageQueue 读取消息的时候，最多可以读 32 个。在上面的场景中，为了更加公平，可以把 **pullBatchSize** 设置成 1。

第三种

强制优先级

TypeA、TypeB、TypeC 三类消息。TypeA 处于第一优先级，要确保只要有 TypeA 消息，必须优先处理；TypeB 处于第二优先级；TypeC 处于第三优先级。对这种要求，或者逻辑更复杂的要求，就要用户自己编码实现优先级控制，如果上述的三类消息在一个 Topic 里，可以使用 PullConsumer，自主控制 MessageQueue 的遍历，以及消息的读取；如果上述三类消息在三个 Topic 下，需要启动三个 Consumer，实现逻辑控制三个 Consumer 的消费。

2.17 底层网络通信 - Netty高性能之道

RocketMQ 底层通信的实现是在 Remoting 模块里，因为借助了 Netty 而没有重复造轮子，RocketMQ 的通信部分没有很多的代码，就是用 Netty 实现了一个自定义协议的客户端/服务器程序。

1. 自定义 ByteBuf 可以从底层解决 ByteBuffer 的一些问题，并且通过“内存池”的设计来提升性能
2. Reactor 主从多线程模型
3. 充分利用了零拷贝，CAS/volatile 高效并发编程特性
4. 无锁串行化设计
5. 管道责任链的编程模型
6. 高性能序列化框架的支持
7. 灵活配置 TCP 协议参数

RocketMQ 消息队列集群主要包括 NameServer、Broker(Master/Slave)、Producer、Consumer 四个角色，基本通讯流程如下：

(1) Broker 启动后需要完成一次将自己注册至 NameServer 的操作；随后每隔 30s 时间定时向 NameServer 上报 Topic 路由信息。

(2) 消息生产者 Producer 作为客户端发送消息时候，需要根据消息的 Topic 从本地缓存的 TopicPublishInfoTable 获取路由信息。如果没有则更新路由信息会从 NameServer 上重新拉取，同时 Producer 会默认每隔 30s 向 NameServer 拉取一次路由信息。

(3) 消息生产者 Producer 根据(2) 中获取的路由信息选择一个队列 (MessageQueue) 进行消息发送；Broker 作为消息的接收者接收消息并落盘存储。

(4) 消息消费者 Consumer 根据(2) 中获取的路由信息，并再完成客户端的负载均衡后，选择其中的某一个或者某几个消息队列来拉取消息并进行消费。

从上面 1) ~ 3) 中可以看出在消息生产者、Broker 和 NameServer 之间都会发生通信（这里只说了 MQ 的部分通信），因此如何设计一个良好的网络通信模块在 MQ 中至关重要，它将决定 RocketMQ 集群整体的消息传输能力与最终的性能。

rocketmq-remoting 模块是 RocketMQ 消息队列中负责网络通信的模块，它几乎被其他所有需要网络通信的模块（诸如 rocketmq-client、rocketmq-broker、rocketmq-namesrv）所依赖和引用。为了实现客户端与服务器之间高效的数据请求与接收，RocketMQ 消息队列自定义了通信协议并在 Netty 的基础之上扩展了通信模块。

RocketMQ 中惯用的套路：

请求报文和响应都使用 RemotingCommand，然后在 Processor 处理器中根据 RequestCode 请求码来匹配对应的处理方法。

处理器通常继承至 NettyRequestProcessor，使用前需要先注册才行，注册方式
remotingServer.registerDefaultProcessor

网络通信核心的东西无非是：

线程模型

私有协议定义

编解码器

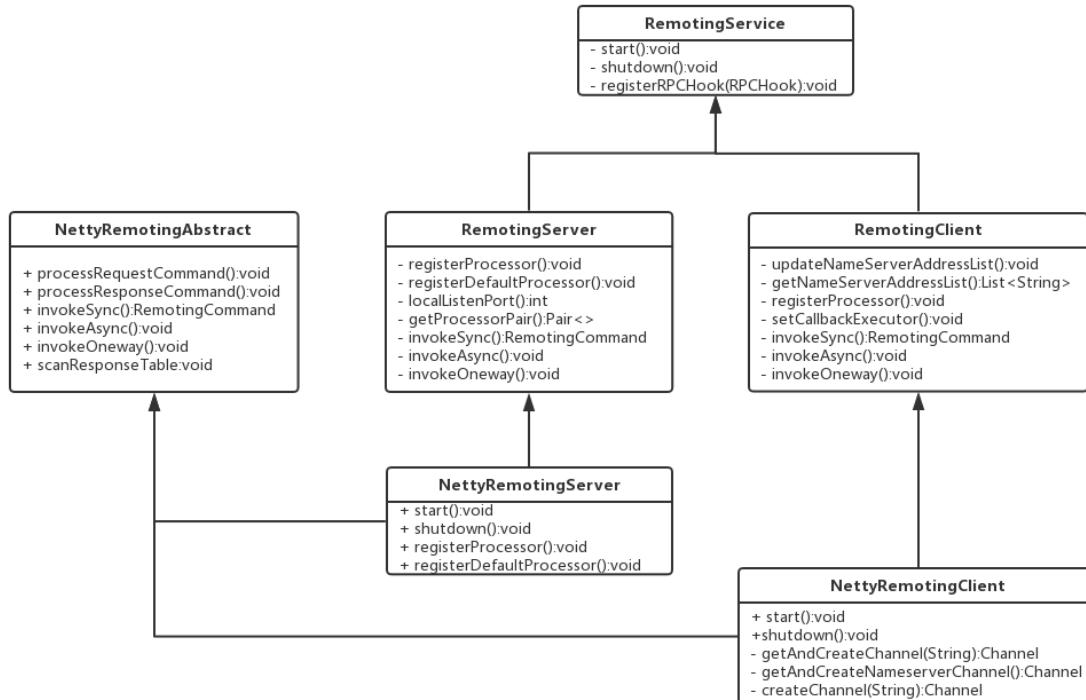
序列化/反序列化

...

既然是基于 Netty 的网络通信，当然少不了一堆自定义实现的 Handler，

例如继承至：SimpleChannelInboundHandler ChannelDuplexHandler

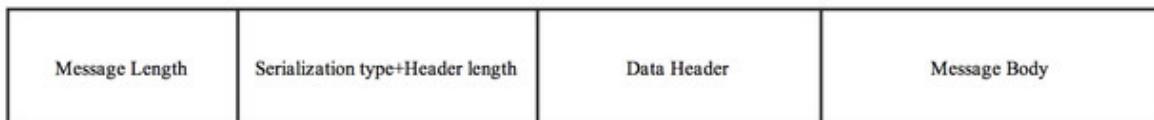
2.17.1 Remoting 通信类结构



2.17.2 协议设计与编解码

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

Header 字段	类型	Request说明	Response说明
code	int	请求操作码，应答方根据不同的请求码进行不同的业务处理	应答响应码。0表示成功，非0则表示各种错误
language	LanguageCode	请求方实现的语言	应答方实现的语言
version	int	请求方程序的版本	应答方程序的版本
opaque	int	相当于requestId，在同一个连接上的不同请求标识码，与响应消息中的相对应	应答不做修改直接返回
flag	int	区分是普通RPC还是onewayRPC得标志	区分是普通RPC还是onewayRPC得标志
remark	String	传输自定义文本信息	传输自定义文本信息
extFields	HashMap<String, String>	请求自定义扩展信息	响应自定义扩展信息

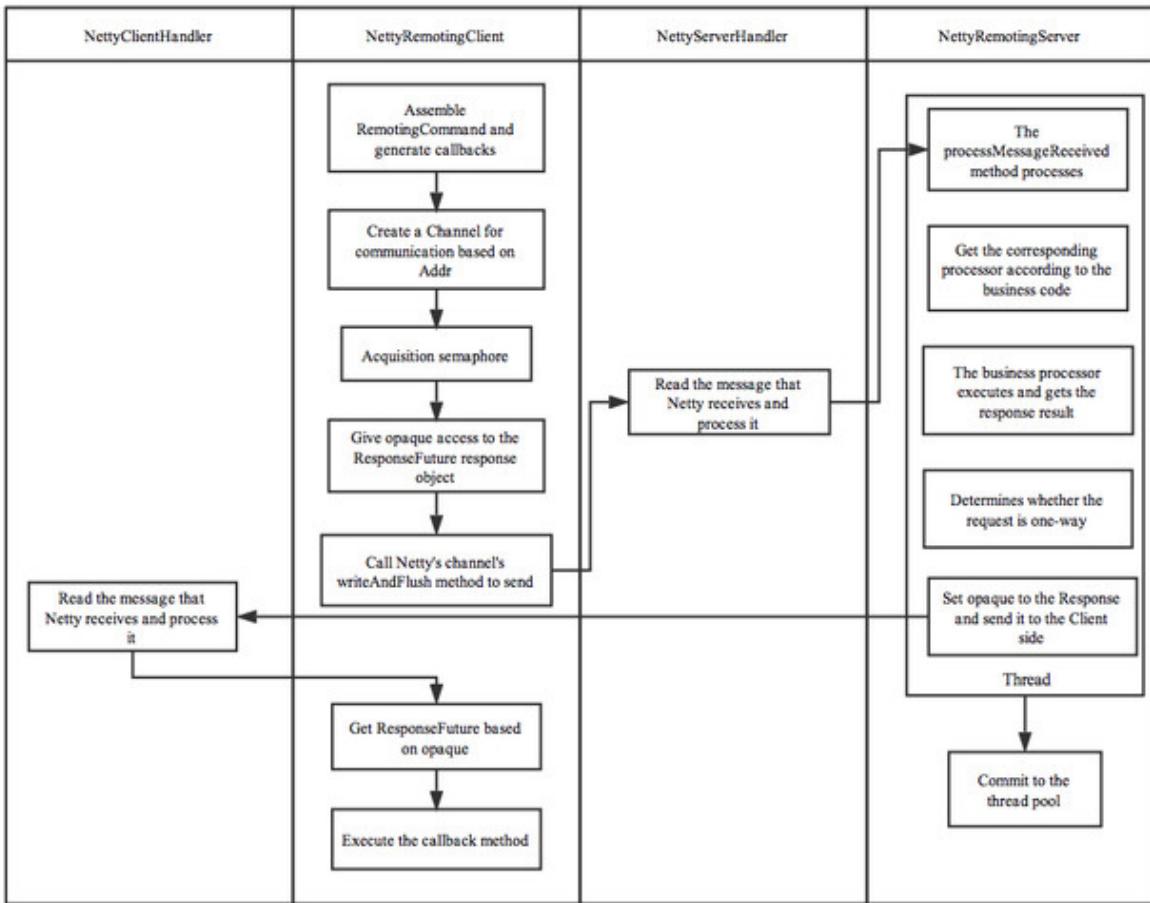


可见传输内容主要可以分为以下4部分：

- (1) 消息长度：总长度，四个字节存储，占用一个int类型；
- (2) 序列化类型&消息头长度：同样占用一个int类型，第一个字节表示序列化类型，后面三个字节表示消息头长度；
- (3) 消息头数据：经过序列化后的消息头数据；
- (4) 消息主体数据：消息主体的二进制字节数据内容；

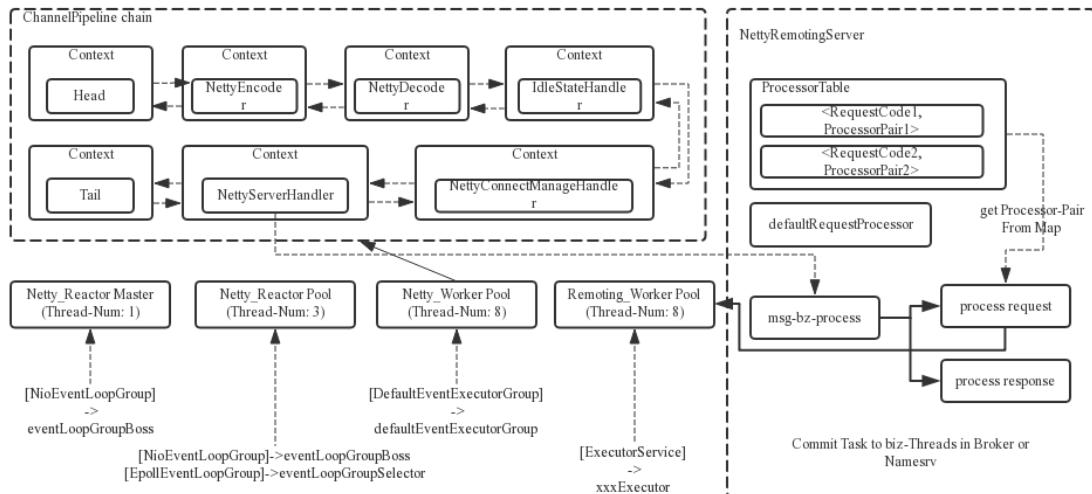
2.17.3 消息的通信方式和流程

在RocketMQ消息队列中支持通信的方式主要有同步(sync)、异步(async)、单向(oneway)三种。其中“单向”通信模式相对简单，一般用在发送心跳包场景下，无需关注其Response。这里，主要介绍RocketMQ的异步通信流程。



2.17.4 Reactor主从多线程模型

RocketMQ的RPC通信采用Netty组件作为底层通信库，同样也遵循了Reactor多线程模型，同时又在这之上做了一些扩展和优化。



上面的框图中可以大致了解RocketMQ中NettyRemotingServer的Reactor 多线程模型。

一个 Reactor 主线程 (eventLoopGroupBoss, 即为上面的1) 负责监听 TCP网络连接请求, 建立好连接, 创建SocketChannel, 并注册到selector上。

RocketMQ的源码中会自动根据OS的类型选择NIO和Epoll, 也可以通过参数配置), 然后监听真正的网络数据。

拿到网络数据后，再丢给Worker线程池（eventLoopGroupSelector，即为上面的“N”，源码中默认设置为3），在真正执行业务逻辑之前需要进行SSL验证、编解码、空闲检查、网络连接管理，这些工作交给defaultEventExecutorGroup（即为上面的“M1”，源码中默认设置为8）去做。

处理业务操作放在业务线程池中执行，根据 RemotingCommand 的业务请求码code去 processorTable这个本地缓存变量中找到对应的 processor，然后封装成task任务后，提交给对应的业务processor处理线程池来执行（sendMessageExecutor，以发送消息为例，即为上面的“M2”）。

从入口到业务逻辑的几个步骤中线程池一直再增加，这跟每一步逻辑复杂性相关，越复杂，需要的并发通道越宽。

线程数	线程名	线程具体说明
1	NettyBoss_%d	Reactor 主线程
N	NettyServerEPOLLSelector%d%d	Reactor 线程池
M1	NettyServerCodecThread_%d	Worker线程池
M2	RemotingExecutorThread_%d	

2.18 限流

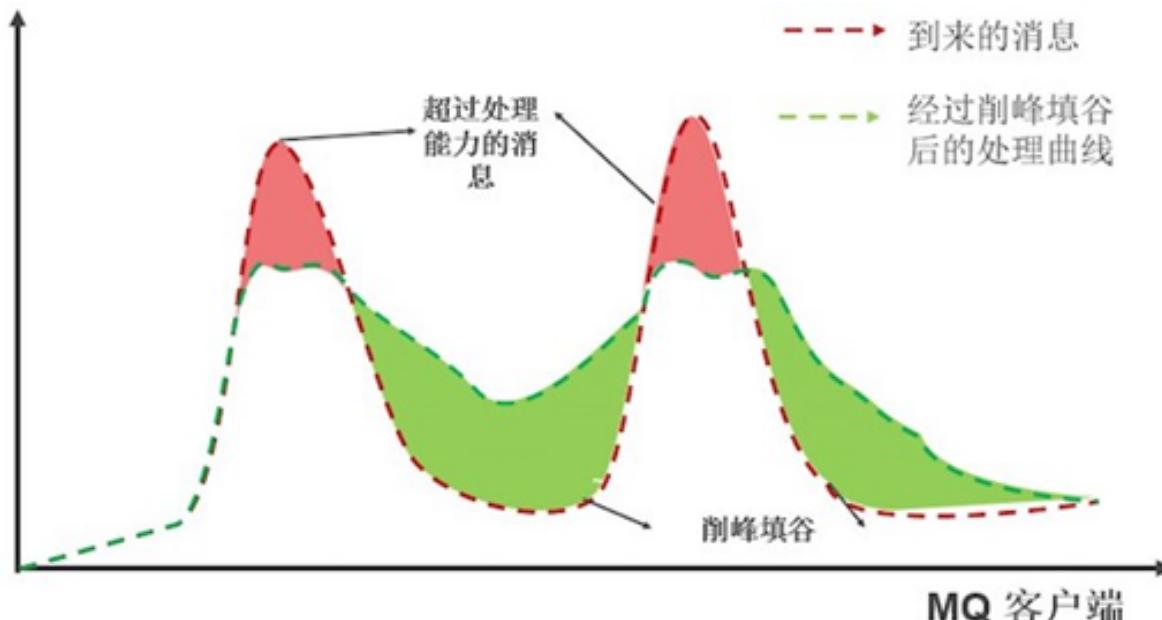
RocketMQ消费端中我们可以：

1. 设置最大消费线程数
2. 每次拉取消息条数等

同时：

1. PushConsumer会判断获取但还未处理的消息个数、消息总大小、Offset的跨度，
2. 任何一个值超过设定的大小就隔一段时间再拉取消息，从而达到流量控制的目的。

在 Apache RocketMQ 中，当消费者去消费消息的时候，无论是通过 pull 的方式还是 push 的方式，都可能会出现大批量的消息突刺。如果此时要处理所有消息，很可能会导致系统负载过高，影响稳定性。但其实可能后面几秒之内都没有消息投递，若直接把多余的消息丢掉则没有充分利用系统处理消息的能力。我们希望可以把消息突刺均摊到一段时间内，让系统负载保持在消息处理水位之下的同时尽可能地处理更多消息，从而起到“削峰填谷”的效果：



上图中红色的部分代表超出消息处理能力的部分。我们可以看到消息突刺往往都是瞬时的、不规律的，其后一段时间系统往往都会有空闲资源。我们希望把红色的那部分消息平摊到后面空闲时去处理，这样既可以保证系统负载处在一个稳定的水位，又可以尽可能地处理更多消息。

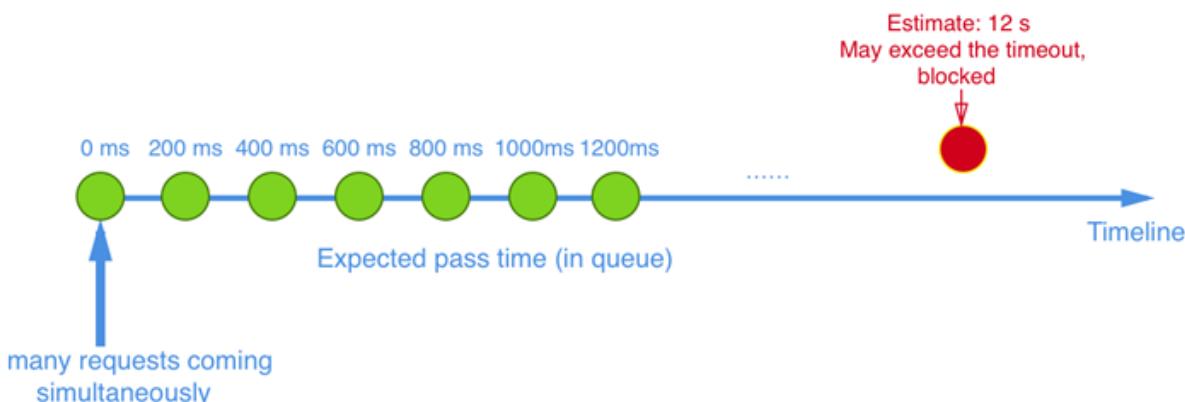
2.18.1 Sentinel 介绍

[Sentinel](#) 是阿里中间件团队开源的，面向分布式服务架构的轻量级流量控制产品，主要以流量为切入点，从流量控制、熔断降级、系统负载保护等多个维度来帮助用户保护服务的稳定性。

2.18.2 Sentinel原理

Sentinel 专门为这种场景提供了[匀速器](#)的特性，可以把突然到来的大量请求以匀速的形式均摊，以固定的间隔时间让请求通过，以稳定的速度逐步处理这些请求，起到“削峰填谷”的效果，从而避免流量突刺造成系统负载过高。同时堆积的请求将会排队，逐步进行处理；当请求排队预计超过最大超时时长的时候则直接拒绝，而不是拒绝全部请求。

比如在 RocketMQ 的场景下配置了匀速模式下请求 QPS 为 5，则会每 200 ms 处理一条消息，多余的处理任务将排队；同时设置了超时时间为 5 s，预计排队时长超过 5s 的处理任务将会直接被拒绝。示意图如下图所示：



RocketMQ 用户可以根据不同的 group 和不同的 topic 分别设置限流规则，限流控制模式设置为匀速器模式 (RuleConstant.CONTROL_BEHAVIOR_RATE_LIMITER)，比如：

```
1 private void initFlowControlRule() {
2     FlowRule rule = new FlowRule();
3     rule.setResource(KEY); // 对应的 key 为 groupName:topicName
4     rule.setCount(5);
5     rule.setGrade(RuleConstant.FLOW_GRADE_QPS);
6     rule.setLimitApp("default");
7
8     // 匀速器模式下，设置了 QPS 为 5，则请求每 200 ms 允许通过 1 个
9     rule.setControlBehavior(RuleConstant.CONTROL_BEHAVIOR_RATE_LIMITER);
10    // 如果更多的请求到达，这些请求会被置于虚拟的等待队列中。等待队列有一个 max timeout,
11    // 如果请求预计的等待时间超过这个时间会直接被 block
12    // 在这里，timeout 为 5s
13
14    rule.setMaxQueueingTimeMs(5 * 1000);
15    FlowRuleManager.loadRules(Collections.singletonList(rule));
}
```

参考：

<https://github.com/alibaba/Sentinel/wiki/Sentinel-%E4%B8%BA-RocketMQ-%E4%BF%9D%E9%A9%BE%E6%8A%A4%E8%88%AA>

```
1 package com.lagou.rocket.demo.consumer;
2
3 import com.alibaba.csp.sentinel.Constants;
4 import com.alibaba.csp.sentinel.Entry;
5 import com.alibaba.csp.sentinel.EntryType;
6 import com.alibaba.csp.sentinel.SphU;
7 import com.alibaba.csp.sentinel.context.ContextUtil;
8 import com.alibaba.csp.sentinel.slots.block.BlockException;
9 import com.alibaba.csp.sentinel.slots.block.RuleConstant;
10 import com.alibaba.csp.sentinel.slots.block.flow.FlowRule;
11 import com.alibaba.csp.sentinel.slots.block.flow.FlowRuleManager;
12 import org.apache.rocketmq.client.consumer.DefaultMQPullConsumer;
13 import org.apache.rocketmq.client.consumer.PullResult;
14 import org.apache.rocketmq.client.exception.MQClientException;
15 import org.apache.rocketmq.common.message.MessageExt;
16 import org.apache.rocketmq.common.message.MessageQueue;
17
18 import java.util.Collections;
19 import java.util.HashMap;
20 import java.util.Map;
21 import java.util.Set;
22 import java.util.concurrent.ExecutorService;
23 import java.util.concurrent.Executors;
24 import java.util.concurrent.atomic.AtomicLong;
25
26 public class PullDemo {
27
28     private static final String GROUP_NAME = "consumer_grp_13_05";
29     private static final String TOPIC_NAME = "tp_demo_13";
30
31     private static final String KEY = String.format("%s:%s", GROUP_NAME,
32     TOPIC_NAME);
33
34     private static final Map<MessageQueue, Long> OFFSET_TABLE = new
35     HashMap<MessageQueue, Long>();
36
37     @SuppressWarnings("PMD.ThreadPoolCreationRule")
38     private static final ExecutorService pool =
39     Executors.newFixedThreadPool(32);
38
39     private static final AtomicLong SUCCESS_COUNT = new AtomicLong(0);
40     private static final AtomicLong FAIL_COUNT = new AtomicLong(0);
41
42     public static void main(String[] args) throws MQClientException {
43         // 初始化哨兵的流控
44         initFlowControlRule();
45
46         DefaultMQPullConsumer consumer = new
47         DefaultMQPullConsumer(GROUP_NAME);
48         consumer.setNamesrvAddr("node1:9876");
49         consumer.start();
50
51         Set<MessageQueue> mqs =
52         consumer.fetchSubscribeMessageQueues(TOPIC_NAME);
53         for (MessageQueue mq : mqs) {
54             System.out.printf("Consuming messages from the queue: %s%n",
55             mq);
```

```
52     SINGLE_MQ:
53     while (true) {
54         try {
55             PullResult pullResult =
56                 consumer.pullBlockIfNotFound(mq, null,
57                     getMessageQueueOffset(mq), 32);
58             if (pullResult.getMsgFoundList() != null) {
59                 for (MessageExt msg :
60                     pullResult.getMsgFoundList()) {
61                     doSomething(msg);
62                 }
63             long nextoffset = pullResult.getNextBeginOffset();
64             // 将每个mq对应的偏移量记录在本地HashMap中
65             putMessageQueueOffset(mq, nextoffset);
66             consumer.updateConsumeOffset(mq, nextoffset);
67             switch (pullResult.getPullStatus()) {
68                 case NO_NEW_MSG:
69                     break SINGLE_MQ;
70                 case FOUND:
71                 case NO_MATCHED_MSG:
72                 case OFFSET_ILLEGAL:
73                 default:
74                     break;
75                 }
76             } catch (Exception e) {
77                 e.printStackTrace();
78             }
79         }
80     }
81     consumer.shutdown();
82 }
83
84 /**
85 * 对每个收到的消息使用一个线程提交任务
86 * @param message
87 */
88 private static void doSomething(MessageExt message) {
89     pool.submit(() -> {
90         Entry entry = null;
91         try {
92             ContextUtil.enter(KEY);
93             entry = Sphu.entry(KEY, EntryType.OUT);
94
95             // 在这里处理业务逻辑，此处只是打印
96             System.out.printf("[%d] [%s] [Success: %d] Receive New
97 Messages: %s %n", System.currentTimeMillis(),
98                     Thread.currentThread().getName(),
99                     SUCCESS_COUNT.addAndGet(1), new String(message.getBody()));
100        } catch (BlockException ex) {
101            // Blocked.
102            System.out.println("Blocked: " + FAIL_COUNT.addAndGet(1));
103        } finally {
104            if (entry != null) {
105                entry.exit();
106            }
107        }
108    }
109 }
```

```

106             ContextUtil.exit();
107         }
108     });
109 }
110
111 private static void initFlowControlRule() {
112     FlowRule rule = new FlowRule();
113     // 消费组名称:主题名称    字符串
114     rule.setResource(KEY);
115     // 根据QPS进行流控
116     rule.setGrade(RuleConstant.FLOW_GRADE_QPS);
117     // 1表示QPS为1, 请求间隔1000ms。
118     // 如果是5, 则表示每秒5个消息, 请求间隔200ms
119     rule.setCount(1);
120     rule.setLimitApp("default");
121
122     // 调用使用固定间隔。如果qps为1, 则请求之间间隔为1s
123
124     rule.setControlBehavior(RuleConstant.CONTROL_BEHAVIOR_RATE_LIMITER);
125     // 如果请求太多, 就将这些请求放到等待队列中
126     // 该队列有超时时间。如果等待队列中请求超时, 则丢弃
127     // 此处设置超时时间为5s
128     rule.setMaxQueueingTimeMs(5 * 1000);
129     FlowRuleManager.loadRules(Collections.singletonList(rule));
130 }
131
132 // 获取指定MQ的偏移量
133 private static long getMessageQueueOffset(MessageQueue mq) {
134     Long offset = OFFSET_TABLE.get(mq);
135     if (offset != null) {
136         return offset;
137     }
138
139     return 0;
140 }
141
142 // 在本地HashMap中记录偏移量
143 private static void putMessageQueueOffset(MessageQueue mq, long
144 offset) {
145     OFFSET_TABLE.put(mq, offset);
146 }
147 }
```

第三部分 RocketMQ高级实战

3.1 生产者

3.1.1 Tags的使用

一个应用尽可能用一个Topic, 而消息子类型则可以用tags来标识。tags可以由应用自由设置, 只有生产者在发送消息设置了tags, 消费方在订阅消息时才可以利用tags通过broker做消息过滤: message.setTags("TagA")。

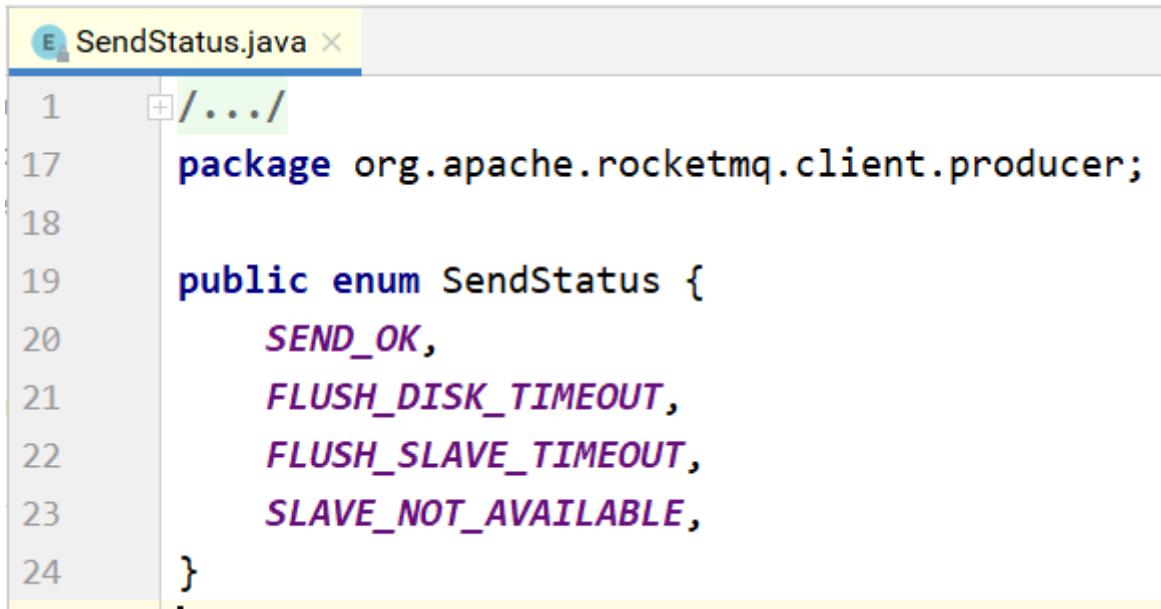
3.1.2 Keys的使用

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);
```

3.1.3 日志的打印

消息发送成功或者失败要打印消息日志，务必要打印SendResult和key字段。send消息方法只要不抛异常，就代表发送成功。发送成功会有多个状态，在SendResult里定义。



```
SendStatus.java  
1  /.../  
17 package org.apache.rocketmq.client.producer;  
18  
19 public enum SendStatus {  
20     SEND_OK,  
21     FLUSH_DISK_TIMEOUT,  
22     FLUSH_SLAVE_TIMEOUT,  
23     SLAVE_NOT_AVAILABLE,  
24 }
```

以下对每个状态进行说明：

- **SEND_OK**

消息发送成功。要注意的是消息发送成功也不意味着它是可靠的。要确保不会丢失任何消息，还应启用同步Master服务器或同步刷盘，即SYNC_MASTER或SYNC_FLUSH。

- **FLUSH_DISK_TIMEOUT**

消息发送成功但是服务器刷盘超时。此时消息已经进入服务器队列（内存），只有服务器宕机，消息才会丢失。消息存储配置参数中可以设置刷盘方式和同步刷盘时间长度。

如果Broker服务器设置了刷盘方式为同步刷盘，即FlushDiskType=SYNC_FLUSH（默认为异步刷盘方式），当Broker服务器未在同步刷盘时间内（默认为5s）完成刷盘，则将返回该状态——刷盘超时。

- **FLUSH_SLAVE_TIMEOUT**

消息发送成功，但是服务器同步到Slave时超时。此时消息已经进入服务器队列，只有服务器宕机，消息才会丢失。

如果Broker服务器的角色是同步Master，即SYNC_MASTER（默认是异步Master即ASYNC_MASTER），并且从Broker服务器未在同步刷盘时间（默认为5秒）内完成与主服务器的同步，则将返回该状态——数据同步到Slave服务器超时。

- **SLAVE_NOT_AVAILABLE**

消息发送成功，但是此时Slave不可用。

如果Broker服务器的角色是同步Master，即**SYNC_MASTER**（默认是异步Master服务器即**ASYNC_MASTER**），但没有配置slave Broker服务器，则将返回该状态——无Slave服务器可用。

3.1.4 消息发送失败处理方式

Producer的send方法本身支持内部重试，重试逻辑如下：

- 至多重试2次（同步发送为2次，异步发送为0次）。
- 如果发送失败，则轮转到下一个Broker。这个方法的总耗时时间不超过**sendMsgTimeout**设置的值，默认10s。
- 如果本身向broker发送消息产生超时异常，就不会再重试。

以上策略也是在一定程度上保证了消息可以发送成功。如果业务对消息可靠性要求比较高，建议应用增加相应的重试逻辑：比如调用**send**同步方法发送失败时，则尝试将消息存储到db，然后由后台线程定时重试，确保消息一定到达Broker。

上述db重试方式为什么没有集成到MQ客户端内部做，而是要求应用自己去完成，主要基于以下几点考虑：

1. MQ的客户端设计为无状态模式，方便任意的水平扩展，且对机器资源的消耗仅仅是cpu、内存、网络。
2. 如果MQ客户端内部集成一个KV存储模块，那么数据只有同步落盘才能较可靠，而同步落盘本身性能开销较大，所以通常会采用异步落盘，又由于应用关闭过程不受MQ运维人员控制，可能经常会发生 kill -9 这样暴力方式关闭，造成数据没有及时落盘而丢失。
3. Producer所在机器的可靠性较低，一般为虚拟机，不适合存储重要数据。综上，建议重试过程交由应用来控制。

3.1.5 选择oneway形式发送

通常消息的发送是这样一个过程：

- 客户端发送请求到服务器
- 服务器处理请求
- 服务器向客户端返回应答

所以，一次消息发送的耗时时间是上述三个步骤的总和，而某些场景要求耗时非常短，但是对可靠性要求并不高，例如日志收集类应用，此类应用可以采用**oneway**形式调用，**oneway形式只发送请求不等待应答**，而发送请求在客户端实现层面仅仅是一个操作系统系统调用的开销，即将数据写入客户端的socket缓冲区，此过程耗时通常在**微秒级**。

3.2 消费者

3.2.1 消费过程幂等

RocketMQ无法避免消息重复（Exactly-Once），所以如果业务对消费重复非常敏感，务必要在业务层面进行去重处理。

可以借助关系数据库进行去重。首先需要确定消息的唯一键，可以是msgId，也可以是消息内容中的唯一标识字段，例如订单Id等。

在消费之前判断唯一键是否在关系数据库中存在。如果不存在则插入，并消费，否则跳过。（实际过程要考虑原子性问题，判断是否存在可以尝试插入，如果报主键冲突，则插入失败，直接跳过）

msgId一定是全局唯一标识符，但是实际使用中，可能会存在相同的消息有两个不同msgId的情况（消费者主动重发、因客户端重投机制导致的重复等），这种情况就需要使业务字段进行重复消费。

3.2.2 消费速度慢的处理方式

提高消费并行度

绝大部分消息消费行为都属于 IO 密集型，即可能是操作数据库，或者调用 RPC，这类消费行为的消费速度在于后端数据库或者外系统的吞吐量。

通过增加消费并行度，可以提高总的消费吞吐量，但是并行度增加到一定程度，**反而会下降**。

所以，应用必须要设置合理的并行度。如下有几种修改消费并行度的方法：

- 同一个 ConsumerGroup 下，通过增加 Consumer 实例数量来提高并行度（需要注意的是超过订阅队列数的 Consumer 实例无效）。可以通过加机器，或者在已有机器启动多个进程的方式。
- 提高单个 Consumer 的消费并行线程，通过修改参数 consumeThreadMin、consumeThreadMax 实现。
- 丢弃部分不重要的消息

批量方式消费

某些业务流程如果支持批量方式消费，则可以很大程度上提高消费吞吐量。

例如订单扣款类应用，一次处理一个订单耗时 1 s，一次处理 10 个订单可能也只耗时 2 s，这样即可大幅度提高消费的吞吐量，通过设置 consumer 的 consumeMessageBatchMaxSize 这个参数，默认是 1，即一次只消费一条消息，例如设置为 N，那么每次消费的消息数小于等于 N。

跳过非重要消息

发生消息堆积时，如果消费速度一直追不上发送速度，如果业务对数据要求不高的话，可以选择丢弃不重要的消息。

例如，当某个队列的消息数堆积到 100000 条以上，则尝试丢弃部分或全部消息，这样就可以快速追上发送消息的速度。示例代码如下：

```
1 public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs,
2     ConsumeConcurrentlyContext context) {
3
4     long offset = msgs.get(0).getQueueOffset();
5     String maxOffset =
6         msgs.get(0).getProperty(Message.PROPERTY_MAX_OFFSET);
7
8     long diff = Long.parseLong(maxOffset) - offset;
9
10    if (diff > 100000) {
11        // TODO 消息堆积情况的特殊处理
12        return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
13    }
14
15    // TODO 正常消费过程
16    return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
17 }
```

3.2.3 优化每条消息消费过程

举例如下，某条消息的消费过程如下：

- 根据消息从 DB 查询【数据 1】
- 根据消息从 DB 查询【数据 2】
- 复杂的业务计算
- 向 DB 插入【数据 3】
- 向 DB 插入【数据 4】

这条消息的消费过程中有4次与 DB 的交互，如果按照每次 5ms 计算，那么总共耗时 20ms，假设业务计算耗时 5ms，那么总过耗时 25ms，所以如果能把 4 次 DB 交互优化为 2 次，那么总耗时就可以优化到 15ms，即总体性能提高了 40%。所以应用如果对时延敏感的话，可以把DB部署在SSD硬盘，相比于SCSI磁盘，前者的RT会小很多。

3.2.4 消费打印日志

如果消息量较少，建议在消费入口方法打印消息，消费耗时等，方便后续排查问题。

```
1 public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs,
2     ConsumeConcurrentlyContext context) {
3
4     log.info("RECEIVE_MSG_BEGIN: " + msgs.toString());
5     // TODO 正常消费过程
6
7     return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
}
```

如果能打印每条消息消费耗时，那么在排查消费慢等线上问题时，会更方便。

3.2.5 其他消费建议

1 关于消费者和订阅

第一件需要注意的事情是，不同的消费组可以独立的消费一些 topic，并且每个消费组都有自己的消费偏移量。

确保同一组内的每个消费者订阅信息保持一致。

2 关于有序消息

消费者将锁定每个消息队列，以确保他们被逐个消费，虽然这将会导致性能下降，但是当你关心消息顺序的时候会很有用。

我们不建议抛出异常，你可以返回 ConsumeOrderlyStatus.SUSPEND_CURRENT_QUEUE_A_MOMENT 作为替代。

3 关于并发消费

顾名思义，消费者将并发消费这些消息，建议你使用它来获得良好性能，我们不建议抛出异常，你可以返回 ConsumeConcurrentlyStatus.RECONSUME_LATER 作为替代。

4 关于消费状态Consume Status

对于并发的消费监听器，你可以返回 RECONSUME_LATER 来通知消费者现在不能消费这条消息，并且希望可以稍后重新消费它。然后，你可以继续消费其他消息。对于有序的消息监听器，因为你关心它的顺序，所以不能跳过消息，但是你可以返回SUSPEND_CURRENT_QUEUE_A_MOMENT 告诉消费者等待片刻。

5 关于Blocking

不建议阻塞监听器，因为它会阻塞线程池，并最终可能会终止消费进程

6 关于线程数设置

消费者使用 ThreadPoolExecutor 在内部对消息进行消费，所以你可以通过设置 setConsumeThreadMin 或 setConsumeThreadMax 来改变它。

7 关于消费位点

当建立一个新的消费组时，需要决定是否需要消费已经存在于 Broker 中的历史消息。

`CONSUME_FROM_LAST_OFFSET` 将会忽略历史消息，并消费之后生成的任何消息。

`CONSUME_FROM_FIRST_OFFSET` 将会消费每个存在于 Broker 中的信息。

也可以使用 `CONSUME_FROM_TIMESTAMP` 来消费在指定时间戳后产生的消息。

```
1 public static void main(String[] args) throws MQClientException {
2     DefaultMQPushConsumer consumer = new
3     DefaultMQPushConsumer("consumer_grp_15_01");
4     consumer.setNamesrvAddr("node1:9886");
5     consumer.subscribe("tp_demo_15", "*");
6
7     // 以下三个选一个使用，如果是根据时间戳进行消费，则需要设置时间戳
8
9     // 从第一个消息开始消费，从头开始
10
11    consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_OFFSET);
12    // 从最后一个消息开始消费，不消费历史消息
13
14    consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_LAST_OFFSET);
15    // 从指定的时间戳开始消费
16    consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_TIMESTAMP);
17    // 指定时间戳的值
18    consumer.setConsumeTimestamp("");
19
20    consumer.setMessageListener(new MessageListenerConcurrently() {
21        @Override
22        public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt>
23            msgs, ConsumeConcurrentlyContext context) {
24
25            // TODO 处理消息的业务逻辑
26
27            return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
28        }
29    });
30
31    consumer.start();
32 }
```

3.3 Broker

3.3.1 Broker 角色

Broker 角色分为 ASYNC_MASTER（异步主机）、SYNC_MASTER（同步主机）以及SLAVE（从机）。如果对消息的可靠性要求比较严格，可以采用 SYNC_MASTER加SLAVE的部署方式。如果对消息可靠性要求不高，可以采用ASYNC_MASTER加SLAVE的部署方式。如果只是测试方便，则可以选择仅 ASYNC_MASTER或仅SYNC_MASTER的部署方式。

3.3.2 FlushDiskType

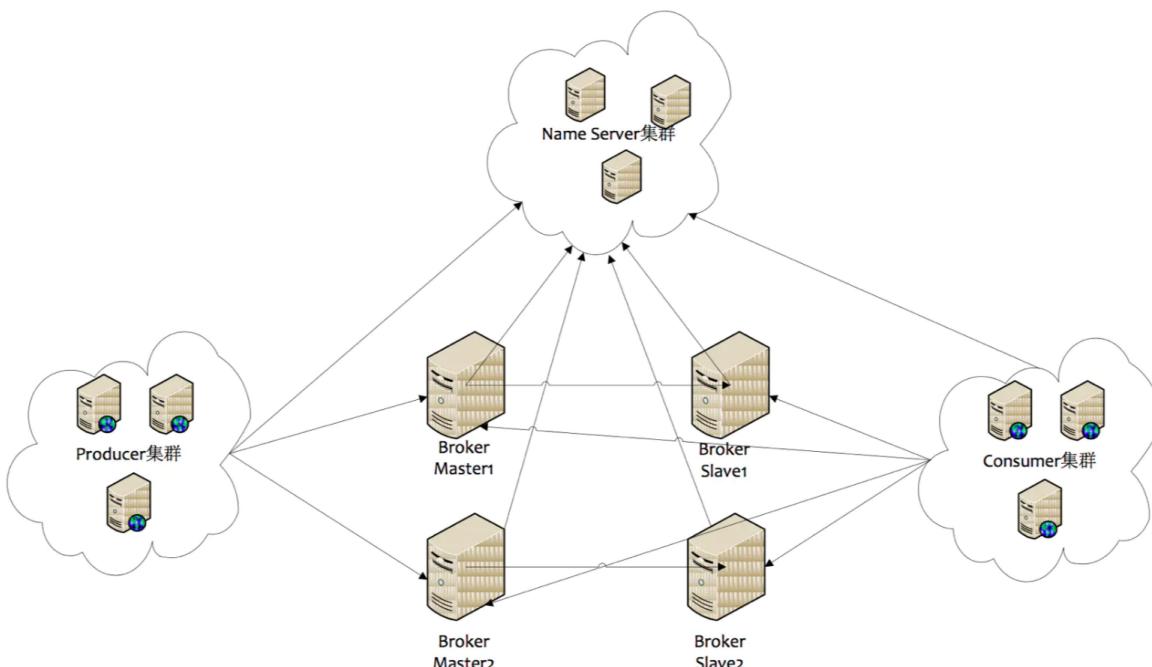
`SYNC_FLUSH`（同步刷新）相比于`ASYNC_FLUSH`（异步处理）会损失很多性能，但是也更可靠，所以需要根据实际的业务场景做好权衡。

3.3.3 Broker 配置

参数名	默认值	说明
listenPort	10911	接受客户端连接的监听端口
namesrvAddr	null	nameServer 地址
brokerIP1	网卡的 InetAddress	当前 broker 监听的 IP
brokerIP2	跟 brokerIP1 一样	存在主从 broker 时, 如果在 broker 主节点上配置了 brokerIP2 属性, broker 从节点会连接主节点配置的 brokerIP2 进行同步
brokerName	null	broker 的名称
brokerClusterName	DefaultCluster	本 broker 所属的 Cluster 名称
brokerId	0	broker id, 0 表示 master, 其他的正整数表示 slave
storePathCommitLog	\$HOME/store/commitlog/	存储 commit log 的路径
storePathConsumerQueue	\$HOME/store/consumequeue/	存储 consume queue 的路径
mappedFileSizeCommitLog	1024 * 1024 * 1024(1G)	commit log 的映射文件大小
deleteWhen	04	在每天的什么时间删除已经超过文件保留时间的 commit log
fileReservedTime	72	以小时计算的文件保留时间
brokerRole	ASYNC_MASTER	SYNC_MASTER/ASYNC_MASTER/SLAVE
flushDiskType	ASYNC_FLUSH	SYNC_FLUSH/ASYNC_FLUSH SYNC_FLUSH 模式下的 broker 保证在收到确认生产者之前将消息刷盘。ASYNC_FLUSH 模式下的 broker 则利用刷盘一组消息的模式, 可以取得更好的性能。

3.4 NameServer

RocketMQ的架构图：



NameServer的设计：

1. NameServer互相独立，彼此没有通信关系，单台NameServer挂掉，不影响其他NameServer。
2. NameServer不去连接别的机器，不主动推消息。
3. 单个Broker (Master、Slave) 与所有NameServer进行定时注册，以便告知NameServer自己还活着。

Broker每隔30秒向所有NameServer发送心跳，心跳包含了自身的topic配置信息。

NameServer每隔10秒，扫描所有还存活的broker连接，如果某个连接的最后更新时间与当前时间差值超过2分钟，则断开此连接，NameServer也会断开此broker下所有与slave的连接。同时更新topic与队列的对应关系，但不通知生产者和消费者。

Broker slave 同步或者异步从Broker master 上拷贝数据。

4. Consumer随机与一个NameServer建立长连接，如果该NameServer断开，则从NameServer列表中查找下一个进行连接。

Consumer主要从NameServer中根据Topic查询Broker的地址，查到就会缓存到客户端，并向提供Topic服务的Master、Slave建立长连接，且定时向Master、Slave发送心跳。

如果Broker宕机，则NameServer会将其剔除，而**Consumer端的定时任务**

`MQClientInstance.this.updateTopicRouteInfoFromNameServer` 每30秒执行一次，将Topic对应的Broker地址拉取下来，此地址只有Slave地址了，此时Consumer从Slave上消费。

消费者与Master和Slave都建有连接，在不同场景有不同的消费规则。

5. Producer随机与一个NameServer建立长连接，每隔30秒（此处时间可配置）从NameServer获取Topic的最新队列情况，如果某个Broker Master宕机，Producer最多30秒才能感知，在这个期间，发往该broker master的消息失败。Producer向提供Topic服务的Master建立长连接，且定时向Master发送心跳。

生产者与所有的master连接，但不能向slave写入。

客户端是先从NameServer寻址的，得到可用Broker的IP和端口信息，然后据此信息连接broker。

综上所述，NameServer在RocketMQ中的作用：

1. NameServer 用来保存活跃的 broker 列表，包括 Master 和 Slave 。
2. NameServer 用来保存所有 topic 和该 topic 所有队列的列表。
3. NameServer 用来保存所有 broker 的 Filter 列表。
4. 命名服务器为客户端，包括生产者，消费者和命令行客户端提供最新的路由信息。

RocketMQ为什么不使用ZooKeeper而自己开发NameServer？

在服务发现领域，ZooKeeper 根本就不能算是最佳的选择。

1. 注册中心是CP还是AP系统？

在分布式系统中，即使是对等部署的服务，因为请求到达的时间，硬件的状态，操作系统的调度，虚拟机的GC等，任何一个时间点，这些对等部署的节点状态也不可能完全一致，而流量不一致的情况下，只要注册中心在A承诺的时间内（例如1s内）将数据收敛到一致状态（即满足最终一致），流量将很快趋于统计学意义上的一致，所以注册中心以最终一致的模型设计在生产实践中完全可以接受。

2. 分区容忍及可用性需求分析

实践中，注册中心不能因为自身的任何原因破坏服务之间本身的可连通性，这是注册中心设计应该遵循的铁律！

在CAP的权衡中，注册中心的可用性比数据强一致性更宝贵，所以整体设计更应该偏向AP，而非CP，数据不一致在可接受范围，而P下舍弃A却完全违反了注册中心不能因为自身的任何原因破坏服务本身的可连通性的原则。

3. 服务规模、容量、服务联通性

当数据中心服务规模超过一定数量，作为注册中心的ZooKeeper性能堪忧。

在服务发现和健康监测场景下，随着服务规模的增大，无论是应用频繁发布时的服务注册带来的写请求，还是刷毫秒级的服务健康状态带来的写请求，还是恨不得整个数据中心的机器或者容器皆与注册中心有长连接带来的连接压力上，ZooKeeper很快就会力不从心，而ZooKeeper的写并不是可扩展的，不可通过加节点解决水平扩展性问题。

4. 注册中心需要持久存储和事务日志么？需要，也需要。

在服务发现场景中，其最核心的数据——实时的健康的服务的地址列表，真的需要数据持久化么？不需要

在服务发现中，服务调用发起方更关注的是其要调用的服务的实时的地址列表和实时健康状态，每次发起调用时，并不关心要调用的服务的历史服务地址列表、过去的健康状态。

但是一个完整的生产可用的注册中心，除了服务的实时地址列表以及实时的健康状态之外，还会存储一些服务的元数据信息，例如服务的版本，分组，所在的数据中心，权重，鉴权策略信息，服务标签等元数据，这些数据需要持久化存储，并且注册中心应该提供对这些元数据的检索的能力。

5. 服务健康检查

使用ZooKeeper作为服务注册中心时，服务的健康检测绑定在了ZooKeeper对于Session的健康监测上，或者说绑定在TCP长链接活性探测上了。

ZK与服务提供者机器之间的TCP长链接活性探测正常的时候，该服务就是健康的么？答案当然是否定的！注册中心应该提供更丰富的健康监测方案，服务的健康与否的逻辑应该开放给服务提供方自己定义，而不是一刀切搞成了TCP活性检测！

健康检测的一大基本设计原则就是尽可能真实的反馈服务本身的真实健康状态，否则一个不敢被服务调用者相信的健康状态判定结果还不如没有健康检测。

6. 注册中心的容灾考虑

如果注册中心（Registry）本身完全宕机了，服务调用链路应该受到影响么？

不应该受到影响。

服务调用（请求响应流）链路应该是弱依赖注册中心，必须仅在服务发布，机器上下线，服务扩缩容等必要时才依赖注册中心。

这需要注册中心仔细的设计自己提供的客户端，客户端中应该有针对注册中心服务完全不可用时做容灾的手段，例如设计客户端缓存数据机制就是行之有效的手段。另外，注册中心的健康检查机制也要仔细设计以便在这种情况下不会出现诸如推空等情况的出现。

ZooKeeper的原生客户端并没有这种能力，所以利用ZooKeeper实现注册中心的时候我们一定要问自己，如果把ZooKeeper所有节点全干掉，你生产上的所有服务调用链路能不受任何影响么？

7. 你有没有ZooKeeper的专家可依靠？

1. 难以掌握的Client/Session状态机
2. 难以承受的异常处理

阿里巴巴是不是完全没有使用 ZooKeeper？并不是。

熟悉阿里巴巴技术体系的都知道，其实阿里巴巴维护了目前国内最大规模的ZooKeeper集群，整体规模有近千台的ZooKeeper服务节点。

在粗粒度分布式锁，分布式选主，主备高可用切换等不需要高TPS支持的场景下有不可替代的作用，而这些需求往往多集中在大数据、离线任务等相关的业务领域，因为大数据领域，讲究分割数据集，并且大部分时间分任务多进程/线程并行处理这些数据集，但是总是有一些点上需要将这些任务和进程统一协调，这时候就是ZooKeeper发挥巨大作用的用武之地。

但是在交易场景交易链路上，在主营业务数据存取，大规模服务发现、大规模健康监测等方面有天然的短板，应该竭力避免在这些场景下引入ZooKeeper，在阿里巴巴的生产实践中，应用对ZooKeeper申请使用的时候要进行严格的场景、容量、SLA需求的评估。

对于ZooKeeper，大数据使用，服务发现不用。

3.5 客户端配置

相对于RocketMQ的Broker集群，生产者和消费者都是客户端。

本节主要描述生产者和消费者公共的行为配置。

DefaultMQProducer、TransactionMQProducer、DefaultMQPushConsumer、DefaultMQPullConsumer都继承于ClientConfig类，ClientConfig为客户端的公共配置类。

客户端的配置都是get、set形式，每个参数都可以用spring来配置，也可以在代码中配置。

例如namesrvAddr这个参数可以这样配置，producer.setNamesrvAddr("192.168.0.1:9876")，其他参数同理。

3.5.1 客户端寻址方式

RocketMQ可以令客户端找到Name Server，然后通过Name Server再找到Broker。如下所示有多种配置方式，优先级由高到低，高优先级会覆盖低优先级。

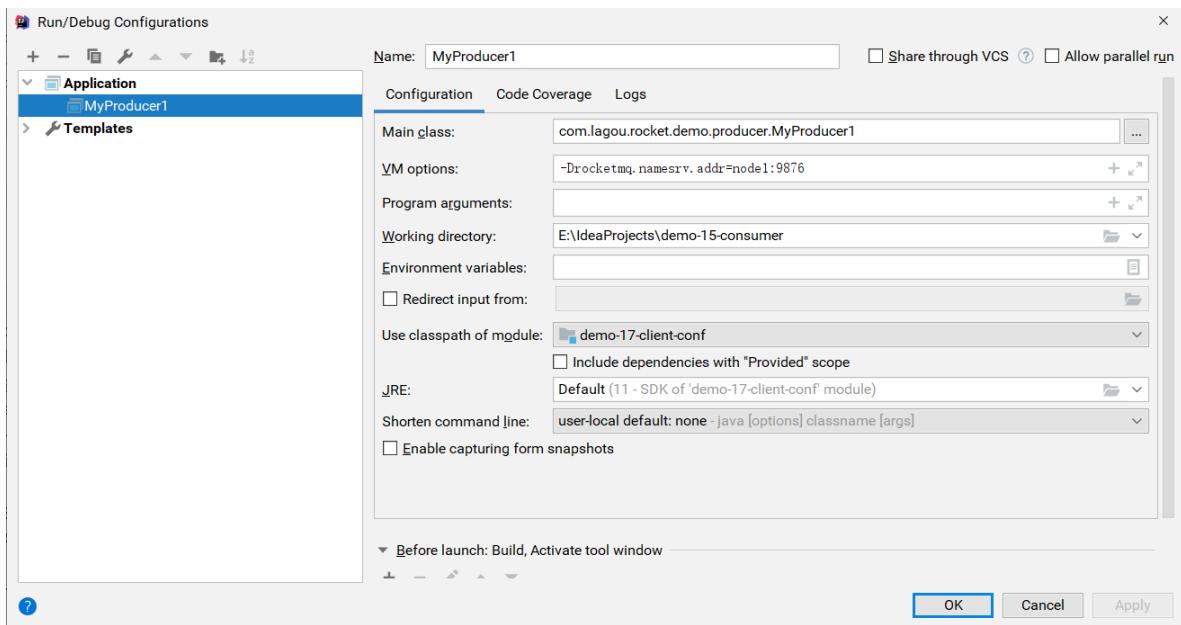
- 代码中指定Name Server地址，多个namesrv地址之间用**分号**分割

```
producer.setNamesrvAddr("192.168.0.1:9876;192.168.0.2:9876");
```

```
consumer.setNamesrvAddr("192.168.0.1:9876;192.168.0.2:9876");
```

- Java启动参数中指定Name Server地址

```
-Drocketmq.namesrv.addr=192.168.0.1:9876;192.168.0.2:9876
```

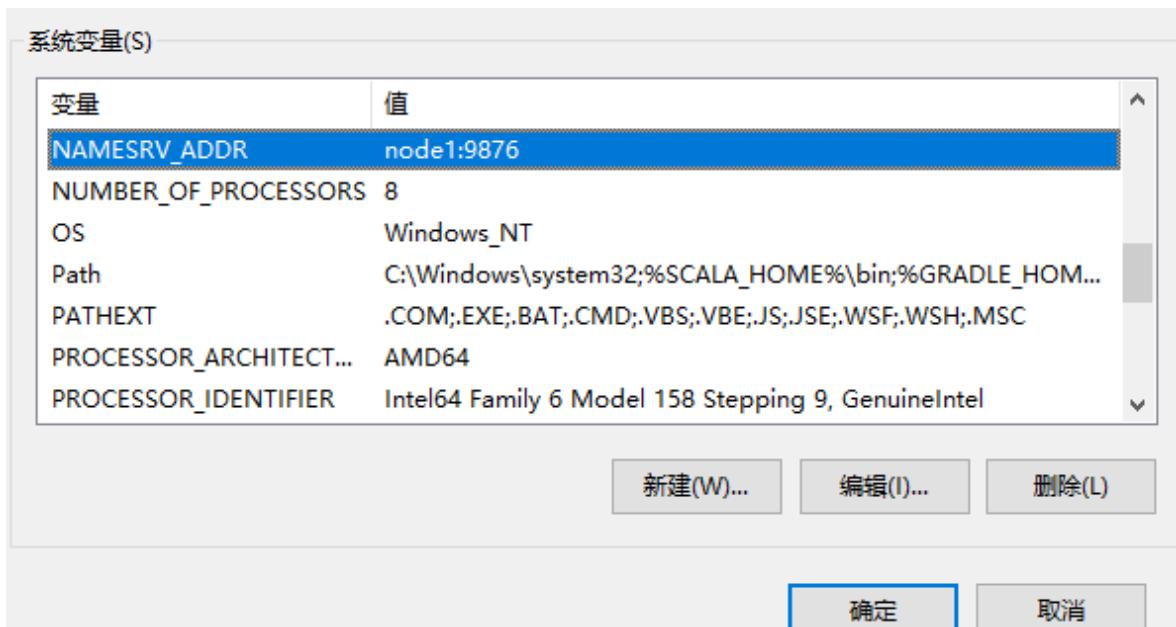


代码：

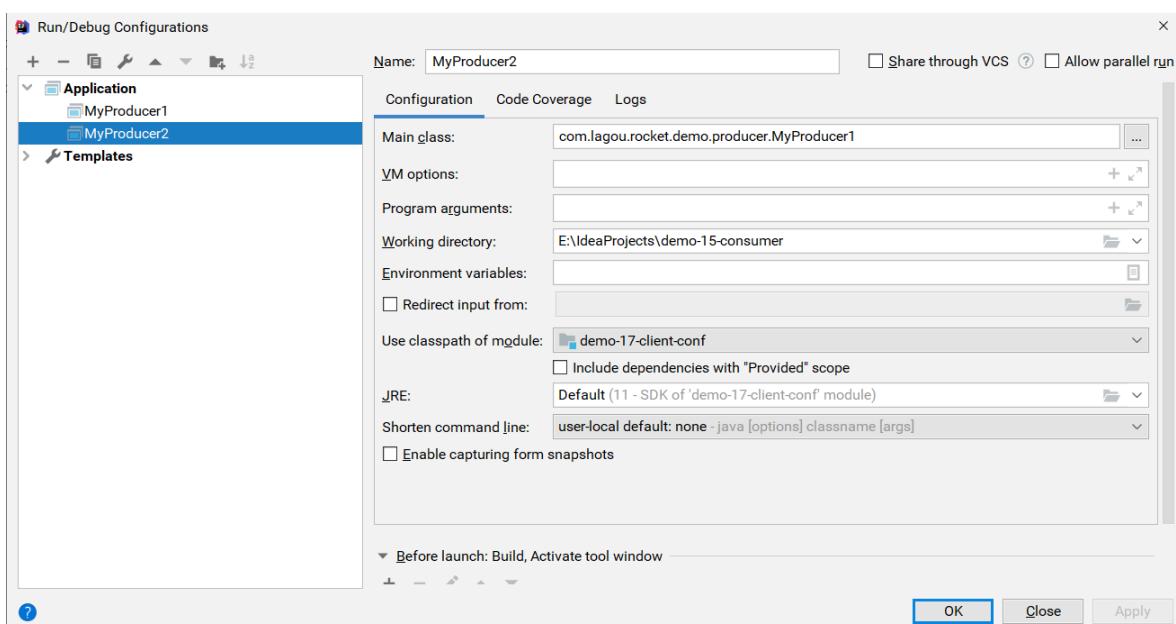
```
1 package com.lagou.rocket.demo.producer;
2
3 import org.apache.rocketmq.client.exception.MQBrokerException;
4 import org.apache.rocketmq.client.exception.MQClientException;
5 import org.apache.rocketmq.client.producer.DefaultMQProducer;
6 import org.apache.rocketmq.client.producer.SendResult;
7 import org.apache.rocketmq.common.message.Message;
8 import org.apache.rocketmq.remoting.exception.RemotingException;
9
10 public class MyProducer1 {
11     public static void main(String[] args) throws MQClientException,
12     RemotingException, InterruptedException, MQBrokerException {
13         DefaultMQProducer producer = new
14             DefaultMQProducer("producer_grp_17_02");
15         // 不使用代码指定，在启动参数中指定
16         // producer.setNamesrvAddr("node1:9876");
17
18         producer.start();
19
20         Message message = new Message("tp_demo_17", "hello
21 Tagou".getBytes());
22         SendResult sendResult = producer.send(message);
23
24         System.out.println(sendResult.getSendStatus());
25         System.out.println(sendResult.getMsgId());
26         System.out.println(sendResult.getOffsetMsgId());
27
28         producer.shutdown();
29     }
30 }
```

- 环境变量指定Name Server地址

```
export NAMESRV_ADDR=192.168.0.1:9876;192.168.0.2:9876
```



代码同上，但是需要重启Idea，一定是关闭之后再启动，让它加载该环境变量。



上图中没有设置VM Options，但是设置了环境变量，运行代码看效果：

The screenshot shows the IntelliJ IDEA interface with the 'Run' tab selected. The title bar says 'Run: MyProducer2'. The main area displays the application's standard output. The output consists of several 'WARNING' messages in red text, followed by some binary data and a success message. At the bottom, it says 'Process finished with exit code 0'.

```
"D:\RunningApps\JetBrains\IntelliJ IDEA 2019.  
WARNING: An illegal reflective access operation has occurred.  
WARNING: Illegal reflective access by io.netty.util.concurrent.DefaultPromise (at DefaultPromise.java:100) to method java.util.concurrent.CompletableFuture<java.lang.Object>.getNow()  
WARNING: Please consider reporting this to the maintainers of io.netty.  
WARNING: Use --illegal-access=warn to enable warning.  
WARNING: All illegal access operations will be reported.  
SEND_OK  
0A4E00A739B078308DB156311ED90000  
C0A8646500002A9F00000000001087C1  
  
Process finished with exit code 0
```

- HTTP静态服务器寻址（默认）

该静态地址，客户端第一次会10s后调用，然后每个2分钟调用一次。

客户端启动后，会定时访问一个静态HTTP服务器，地址如下：<http://jmenv.tbsite.net:8080/rocketmq/nsaddr>，这个URL的返回内容如下：

192.168.0.1:9876;192.168.0.2:9876

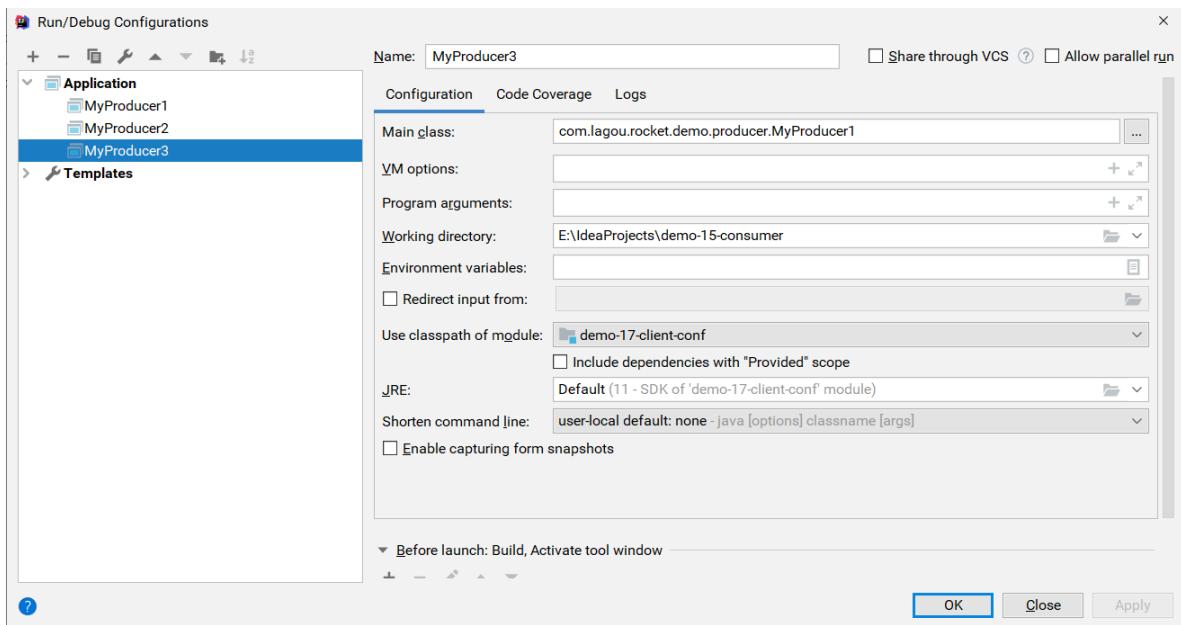
源码：

org.apache.rocketmq.common.MixAll.java中：

The screenshot shows the code editor for the 'MixAll.java' file. The cursor is at line 111, which contains the return statement 'return wsAddr;'. The code implements a static method 'getWSAddr' that constructs a URL by combining properties from the system and default values. It handles cases where the domain name contains a port number.

```
public static String getWSAddr() {  
    // 首先到系统中查找"rocketmq.namesrv.domain"名字的属性：我们的设置是：node1:9876  
    // 如果没有，则使用DEFAULT_NAMESRV_ADDR_LOOKUP的值：jmenv.tbsite.net  
    String wsDomainName = System.getProperty("rocketmq.namesrv.domain", DEFAULT_NAMESRV_ADDR_LOOKUP);  
    // 到系统中查找"rocketmq.namesrv.domain.subgroup"名字属性的值，如果没有则使用nsaddr  
    String wsDomainSubgroup = System.getProperty("rocketmq.namesrv.domain.subgroup", def: "nsaddr");  
    // http://jmenv.tbsite.net:8080/rocketmq/nsaddr  
    String wsAddr = "http://" + wsDomainName + ":" + wsDomainSubgroup;  
  
    // 如果wsDomainName中包含冒号，表示已经包含了端口号了，则直接拼接，不用默认的8080  
    if (wsDomainName.indexOf(":") > 0) {  
        wsAddr = "http://" + wsDomainName + "/rocketmq/" + wsDomainSubgroup;  
    }  
    // 返回  
    return wsAddr;  
}
```

没有设置VM Options，也没有设置环境变量。



同时开发一个服务，让客户端来访问：

```
6 @RestController
7 public class NameServerController {
8
9     @RequestMapping("/rocketmq/nsaddr")
10    public String getName() {
11        return "192.168.100.101:9876";
12    }
13
14 }
```

启动springboot项目，访问浏览器：

localhost:8080/rocketmq/nsaddr

192.168.100.101:9876

修改/etc/hosts文件：

```
33 # http://jmenv.tbsite.net:8080/rocketmq/nsaddr
34
35 127.0.0.1 jmenv.tbsite.net
```

在不设置nameserver地址时，依然可以访问，发送消息。

Run: DemoNameserverAddrApplication MyProducer3

```
"D:\RunningApps\JetBrains\IntelliJ IDEA 2019.2.3\bin\java.exe" -Didea.version=2019.2.3 -Dfile.encoding=UTF-8 -classpath "D:\RunningApps\JetBrains\IntelliJ IDEA 2019.2.3\lib\idea_rt.jar" com.intellij.rt.execution.application.AppMain DemoNameserverAddrApplication
```

WARNING: An illegal reflective access operation has occurred

WARNING: Illegal reflective access by io.reflectasm.Reflectasm (file: D:\RunningApps\JetBrains\IntelliJ IDEA 2019.2.3\lib\reflectasm-0.9.1.jar) found access to field org.apache.kafka.common.internals.ConsumerRecord.value

WARNING: Please consider reporting this to the maintainers of io.reflectasm

WARNING: Use --illegal-access=warn to enable warning for all illegal reflective access operations

WARNING: All illegal access operations will be denied in a future release

SEND_OK

0A4E00A7228878308DB1565610CA0000

C0A8646500002A9F0000000000108865

Process finished with exit code 0

推荐使用HTTP静态服务器寻址方式，好处是客户端部署简单，且Name Server集群可以热升级。因为只需要修改域名解析，客户端不需要重启。

3.5.2 客户端的公共配置

参数名	默认值	说明
namesrvAddr		Name Server地址列表，多个NameServer 地址用分号隔开
clientIP	本机IP	客户端本机IP地址，某些机器会发生无法识别客户端IP地址情况，需要应用在代码中强制指定
instanceName	DEFAULT	客户端实例名称，客户端创建的多个 Producer、Consumer实际是共用一个内部实例（这个实例包含网络连接、线程资源等）
clientCallbackExecutorThreads	4	通信层异步回调线程数
pollNameServerInterval	30000	轮询Name Server间隔时间，单位毫秒
heartbeatBrokerInterval	30000	向Broker发送心跳间隔时间，单位毫秒
persistConsumerOffsetInterval	5000	持久化Consumer消费进度间隔时间，单位毫秒

3.5.3 Producer配置

参数名	默认值	说明
producerGroup	DEFAULT_PRODUCER	Producer组名，多个Producer如果属于一个应用，发送同样的消息，则应该将它们归为同一组
createTopicKey	TBW102	在发送消息时，自动创建服务器不存在的topic，需要指定Key，该Key可用于配置发送消息所在topic的默认路由。
defaultTopicQueueNums	4	在发送消息，自动创建服务器不存在的topic时，默认创建的队列数
sendMsgTimeout	10000	发送消息超时时间，单位毫秒
compressMsgBodyOverHowmuch	4096	消息Body超过多大开始压缩 (Consumer收到消息会自动解压 缩)，单位字节
retryAnotherBrokerWhenNotStoreOK	FALSE	如果发送消息返回sendResult，但是sendStatus!=SEND_OK，是否重试发送
retryTimesWhenSendFailed	2	如果消息发送失败，最大重试次数，该参数只对同步发送模式起作用
maxMessageSize	4MB	客户端限制的消息大小，超过报错，同时服务端也会限制，所以需要跟服务端配合使用。
transactionCheckListener		事务消息回查监听器，如果发送事务消息，必须设置
checkThreadPoolMinSize	1	Broker回查Producer事务状态时，线程池最小线程数
checkThreadPoolMaxSize	1	Broker回查Producer事务状态时，线程池最大线程数
checkRequestHoldMax	2000	Broker回查Producer事务状态时，Producer本地缓冲请求队列大小
RPCHook	null	该参数是在Producer创建时传入的，包含消息发送前的预处理和消息响应后的处理两个接口，用户可以在第一个接口中做一些安全控制或者其他操作。

3.5.4 PushConsumer配置

参数名	默认值	说明
consumerGroup	DEFAULT_CONSUMER	Consumer组名，多个Consumer如果属于一个应用，订阅同样的消息，且消费逻辑一致，则应该将它们归为同一组
messageModel	CLUSTERING	消费模型支持集群消费和广播消费两种
consumeFromWhere	CONSUME_FROM_LAST_OFFSET	Consumer启动后，默认从上次消费的位置开始消费，这包含两种情况：一种是上次消费的位置未过期，则消费从上次中止的位置进行；一种是上次消费位置已经过期，则从当前队列第一条消息开始消费
consumeTimestamp	半个小时前	只有当consumeFromWhere值为CONSUME_FROM_TIMESTAMP时才起作用。
allocateMessageQueueStrategy	AllocateMessageQueueAveragely	Rebalance算法实现策略
subscription		订阅关系
messageListener		消息监听器
offsetStore		消费进度存储
consumeThreadMin	10	消费线程池最小线程数
consumeThreadMax	20	消费线程池最大线程数
consumeConcurrentlyMaxSpan	2000	单队列并行消费允许的最大跨度
pullThresholdForQueue	1000	拉消息本地队列缓存消息最大数
pullInterval	0	拉消息间隔，由于是长轮询，所以为0，但是如果应用为了流控，也可以设置大于0的值，单位毫秒
consumeMessageBatchMaxSize	1	批量消费，一次消费多少条消息
pullBatchSize	32	批量拉消息，一次最多拉多少条

3.5.5 PullConsumer配置

参数名	默认值	说明
consumerGroup	DEFAULT_CONSUMER	Consumer组名，多个Consumer如果属于一个应用，订阅同样的消息，且消费逻辑一致，则应该将它们归为同一组
brokerSuspendMaxTimeMillis	20000	长轮询，Consumer拉消息请求在Broker挂起最长时间，单位毫秒
consumerTimeoutMillisWhenSuspend	30000	长轮询，Consumer拉消息请求在Broker挂起超过指定时间，客户端认为超时，单位毫秒
consumerPullTimeoutMillis	10000	非长轮询，拉消息超时时间，单位毫秒
messageModel	BROADCASTING	消息支持两种模式：集群消费和广播消费
messageQueueListener		监听队列变化
offsetStore		消费进度存储
registerTopics		注册的topic集合
allocateMessageQueueStrategy	AllocateMessageQueueAveragely	Rebalance算法实现策略

3.5.6 Message数据结构

字段名	默认值	说明
Topic	null	必填，消息所属topic的名称
Body	null	必填，消息体
Tags	null	选填，消息标签，方便服务器过滤使用。目前只支持每个消息设置一个tag
Keys	null	选填，代表这条消息的业务关键词，服务器会根据keys创建哈希索引，设置后，可以在Console系统根据Topic、Keys来查询消息，由于是哈希索引，请尽可能保证key唯一，例如订单号，商品Id等。
Flag	0	选填，完全由应用来设置，RocketMQ不做干预
DelayTimeLevel	0	选填，消息延时级别，0表示不延时，大于0会延时特定的时间才会被消费
WaitStoreMsgOK	TRUE	选填，表示消息是否在服务器落盘后才返回应答。

3.6 系统配置

本小节主要介绍系统 (JVM/OS) 相关的配置。

3.6.1 JVM选项

设置Xms和Xmx一样大，防止JVM重新调整堆空间大小影响性能。

```
1 | -server -Xms8g -Xmx8g -Xmn4g
```

设置DirectByteBuffer内存大小。当DirectByteBuffer占用达到这个值，就会触发Full GC。

```
1 | -XX:MaxDirectMemorySize=15g
```

如果不太关心RocketMQ的启动时间，可以设置pre-touch，这样在JVM启动的时候就会分配完整的页空间。

```
1 | -XX:+AlwaysPreTouch
```

禁用偏向锁可能减少JVM的停顿，因为偏向锁在线程需要获取锁之前会判断当前线程是否拥有锁，如果拥有，就不用再去获取锁了。

在并发小的时候使用偏向锁有利于提升JVM效率，在高并发场合禁用掉。

```
1 | -XX:-UseBiasedLocking
```

推荐使用JDK1.8的G1垃圾回收器：

当在G1的GC日志中看到 `to-space overflow` 或者 `to-space exhausted` 的时候，表示G1没有足够的内存使用的（可能是 survivor 区不够了，可能是老年代不够了，也可能是两者都不够了），这时候表示Java堆占用大小已经达到了最大值。比如：924.897: [GC pause (G1 Evacuation Pause) (mixed) (to-space exhausted), 0.1957310 secs] | 924.897: [GC pause (G1 Evacuation Pause) (mixed) (to-space overflow), 0.1957310 secs] 为了解决这个问题，请尝试做以下调整：

1. 增加预留内存：增大参数 `-XX:G1ReservePercent` 的值（相应的增加堆内存）来增加预留内存；
2. 更早的开始标记周期：减小 `-XX:InitiatingHeapOccupancyPercent` 参数的值，以更早的开始标记周期；
3. 增加并发收集线程数：增大 `-XX:ConcGCThreads` 参数值，以增加并行标记线程数。

对G1而言，大小超过region大小50%的对象将被认为是大对象，这种大对象将直接被分配到老年代的humongous regions中，humongous regions是连续的region集合，StartsHumongous表记集合从那里开始，ContinuesHumongous标记连续集合。

在分配大对象之前，将会检查标记阈值，如果有必要的话，还会启动并发周期。

死亡的大对象会在标记周期的清理阶段和发生Full GC的时候被清理。

为了减少复制开销，任何转移阶段都不包含大对象的复制。在Full GC时，G1在原地压缩大对象。

因为每个独立的humongous regions只包含一个大对象，因此从大对象的结尾到它占用的最后一个region的结尾的那部分空间时没有被使用的，对于那些大小略大于region整数倍的对象，这些没有被使用的内存将导致内存碎片化。

如果你看到因为大对象的分配导致不断的启动并发收集，并且这种分配使得老年代碎片化不断加剧，那么请增加`-XX:G1HeapRegionSize`参数的值，这样的话，大对象将不再被G1认为是大对象，它会走普通对象的分配流程。

```
1 # G1回收器将堆空间划分为1024个region，此选项指定堆空间region的大小
2 -XX:+UseG1GC -XX:G1HeapRegionSize=16m
3 -XX:G1ReservePercent=25
4 -XX:InitiatingHeapOccupancyPercent=30
```

上述设置可能有点儿激进，但是对于生产环境，性能很好。

`-XX:MaxGCPauseMillis`不要设置的太小，否则VM会使用小的年轻代空间以达到此设置的值，同时引起很频繁的minor GC。

推荐使用GC log文件：

```
1 -XX:+UseGCLLogFileRotation
2 -XX:NumberOfGCLogFiles=5
3 -XX:GCLogFileSize=30m
```

如果写GC文件增加了Broker的延迟，可以考虑将GC log文件写到内存文件系统：

```
1 | -xloggc:/dev/shm/mq_gc_%p.log123
```

3.6.2 Linux内核参数

os.sh脚本在bin文件夹中列出了许多内核参数，可以进行微小的更改然后用于生产用途。下面的参数需要注意，更多细节请参考/proc/sys/vm/*的[文档](#)

```
1 | # 获取内核参数值  
2 | sysctl vm.extra_free_kbytes  
3 | # 设置内核参数值  
4 | sudo sysctl -w vm.overcommit_memory=1
```

- **vm.extra_free_kbytes**, 告诉VM在后台回收 (kswapd) 启动的阈值与直接回收 (通过分配进程) 的阈值之间保留额外的可用内存。RocketMQ使用此参数来避免内存分配中的长延迟。(与具体内核版本相关)
- **vm.min_free_kbytes**, 如果将其设置为低于1024KB, 将会巧妙的将系统破坏, 并且系统在高负载下容易出现死锁。
- **vm.max_map_count**, 限制一个进程可能具有的最大内存映射区域数。RocketMQ将使用 mmap加载CommitLog和ConsumeQueue, 因此建议将为此参数设置较大的值。
- **vm.swappiness**, 定义内核交换内存页面的积极程度。较高的值会增加攻击性, 较低的值会减少交换量。建议将值设置为10来避免交换延迟。
- **File descriptor limits**, RocketMQ需要为文件 (CommitLog和ConsumeQueue) 和网络连接打开文件描述符。我们建议设置文件描述符的值为655350。

```
1 | echo '* hard nofile 655350' >> /etc/security/limits.conf
```

- [Disk scheduler](#), RocketMQ建议使用I/O截止时间调度器, 它试图为请求提供有保证的延迟。

```
echo 'deadline' > /sys/block/${DISK}/queue/scheduler
```

3.7 动态扩缩容

3.7.1 动态增减Namesrv机器

NameServer是RocketMQ集群的协调者，集群的各个组件是通过NameServer获取各种属性和地址信息的。

主要功能包括两部分：

1. 一个各个Broker定期上报自己的状态信息到NameServer；
2. 另一个是各个客户端，包括Producer、Consumer，以及命令行工具，通过NameServer获取最新的状态信息。

所以，在启动Broker、生产者和消费者之前，必须告诉它们NameServer的地址，为了提高可靠性，建议启动多个NameServer。NameServer占用资源不多，可以和Broker部署在同一台机器。有多个NameServer后，减少某个NameServer不会对其他组件产生影响。

有四种种方式可设置NameServer的地址，下面按优先级由高到低依次介绍：

1) 通过代码设置，比如在Producer中，通过Producer.setNamesrvAddr ("name-server1-ip: port; name-server2-ip: port") 来设置。

在mqadmin命令行工具中，是通过-n name-server-ip1: port; name-server-ip2: port参数来设置的，如果自定义了命令行工具，也可以通过defaultMQAdminExt.setNamesrvAddr ("name-server1-ip: port; name-server2-ip: port") 来设置。

2) 使用Java启动参数设置，对应的option是rocketmq.namesrv.addr。

3) 通过Linux环境变量设置，在启动前设置变量：NAMESRV_ADDR。

4) 通过HTTP服务来设置，当上述方法都没有使用，程序会向一个HTTP地址发送请求来获取NameServer地址，默认的URL是<http://jmenv.tbsite.net:8080/rocketmq/nsaddr>（淘宝的测试地址），通过rocketmq.namesrv.domain参数来覆盖jmenv.tbsite.net；通过rocketmq.namesrv.domain.subgroup参数来覆盖nsaddr。

第4种方式看似繁琐，但它是唯一支持动态增加NameServer，无须重启其他组件的方式。使用这种方式后其他组件会每隔2分钟请求一次该URL，获取最新的NameServer地址。

3.7.2 动态增减Broker机器

由于业务增长，需要对集群进行扩容的时候，可以动态增加Broker角色的机器。只增加Broker不会对原有的Topic产生影响，原来创建好的Topic中数据的读写依然在原来的那些Broker上进行。

集群扩容后，一是可以把新建的Topic指定到新的Broker机器上，均衡利用资源；另一种方式是通过updateTopic命令更改现有的Topic配置，在新加的Broker上创建新的队列。比如TestTopic是现有的一个Topic，因为数据量增大需要扩容，新增的一个Broker机器地址是192.168.0.1: 10911，这个时候执行下面的命令：sh./bin/mqadmin updateTopic -b 192.168.0.1: 10911-t TestTopic-n 192.168.0.100: 9876，结果是在新增的Broker机器上，为TestTopic新创建了8个读写队列。

```
mqadmin updateTopic -b <arg> | -c <arg> [-h] [-n <arg>] [-o <arg>] [-p <arg>] [-r <arg>] [-s <arg>] -t <arg> [-u <arg>] [-w <arg>]
```

```
1 [root@node1 ~]# mqadmin topicStatus -n localhost:9876 -t tp_demo_07
2 [root@node1 ~]# mqadmin updateTopic -b node2:10911 -t tp_demo_07 -n
'node1:9876;node2:9876' -w 8 -r 8
3 [root@node1 ~]# mqadmin topicStatus -n localhost:9876 -t tp_demo_07
```

如果因为业务变动或者置换机器需要减少Broker，此时该如何操作呢？减少Broker要看是否有持续运行的Producer，当一个Topic只有一个Master Broker，停掉这个Broker后，消息的发送肯定会受到影响，需要在停止这个Broker前，停止发送消息。

当某个Topic有多个Master Broker，停了其中一个，这时候是否会丢失消息呢？答案和Producer使用的发送消息的方式有关，如果使用同步方式send (msg) 发送，在DefaultMQProducer内部有个自动重试逻辑，其中一个Broker停了，会自动向另一个Broker发消息，不会发生丢消息现象。如果使用异步方式发送send (msg, callback)，或者用sendOneWay方式，会丢失切换过程中的消息。因为在异步和sendOneWay这两种发送方式下，Producer.setRetryTimesWhenSendFailed设置不起作用，发送失败不会重试。DefaultMQProducer默认每30秒到NameServer请求最新的路由消息，Producer如果获取不到已停止的Broker下的队列信息，后续就自动不再向这些队列发送消息。

如果Producer程序能够暂停，在有一个Master和一个Slave的情况下也可以顺利切换。可以关闭Producer后关闭Master Broker，这个时候所有的读取都会被定向到Slave机器，消费消息不受影响。把Master Broker机器置换完后，基于原来的数据启动这个Master Broker，然后再启动Producer程序正常发送消息。

用Linux的kill pid命令就可以正确地关闭Broker，BrokerController下有个shutdown函数，这个函数被加到了ShutdownHook里，当用Linux的kill命令时（不能用kill-9），shutdown函数会先被执行。也可以通过RocketMQ提供的工具（mqshutdown broker）来关闭Broker，它们的原理是一样的。

3.8 各种故障对消息的影响

我们期望消息队列集群一直可靠稳定地运行，但有时候故障是难免的，本节我们列出可能的故障情况，看看如何处理：

- 1) Broker正常关闭，启动；
- 2) Broker异常Crash，然后启动；
- 3) OS Crash，重启；
- 4) 机器断电，但能马上恢复供电；
- 5) 磁盘损坏；
- 6) CPU、主板、内存等关键设备损坏。

假设现有的RocketMQ集群，每个Topic都配有多Master角色的Broker供写入，并且每个Master都至少有一个Slave机器（用两台物理机就可以实现上述配置），我们来看看在上述情况下消息的可靠性情况。

第1种情况属于可控的软件问题，内存中的数据不会丢失。如果重启过程中有持续运行的Consumer，Master机器出故障后，Consumer会自动重连到对应的Slave机器，不会有消息丢失和偏差。当Master角色的机器重启以后，Consumer又会重新连接到Master机器（注意在启动Master机器的时候，如果Consumer正在从Slave消费消息，不要停止Consumer。假如此时先停止Consumer后再启动Master机器，然后再启动Consumer，这个时候Consumer就会去读Master机器上已经滞后的offset值，造成消息大量重复）。

如果第1种情况出现时有持续运行的Producer，一台Master出故障后，Producer只能向Topic下其他的Master机器发送消息，如果Producer采用同步发送方式，不会有消息丢失。

第2、3、4种情况属于软件故障，内存的数据可能丢失，所以刷盘策略不同，造成的影响也不同，如果Master、Slave都配置成SYNC_FLUSH，可以达到和第1种情况相同的效果。

第5、6种情况属于硬件故障，发生第5、6种情况的故障，原有机器的磁盘数据可能会丢失。如果Master和Slave机器间配置成同步复制方式，某一台机器发生5或6的故障，也可以达到消息不丢失的效果。如果Master和Slave机器间是异步复制，两次Sync间的消息会丢失。

总的来说，当设置成：

- 1) 多Master，每个Master带有Slave；
- 2) 主从之间设置成SYNC_MASTER；
- 3) Producer用同步方式写；
- 4) 刷盘策略设置成SYNC_FLUSH。

就可以消除单点依赖，即使某台机器出现极端故障也不会丢消息。

第四部分 RocketMQ集群与运维

4.1 集群搭建方式

4.1.1 集群特点

- NameServer是一个几乎无状态节点，可集群部署，节点之间无任何信息同步。

- Broker部署相对复杂，Broker分为Master与Slave，一个Master可以对应多个Slave，但是一个Slave只能对应一个Master，Master与Slave的对应关系通过指定相同的BrokerName，不同的BrokerId来定义，BrokerId为0表示Master，非0表示Slave。Master也可以部署多个。每个Broker与NameServer集群中的所有节点建立长连接，定时注册Topic信息到所有NameServer。
- Producer与NameServer集群中的其中一个节点（随机选择）建立长连接，定期从NameServer取Topic路由信息，并向提供Topic服务的Master建立长连接，且定时向Master发送心跳。Producer完全无状态，可集群部署。
- Consumer与NameServer集群中的其中一个节点（随机选择）建立长连接，定期从NameServer取Topic路由信息，并向提供Topic服务的Master、Slave建立长连接，且定时向Master、Slave发送心跳。Consumer既可以从Master订阅消息，也可以从Slave订阅消息，订阅规则由Broker配置决定。

4.1.2 集群模式

1) 单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，可以用于本地测试。

2) 多Master模式

一个集群无Slave，全是Master，例如2个Master或者3个Master，这种模式的优缺点如下：

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

3) 多Master多Slave模式（异步）

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），这种模式的优缺点如下：

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

4) 多Master多Slave模式（同步）

每个Master配置一个Slave，有多对Master-Slave，HA采用同步双写方式，即只有主备都写成功，才向应用返回成功，这种模式的优缺点如下：

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

4.2 集群的搭建

4.2.0 前置配置

1. 安装JDK 11.0.5
2. 修改RocketMQ的启动脚本:
 - bin/runserver.sh
 - bin/runbroker.sh
 - bin/tools.sh

bin/runserver.sh:

```
1 #!/bin/sh
2
3 # Licensed to the Apache Software Foundation (ASF) under one or more
4 # contributor license agreements. See the NOTICE file distributed with
5 # this work for additional information regarding copyright ownership.
6 # The ASF licenses this file to You under the Apache License, Version 2.0
7 # (the "License"); you may not use this file except in compliance with
8 # the License. You may obtain a copy of the License at
9 #
10 #     http://www.apache.org/licenses/LICENSE-2.0
11 #
12 # Unless required by applicable law or agreed to in writing, software
13 # distributed under the License is distributed on an "AS IS" BASIS,
14 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
15 # See the License for the specific language governing permissions and
16 # limitations under the License.
17
18 =====
=====
19 # Java Environment Setting
20 =====
=====
21 error_exit ()
22 {
23 echo "ERROR: $1 !"
24 exit 1
25 }
26
27 [ ! -e "$JAVA_HOME/bin/java" ] && JAVA_HOME=$HOME/jdk/java
28 [ ! -e "$JAVA_HOME/bin/java" ] && JAVA_HOME=/usr/java
29 [ ! -e "$JAVA_HOME/bin/java" ] && error_exit "Please set the JAVA_HOME
variable in your environment, we need java(x64)!"
30
31 export JAVA_HOME
32 export JAVA="$JAVA_HOME/bin/java"
33 export BASE_DIR=$(dirname $0)/..
34 export
35 CLASSPATH=.:${BASE_DIR}/conf:${JAVA_HOME}/jre/lib/ext:${BASE_DIR}/lib/*
36 =====
=====
37 # JVM Configuration
38 =====
```

```

39 JAVA_OPT="${JAVA_OPT} -server -Xms256m -Xmx256m -Xmn128m -
40 XX:Metaspacesize=64m -XX:MaxMetaspacesize=160m"
41 JAVA_OPT="${JAVA_OPT} -XX:CMSInitiatingOccupancyFraction=70 -
42 XX:+CMSParallelRemarkEnabled -XX:SoftRefLRUPolicyMSPerMB=0 -
43 XX:+CMSClassUnloadingEnabled -XX:SurvivorRatio=8"
44 JAVA_OPT="${JAVA_OPT} -verbose:gc -Xlog:gc:/dev/shm/rmq_srv_gc.log -
45 XX:+PrintGCDetails"
46 JAVA_OPT="${JAVA_OPT} -XX:-OmitStackTraceInFastThrow"
47 JAVA_OPT="${JAVA_OPT} -XX:-UseLargePages"
48 # JAVA_OPT="${JAVA_OPT} -
49 Djava.ext.dirs=${JAVA_HOME}/jre/lib/ext:${BASE_DIR}/lib"
50 #JAVA_OPT="${JAVA_OPT} -Xdebug -
51 Xrunjdwp:transport=dt_socket,address=9555,server=y,suspend=n"
52 JAVA_OPT="${JAVA_OPT} ${JAVA_OPT_EXT}"
53 JAVA_OPT="${JAVA_OPT} -cp ${CLASSPATH}"
54
55 $JAVA ${JAVA_OPT} $@

```

bin/runbroker.sh:

```

1 #!/bin/sh
2
3 # Licensed to the Apache Software Foundation (ASF) under one or more
4 # contributor license agreements. See the NOTICE file distributed with
5 # this work for additional information regarding copyright ownership.
6 # The ASF licenses this file to You under the Apache License, Version 2.0
7 # (the "License"); you may not use this file except in compliance with
8 # the License. You may obtain a copy of the License at
9 #
10 #     http://www.apache.org/licenses/LICENSE-2.0
11 #
12 # Unless required by applicable law or agreed to in writing, software
13 # distributed under the License is distributed on an "AS IS" BASIS,
14 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
15 # See the License for the specific language governing permissions and
16 # limitations under the License.
17
18 =====
19 # Java Environment Setting
20 =====
21 =====
22 error_exit () {
23     echo "ERROR: $1 !!""
24     exit 1
25 }
26
27 [ ! -e "$JAVA_HOME/bin/java" ] && JAVA_HOME=$HOME/jdk/java
28 [ ! -e "$JAVA_HOME/bin/java" ] && JAVA_HOME=/usr/java
29 [ ! -e "$JAVA_HOME/bin/java" ] && error_exit "Please set the JAVA_HOME
variable in your environment, we need java(x64)!"
30
31 export JAVA_HOME
32 export JAVA="$JAVA_HOME/bin/java"
33 export BASE_DIR=$(dirname $0)/..

```

```

34 export
35 CLASSPATH=.:${JAVA_HOME}/jre/lib/ext:${BASE_DIR}/lib/*:${BASE_DIR}/conf:${CL
ASSPATH}
36 #=====
37 # JVM Configuration
38 #=====
39 JAVA_OPT="${JAVA_OPT} -server -Xms256m -Xmx256m -Xmn128m"
40 JAVA_OPT="${JAVA_OPT} -XX:+UseG1GC -XX:G1HeapRegionSize=16m -
XX:G1ReservePercent=25 -XX:InitiatingHeapOccupancyPercent=30 -
XX:SoftRefLRUPolicyMSPerMB=0"
41 JAVA_OPT="${JAVA_OPT} -verbose:gc -Xloggc:/dev/shm/mq_gc_%p.log -
XX:+PrintGCDetails"
42 JAVA_OPT="${JAVA_OPT} -XX:-OmitStackTraceInFastThrow"
43 JAVA_OPT="${JAVA_OPT} -XX:+AlwaysPreTouch"
44 JAVA_OPT="${JAVA_OPT} -XX:MaxDirectMemorySize=15g"
45 JAVA_OPT="${JAVA_OPT} -XX:-UseLargePages -XX:-UseBiasedLocking"
46 #JAVA_OPT="${JAVA_OPT} -Xdebug -
Xrunjdwp:transport=dt_socket,address=9555,server=y,suspend=n"
47 JAVA_OPT="${JAVA_OPT} ${JAVA_OPT_EXT}"
48 JAVA_OPT="${JAVA_OPT} -cp ${CLASSPATH}"
49
50 numactl --interleave=all pwd > /dev/null 2>&1
51 if [ $? -eq 0 ]
52 then
53     if [ -z "$RMQ_NUMA_NODE" ] ; then
54         numactl --interleave=all $JAVA ${JAVA_OPT} @@
55     else
56         numactl --cpunodebind=$RMQ_NUMA_NODE --membind=$RMQ_NUMA_NODE $JAVA
$JAVA_OPT @@
57     fi
58 else
59     $JAVA ${JAVA_OPT} --add-exports=java.base/jdk.internal.ref=ALL-UNNAMED
@@
60 fi

```

bin/tools.sh:

```

1 #!/bin/sh
2
3 # Licensed to the Apache Software Foundation (ASF) under one or more
4 # contributor license agreements. See the NOTICE file distributed with
5 # this work for additional information regarding copyright ownership.
6 # The ASF licenses this file to You under the Apache License, Version 2.0
7 # (the "License"); you may not use this file except in compliance with
8 # the License. You may obtain a copy of the License at
9 #
10 #     http://www.apache.org/licenses/LICENSE-2.0
11 #
12 # Unless required by applicable law or agreed to in writing, software
13 # distributed under the License is distributed on an "AS IS" BASIS,
14 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
15 # See the License for the specific language governing permissions and
16 # limitations under the License.
17

```

```

18 #=====
19 =====
20 # Java Environment Setting
21 =====
22 =====
23 error_exit ()
24 {
25     echo "ERROR: $1 !!" 
26     exit 1
27 }
28 [ ! -e "$JAVA_HOME/bin/java" ] && JAVA_HOME=$HOME/jdk/java
29 [ ! -e "$JAVA_HOME/bin/java" ] && JAVA_HOME=/usr/java
30 [ ! -e "$JAVA_HOME/bin/java" ] && error_exit "Please set the JAVA_HOME
variable in your environment, we need java(x64)!"
31 export JAVA_HOME
32 export JAVA="$JAVA_HOME/bin/java"
33 export BASE_DIR=$(dirname $0)/..
34 # export CLASSPATH=.:${BASE_DIR}/conf:${CLASSPATH}
35 export
36 CLASSPATH=.:${JAVA_HOME}/jre/lib/ext:${BASE_DIR}/lib/*:${BASE_DIR}/conf:${CL
ASSPATH}
37 =====
38 # JVM Configuration
39 =====
40 JAVA_OPT="${JAVA_OPT} -server -Xms256m -Xmx256m -Xmn256m -XX:PermSize=128m
-XX:MaxPermSize=128m"
41 # JAVA_OPT="${JAVA_OPT} -
Djava.ext.dirs=${BASE_DIR}/lib:${JAVA_HOME}/jre/lib/ext"
42 JAVA_OPT="${JAVA_OPT} -cp ${CLASSPATH}"
43 $JAVA ${JAVA_OPT} $@

```

4.2.1 单Master模式

这是最简单也是风险最大的模式，一旦broker重启或宕机，整个服务不可用。不推荐在生产环境使用，一般用于开发或本地测试。

步骤如下：

4.2.1.1 启动NameServer

```

1 ### 首先启动NameServer
2 $ nohup sh mqnamesrv &
3
4 ### 检查Name Server是否启动成功
5 $ tail -f ~/logs/rocketmq/logs/namesrv.log
6 The Name Server boot success...

```

只要在namesrv.log中看到“The Name Server boot success..”就表示NameServer启动成功了。

4.2.1.2 启动Broker

```
1   ### 首先启动broker  
2   $ nohup sh mqbroker -n localhost:9876 &  
3  
4   ### 检查broker是否启动成功。如果看到broker的下面的语句表示启动成功  
5   $ tail -f ~/logs/rocketmq/logs/broker.log  
6   The broker[broker-a,192.169.1.2:10911] boot success...
```

如果在broker.log中看到“The broker[brokerName,ip:port] boot success..”就表示broker启动成功。

4.2.2 多Master模式

多Master模式意味着所有的节点都是Master节点，没有Slave节点。优缺点如下：

- 优点：

1. 配置简单。
2. 一个服务器节点的停用或重启（维护目的）不会对应用造成大的影响。
3. 如果磁盘配置了RAID10，就不会由消息的丢失。（因为RAID10非常可靠，即使服务器不可恢复，在异步刷盘时会丢失少量数据，同步刷盘不会丢数据）。
4. 该模式性能最好。

- 缺点：

1. 在单个节点停用期间，消费者无法消费该节点上的数据，也不能订阅该节点上的数据，除非节点恢复。消息的实时性受到影响。

启动步骤如下：

4.2.2.1 启动NameServer

```
1   ### 首先启动NameServer  
2   $ nohup sh mqnamesrv &  
3  
4   ### 检查NameServer是否启动成功  
5   $ tail -f ~/logs/rocketmq/logs/namesrv.log  
6   The Name Server boot success...
```

4.2.2.2 启动Broker集群

```
1   ### 启动第一个broker，假定namesrv在192.168.1.1上。注意这里的配置文件的位置  
2   $ nohup sh mqbroker -n 192.168.1.1:9876 -c $ROCKETMQ_HOME/conf/2m-  
noslave/broker-a.properties &  
3  
4   ### 在第二台服务器上启动另一个broker。  
5   $ nohup sh mqbroker -n 192.168.1.1:9876 -c $ROCKETMQ_HOME/conf/2m-  
noslave/broker-b.properties &  
6  
7   ...
```

上面的NameServer是一台，IP地址直接写，如果是多台NameServer，则需要在-n后接多个NameServer的地址，使用分号分开。由于shell对分号敏感，可以使用单引号引起多个NameServer的地址，禁止shell对分号的解释。

4.2.3 多Master和Slave模式-异步复制

每个Master节点配置一个或多个Slave节点，组成一组，有多组这样的组合组成集群。HA使用**异步复制**，Master和Slave节点间有毫秒级的消息同步延迟。优缺点如下：

- 优点:

1. 磁盘坏掉，会有少量消息丢失，但是消息的实时性不会受到影响。
2. 同时，如果Master宕机，消费者依然可以从Slave节点消费，并且这个转换是透明的，也无需运维手动介入。
3. 性能和多Master模式差不多，弱一点点。

- 缺点:

1. 如果Master宕机，磁盘坏掉，会丢失少量消息。

启动步骤如下：

4.2.3.1 启动NameServer

```
1  ### 首先启动NameServer
2  $ nohup sh mqnamesrv &
3
4  ### 检查NameServer是否启动成功
5  $ tail -f ~/logs/rocketmqlogs/namesrv.log
6  The Name Server boot success...
```

4.2.3.2 启动Broker集群

```
1  ### 启动Master的broker: broker-a
2  $ nohup sh mqbroker -n 192.168.1.1:9876 -c $ROCKETMQ_HOME/conf/2m-2s-
   async/broker-a.properties &
3
4  ### 在另外一台服务器上启动另一个Master的broker: broker-b
5  $ nohup sh mqbroker -n 192.168.1.1:9876 -c $ROCKETMQ_HOME/conf/2m-2s-
   async/broker-b.properties &
6
7  ### 在另一台服务器上启动broker-a的slave
8  $ nohup sh mqbroker -n 192.168.1.1:9876 -c $ROCKETMQ_HOME/conf/2m-2s-
   async/broker-a-s.properties &
9
10 ### 在另一台服务器上启动broker-b的slave
11 $ nohup sh mqbroker -n 192.168.1.1:9876 -c $ROCKETMQ_HOME/conf/2m-2s-
   async/broker-b-s.properties &
```

以上启动的是2M-2S-Async模式，双主双从，主从异步复制模式。

4.2.4 多Master和Slave模式-同步双写

该模式中，每个Master节点配置多个Slave节点，它们构成一组，多组构成集群。HA使用同步双写，即，只有消息在Master节点和多个Slave节点上写成功，才返回生产者消息发送成功。

ASYNC_MASTER

SYNC_MASTER

优缺点如下：

- 优点:

1. 数据和服务都没有单点故障
2. 当Master主机宕机，消息的消费不会延迟。
3. 服务和数据的高可用。

- 缺点:

1. 该模式的性能比异步复制的模式低10%左右。
2. 发送消息的延迟稍微高一点。
3. 当前版本中，如果Master节点宕机，Slave节点不能自动切换为Master模式。

启动步骤如下：

4.2.4.1 启动NameServer

```

1  ### 首先启动NameServer
2  $ nohup sh mqnamesrv &
3
4  ### 检查NameServer是否启动成功。
5  $ tail -f ~/logs/rocketmq/logs/namesrv.log
6  The Name Server boot success...

```

4.2.4.2 启动Broker集群

```

1  ### 在一个节点启动broker-a (MASTER)
2  $ nohup sh mqbroker -n 192.168.1.1:9876 -c $ROCKETMQ_HOME/conf/2m-2s-
sync/broker-a.properties &
3
4  ### 在另一个节点启动broker-b (MASTER)
5  $ nohup sh mqbroker -n 192.168.1.1:9876 -c $ROCKETMQ_HOME/conf/2m-2s-
sync/broker-b.properties &
6
7  ### 在另一个节点启动broker-a的同步slave节点: broker-a-s
8  $ nohup sh mqbroker -n 192.168.1.1:9876 -c $ROCKETMQ_HOME/conf/2m-2s-
sync/broker-a-s.properties &
9
10 ### 在另一个节点启动broker-b的同步slave节点: broker-b-s
11 $ nohup sh mqbroker -n 192.168.1.1:9876 -c $ROCKETMQ_HOME/conf/2m-2s-
sync/broker-b-s.properties &

```

上述配置中，通过相同的brokerName不同的brokerId将Master和Slave组合为一组。Master的brokerId必须是0，Slave的brokerId必须大于0，且不能相同。

作业：搭建包含3个NameServer，2m-2s-sync。

4.3 mqadmin管理工具

注意：

1. 执行命令方法：`./mqadmin {command} {args}`
2. 几乎所有命令都需要配置-n表示NameServer地址，格式为ip:port
3. 几乎所有命令都可以通过-h获取帮助
4. 如果既有Broker地址 (-b) 配置项又有clusterName (-c) 配置项，则优先以Broker地址执行命令，如果不配置Broker地址，则对集群中所有主机执行命令，只支持一个Broker地址。-b格式为ip:port，port默认是10911
5. 在tools下可以看到很多命令，但并不是所有命令都能使用，只有在MQAdminStartup中初始化的命令才能使用，你也可以修改这个类，增加或自定义命令
6. 由于版本更新问题，少部分命令可能未及时更新，遇到错误请直接阅读相关命令源码

4.3.1 Topic相关

启动集群:

```
1 [root@node1 ~]# nohup sh mqnamesrv &
2 [root@node1 ~]# nohup sh mqbroker -n node1:9876 -c /opt/rocketmq/conf/2m-2s-
sync/broker-a.properties &
3 [root@node2 ~]# nohup sh mqbroker -n node1:9876 -c /opt/rocketmq/conf/2m-2s-
sync/broker-a-s.properties &
4 [root@node3 ~]# nohup sh mqbroker -n node1:9876 -c /opt/rocketmq/conf/2m-2s-
sync/broker-b.properties &
5 [root@node4 ~]# nohup sh mqbroker -n node1:9876 -c /opt/rocketmq/conf/2m-2s-
sync/broker-b-s.properties &
```

4.3.1.1 命令列表

名称	含义	命令选项	说明
updateTopic	创建更新Topic配置	-b	Broker 地址, 表示 topic 所在 Broker, 只支持单台Broker, 地址为 ip:port
		-c	cluster 名称, 表示 topic 所在集群 (集群可通过 clusterList 查询)
		-h	打印帮助
		-n	NameServer服务地址, 格式 ip:port
		-p	指定新topic的读写权限(W=2\
		-r	可读队列数 (默认为 8)
		-w	可写队列数 (默认为 8)
		-t	topic 名称 (名称只能使用字符 ^[a-zA-Z0-9_-]+\$)
deleteTopic	删除Topic	-c	cluster 名称, 表示删除某集群下的某个 topic (集群 可通过 clusterList 查询)
		-h	打印帮助
		-n	NameServer 服务地址, 格式 ip:port
		-t	topic 名称 (名称只能使用字符 ^[a-zA-Z0-9_-]+\$)
topicList	查看 Topic 列表信息	-h	打印帮助
		-c	不配置-c只返回topic列表, 增加-c返回 clusterName, topic, consumerGroup 信息, 即topic的所属集群和订阅关系, 没有参数
		-n	NameServer 服务地址, 格式 ip:port
topicRoute	查看 Topic 路由信息	-t	topic 名称
		-h	打印帮助
		-n	NameServer 服务地址, 格式 ip:port
topicStatus	查看 Topic 消息队列 offset	-t	topic 名称
		-h	打印帮助
		-n	NameServer 服务地址, 格式 ip:port

名称	含义	命令选项	说明
topicClusterList	查看 Topic 所在集群列表	-t	topic 名称
		-h	打印帮助
		-n	NameServer 服务地址, 格式 ip:port
updateTopicPerm	更新 Topic 读写权限	-t	topic 名称
		-h	打印帮助
		-n	NameServer 服务地址, 格式 ip:port
		-b	Broker 地址, 表示 topic 所在 Broker, 只支持单台Broker, 地址为 ip:port
		-p	指定新 topic 的读写权限(W=2\
		-c	cluster 名称, 表示 topic 所在集群 (集群可通过 clusterList 查询) , -b优先, 如果没有-b, 则对集群中所有Broker执行命令
updateOrderConf	从NameServer上创建、删除、获取特定命名空间的kv配置, 目前还未启用	-h	打印帮助
		-n	NameServer 服务地址, 格式 ip:port
		-t	topic, 键
		-v	orderConf, 值
		-m	method, 可选get、put、delete
allocateMQ	以平均负载算法计算消费者列表负载消息队列的负载结果	-t	topic 名称
		-h	打印帮助
		-n	NameServer 服务地址, 格式 ip:port
		-i	ipList, 用逗号分隔, 计算这些ip去负载Topic的消息队列
statsAll	打印Topic订阅关系、TPS、积累量、24h读写总量等信息	-h	打印帮助

名称	含义	命令选项	说明
		-n	NameServer 服务地址, 格式 ip:port
		-a	是否只打印活跃topic
		-t	指定topic

4.3.1.2 具体操作

```

1 # 查看指定NameServer下的主题
2 [root@node4 ~]# mqadmin topicList -n node1:9876
3 # 查看指定NameServer, 指定集群名称下的主题
4 [root@node4 ~]# mqadmin topicList -n node1:9876 -c DefaultCluster
5 # 创建主题, 指定NameServer, 指定Broker 指定主题名称, 指定主题的写队列个数, 指定读主题队列个数
6 [root@node4 ~]# mqadmin updateTopic -b node1:10911 -r 3 -w 3 -t tp_admin_01
7 # 描述主题, 指定NameServer, 指定主题名称
8 [root@node4 ~]# mqadmin topicStatus -t tp_admin_01 -n node1:9876
9 # 创建主题, 指定NameServer, 指定集群名称, 指定主题名称, 指定读主题队列个数, 指定写主题队列个数
10 [root@node4 ~]# mqadmin updateTopic -c DefaultCluster -n node1:9876 -r 3 -w 3 -t tp_admin_02
11 # 查看指定主题的状态, 指定NameServer地址, 指定主题名称
12 [root@node4 ~]# mqadmin topicStatus -t tp_admin_02 -n node1:9876
13 #删除主题, 指定NameServer地址, 指定集群名称, 指定主题名称
14 [root@node3 ~]# mqadmin deleteTopic -n node1:9876 -c DefaultCluster -t tp_admin_03
15 # 查看主题所在的集群, 指定NameServer地址, 指定主题名称。因为不同集群可以拥有同名的主题,
并且不同集群可以注册到同一个NameServer
16 [root@node3 ~]# mqadmin topicClusterList -n node1:9876 -t tp_admin_02
17 # 计算消费的负载均衡, 不同的-i列表, 计算不同的消费平衡负载结果
18 [root@node3 ~]# mqadmin allocateMQ -n node1:9876 -t tp_admin_02 -i
node1,node3
19 [root@node3 ~]# mqadmin allocateMQ -n node1:9876 -t tp_admin_02 -i
node1,node2,node3,node4
20 # 打印Topic订阅关系、TPS、积累量、24h读写总量等信息
21 [root@node3 ~]# mqadmin statsAll -n node1:9876

```

4.3.2 集群相关

4.3.2.1 命令列表

名称	含义	命令选项	说明
clusterList	查看集群信息，集群、BrokerName、BrokerId、TPS等信息	-m	打印更多信息 (增加打印出如下信息 #InTotalYest, #OutTotalYest, #InTotalToday ,#OutTotalToday)
		-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
		-i	打印间隔，单位秒
clusterRT	发送消息检测集群各Broker RT。消息发往 \${BrokerName} Topic。	-a	amount, 每次探测的总数, RT = 总时间 / amount
		-s	消息大小，单位B
		-c	探测哪个集群
		-p	是否打印格式化日志
		-h	打印帮助
		-m	所属机房，打印使用
		-i	发送间隔，单位秒
		-n	NameServer 服务地址，格式 ip:port

4.3.2.2 具体操作

```

1 # 查看集群信息，集群、BrokerName、BrokerId、TPS等信息
2 [root@node3 ~]# mqadmin clusterList -n node1:9876 -i 1 -m
3 # 检查集群中broker的延迟。
4 [root@node3 ~]# mqadmin clusterRT -a 5 -s 1048576 -c defaultCluster -p true -
  i 2 -n node1:9876

```

4.3.3 Broker相关

4.3.3.1 命令列表

名称	含义	命令选项	说明
updateBrokerConfig	更新 Broker 配置文件，会修改 Broker.conf	-b	Broker 地址，格式为ip:port
		-c	cluster 名称
		-k	key 值
		-v	value 值
		-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
brokerStatus	查看 Broker 统计信息、运行状态（你想要的信息几乎都在里面）	-b	Broker 地址，地址为ip:port
		-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
brokerConsumeStats	Broker中各个消费者的消费情况，按Message Queue维度返回Consume Offset, Broker Offset, Diff, TTimestamp等信息	-b	Broker 地址，地址为ip:port
		-t	请求超时时间
		-l	diff阈值，超过阈值才打印
		-o	是否为顺序topic，一般为false
		-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
getBrokerConfig	获取Broker配置	-b	Broker 地址，地址为ip:port
		-n	NameServer 服务地址，格式 ip:port
wipeWritePerm	从NameServer上清除 Broker写权限	-b	Broker 地址，地址为ip:port
		-n	NameServer 服务地址，格式 ip:port
		-h	打印帮助

名称	含义	命令选项	说明
cleanExpiredCQ	清理Broker上过期的Consume Queue, 如果手动减少对列数可能产生过期队列	-n	NameServer 服务地址, 格式 ip:port
		-h	打印帮助
		-b	Broker 地址, 地址为ip:port
		-c	集群名称
cleanUnusedTopic	清理Broker上不使用的Topic, 从内存中释放Topic的Consume Queue, 如果手动删除Topic会产生不使用的Topic	-n	NameServer 服务地址, 格式 ip:port
		-h	打印帮助
		-b	Broker 地址, 地址为ip:port
		-c	集群名称
sendMsgStatus	向Broker发消息, 返回发送状态和RT	-n	NameServer 服务地址, 格式 ip:port
		-h	打印帮助
		-b	BrokerName, 注意不同于Broker地址
		-s	消息大小, 单位B
		-c	发送次数

4.3.3.2 具体操作

```
1 # 查看broker状态
2 [root@node3 ~]# mqadmin brokerStatus -b node1:10911
3 [root@node3 ~]# mqadmin brokerStatus -n node1:9876 -b node2:10911
4 # 修改节点的配置，配置文件也会修改
5 [root@node3 ~]# mqadmin updateBrokerConfig -n node1:9876 -b node2:10911 -c
DefaultCluster -k brokerRole -v ASYNC_MASTER -k brokerId -v 0 -k brokerName
-v 'broker-c'
6 # 获取Broker配置
7 [root@node1 ~]# mqadmin getBrokerConfig -n node1:9876 -b node3:10911
8 # 清理Broker上不使用的Topic，从内存中释放Topic的Consume Queue，如果手动删除Topic会产生不使用的Topic
9 [root@node1 ~]# mqadmin cleanUnusedTopic -n node1:9876 -b node1:10911 -c
DefaultCluster
10 # 向Broker发消息，返回发送状态和RT
11 [root@node3 ~]# mqadmin sendMsgStatus -n node1:9876 -b broker-b -s 128 -c 5
```

4.3.4 消息相关

4.3.4.1 命令列表

名称	含义	命令选项	说明
queryMsgById	根据offsetMsgId查询msg，如果使用开源控制台，应使用offsetMsgId，此命令还有其他参数，具体作用请阅读QueryMsgByIdSubCommand。	-i	msgId
		-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
queryMsgByKey	根据消息 Key 查询消息	-k	msgKey
		-t	Topic 名称
		-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
queryMsgByOffset	根据 Offset 查询消息	-b	Broker 名称，(这里需要注意 填写的是 Broker 的名称，不是 Broker 的地址，Broker 名称可以在 clusterList 查到)
		-i	query 队列 id
		-o	offset 值
		-t	topic 名称
		-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
queryMsgByUniqueKey	根据msgId查询，msgId不同于offsetMsgId，区别详见常见运维问题。-g, -d配合使用，查到消息后尝试让特定的消费者消费消息并返回消费结果	-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
		-i	unique msg id
		-g	consumerGroup
		-d	clientId
		-t	topic名称

名称	含义	命令选项	说明
checkMsgSendRT	检测向topic发消息的RT, 功能类似clusterRT	-h	打印帮助
		-n	NameServer 服务地址, 格式 ip:port
		-t	topic名称
		-a	探测次数
		-s	消息大小
sendMessage	发送一条消息, 可以根据配置发往特定Message Queue, 或普通发送。	-h	打印帮助
		-n	NameServer 服务地址, 格式 ip:port
		-t	topic名称
		-p	body, 消息体
		-k	keys
		-c	tags
		-b	BrokerName
		-i	queueId
consumeMessage	消费消息。可以根据offset、开始&结束时间戳、消息队列消费消息, 配置不同执行不同消费逻辑, 详见ConsumeMessageCommand。	-h	打印帮助
		-n	NameServer 服务地址, 格式 ip:port
		-t	topic名称
		-b	BrokerName
		-o	从offset开始消费
		-i	queueId
		-g	消费者分组
		-s	开始时间戳, 格式详见-h
		-d	结束时间戳
		-c	消费多少条消息

名称	含义	命令选项	说明
printMsg	从Broker消费消息并打印，可选时间段	-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
		-t	topic名称
		-c	字符集，例如UTF-8
		-s	subExpress，过滤表达式
		-b	开始时间戳，格式参见-h
		-e	结束时间戳
		-d	是否打印消息体
printMsgByQueue	类似printMsg，但指定Message Queue	-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
		-t	topic名称
		-i	queueId
		-a	BrokerName
		-c	字符集，例如UTF-8
		-s	subExpress，过滤表达式
		-b	开始时间戳，格式参见-h
		-e	结束时间戳
		-p	是否打印消息
		-d	是否打印消息体
		-f	是否统计tag数量并打印
resetOffsetByTime	按时间戳重置offset，Broker和consumer都会重置	-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
		-g	消费者分组

名称	含义	命令选项	说明
		-t	topic名称
		-s	重置为此时间戳对应的offset
		-f	是否强制重置, 如果false, 只支持回溯offset, 如果true, 不管时间戳对应offset与consumeOffset关系
		-c	是否重置c++客户端offset

4.3.4.2 具体操作

```

1 # 根据MsgKey查询消息, 指定NameServer地址, 指定主题名称, 指定MsgKey
2 [root@node3 ~]# mqadmin queryMsgByKey -n node1:9876 -t tp_admin_01 -k 00100
3 # 根据UNIQ_KEY查询消息, 指定NameServer地址, 指定UNIQ_KEY, 指定消费组名称, 指定主题名
4 # 指定发送消息的大小, 以测试延迟 1MB
5 [root@node3 ~]# mqadmin checkMsgSendRT -n node1:9876 -t tp_admin_01 -a 5 -s
6 128
7 # 指定发送消息的大小, 以测试延迟 1MB
8 [root@node3 ~]# mqadmin checkMsgSendRT -n node1:9876 -t tp_admin_01 -a 5 -s
9 1048576
10 # 消费消息, 指定NameServer, 指定主题, 指定broker名称, 指定MQ的id。
11 [root@node3 ~]# mqadmin consumeMessage -n node1:9876 -t tp_admin_01 -o 0 -b
12 broker-a -i 0
13 # 发送消息, 指定主题, 指定NameServer地址, 指定消息体
14 [root@node3 ~]# mqadmin sendMessage -n node1:9876 -t tp_admin_01 -p 'hello
15 lagou console'
16 # 发送消息, 指定主题, 指定NameServer地址, 指定消息体, 指定keys, 指定tags
17 [root@node3 ~]# mqadmin sendMessage -n node1:9876 -t tp_admin_01 -p 'hello
18 lagou console' -k '00100' -c 'test'
19 # 发送消息, 指定NameServer, 指定消息体, 指定Broker, 指定主题
20 [root@node3 ~]# mqadmin sendMessage -n node1:9876 -p 'hello lagou test 01'
21 -b broker-a -i 0 -t 'tp_admin_01'
22 # 查看偏移量, 指定NameServer, 指定主题
23 [root@node1 ~]# mqadmin topicStatus -n node1:9876 -t tp_admin_01
24 # 打印消息, 指定NameServer地址, 指定主题, 指定标签过滤, 指定是否打印消息体
25 [root@node3 ~]# mqadmin printMsg -n node1:9876 -t tp_admin_01 -s "*" -d
26 true

```

4.3.5 消费者、消费组相关

4.3.5.1 命令列表

名称	含义	命令选项	说明
consumerProgress	查看订阅组消费状态， 可以查看具体的client IP的消息 积累量	-g	消费者所属组名
		-s	是否打印client IP
		-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
consumerStatus	查看消费者状态， 包括同一个分组中是否都是相同的订阅， 分析Process Queue是否堆积， 返回消费者jstack结果，内容较多，使用者参见 ConsumerStatusSubCommand	-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
		-g	consumer group
		-i	clientId
		-s	是否执行jstack
updateSubGroup	更新或创建订阅关系	-n	NameServer 服务地址，格式 ip:port
		-h	打印帮助
		-b	Broker地址
		-c	集群名称
		-g	消费者分组名称
		-s	分组是否允许消费
		-m	是否从最小offset开始消费
		-d	是否是广播模式
		-q	重试队列数量
		-r	最大重试次数

名称	含义	命令选项	说明
		-i	当slaveReadEnable开启时有效，且还未达到从slave消费时建议从哪个BrokerId消费，可以配置备机id，主动从备机消费
		-w	如果Broker建议从slave消费，配置决定从哪个slave消费，配置BrokerId，例如1
		-a	当消费者数量变化时是否通知其他消费者负载均衡
deleteSubGroup	从Broker删除订阅关系	-n	NameServer 服务地址，格式 ip:port
		-h	打印帮助
		-b	Broker地址
		-c	集群名称
		-g	消费者分组名称
cloneGroupOffset	在目标群组中使用源群组的offset	-n	NameServer 服务地址，格式 ip:port
		-h	打印帮助
		-s	源消费者组
		-d	目标消费者组
		-t	topic名称
		-o	暂未使用

4.3.5.2 具体操作

```

1 package com.lagou.rocket.demo.producer;
2
3 import org.apache.rocketmq.client.exception.MQBrokerException;
4 import org.apache.rocketmq.client.exception.MQClientException;
5 import org.apache.rocketmq.client.producer.DefaultMQProducer;
6 import org.apache.rocketmq.common.message.Message;
7 import org.apache.rocketmq.remoting.exception.RemotingException;
8
9 public class MyProducer {
10     public static void main(String[] args) throws MQClientException,
11     RemotingException, InterruptedException, MQBrokerException {
12         DefaultMQProducer producer = new DefaultMQProducer("mygrp");
13     }
14 }
```

```

12     producer.setNamesrvAddr("node1:9876");
13     producer.start();
14
15     Message message = null;
16     for (int i = 0; i < 1000; i++) {
17         message = new Message("tp_admin_01", ("hello lagou - " +
18             i).getBytes());
19         producer.send(message);
20         Thread.sleep(1000);
21     }
22
23     producer.shutdown();
24 }
```

```

1 package com.lagou.rocket.demo.consumer;
2
3 import org.apache.rocketmq.client.consumer.DefaultMQPullConsumer;
4 import org.apache.rocketmq.client.consumer.PullResult;
5 import org.apache.rocketmq.client.exception.MQBrokerException;
6 import org.apache.rocketmq.client.exception.MQClientException;
7 import org.apache.rocketmq.common.message.MessageQueue;
8 import org.apache.rocketmq.remoting.exception.RemotingException;
9
10 import java.util.HashMap;
11 import java.util.Map;
12 import java.util.Set;
13
14 public class MyConsumer {
15     public static void main(String[] args) throws MQClientException,
16     RemotingException, InterruptedException, MQBrokerException {
17         DefaultMQPullConsumer consumer = new
18 DefaultMQPullConsumer("mygrp_consume");
19         consumer.setNamesrvAddr("node1:9876");
20         consumer.start();
21
22         Map<MessageQueue, Long> offsetMap = new HashMap<>();
23
24         Set<MessageQueue> messageQueues =
25 consumer.fetchSubscribeMessageQueues("tp_admin_01");
26         while (true) {
27             for (MessageQueue messageQueue : messageQueues) {
28                 Long aLong = offsetMap.get(messageQueue);
29                 if (aLong == null) {
30                     offsetMap.put(messageQueue, 0L);
31                 }
32                 PullResult pull = consumer.pull(messageQueue, "*",
33 offsetMap.get(messageQueue), 1);
34                 System.out.println(pull.getMsgFoundList().size() + "\t" +
35 pull.getMsgFoundList().get(0).getMsgId());
36                 offsetMap.put(messageQueue, pull.getNextBeginOffset());
37                 Thread.sleep(1000);
38             }
39         }
40     }
41 }
```

```

1 # 查看消费者状态
2 [root@node3 ~]# mqadmin consumerStatus -n node1:9876 -g mygrp_consume -s

```

4.3.6 连接相关

4.3.6.1 命令列表

名称	含义	命令选项	说明
consumerConnection	查询 Consumer 的网络连接	-g	消费者所属组名
		-n	NameServer 服务地址, 格式 ip:port
		-h	打印帮助
producerConnection	查询 Producer 的网络连接	-g	生产者所属组名
		-t	主题名称
		-n	NameServer 服务地址, 格式 ip:port
		-h	打印帮助

4.3.6.2 具体操作

```

1 package com.lagou.rocket.demo.producer;
2
3 import org.apache.rocketmq.client.exception.MQBrokerException;
4 import org.apache.rocketmq.client.exception.MQClientException;
5 import org.apache.rocketmq.client.producer.DefaultMQProducer;
6 import org.apache.rocketmq.common.message.Message;
7 import org.apache.rocketmq.remoting.exception.RemotingException;
8
9 public class MyProducer {
10     public static void main(String[] args) throws MQClientException,
11     RemotingException, InterruptedException, MQBrokerException {
12         DefaultMQProducer producer = new DefaultMQProducer("mygrp");
13         producer.setNamesrvAddr("node1:9876");
14         producer.start();
15
16         Message message = null;
17         for (int i = 0; i < 1000; i++) {
18             message = new Message("tp_admin_01", ("hello lagou - " +
19             i).getBytes());
20             producer.send(message);
21             Thread.sleep(1000);
22         }
23     }
24 }

```

```
20     }
21
22     producer.shutdown();
23 }
24 }
```

```
1 package com.lagou.rocket.demo.consumer;
2
3 import org.apache.rocketmq.client.consumer.DefaultMQPullConsumer;
4 import org.apache.rocketmq.client.consumer.PullResult;
5 import org.apache.rocketmq.client.exception.MQBrokerException;
6 import org.apache.rocketmq.client.exception.MQClientException;
7 import org.apache.rocketmq.common.message.MessageQueue;
8 import org.apache.rocketmq.remoting.exception.RemotingException;
9
10 import java.util.HashMap;
11 import java.util.Map;
12 import java.util.Set;
13
14 public class MyConsumer {
15     public static void main(String[] args) throws MQClientException,
16     RemotingException, InterruptedException, MQBrokerException {
17         DefaultMQPullConsumer consumer = new
18         DefaultMQPullConsumer("mygrp_consume");
19         consumer.setNamesrvAddr("node1:9876");
20         consumer.start();
21
22         Map<MessageQueue, Long> offsetMap = new HashMap<>();
23
24         Set<MessageQueue> messageQueues =
25         consumer.fetchSubscribeMessageQueues("tp_admin_01");
26         while (true) {
27             for (MessageQueue messageQueue : messageQueues) {
28                 Long aLong = offsetMap.get(messageQueue);
29                 if (aLong == null) {
30                     offsetMap.put(messageQueue, 0L);
31                 }
32                 PullResult pull = consumer.pull(messageQueue, "*",
33                 offsetMap.get(messageQueue), 1);
34                 System.out.println(pull.getMsgFoundList().size() + "\t" +
35                 pull.getMsgFoundList().get(0).getMsgId());
36                 offsetMap.put(messageQueue, pull.getNextBeginOffset());
37                 Thread.sleep(1000);
38             }
39         }
40     }
41 }
```

```

1 # 查看生产者连接
2 [root@node3 bin]# mqadmin producerConnection -g mygrp -t tp_admin_01 -n
  node1:9876
3 # 查看消费者连接
4 [root@node3 bin]# mqadmin consumerConnection -g mygrp_consume -n node1:9876

```

4.3.7 NameServer相关

4.3.7.1 命令列表

名称	含义	命令选项	说明
updateKvConfig	更新NameServer的kv配置, 目前还未使用	-s -k -v	命名空间 key value
		-n	NameServer 服务地址, 格式 ip:port
		-h	打印帮助
deleteKvConfig	删除NameServer的kv配置	-s -k -n	命名空间 key NameServer 服务地址, 格式 ip:port
		-h	打印帮助
getNamesrvConfig	获取NameServer配置	-n -h	NameServer 服务地址, 格式 ip:port 打印帮助
updateNamesrvConfig	修改NameServer配置	-n -h -k -v	NameServer 服务地址, 格式 ip:port 打印帮助 key value

4.3.7.2 具体操作

```
1 # 获取NameServer配置信息
2 [root@node2 ~]# mqadmin getNamesrvConfig -n node1:9876
3 # 修改NameServer的配置
4 [root@node2 ~]# mqadmin updateNamesrvConfig -n node1:9876 -k
serverworkerThreads -v 10
```

4.3.8 其他

4.3.8.1 命令列表

名称	含义	命令选项	说明
startMonitoring	开启监控进程，监控消息误删、重试队列消息数等	-n	NameServer 服务地址，格式 ip:port
		-h	打印帮助

4.3.8.2 具体操作

```
1 [root@node4 ~]# mqadmin startMonitoring -n node1:9876
```

4.4 运维常见问题

4.4.1 RocketMQ的mqadmin命令报错问题

问题描述：有时候在部署完RocketMQ集群后，尝试执行“mqadmin”一些运维命令，会出现下面的异常信息：

```
1 org.apache.rocketmq.remoting.exception.RemotingConnectException: connect to
<null> failed
```

解决方法：可以在部署RocketMQ集群的虚拟机上执行 `export NAMESRV_ADDR=ip:9876` (ip指的是集群中部署NameServer组件的机器ip地址) 命令之后再使用“mqadmin”的相关命令进行查询，即可得到结果。

4.4.2 RocketMQ生产端和消费端版本不一致导致不能正常消费的问题

问题描述：同一个生产端发出消息，A消费端可消费，B消费端却无法消费，rocketMQ Console中出现：

```
1 Not found the consumer group consume stats, because return offset table is
empty, maybe the consumer not consume any message的异常消息。
```

解决方案：RocketMQ 的jar包：rocketmq-client等包应该保持生产端，消费端使用相同的version。

4.4.3 新增一个topic的消费组时，无法消费历史消息的问题

问题描述：当同一个topic的新增消费组启动时，消费的消息是当前的offset的消息，并未获取历史消息。

解决方案：rocketmq默认策略是从消息队列尾部，即跳过历史消息。如果想消费历史消息，则需要设置：

`org.apache.rocketmq.client.consumer.DefaultMQPushConsumer#setConsumeFromWhere`。常用的有以下三种配置：

- 默认配置,一个新的订阅组第一次启动从队列的最后位置开始消费,后续再启动接着上次消费的进度开始消费,即跳过历史消息;

```
1 | consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_LAST_OFFSET);
```

- 一个新的订阅组第一次启动从队列的最前位置开始消费,后续再启动接着上次消费的进度开始消费,即消费Broker未过期的历史消息;

```
1 | consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_OFFSET);
```

- 一个新的订阅组第一次启动从指定时间点开始消费,后续再启动接着上次消费的进度开始消费,和`consumer.setConsumeTimestamp()`配合使用,默认是半个小时以前;

```
1 | consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_TIMESTAMP);
```

4.4.4 如何开启从Slave读数据功能

在某些情况下, Consumer需要将消费位点重置到1-2天前,这时在内存有限的Master Broker上, CommitLog会承载比较重的IO压力,影响到该Broker的其它消息的读与写。可以开启`slaveReadEnable=true`,当Master Broker发现Consumer的消费位点与CommitLog的最新值的差值的容量超过该机器内存的百分比(`accessMessageInMemoryMaxRatio=40%`) ,会推荐Consumer从Slave Broker中去读取数据,降低Master Broker的IO。

4.4.5 性能调优问题

异步刷盘建议使用**自旋锁**,同步刷盘建议使用**重入锁**,调整Broker配置项`useReentrantLockWhenPutMessage`,默认为false;异步刷盘建议开启`TransientStorePoolEnable`;建议关闭`transferMsgByHeap`,提高拉消息效率;同步刷盘建议适当增大`sendMessageThreadPoolNums`,具体配置需要经过压测。

4.4.6 在RocketMQ中msgId和offsetMsgId的含义与区别

使用RocketMQ完成生产者客户端消息发送后,通常会看到如下日志打印信息:

```
1 | SendResult [sendStatus=SEND_OK, msgId=0A42333A0DC818B4AAC246C290FD0000,  
offsetMsgId=0A42333A00002A9F000000000134F1F5, messageQueue=MessageQueue  
[topic=topicTest1, BrokerName=mac.local, queueId=3], queueOffset=4]
```

- `msgId`,对于客户端来说`msgId`是由客户端producer实例端生成的,具体来说,调用方法`MessageClientIDSetter.createUniqueIdBuffer()`生成唯一的Id;

```
700     byte[] prevBody = msg.getBody();
701     try {
702         //for MessageBatch, ID has been set in the generating process
703         if (!(msg instanceof MessageBatch)) {
704             MessageClientIDSetter.setUniqID(msg);
705         }
706     } catch (Exception e) {
707         log.error("Error occurred while setting unique ID for message: " + msg);
708     }
709 }
```

```
116     buffer.putInt((int) (System.currentTimeMillis() - startTime));
117     buffer.putShort((short) COUNTER.getAndIncrement());
118     return buffer.array();
119 }
120
121 @Public
122 public static void setUniqID(final Message msg) {
123     if (msg.getProperty(MessageConst.PROPERTY_UNIQ_CLIENT_MESSAGE_ID_KEYIDX) == null) {
124         msg.putProperty(MessageConst.PROPERTY_UNIQ_CLIENT_MESSAGE_ID_KEYIDX, createUniqID());
125     }
126 }
```

```
101
102 @Public
103 public static String createUniqID() {
104     StringBuilder sb = new StringBuilder( capacity: LEN * 2 );
105     sb.append(FIX_STRING);
106     sb.append(UtilAll.bytes2string(createUniqIDBuffer()));
107     return sb.toString();
108 }
```

```
109 private static byte[] createUniqIDBuffer() {
110     ByteBuffer buffer = ByteBuffer.allocate(4 + 2);
111     long current = System.currentTimeMillis();
112     if (current >= nextStartTime) {
113         setStartTime(current);
114     }
115     buffer.position(0);
116     buffer.putInt((int) (System.currentTimeMillis() - startTime));
117     buffer.putShort((short) COUNTER.getAndIncrement());
118     return buffer.array();
119 }
```

- offsetMsgId, offsetMsgId是由Broker服务端在写入消息时生成的（采用“IP地址+Port端口”与“CommitLog的物理偏移量地址”做了一个字符串拼接），其中offsetMsgId就是在RocketMQ控制台直接输入查询的那个messageId。

```

16     Message message = new Message( topic: "tp_demo_18", "hello lagou".getBytes());
17
18     SendResult sendResult = producer.send(message);
19
20     System.out.println(sendResult.getId());
21     System.out.println(sendResult.getOffset());
22
MyProducer > main()
Run: MyProducer
D:\RunningApps\JetBrains\IntelliJ IDEA 2019.3.2\jbr\bin\java.exe" "-javaagent:D:\RunningApps\JetBrains\IntelliJ IDEA 2019.3.2\lib\idea_rt.jar=53346:D:\RunningApps\JetBrains\IntelliJ IDEA 2019.3.2\bin" "-Dfile.encoding=UTF-8" "-jar" "D:\RunningApps\JetBrains\IntelliJ IDEA 2019.3.2\lib\idea_rt.jar" "-java-home=D:\RunningApps\JetBrains\IntelliJ IDEA 2019.3.2\jbr"
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by io.netty.util.internal.PlatformDependent$1 (file:/D:/RunningApps/JetBrains/IntelliJ%20IDEA%202019.3.2/lib/idea_rt.jar) to method java.nio.channels.FileChannel.interrupt()
WARNING: Please consider reporting this to the maintainers of io.netty.util.internal
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective
WARNING: All illegal access operations will be denied in a future release
0A4E00A723BC78308DB15CC0C5B40000
C0A8646900002A9F0000000000000000

```

第五章 RocketMQ源码剖析

5.1 环境搭建

依赖工具

- JDK : 1.8+
- Maven
- IntelliJ IDEA

5.1.1 源码拉取

从官方仓库 <https://github.com/apache/rocketmq> clone 或者 download 源码。

Mirror of Apache RocketMQ

rocketmq

Branch: master ▾ New pull request

1,123 commits 35 branches 9 releases 165 contributors Apache-2.0

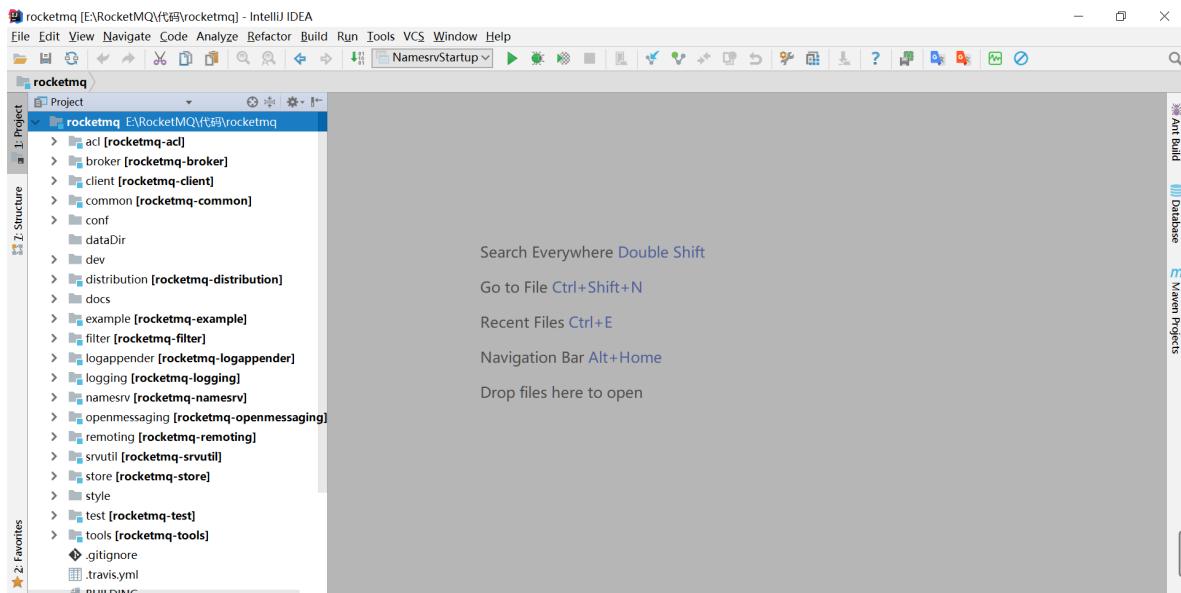
Clone with HTTPS ⓘ Use SSH
https://github.com/apache/rocketmq.git

Branch	Last Commit	Time Ago
zongtanghu Merge remote-tracking branch 'origin/release-4.5.2'	[maven-release-plugin] prepare release rocketmq-all-4.5.2	9 days ago
.github Update issue_template.md		
acl [maven-release-plugin] prepare release rocketmq-all-4.5.2		
broker [maven-release-plugin] prepare release rocketmq-all-4.5.2		
client [maven-release-plugin] prepare release rocketmq-all-4.5.2		
common [maven-release-plugin] prepare release rocketmq-all-4.5.2		9 days ago
dev [ROCKETMQ-302] TLP clean up, removes incubating related info from cod...		2 years ago
distribution [maven-release-plugin] prepare release rocketmq-all-4.5.2		9 days ago

源码目录结构:

- broker: broker 模块 (broke 启动进程)
- client : 消息客户端, 包含消息生产者、消息消费者相关类
- common : 公共包
- dev : 开发者信息 (非源代码)
- distribution : 部署实例文件夹 (非源代码)
- example: RocketMQ 例代码
- filter : 消息过滤相关基础类
- filtersrv: 消息过滤服务器实现相关类 (Filter启动进程)
- logappender: 日志实现相关类
- namesrv: NameServer实现相关类 (NameServer启动进程)
- openmessageing: 消息开放标准
- remoting: 远程通信模块, 基于Netty
- srcutil: 服务工具类
- store: 消息存储实现相关类
- style: checkstyle相关实现
- test: 测试相关类
- tools: 工具类, 监控命令相关实现类

5.1.2 导入IDEA

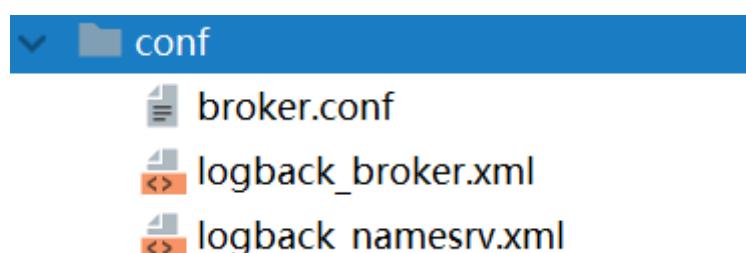


执行安装

```
1 | clean install -Dmaven.test.skip=true
```

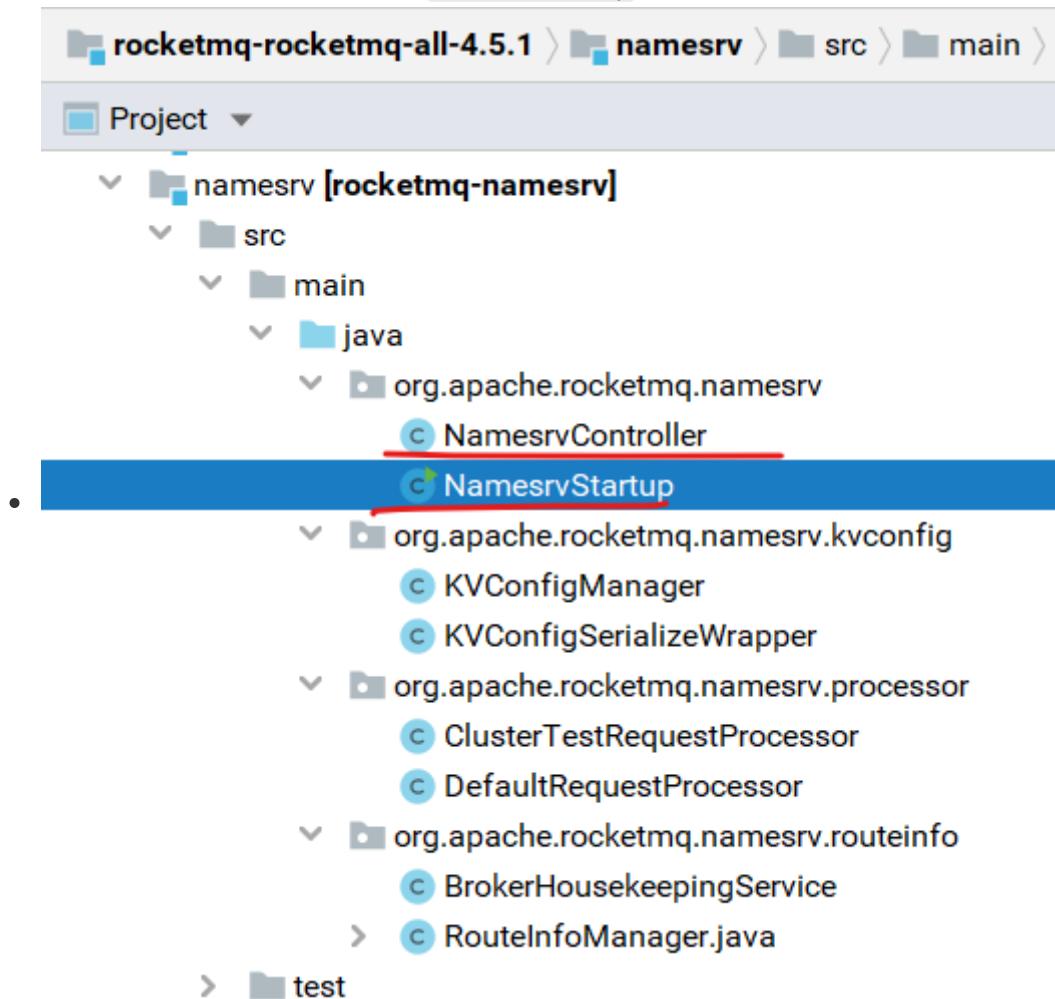
5.1.3 调试

创建 conf 配置文件夹,从 distribution 拷贝 broker.conf 和 logback_broker.xml 和 logback_namesrv.xml

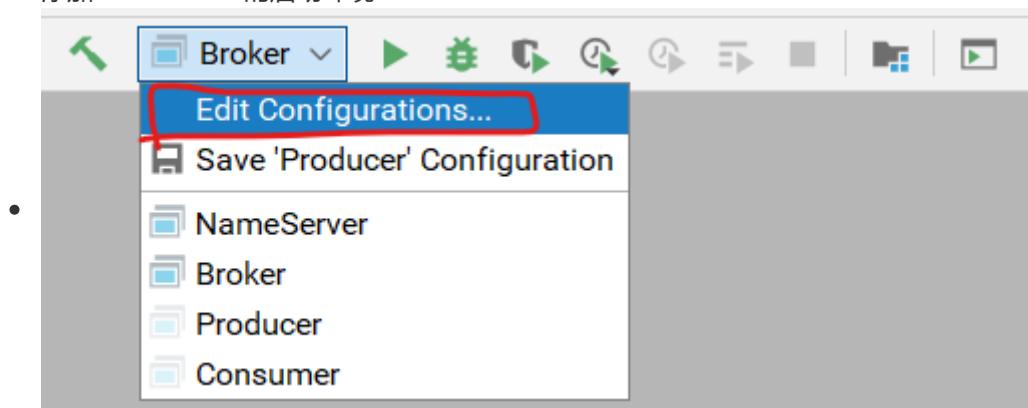


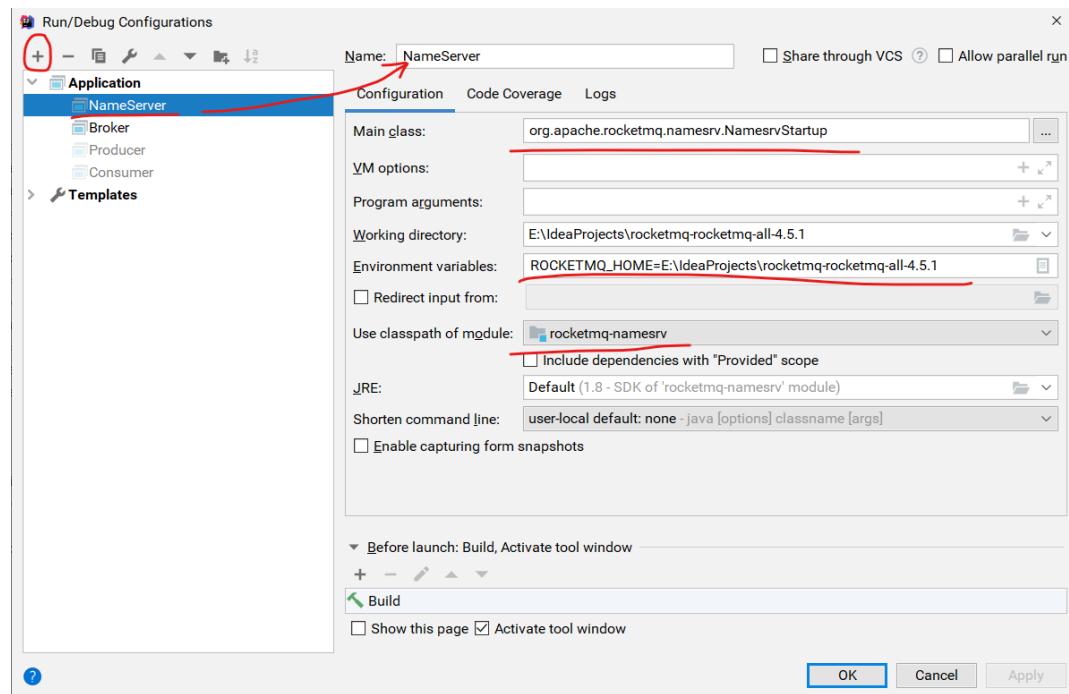
5.1.4 启动NameServer

- 配置NameServer的启动环境:
- NameServer启动的类就是这里的 NamesrvStartup。

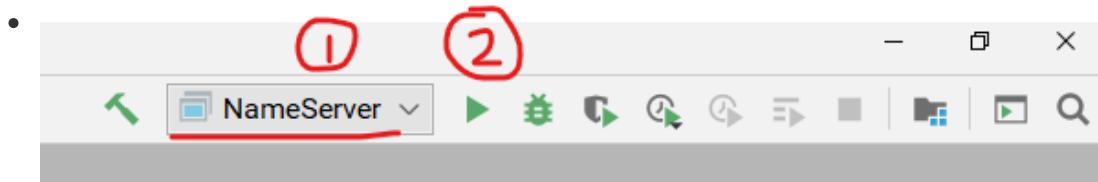


- 添加NameServer的启动环境:





- 启动NameServer：

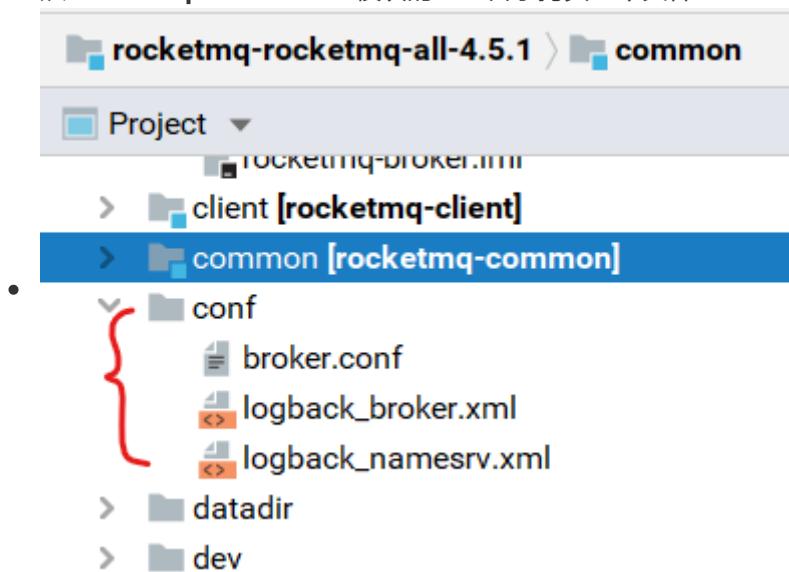


控制台打印结果

```
1 | The Name Server boot success. serializeType=JSON
```

5.1.5 启动Broker

- 在项目根目录下创建数据文件夹 dataDir
- 从rocketmq-distribution模块的conf目录拷贝三个文件：



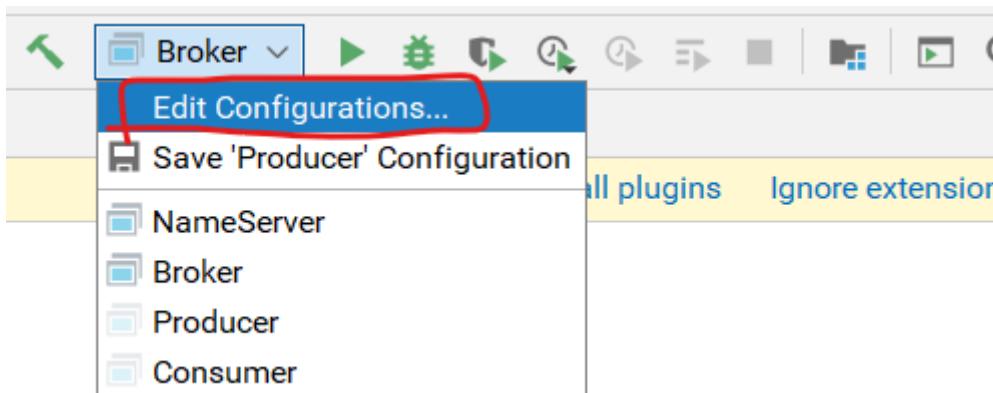
- conf文件内容修改：

```

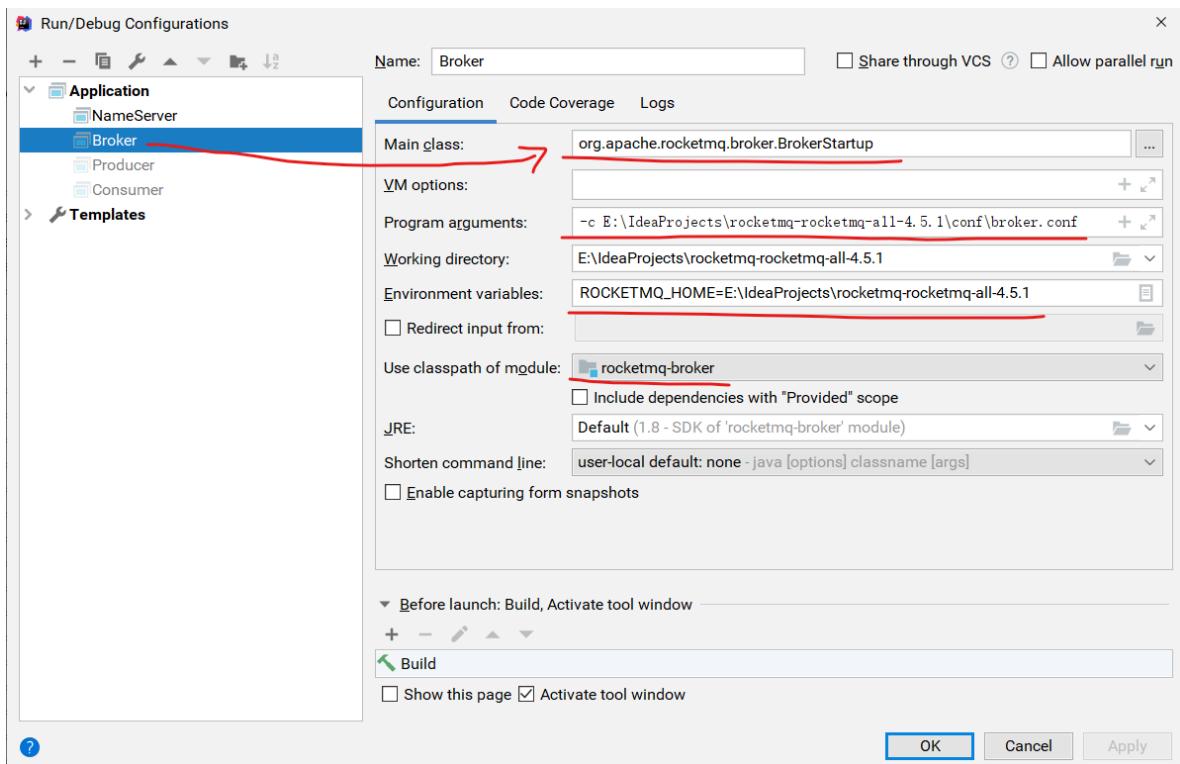
1 brokerClusterName = DefaultCluster
2 brokerName = broker-a
3 brokerId = 0
4 deletewhen = 04
5 fileReservedTime = 48
6 brokerRole = ASYNC_MASTER
7 flushDiskType = ASYNC_FLUSH
8
9 # namesrvAddr地址
10 namesrvAddr=127.0.0.1:9876
11 # 启用自动创建主题
12 autoCreateTopicEnable=true
13
14 # 存储路径
15 storePathRootDir=E:\\IdeaProjects\\rocketmq-rocketmq-all-4.5.1\\dataDir
16 # commitLog路径
17 storePathCommitLog=E:\\IdeaProjects\\rocketmq-rocketmq-all-
4.5.1\\dataDir\\commitlog
18 # 消息队列存储路径
19 storePathConsumeQueue=E:\\IdeaProjects\\rocketmq-rocketmq-all-
4.5.1\\dataDir\\consumequeue
20 # 消息索引存储路径
21 storePathIndex=E:\\IdeaProjects\\rocketmq-rocketmq-all-
4.5.1\\dataDir\\index
22 # checkpoint文件路径
23 storeCheckpoint=E:\\IdeaProjects\\rocketmq-rocketmq-all-
4.5.1\\dataDir\\checkpoint
24 # abort文件存储路径
25 abortFile=E:\\IdeaProjects\\rocketmq-rocketmq-all-4.5.1\\dataDir\\abort

```

- 启动 BrokerStartup,配置 broker.conf 和 ROCKETMQ_HOME



现在下图中已经存在NameServer或Broker的启动环境了，如果没有，可以点击左上角的“+”，选择创建Application环境，在Name一栏修改名字。



5.1.6 发送消息

- 进入example模块的`org.apache.rocketmq.example.quickstart`
- 指定Namesrv地址

```

1 DefaultMQProducer producer = new
DefaultMQProducer("please_rename_unique_group_name");
2 producer.setNamesrvAddr("127.0.0.1:9876");

```

- 运行`main`方法，发送消息

5.1.7 消费消息

- 进入example模块的`org.apache.rocketmq.example.quickstart`
- 指定Namesrv地址

```

1 DefaultMQPushConsumer consumer = new
DefaultMQPushConsumer("please_rename_unique_group_name_4");
2 consumer.setNamesrvAddr("127.0.0.1:9876");

```

- 运行`main`方法，消费消息

5.2 NameServer

5.2.1 架构设计

消息中间件的设计思路一般是基于主题订阅发布的机制。

生产者（Producer）发送消息到消息服务器的某个主题，消息服务器负责将消息持久化存储。

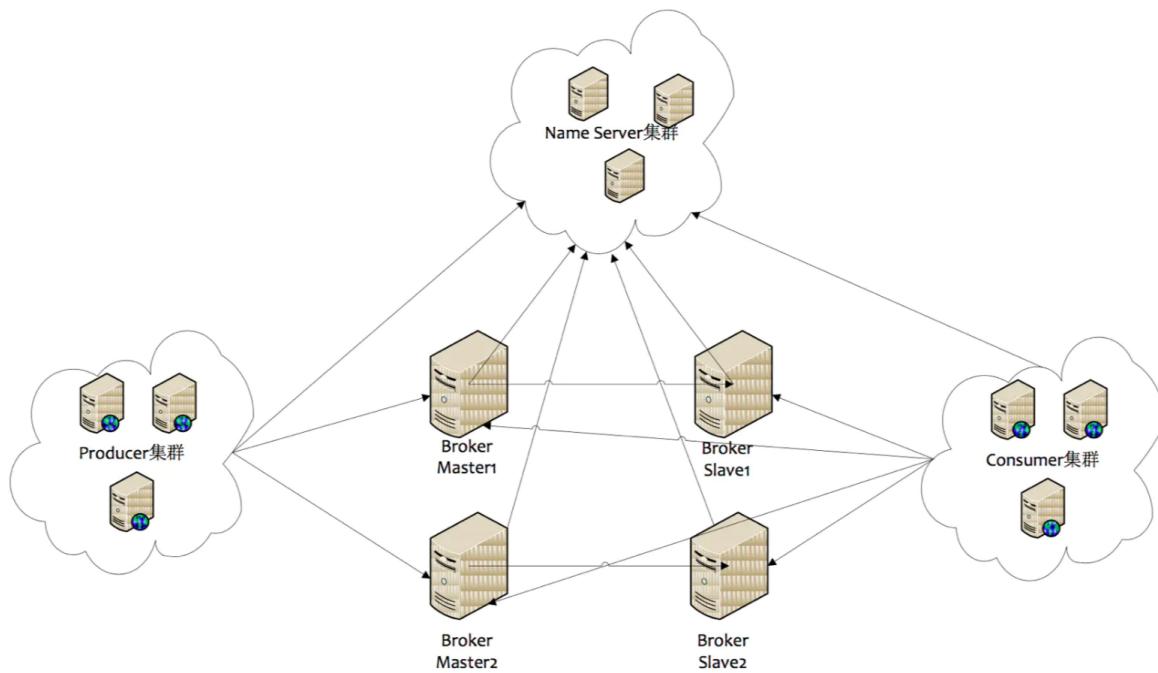
消息消费者（Consumer）订阅该兴趣的主题，消息服务器根据订阅信息（路由信息）将消息推送
到消费者（Push模式）或者消费者主动向消息服务器拉去（Pull模式），从而实现消息生产者与消息消
费者解耦。

为了避免消息服务器的单点故障，部署多台消息服务器共同承担消息的存储。

生产者如何知道消息要发送到哪台消息服务器呢？

如果某一台消息服务器宕机了，生产者如何在不重启服务情况下感知呢？

NameServer就是为了解决以上问题设计的。



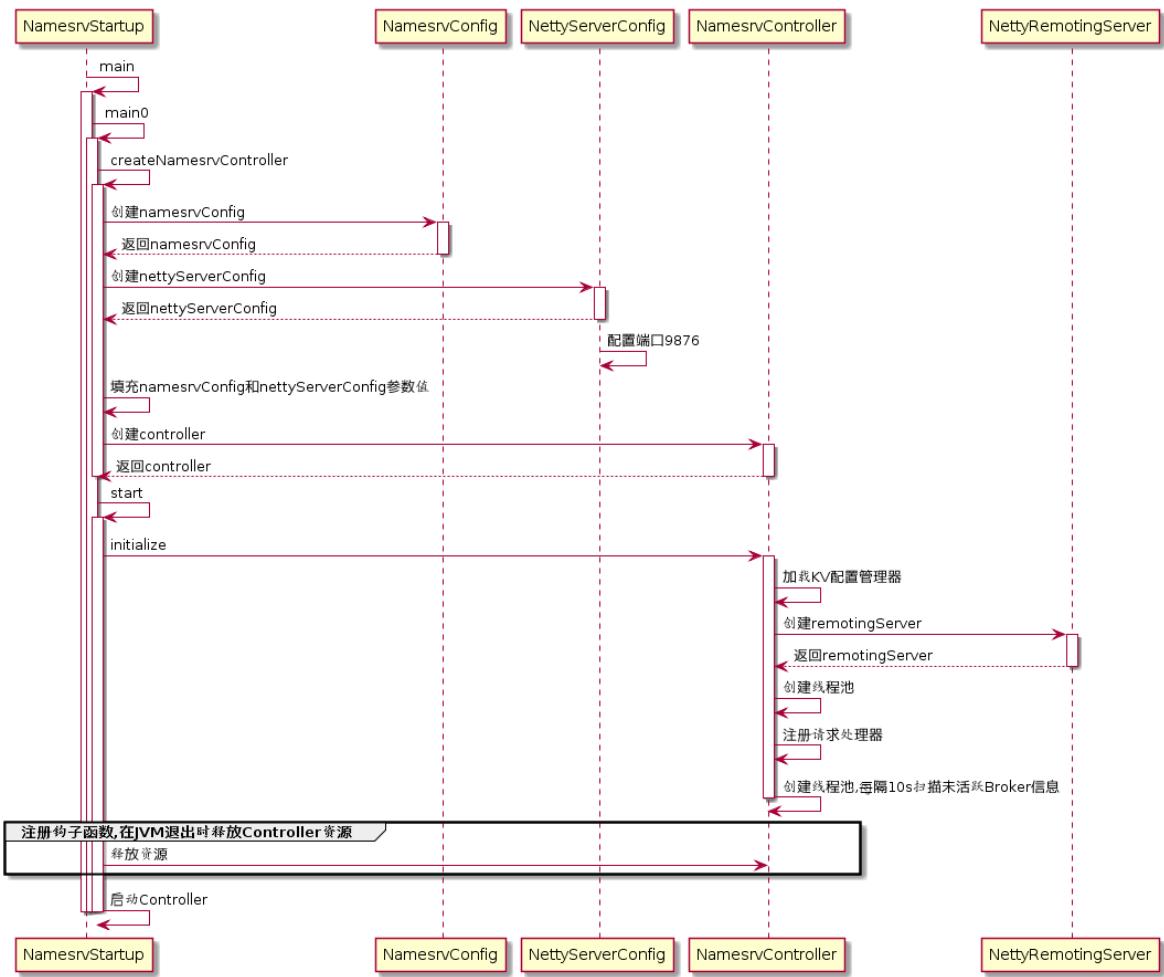
Broker消息服务器在启动时向所有NameServer注册，生产者（Producer）在发送消息之前先从
NameServer获取Broker服务器地址列表，然后根据负载均衡算法从列表中选择一台服务器进行发送。

NameServer与每台Broker保持长连接，并间隔30S检测Broker是否存活，如果检测到Broker宕
机，则从路由注册表中删除。

但是路由变化不会马上通知生产者。这样设计的目的是为了降低NameServer实现的复杂度，在消
息发送端提供容错机制保证消息发送的可用性。

NameServer本身的高可用是通过部署多台NameServer来实现，但彼此之间不通讯，也就
是NameServer服务器之间在某一个时刻的数据并不完全相同，但这对消息发送并不会造成任何影响，这
也是NameServer设计的一个亮点，总之，**RocketMQ设计追求简单高效**。

5.2.2 启动流程



启动类: `org.apache.rocketmq.namesrv.NamesrvStartup`

步骤一

解析配置文件，填充NameServerConfig、NettyServerConfig属性值，并创建NamesrvController

代码: `NamesrvController#createNamesrvController`

```

1 //创建NamesrvConfig
2 final NamesrvConfig namesrvConfig = new NamesrvConfig();
3 //创建NettyServerConfig
4 final NettyServerConfig nettyServerConfig = new NettyServerConfig();
5 //设置启动端口号
6 nettyServerConfig.setListenPort(9876);
7 //解析启动-c参数
8 if (commandLine.hasOption('c')) {
9     String file = commandLine.getOptionValue('c');
10    if (file != null) {
11        InputStream in = new BufferedInputStream(new
12 FileInputStream(file));
13        properties = new Properties();
14        properties.load(in);
15        MixAll.properties2Object(properties, namesrvConfig);
16        MixAll.properties2Object(properties, nettyServerConfig);
17
18        namesrvConfig.setConfigStorePath(file);
19
20        System.out.printf("load config properties file OK, %s%n", file);
21        in.close();
22    }
23}

```

```

22 }
23 //解析启动-p参数
24 if (commandLine.hasOption('p')) {
25     InternalLogger console =
26         InternalLoggerFactory.getLogger(LoggerName.NAMESRV_CONSOLE_NAME);
27     MixAll.printObjectProperties(console, namesrvConfig);
28     MixAll.printObjectProperties(console, nettyServerConfig);
29     System.exit(0);
30 }
31 //将启动参数填充到namesrvConfig,nettyServerConfig
32 MixAll.properties2Object(ServerUtil.commandLine2Properties(commandLine),
33                         namesrvConfig);
34
35 //创建NameServerController
36 final NamesrvController controller = new NamesrvController(namesrvConfig,
37                 nettyServerConfig);

```

NamesrvConfig属性

```

1 private String rocketmqHome =
2     System.getProperty(MixAll.ROCKETMQ_HOME_PROPERTY,
3     System.getenv(MixAll.ROCKETMQ_HOME_ENV));
4 private String kvConfigPath = System.getProperty("user.home") +
5     File.separator + "namesrv" + File.separator + "kvConfig.json";
6 private String configStorePath = System.getProperty("user.home") +
7     File.separator + "namesrv" + File.separator + "namesrv.properties";
8 private String productEnvName = "center";
9 private boolean clusterTest = false;
10 private boolean orderMessageEnable = false;

```

rocketmqHome: rocketmq主目录

kvConfig: NameServer存储KV配置属性的持久化路径

configStorePath: nameServer默认配置文件路径

orderMessageEnable: 是否支持顺序消息

NettyServerConfig属性

```

1 private int listenPort = 8888;
2 private int serverWorkerThreads = 8;
3 private int serverCallbackExecutorThreads = 0;
4 private int serverSelectorThreads = 3;
5 private int serverOnewaySemaphoreValue = 256;
6 private int serverAsyncSemaphoreValue = 64;
7 private int serverChannelMaxIdleTimeSeconds = 120;
8 private int serverSocketSndBufSize = NettySystemConfig.socketsndbufsize;
9 private int serverSocketRcvBufSize = NettySystemConfig.socketrcvbufsize;
10 private boolean serverPooledByteBufAllocatorEnable = true;
11 private boolean useEpollNativeSelector = false;

```

listenPort: NameServer监听端口，该值默认会被初始化为9876 **serverWorkerThreads:**

Netty业务线程池线程个数 **serverCallbackExecutorThreads:** Netty public任务线程池线程个数，Netty网络设计，根据业务类型会创建不同的线程池，比如处理消息发送、消息消费、心跳检测等。如果该业务类型未注册线程池，则由public线程池执行。 **serverSelectorThreads:** IO线程池个数，主要是NameServer、Broker端解析请求、返回相应的线程个数，这类线程主要是处理网路请求的，解析

请求包，然后转发到各个业务线程池完成具体的操作，然后将结果返回给调用方；
serverOnewaySemaphoreValue: send oneway消息请求并发读（Broker端参数）；
serverAsyncSemaphoreValue: 异步消息发送最大并发度; **serverChannelMaxIdleTimeSeconds**: 网络连接最大的空闲时间，默认120s。 **serverSocketSndBufSize**: 网络socket发送缓冲区大小。
serverSocketRcvBufSize: 网络接收端缓存区大小。 **serverPooledByteBufAllocatorEnable**: ByteBuffer是否开启缓存; **useEpollNativeSelector**: 是否启用Epoll IO模型。

步骤二

根据启动属性创建NamesrvController实例，并初始化该实例。NameServerController实例为NameServer核心控制器

代码: NamesrvController#initialize

```
1 public boolean initialize() {
2     //加载KV配置
3     this.kvConfigManager.load();
4     //创建NettyServer网络处理对象
5     this.remotingServer = new NettyRemotingServer(this.nettyServerConfig,
6         this.brokerHousekeepingService);
7     //开启定时任务：每隔10s扫描一次Broker，移除不活跃的Broker
8     this.remotingExecutor =
9
10    Executors.newFixedThreadPool(nettyServerConfig.getServerWorkerThreads(),
11        new ThreadFactoryImpl("RemotingExecutorThread"));
12    this.registerProcessor();
13    this.scheduledExecutorService.scheduleAtFixedRate(new Runnable() {
14        @Override
15        public void run() {
16            NamesrvController.this.routeInfoManager.scanNotActiveBroker();
17        }
18    }, 5, 10, TimeUnit.SECONDS);
19    //开启定时任务：每隔10min打印一次KV配置
20    this.scheduledExecutorService.scheduleAtFixedRate(new Runnable() {
21
22        @Override
23        public void run() {
24            NamesrvController.this.kvConfigManager.printAllPeriodically();
25        }
26    }, 1, 10, TimeUnit.MINUTES);
27    return true;
28 }
```

步骤三

在JVM进程关闭之前，先将线程池关闭，及时释放资源

代码: NamesrvStartup#start

```

1 //注册JVM钩子函数代码
2 Runtime.getRuntime().addShutdownHook(new ShutdownHookThread(log, new
3 Callable<Void>() {
4     @Override
5     public void call() throws Exception {
6         //释放资源
7         controller.shutdown();
8         return null;
9     }
10 });

```

5.2.3 路由管理

NameServer的主要作用是为消息的生产者和消息消费者提供关于主题Topic的路由信息，那么NameServer需要存储路由的基础信息，还要管理Broker节点，包括路由注册、路由删除等。

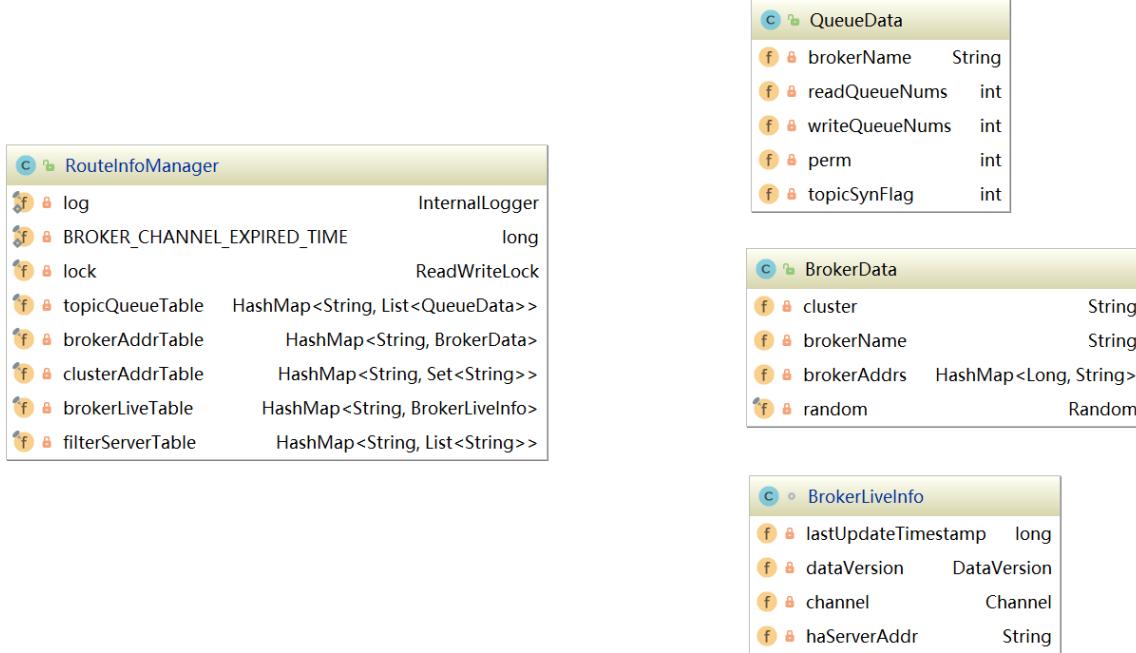
5.2.3.1 路由元信息

代码: RouteInfoManager

```

1 private final HashMap<String/* topic */, List<QueueData>> topicQueueTable;
2 private final HashMap<String/* brokerName */, BrokerData> brokerAddrTable;
3 private final HashMap<String/* clusterName */, Set<String/* brokerName */>>
clusterAddrTable;
4 private final HashMap<String/* brokerAddr */, BrokerLiveInfo>
brokerLiveTable;
5 private final HashMap<String/* brokerAddr */, List<String>/> filterServerTable;

```



Powered by yFiles

topicQueueTable: Topic消息队列路由信息，消息发送时根据路由表进行负载均衡

brokerAddrTable: Broker基础信息，包括brokerName、所属集群名称、主备Broker地址

clusterAddrTable: Broker集群信息，存储集群中所有Broker名称

brokerLiveTable: Broker状态信息，NameServer每次收到心跳包是会替换该信息

filterServerTable: Broker上的FilterServer列表，用于类模式消息过滤。

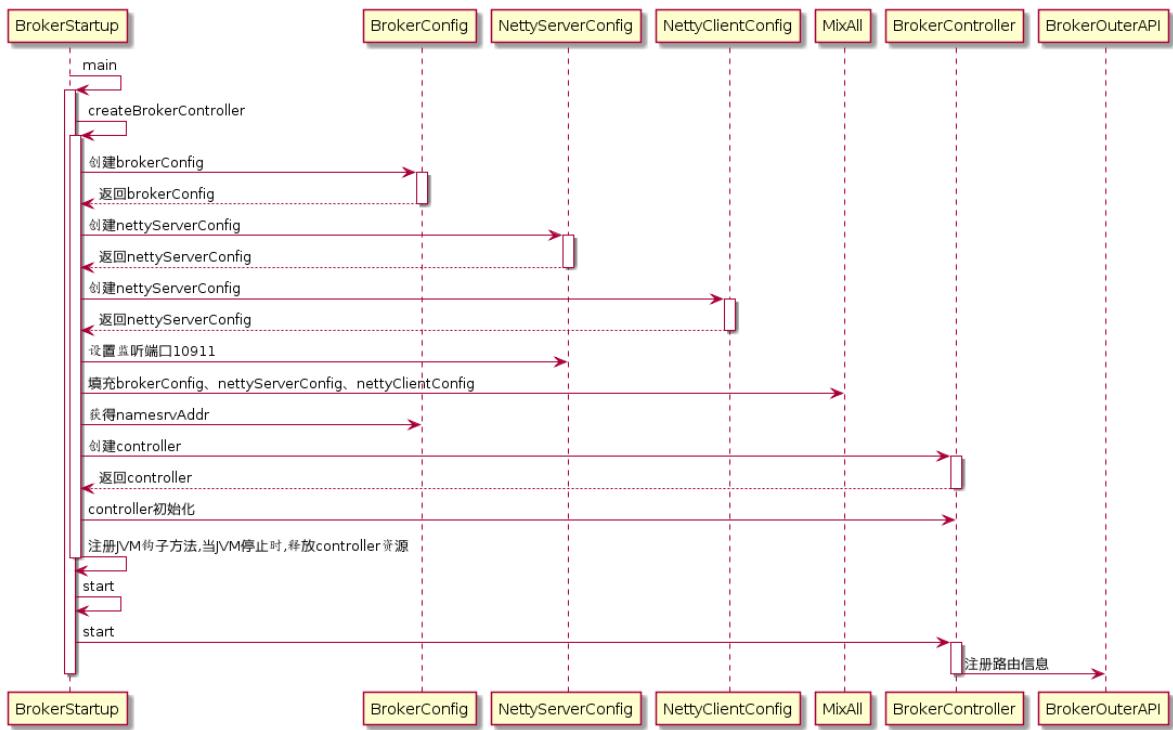
RocketMQ基于订阅发布机制，一个Topic拥有多个消息队列，一个Broker为每一个主题创建4个读队列和4个写队列。多个Broker组成一个集群，集群由相同的多台Broker组成Master-Slave架构，brokerId为0代表Master，大于0为Slave。BrokerLiveInfo中的lastUpdateTimestamp存储上次收到Broker心跳包的时间。

```
topicQueueTable:{  
    "topic1": [  
        {  
            "brokerName": "broker-a",  
            "readQueueNums": 4,  
            "readQueueNums": 4,  
            "perm": 6, // 读写权限，具体含义请参考PermName  
            "topicSynFlag": 0 // topic同步标记，具体含义请参考TopicSysFlag  
        },  
        {  
            "brokerName": "broker-b",  
            "readQueueNums": 4,  
            "readQueueNums": 4,  
            "perm": 6, // 读写权限，具体含义请参考PermName  
            "topicSynFlag": 0 // topic同步标记，具体含义请参考TopicSysFlag  
        }  
    ],  
    "topic other": []  
}  
  
brokerLiveTable : {  
    "192.168.56.1:10000" : {  
        "lastUpdateTimestamp": 1518270318980,  
        "dataVersion": versionObj,  
        "channel": channelObj,  
        "haServerAddr": "192.168.56.2:10000"  
    },  
    "192.168.56.2:10000" : {  
        "lastUpdateTimestamp": 1518270318980,  
        "dataVersion": versionObj,  
        "channel": channelObj,  
        "haServerAddr": ""  
    },  
    "192.168.56.3:10000" : {  
        "lastUpdateTimestamp": 1518270318980,  
        "dataVersion": versionObj,  
        "channel": channelObj,  
        "haServerAddr": "192.168.56.4:10000"  
    },  
    "192.168.56.4:10000" : {  
        "lastUpdateTimestamp": 1518270318980,  
        "dataVersion": versionObj,  
        "channel": channelObj,  
        "haServerAddr": ""  
    },  
}
```

```
clusterAddrTable: {  
    "c1" : [{"broker-a", "broker-b"}]  
}  
  
brokerAddrTable:  
    "broker-a": {  
        "cluster": "c1",  
        "brokerName": "broker-a",  
        "brokerAddrs": {  
            0: "192.168.56.1:10000",  
            1: "192.168.56.2:10000"  
        }  
    },  
    "broker-b": {  
        "cluster": "c1",  
        "brokerAddrs": {  
            0: "192.168.56.3:10000",  
            1: "192.168.56.4:10000"  
        }  
    }
```

5.2.3.2 路由注册

1) 发送心跳包



RocketMQ路由注册是通过Broker与NameServer的心跳功能实现的。Broker启动时向集群中所有的NameServer发送心跳信息，每隔30s向集群中所有NameServer发送心跳包，NameServer收到心跳包时会更新brokerLiveTable缓存中BrokerLiveInfo的lastUpdateTimeStamp信息，然后NameServer每隔10s扫描brokerLiveTable，如果连续120S没有收到心跳包，NameServer将移除Broker的路由信息同时关闭Socket连接。

代码: *BrokerController#start*

```

1 //注册Broker信息
2 this.registerBrokerAll(true, false, true);
3 //每隔30s上报Broker信息到NameServer
4 this.scheduledExecutorService.scheduleAtFixedRate(new Runnable() {
5
6     @Override
7     public void run() {
8         try {
9             BrokerController.this.registerBrokerAll(true, false,
10                brokerConfig.isForceRegister());
11         } catch (Throwable e) {
12             log.error("registerBrokerAll Exception", e);
13         }
14     }, 1000 * 10, Math.max(10000,
15                Math.min(brokerConfig.getRegisterNameServerPeriod(), 60000)),
16                TimeUnit.MILLISECONDS);

```

代码: *BrokerOuterAPI#registerBrokerAll*

```

1 //获得nameServer地址信息
2 List<String> nameServerAddressList =
3     this.remotingClient.getNameServerAddressList();
4 //遍历所有nameserver列表
5 if (nameServerAddressList != null && nameServerAddressList.size() > 0) {
6     //封装请求头

```

```

7     final RegisterBrokerRequestHeader requestHeader = new
8     RegisterBrokerRequestHeader();
9     requestHeader.setBrokerAddr(brokerAddr);
10    requestHeader.setBrokerId(brokerId);
11    requestHeader.setBrokerName(brokerName);
12    requestHeader.setClusterName(clusterName);
13    requestHeader.setHaServerAddr(haServerAddr);
14    requestHeader.setCompressed(compressed);
15    //封装请求体
16    RegisterBrokerBody requestBody = new RegisterBrokerBody();
17    requestBody.setTopicConfigSerializeWrapper(topicConfigWrapper);
18    requestBody.setFilterServerList(filterServerList);
19    final byte[] body = requestBody.encode(compressed);
20    final int bodyCrc32 = UtilAll.crc32(body);
21    requestHeader.setBodyCrc32(bodyCrc32);
22    final CountDownLatch countDownLatch = new
23    CountDownLatch(nameServerAddressList.size());
24    for (final String namesrvAddr : nameServerAddressList) {
25        brokerOuterExecutor.execute(new Runnable() {
26            @Override
27            public void run() {
28                try {
29                    //分别向NameServer注册
30                    RegisterBrokerResult result =
31                    registerBroker(namesrvAddr, oneWay, timeoutMills, requestHeader, body);
32                    if (result != null) {
33                        registerBrokerResultList.add(result);
34                    }
35                } catch (Exception e) {
36                    log.warn("registerBroker Exception, {}", namesrvAddr,
37                            e);
38                } finally {
39                    countDownLatch.countDown();
40                }
41            }
42        });
43    }
44    try {
45        countDownLatch.await(timeoutMills, TimeUnit.MILLISECONDS);
46    } catch (InterruptedException e) {
47    }

```

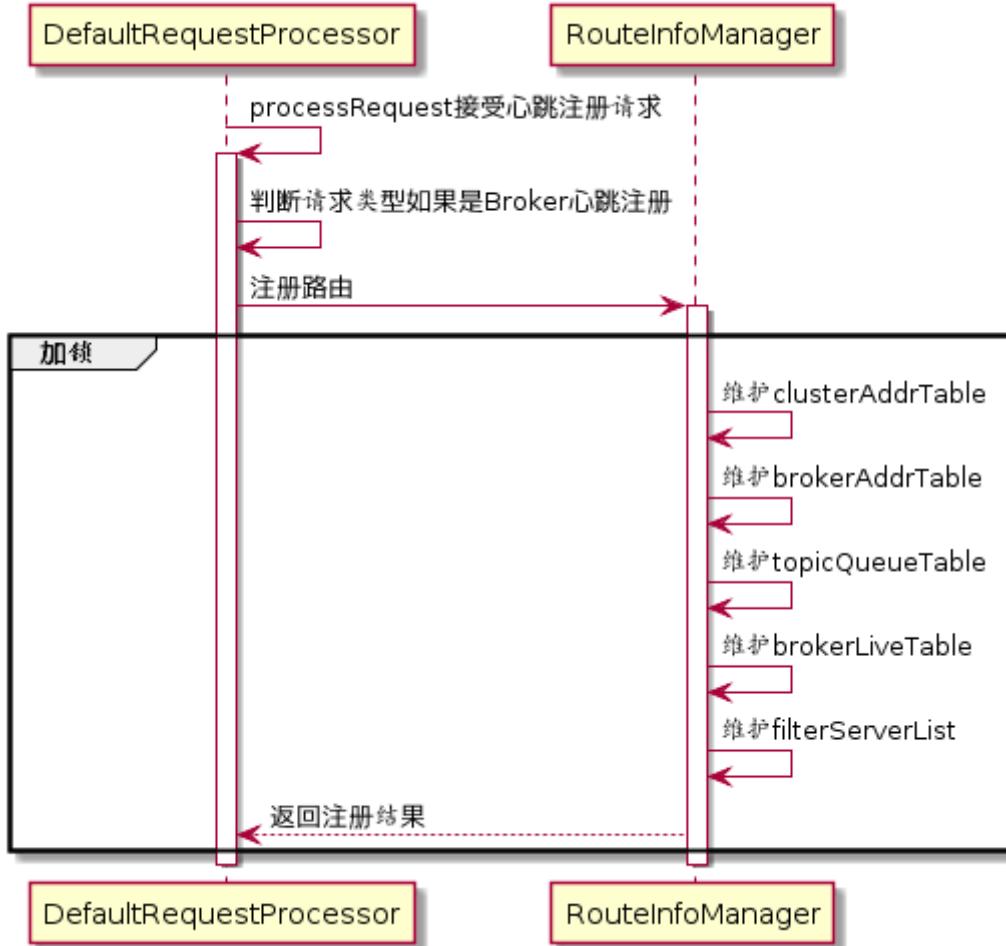
代码: BrokerOutAPI#registerBroker

```

1 if (oneway) {
2     try {
3         this.remotingClient.invokeOneWay(namesrvAddr, request, timeoutMills);
4     } catch (RemotingTooMuchRequestException e) {
5         // Ignore
6     }
7     return null;
8 }
9 RemotingCommand response = this.remotingClient.invokeSync(namesrvAddr,
request, timeoutMills);

```

2) 处理心跳包



`org.apache.rocketmq.namesrv.processor.DefaultRequestProcessor` 网路处理类解析请求类型，如果请求类型是为`REGISTER_BROKER`，则将请求转发到`RouteInfoManager#registerBroker`

代码: DefaultRequestProcessor#processRequest

```

1 //判断是注册Broker信息
2 case RequestCode.REGISTER_BROKER:
3     Version brokerVersion = MQVersion.value2Version(request.getVersion());
4     if (brokerVersion.ordinal() >= MQVersion.Version.V3_0_11.ordinal()) {
5         return this.registerBrokerWithFilterServer(ctx, request);
6     } else {
7         //注册Broker信息
8         return this.registerBroker(ctx, request);
9     }

```

代码: DefaultRequestProcessor#registerBroker

```

1 RegisterBrokerResult result =
2     this.namesrvController.getRouteInfoManager().registerBroker(
3         requestHeader.getClusterName(),
4         requestHeader.getBrokerAddr(),
5         requestHeader.getBrokerName(),
6         requestHeader.getBrokerId(),
7         requestHeader.getHaServerAddr(),
8         topicConfigwrapper,
9         null,
10        ctx.channel()
11    );

```

代码: RouteInfoManager#registerBroker

维护路由信息

```

1 //加锁
2 this.lock.writeLock().lockInterruptibly();
3 //维护clusterAddrTable
4 Set<String> brokerNames = this.clusterAddrTable.get(clusterName);
5 if (null == brokerNames) {
6     brokerNames = new HashSet<String>();
7     this.clusterAddrTable.put(clusterName, brokerNames);
8 }
9 brokerNames.add(brokerName);

```

```

1 //维护brokerAddrTable
2 BrokerData brokerData = this.brokerAddrTable.get(brokerName);
3 //第一次注册，则创建brokerData
4 if (null == brokerData) {
5     registerFirst = true;
6     brokerData = new BrokerData(clusterName, brokerName, new HashMap<Long,
7         String>());
7     this.brokerAddrTable.put(brokerName, brokerData);
8 }
9 //非第一次注册，更新Broker
10 Map<Long, String> brokerAddrsMap = brokerData.getBrokerAddrs();
11 Iterator<Entry<Long, String>> it = brokerAddrsMap.entrySet().iterator();
12 while (it.hasNext()) {
13     Entry<Long, String> item = it.next();
14     if (null != brokerAddr && brokerAddr.equals(item.getValue()) &&
15     brokerId != item.getKey()) {
16         it.remove();
17     }
18     String oldAddr = brokerData.getBrokerAddrs().put(brokerId, brokerAddr);
19     registerFirst = registerFirst || (null == oldAddr);

```

```

1 //维护topicQueueTable
2 if (null != topicConfigWrapper && MixAll.MASTER_ID == brokerId) {
3     if (this.isBrokerTopicConfigChanged(brokerAddr,
4         topicConfigWrapper.getDataVersion()) ||
5             registerFirst) {
6         ConcurrentMap<String, TopicConfig> tcTable =
7             topicConfigWrapper.getTopicConfigTable();
8         if (tcTable != null) {
9             for (Map.Entry<String, TopicConfig> entry : tcTable.entrySet())
10            {
11                this.createAndUpdateQueueData(brokerName,
12                    entry.getValue());
13            }
14        }
15    }
16 }

```

代码: RouteInfoManager#createAndUpdateQueueData

```

1 private void createAndUpdateQueueData(final String brokerName, final
2 TopicConfig topicConfig) {
3     //创建QueueData
4     QueueData queueData = new QueueData();
5     queueData.setBrokerName(brokerName);
6     queueData.setWriteQueueNums(topicConfig.getWriteQueueNums());
7     queueData.setReadQueueNums(topicConfig.getReadQueueNums());
8     queueData.setPerm(topicConfig.getPerm());
9     queueData.setTopicSysFlag(topicConfig.getTopicSysFlag());
10    //获得topicQueueTable中队列集合
11    List<QueueData> queueDataList =
12        this.topicQueueTable.get(topicConfig.getTopicName());
13    //topicQueueTable为空,则直接添加queueData到队列集合
14    if (null == queueDataList) {
15        queueDataList = new LinkedList<QueueData>();
16        queueDataList.add(queueData);
17        this.topicQueueTable.put(topicConfig.getTopicName(),
18            queueDataList);
19        log.info("new topic registered, {} {}", topicConfig.getTopicName(),
20            queueData);
21    } else {
22        //判断是否是新的队列
23        boolean addNewOne = true;
24        Iterator<QueueData> it = queueDataList.iterator();
25        while (it.hasNext()) {
26            QueueData qd = it.next();
27            //如果brokerName相同,代表不是新的队列
28            if (qd.getBrokerName().equals(brokerName)) {
29                if (qd.equals(queueData)) {
30                    addNewOne = false;
31                } else {
32                    log.info("topic changed, {} OLD: {} NEW: {}",
33                        topicConfig.getTopicName(), qd,
34                            queueData);
35                    it.remove();
36                }
37            }
38        }
39    }
40 }

```

```

33     }
34     //如果是新的队列，则添加队列到queueDataList
35     if (addNewOne) {
36         queueDataList.add(queueData);
37     }
38 }
39 }
```

```

1 //维护brokerLiveTable
2 BrokerLiveInfo prevBrokerLiveInfo = this.brokerLiveTable.put(brokerAddr, new
BrokerLiveInfo(
3     System.currentTimeMillis(),
4     topicConfigWrapper.getDataVersion(),
5     channel,
6     haserverAddr));
```

```

1 //维护filterServerList
2 if (filterServerList != null) {
3     if (filterServerList.isEmpty()) {
4         this.filterServerTable.remove(brokerAddr);
5     } else {
6         this.filterServerTable.put(brokerAddr, filterServerList);
7     }
8 }
9
10 if (MixAll.MASTER_ID != brokerId) {
11     String masterAddr = brokerData.getBrokerAddrs().get(MixAll.MASTER_ID);
12     if (masterAddr != null) {
13         BrokerLiveInfo brokerLiveInfo =
14             this.brokerLiveTable.get(masterAddr);
15         if (brokerLiveInfo != null) {
16             result.setHaserverAddr(brokerLiveInfo.getHaserverAddr());
17             result.setMasterAddr(masterAddr);
18         }
19     }
}
```

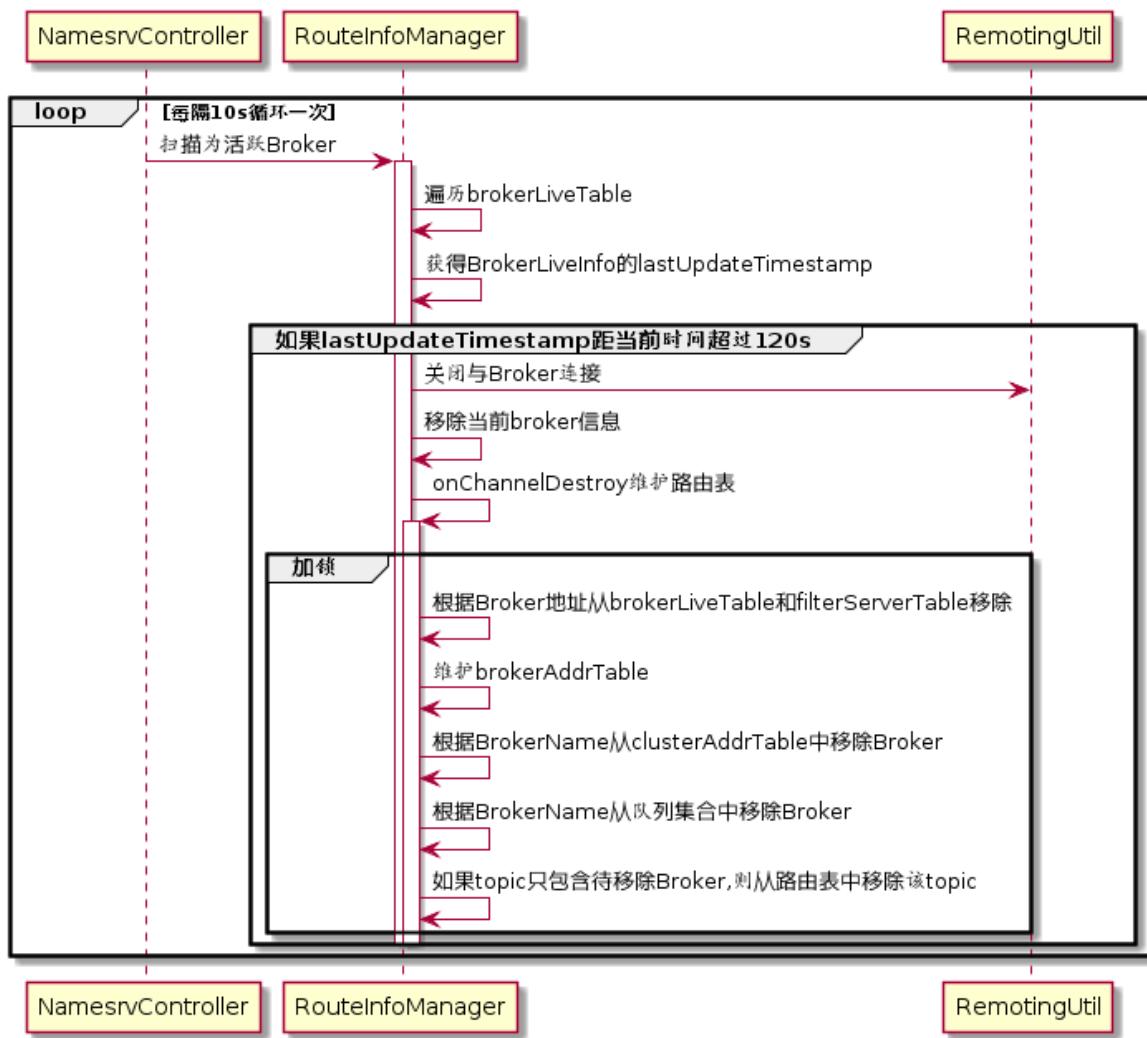
5.2.3.3 路由删除

Broker 每隔30s向 NameServer 发送一个心跳包，心跳包包含 BrokerId，Broker 地址，Broker 名称，Broker 所属集群名称、Broker 关联的 FilterServer 列表。但是如果 Broker 宕机，NameServer 无法收到心跳包，此时 NameServer 如何来剔除这些失效的 Broker 呢？NameServer 会每隔10s扫描 brokerLiveTable 状态表，如果 BrokerLive 的 lastUpdateTimestamp 的时间戳距当前时间超过120s，则认为 Broker 失效，移除该 Broker，关闭与 Broker 连接，同时更新 topicQueueTable、brokerAddrTable、brokerLiveTable、filterServerTable。

RocketMQ 有两个触发点来删除路由信息：

- NameServer 定期扫描 brokerLiveTable 检测上次心跳包与当前系统的时间差，如果时间超过 120s，则需要移除 broker。
- Broker 在正常关闭的情况下，会执行 unregisterBroker 指令

这两种方式路由删除的方法都是一样的，就是从相关路由表中删除与该 broker 相关的信息。



代码: NamesrvController#initialize

```

1 //每隔10s扫描一次为活跃Broker
2 this.scheduledExecutorService.scheduleAtFixedRate(new Runnable() {
3
4     @Override
5     public void run() {
6         NamesrvController.this.routeInfoManager.scanNotActiveBroker();
7     }
8 }, 5, 10, TimeUnit.SECONDS);

```

代码: RouteInfoManager#scanNotActiveBroker

```

1 public void scanNotActiveBroker() {
2     //获得brokerLiveTable
3     Iterator<Entry<String, BrokerLiveInfo>> it =
4     this.brokerLiveTable.entrySet().iterator();
5     //遍历brokerLiveTable
6     while (it.hasNext()) {
7         Entry<String, BrokerLiveInfo> next = it.next();
8         Long last = next.getValue().getLastUpdateTimestamp();
9         //如果收到心跳包的时间距当时时间是否超过120s
10        if ((last + BROKER_CHANNEL_EXPIRED_TIME) <
11            System.currentTimeMillis()) {
12            //关闭连接
13            RemotingUtil.closeChannel(next.getValue().getChannel());
14            //移除broker
15        }
16    }
17 }

```

```

13         it.remove();
14         //维护路由表
15         this.onChannelDestroy(next.getKey(),
16             next.getValue().getChannel());
17     }
18 }

```

代码: RouteInfoManager#onChannelDestroy

```

1 //申请写锁,根据brokerAddress从brokerLiveTable和filterServerTable移除
2 this.lock.writeLock().lockInterruptibly();
3 this.brokerLiveTable.remove(brokerAddrFound);
4 this.filterServerTable.remove(brokerAddrFound);

```

```

1 //维护brokerAddrTable
2 String brokerNameFound = null;
3 boolean removeBrokerName = false;
4 Iterator<Entry<String, BrokerData>> itBrokerAddrTable
= this.brokerAddrTable.entrySet().iterator();
5 //遍历brokerAddrTable
6 while (itBrokerAddrTable.hasNext() && (null == brokerNameFound)) {
7     BrokerData brokerData = itBrokerAddrTable.next().getValue();
8     //遍历broker地址
9     Iterator<Entry<Long, String>> it =
10    brokerData.getBrokerAddrs().entrySet().iterator();
11     while (it.hasNext()) {
12         Entry<Long, String> entry = it.next();
13         Long brokerId = entry.getKey();
14         String brokerAddr = entry.getValue();
15         //根据broker地址移除brokerAddr
16         if (brokerAddr.equals(brokerAddrFound)) {
17             brokerNameFound = brokerData.getBrokerName();
18             it.remove();
19             log.info("remove brokerAddr[{}, {}] from brokerAddrTable,
because channel destroyed",
20                     brokerId, brokerAddr);
21             break;
22         }
23     }
24     //如果当前主题只包含待移除的broker,则移除该topic
25     if (brokerData.getBrokerAddrs().isEmpty()) {
26         removeBrokerName = true;
27         itBrokerAddrTable.remove();
28         log.info("remove brokerName[{}] from brokerAddrTable, because
channel destroyed",
29                 brokerData.getBrokerName());
30     }
}

```

```

1 //维护clusterAddrTable
2 if (brokerNameFound != null && removeBrokerName) {
3     Iterator<Entry<String, Set<String>>> it =
4     this.clusterAddrTable.entrySet().iterator();
5     //遍历clusterAddrTable
6     while (it.hasNext()) {

```

```

6     Entry<String, Set<String>> entry = it.next();
7     //获得集群名称
8     String clusterName = entry.getKey();
9     //获得集群中brokerName集合
10    Set<String> brokerNames = entry.getValue();
11    //从brokerNames中移除brokerNameFound
12    boolean removed = brokerNames.remove(brokerNameFound);
13    if (removed) {
14        log.info("remove brokerName[{}], clusterName[{}] from
clusterAddrTable, because channel destroyed",
15                brokerNameFound, clusterName);
16
17        if (brokerNames.isEmpty()) {
18            log.info("remove the clusterName[{}] from clusterAddrTable,
because channel destroyed and no broker in this cluster",
19                    clusterName);
20            //如果集群中不包含任何broker,则移除该集群
21            it.remove();
22        }
23
24        break;
25    }
26 }
27 }
```

```

1 //维护topicQueueTable队列
2 if (removeBrokerName) {
3     //遍历topicQueueTable
4     Iterator<Entry<String, List<QueueData>>> itTopicQueueTable =
5         this.topicQueueTable.entrySet().iterator();
6     while (itTopicQueueTable.hasNext()) {
7         Entry<String, List<QueueData>> entry = itTopicQueueTable.next();
8         //主题名称
9         String topic = entry.getKey();
10        //队列集合
11        List<QueueData> queueDataList = entry.getValue();
12        //遍历该主题队列
13        Iterator<QueueData> itQueueData = queueDataList.iterator();
14        while (itQueueData.hasNext()) {
15            //从队列中移除为活跃broker信息
16            QueueData queueData = itQueueData.next();
17            if (queueData.getBrokerName().equals(brokerNameFound)) {
18                itQueueData.remove();
19                log.info("remove topic[{} {}], from topicQueueTable,
because channel destroyed",
20                        topic, queueData);
21            }
22        }
23        //如果该topic的队列为空,则移除该topic
24        if (queueDataList.isEmpty()) {
25            itTopicQueueTable.remove();
26            log.info("remove topic[{}] all queue, from topicQueueTable,
because channel destroyed",
27                    topic);
28        }
29    }
30 }
```

```
1 //释放写锁
2 finally {
3     this.lock.writeLock().unlock();
4 }
```

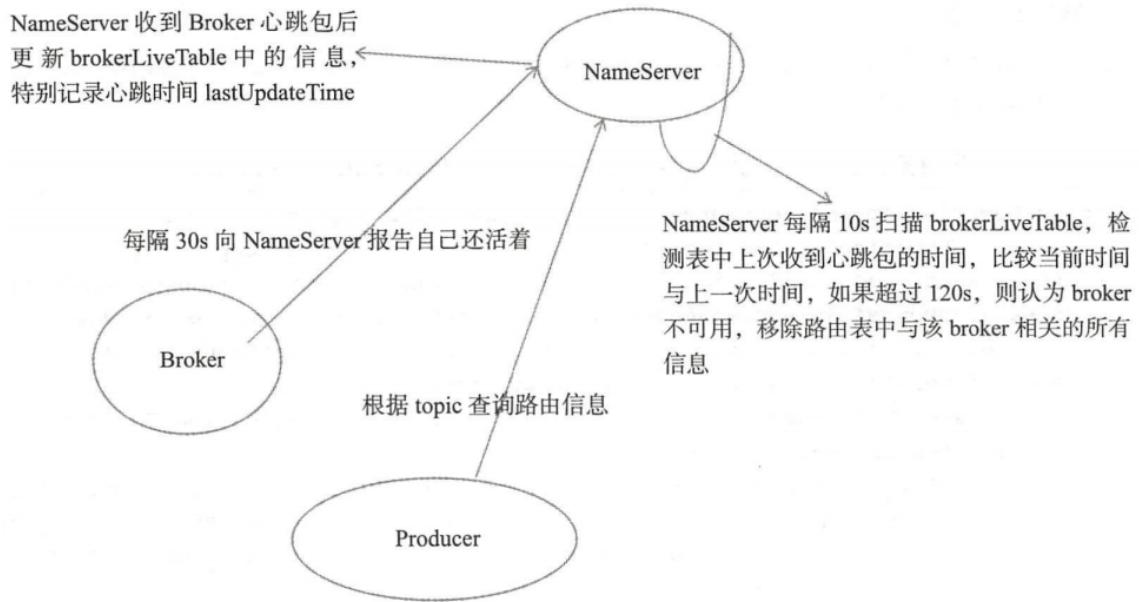
5.2.3.4 路由发现

RocketMQ路由发现是非实时的，当Topic路由出现变化后，NameServer不会主动推送给客户端，而是由客户端定时拉取主题最新的路由。

代码: `DefaultRequestProcessor#$routeInfoByTopic`

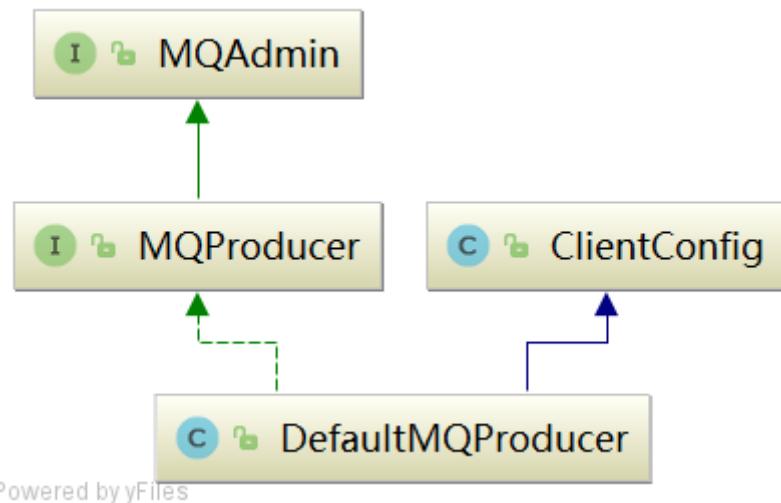
```
1 public RemotingCommand getRouteInfoByTopic(ChannelHandlerContext ctx,
2     RemotingCommand request) throws RemotingCommandException {
3     final RemotingCommand response =
4         RemotingCommand.createResponseCommand(null);
5     final GetRouteInfoRequestHeader requestHeader =
6         (GetRouteInfoRequestHeader)
7     request.decodeCommandCustomHeader(GetRouteInfoRequestHeader.class);
8     //调用RouteInfoManager的方法，从路由表topicQueueTable、brokerAddrTable、
9     filterServerTable中分别填充TopicRouteData的List<QueueData>、List<BrokerData>、
10    filterServer
11    TopicRouteData topicRouteData =
12    this.namesrvController.getRouteInfoManager().pickupTopicRouteData(requestHe
13    ader.getTopic());
14    //如果找到主题对应你的路由信息并且该主题为顺序消息，则从NameServer KVConfig中获取
15    //关于顺序消息相关的配置填充路由信息
16    if (topicRouteData != null) {
17        if
18            (this.namesrvController.getNamesrvConfig().isOrderMessageEnable()) {
19                String orderTopicConf =
20
21                this.namesrvController.getKvConfigManager().getKVConfig(NamesrvUtil.NAMESP
22                ACE_ORDER_TOPIC_CONFIG,
23                    requestHeader.getTopic());
24                topicRouteData.setOrderTopicConf(orderTopicConf);
25            }
26
27            byte[] content = topicRouteData.encode();
28            response.setBody(content);
29            response.setCode(ResponseCode.SUCCESS);
30            response.setRemark(null);
31            return response;
32        }
33
34        response.setCode(ResponseCode.TOPIC_NOT_EXIST);
35        response.setRemark("No topic route info in name server for the topic: "
36        + requestHeader.getTopic()
37            + FAQUrl.suggestTodo(FAQUrl.APPLY_TOPIC_URL));
38        return response;
39    }
```

5.2.4 小结



5.3 Producer

消息生产者的代码都在client模块中，相对于RocketMQ来讲，消息生产者就是客户端，也是消息的提供者。



5.3.1 方法和属性

5.3.1.1 主要方法介绍

MQAdmin		
(m)	createTopic(String, String, int)	void
(m)	createTopic(String, String, int, int)	void
(m)	searchOffset(MessageQueue, long)	long
(m)	maxOffset(MessageQueue)	long
(m)	minOffset(MessageQueue)	long
(m)	earliestMsgStoreTime(MessageQueue)	long
(m)	viewMessage(String)	MessageExt
(m)	queryMessage(String, String, int, long, long)	QueryResult
(m)	viewMessage(String, String)	MessageExt

Powered by yFiles

- ```

1 //创建主题
2 void createTopic(final String key, final String newTopic, final int
queueNum) throws MQClientException;
```
- ```

1 //根据时间戳从队列中查找消息偏移量
2 long searchOffset(final MessageQueue mq, final long timestamp)
```
- ```

1 //查找消息队列中最大的偏移量
2 long maxOffset(final MessageQueue mq) throws MQClientException;
```
- ```

1 //查找消息队列中最小的偏移量
2 long minOffset(final MessageQueue mq)
```
- ```

1 //根据偏移量查找消息
2 MessageExt viewMessage(final String offsetMsgId) throws
RemotingException, MQBrokerException,
InterruptedException, MQClientException;
3
```
- ```

1 //根据条件查找消息
2 QueryResult queryMessage(final String topic, final String key, final
int maxNum, final long begin,
final long end) throws MQClientException,
InterruptedException;
```
- ```

1 //根据消息ID和主题查找消息
2 MessageExt viewMessage(String topic, String msgId) throws
RemotingException, MQBrokerException, InterruptedException,
MQClientException;
```

| MQProducer                                                              |                       |
|-------------------------------------------------------------------------|-----------------------|
| (m) start()                                                             | void                  |
| (m) shutdown()                                                          | void                  |
| (m) fetchPublishMessageQueues(String)                                   | List<MessageQueue>    |
| (m) send(Message)                                                       | SendResult            |
| (m) send(Message, long)                                                 | SendResult            |
| (m) send(Message, SendCallback)                                         | void                  |
| (m) send(Message, SendCallback, long)                                   | void                  |
| (m) sendOneway(Message)                                                 | void                  |
| (m) send(Message, MessageQueue)                                         | SendResult            |
| (m) send(Message, MessageQueue, long)                                   | SendResult            |
| (m) send(Message, MessageQueue, SendCallback)                           | void                  |
| (m) send(Message, MessageQueue, SendCallback, long)                     | void                  |
| (m) sendOneway(Message, MessageQueue)                                   | void                  |
| (m) send(Message, MessageQueueSelector, Object)                         | SendResult            |
| (m) send(Message, MessageQueueSelector, Object, long)                   | SendResult            |
| (m) send(Message, MessageQueueSelector, Object, SendCallback)           | void                  |
| (m) send(Message, MessageQueueSelector, Object, SendCallback, long)     | void                  |
| (m) sendOneway(Message, MessageQueueSelector, Object)                   | void                  |
| (m) sendMessageInTransaction(Message, LocalTransactionExecuter, Object) | TransactionSendResult |
| (m) sendMessageInTransaction(Message, Object)                           | TransactionSendResult |
| (m) send(Collection<Message>)                                           | SendResult            |
| (m) send(Collection<Message>, long)                                     | SendResult            |
| (m) send(Collection<Message>, MessageQueue)                             | SendResult            |
| (m) send(Collection<Message>, MessageQueue, long)                       | SendResult            |

Powered by yFiles

- ```

1 //启动
2 void start() throws MQClientException;

```
- ```

1 //关闭
2 void shutdown();

```
- ```

1 //查找该主题下所有消息
2 List<MessageQueue> fetchPublishMessageQueues(final String topic)
    throws MQClientException;

```
- ```

1 //同步发送消息
2 SendResult send(final Message msg) throws MQClientException,
 RemotingException, MQBrokerException,
 InterruptedException;
3

```
- ```

1 //同步超时发送消息
2 SendResult send(final Message msg, final long timeout) throws
    MQClientException,
    RemotingException, MQBrokerException, InterruptedException;
3

```

- ```
1 //异步发送消息
2 void send(final Message msg, final SendCallback sendCallback) throws
MQClientException,
RemotingException, InterruptedException;
```
- ```
1 //异步超时发送消息
2 void send(final Message msg, final SendCallback sendCallback, final
long timeout)
throws MQClientException, RemotingException,
InterruptedException;
```
- ```
1 //发送单向消息
2 void sendOneWay(final Message msg) throws MQClientException,
RemotingException,
InterruptedException;
```
- ```
1 //选择指定队列同步发送消息
2 SendResult send(final Message msg, final MessageQueue mq) throws
MQClientException,
RemotingException, MQBrokerException, InterruptedException;
```
- ```
1 //选择指定队列异步发送消息
2 void send(final Message msg, final MessageQueue mq, final
SendCallback sendCallback)
throws MQClientException, RemotingException,
InterruptedException;
```
- ```
1 //选择指定队列单项发送消息
2 void sendOneWay(final Message msg, final MessageQueue mq) throws
MQClientException,
RemotingException, InterruptedException;
```
- ```
1 //批量发送消息
2 SendResult send(final Collection<Message> msgs) throws
MQClientException, RemotingException,
MQBrokerException, InterruptedException;
```

### 5.3.1.2 属性介绍

| DefaultMQProducer |                                  |                       |
|-------------------|----------------------------------|-----------------------|
| f                 | log                              | InternalLogger        |
| f                 | defaultMQProducerImpl            | DefaultMQProducerImpl |
| f                 | producerGroup                    | String                |
| f                 | createTopicKey                   | String                |
| f                 | defaultTopicQueueNums            | int                   |
| f                 | sendMsgTimeout                   | int                   |
| f                 | compressMsgBodyOverHowmuch       | int                   |
| f                 | retryTimesWhenSendFailed         | int                   |
| f                 | retryTimesWhenSendAsyncFailed    | int                   |
| f                 | retryAnotherBrokerWhenNotStoreOK | boolean               |
| f                 | maxMessageSize                   | int                   |
| f                 | traceDispatcher                  | TraceDispatcher       |

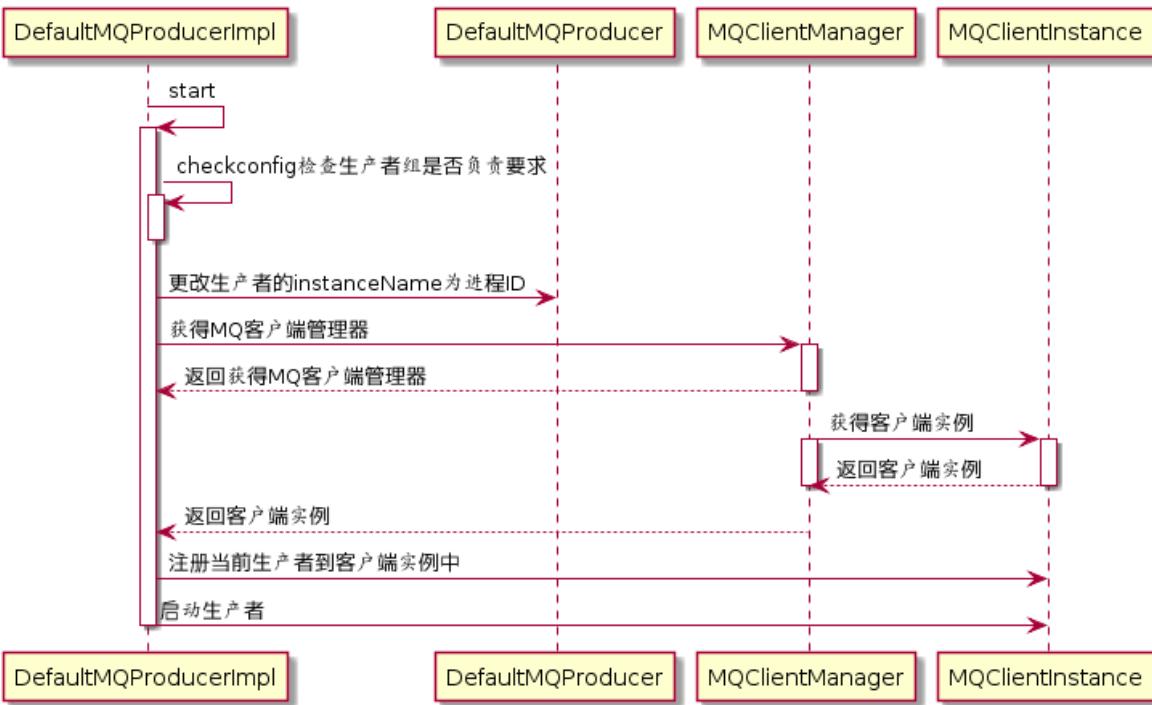
Powered by yFiles

```

1 producerGroup: 生产者所属组
2 createTopicKey: 默认Topic
3 defaultTopicQueueNums: 默认主题在每一个Broker队列数量
4 sendMsgTimeout: 发送消息默认超时时间, 默认3s
5 compressMsgBodyOverHowmuch: 消息体超过该值则启用压缩, 默认4k
6 retryTimesWhenSendFailed: 同步方式发送消息重试次数, 默认为2, 总共执行3次
7 retryTimesWhenSendAsyncFailed: 异步方法发送消息重试次数, 默认为2
8 retryAnotherBrokerWhenNotStoreOK: 消息重试时选择另外一个Broker时, 是否不等待存储结果就返回, 默认为false
9 maxMessageSize: 允许发送的最大消息长度, 默认为4M

```

### 5.3.2 启动流程



代码: `DefaultMQProducerImpl#start`

```

1 //检查生产者组是否满足要求
2 this.checkConfig();
3 //更改当前instanceName为进程ID
4 if
5 (!this.defaultMQProducer.getProducerGroup().equals(MixAll.CLIENT_INNER_PRODUCER_GROUP)) {
6 this.defaultMQProducer.changeInstanceIdToPID();
7 }
8 //获得MQ客户端实例
9 this.mqClientFactory =
10 MQClientManager.getInstance().getAndCreateMQClientInstance(this.defaultMQProducer, rpcHook);

```

整个JVM中只存在一个MQClientManager实例，维护一个MQClientInstance缓存表

```

ConcurrentMap<String/* clientId */, MQClientInstance> factoryTable = new
ConcurrentHashMap<String,MQClientInstance>();

```

同一个clientId只会创建一个MQClientInstance。

**MQClientInstance**封装了RocketMQ网络处理API，是消息生产者和消息消费者与NameServer、Broker打交道的网络通道

代码: `MQClientManager#getAndCreateMQClientInstance`

```

1 public MQClientInstance getAndCreateMQClientInstance(final ClientConfig
clientConfig,
2 RPCHook rpcHook) {
3
4 //构建客户端ID
5 String clientId = clientConfig.buildMQClientId();
6
7 //根据客户端ID或者客户端实例
8 MQClientInstance instance = this.factoryTable.get(clientId);
9
10 //实例如果为空就创建新的实例，并添加到实例表中
11 if (null == instance) {
12 instance =

```

```

10 new MQClientInstance(clientConfig.cloneClientConfig(),
11 this.factoryIndexGenerator.getAndIncrement(), clientId,
12 rpcHook);
13 MQClientInstance prev = this.factoryTable.putIfAbsent(clientId,
14 instance);
15 if (prev != null) {
16 instance = prev;
17 log.warn("Returned Previous MQClientInstance for clientId:" +
18 "{}", clientId);
19 } else {
20 log.info("Created new MQClientInstance for clientId:{}",
21 clientId);
22 }
23 }
24
25 return instance;
26 }

```

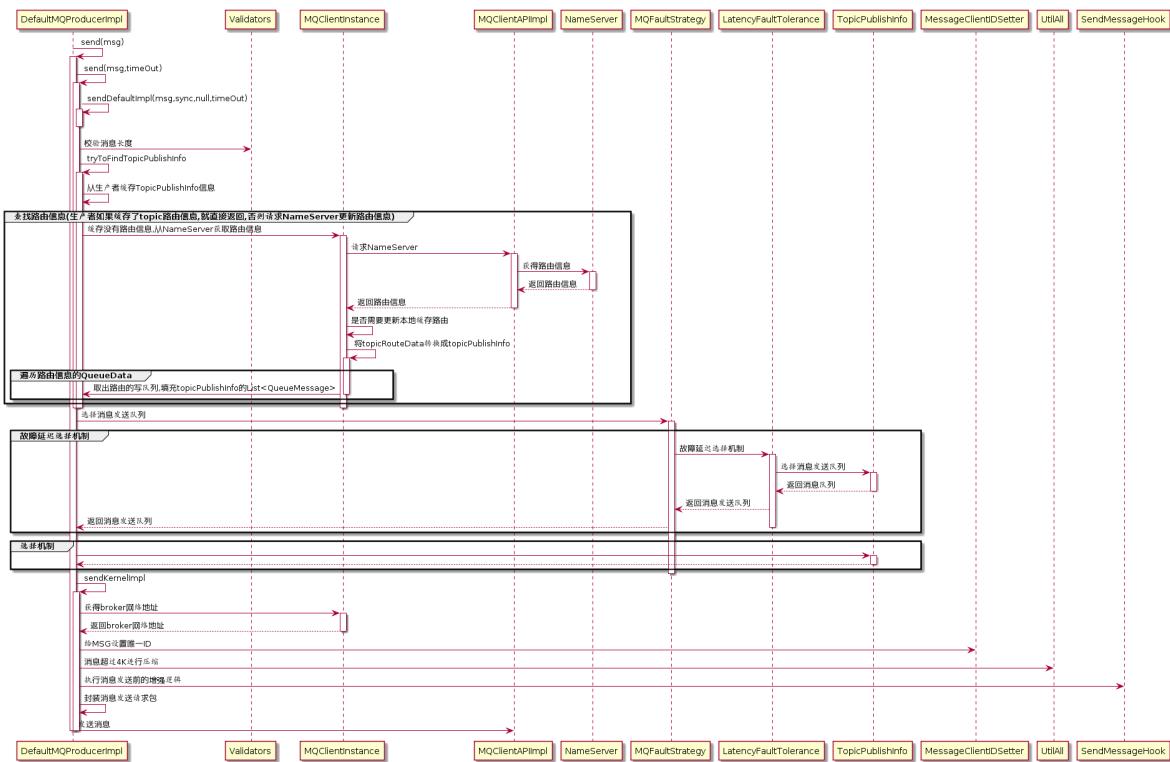
**代码: DefaultMQProducerImpl#start**

```

1 //注册当前生产者到到MQClientInstance管理中,方便后续调用网路请求
2 boolean registerOK =
3 mQClientFactory.registerProducer(this.defaultMQProducer.getProducerGroup(),
4 this);
5 if (!registerOK) {
6 this.serviceState = ServiceState.CREATE_JUST;
7 throw new MQClientException("The producer group[" +
8 this.defaultMQProducer.getProducerGroup()
9 + "] has been created before, specify another name please." +
10 FAQUrl.suggestTodo(FAQUrl.GROUP_NAME_DUPLICATE_URL),
11 null);
12 }
13 //启动生产者
14 if (startFactory) {
15 mQClientFactory.start();
16 }

```

### 5.3.3 消息发送



代码: `DefaultMQProducerImpl#send(Message msg)`

```

1 //发送消息
2 public SendResult send(Message msg) {
3 return send(msg, this.defaultMQProducer.getSendMsgTimeout());
4 }
```

代码: `DefaultMQProducerImpl#send(Message msg,long timeout)`

```

1 //发送消息,默认超时时间为3s
2 public SendResult send(Message msg, long timeout){
3 return this.sendDefaultImpl(msg, CommunicationMode.SYNC, null, timeout);
4 }
```

代码: `DefaultMQProducerImpl#sendDefaultImpl`

```

1 //校验消息
2 validators.checkMessage(msg, this.defaultMQProducer);
```

### 5.3.3.1 验证消息

代码: `Validators#checkMessage`

```

1 public static void checkMessage(Message msg, DefaultMQProducer
defaultMQProducer)
2 throws MQClientException {
3 //判断是否为空
4 if (null == msg) {
5 throw new MQClientException(ResponseCode.MESSAGE_ILLEGAL, "the
message is null");
6 }
7 // 校验主题
8 validators.checkTopic(msg.getTopic());
```

```

10 // 校验消息体
11 if (null == msg.getBody()) {
12 throw new MQClientException(ResponseCode.MESSAGE_ILLEGAL, "the
13 message body is null");
14 }
15
16 if (0 == msg.getBody().length) {
17 throw new MQClientException(ResponseCode.MESSAGE_ILLEGAL, "the
18 message body length is zero");
19 }
20
21 if (msg.getBody().length > defaultMQProducer.getMaxMessageSize()) {
22 throw new MQClientException(ResponseCode.MESSAGE_ILLEGAL,
23 "the message body size over max value, MAX: " +
24 defaultMQProducer.getMaxMessageSize());
25 }

```

### 5.3.3.2 查找路由

**代码: DefaultMQProducerImpl#tryToFindTopicPublishInfo**

```

1 private TopicPublishInfo tryToFindTopicPublishInfo(final String topic) {
2 //从缓存中获得主题的路由信息
3 TopicPublishInfo topicPublishInfo =
4 this.topicPublishInfoTable.get(topic);
5 //路由信息为空，则从NameServer获取路由
6 if (null == topicPublishInfo || !topicPublishInfo.ok()) {
7 this.topicPublishInfoTable.putIfAbsent(topic, new
8 TopicPublishInfo());
9 this.mQClientFactory.updateTopicRouteInfoFromNameServer(topic);
10 topicPublishInfo = this.topicPublishInfoTable.get(topic);
11 }
12
13 if (topicPublishInfo.isHaveTopicRouterInfo() || topicPublishInfo.ok())
14 {
15 return topicPublishInfo;
16 } else {
17 //如果未找到当前主题的路由信息，则用默认主题继续查找
18 this.mQClientFactory.updateTopicRouteInfoFromNameServer(topic,
19 true, this.defaultMQProducer);
20 topicPublishInfo = this.topicPublishInfoTable.get(topic);
21 return topicPublishInfo;
22 }
23 }

```

| TopicPublishInfo |                                     |
|------------------|-------------------------------------|
| f                | orderTopic boolean                  |
| f                | haveTopicRouterInfo boolean         |
| f                | messageQueueList List<MessageQueue> |
| f                | sendWhichQueue ThreadLocalIndex     |
| f                | topicRouteData TopicRouteData       |

| TopicRouteData |                                                 |
|----------------|-------------------------------------------------|
| f              | orderTopicConf String                           |
| f              | queueDatas List<QueueData>                      |
| f              | brokerDatas List<BrokerData>                    |
| f              | filterServerTable HashMap<String, List<String>> |

Powered by yFiles

**代码: TopicPublishInfo**

```

1 public class TopicPublishInfo {
2 private boolean orderTopic = false; //是否是顺序消息
3 private boolean haveTopicRouterInfo = false;
4 private List<MessageQueue> messageQueueList = new ArrayList<MessageQueue>
5 (); //该主题消息队列
6 private volatile ThreadLocalIndex sendwhichQueue = new
7 ThreadLocalIndex(); //每选择一次消息队列,该值+1
8 private TopicRouteData topicRouteData; //关联Topic路由元信息
9 }

```

**代码: MQClientInstance#updateTopicRouteInfoFromNameServer**

```

1 TopicRouteData topicRouteData;
2 //使用默认主题从NameServer获取路由信息
3 if (isDefault && defaultMQProducer != null) {
4 // TBW102
5 topicRouteData =
6 this.mQClientAPIImpl.getDefaultTopicRouteInfoFromNameServer(defaultMQProducer
7 .getCreateTopicKey(),
8 1000 * 3);
9 if (topicRouteData != null) {
10 for (QueueData data : topicRouteData.getQueueDatas()) {
11 int queueNums =
12 Math.min(defaultMQProducer.getDefaultTopicQueueNums(),
13 data.getReadQueueNums());
14 data.setReadQueueNums(queueNums);
15 data.setWriteQueueNums(queueNums);
16 }
17 }
18 } else {
19 //使用指定主题从NameServer获取路由信息
20 topicRouteData =
21 this.mQClientAPIImpl.getTopicRouteInfoFromNameServer(topic, 1000 * 3);
22 }

```

**代码: MQClientInstance#updateTopicRouteInfoFromNameServer**

```

1 //判断路由是否需要更改
2 TopicRouteData old = this.topicRouteTable.get(topic);
3 boolean changed = topicRouteDataIsChange(old, topicRouteData);
4 if (!changed) {
5 changed = this.isNeedUpdateTopicRouteInfo(topic);
6 } else {
7 log.info("the topic[{}] route info changed, old[{}], new[{}]", topic,
8 old, topicRouteData);
9 }

```

**代码: MQClientInstance#updateTopicRouteInfoFromNameServer**

```

1 if (changed) {
2 //将topicRouteData转换为发布队列
3 TopicPublishInfo publishInfo = topicRouteData2TopicPublishInfo(topic,
4 topicRouteData);
5 publishInfo.setHaveTopicRouterInfo(true);
6 //遍历生产
7 }

```

```

6 Iterator<Entry<String, MQProducerInner>> it =
7 this.producerTable.entrySet().iterator();
8 while (it.hasNext()) {
9 Entry<String, MQProducerInner> entry = it.next();
10 MQProducerInner impl = entry.getValue();
11 if (impl != null) {
12 //生产者不为空时，更新publishInfo信息
13 impl.updateTopicPublishInfo(topic, publishInfo);
14 }
15 }

```

**代码: MQClientInstance#topicRouteData2TopicPublishInfo**

```

1 public static TopicPublishInfo topicRouteData2TopicPublishInfo(final String
topic, final TopicRouteData route) {
2 //创建TopicPublishInfo对象
3 TopicPublishInfo info = new TopicPublishInfo();
4 //关联topicRoute
5 info.setTopicRouteData(route);
6 //顺序消息，更新TopicPublishInfo
7 if (route.getOrderTopicConf() != null &&
route.getOrderTopicConf().length() > 0) {
8 String[] brokers = route.getOrderTopicConf().split(";");
9 for (String broker : brokers) {
10 String[] item = broker.split(":");
11 int nums = Integer.parseInt(item[1]);
12 for (int i = 0; i < nums; i++) {
13 MessageQueue mq = new MessageQueue(topic, item[0], i);
14 info.getMessageQueueList().add(mq);
15 }
16 }
17
18 info.setorderTopic(true);
19 } else {
20 //非顺序消息更新TopicPublishInfo
21 List<QueueData> qds = route.getQueueDatas();
22 Collections.sort(qds);
23 //遍历topic队列信息
24 for (QueueData qd : qds) {
25 //是否是写队列
26 if (PermName.isWriteable(qd.getPerm())) {
27 BrokerData brokerData = null;
28 //遍历写队列Broker
29 for (BrokerData bd : route.getBrokerDatas()) {
30 //根据名称获得读队列对应的Broker
31 if (bd.getBrokerName().equals(qd.getBrokerName()))
32 brokerData = bd;
33 break;
34 }
35 }
36
37 if (null == brokerData) {
38 continue;
39 }
40 }

```

```

41 if
42 (!brokerData.getBrokerAddrs().containsKey(MixAll.MASTER_ID)) {
43 continue;
44 }
45 //封装TopicPublishInfo写队列
46 for (int i = 0; i < qd.getWriterQueueNums(); i++) {
47 MessageQueue mq = new MessageQueue(topic,
48 qd.getBrokerName(), i);
49 info.getMessageQueueList().add(mq);
50 }
51 }
52 info.setOrderTopic(false);
53 }
54 //返回TopicPublishInfo对象
55 return info;
56 }
```

### 5.3.3.3 选择队列

- 默认不启用Broker故障延迟机制

**代码: TopicPublishInfo#selectOneMessageQueue(lastBrokerName)**

```

1 public MessageQueue selectOneMessageQueue(final String lastBrokerName) {
2 //第一次选择队列
3 if (lastBrokerName == null) {
4 return selectOneMessageQueue();
5 } else {
6 //sendwhichQueue
7 int index = this.sendwhichQueue.getAndIncrement();
8 //遍历消息队列集合
9 for (int i = 0; i < this.messageQueueList.size(); i++) {
10 //sendwhichQueue自增后取模
11 int pos = Math.abs(index++) % this.messageQueueList.size();
12 if (pos < 0)
13 pos = 0;
14 //规避上次Broker队列
15 MessageQueue mq = this.messageQueueList.get(pos);
16 if (!mq.getBrokerName().equals(lastBrokerName))
17 return mq;
18 }
19 }
20 //如果以上情况都不满足,返回sendwhichQueue取模后的队列
21 return selectOneMessageQueue();
22 }
23 }
```

**代码: TopicPublishInfo#selectOneMessageQueue()**

```

1 //第一次选择队列
2 public MessageQueue selectOneMessageQueue() {
3 //sendwhichQueue自增
4 int index = this.sendwhichQueue.getAndIncrement();
5 //对队列大小取模
6 int pos = Math.abs(index) % this.messageQueueList.size();
7 if (pos < 0)
8 pos = 0;
9 //返回对应的队列
10 return this.messageQueueList.get(pos);
11 }

```

- 启用Broker故障延迟机制

```

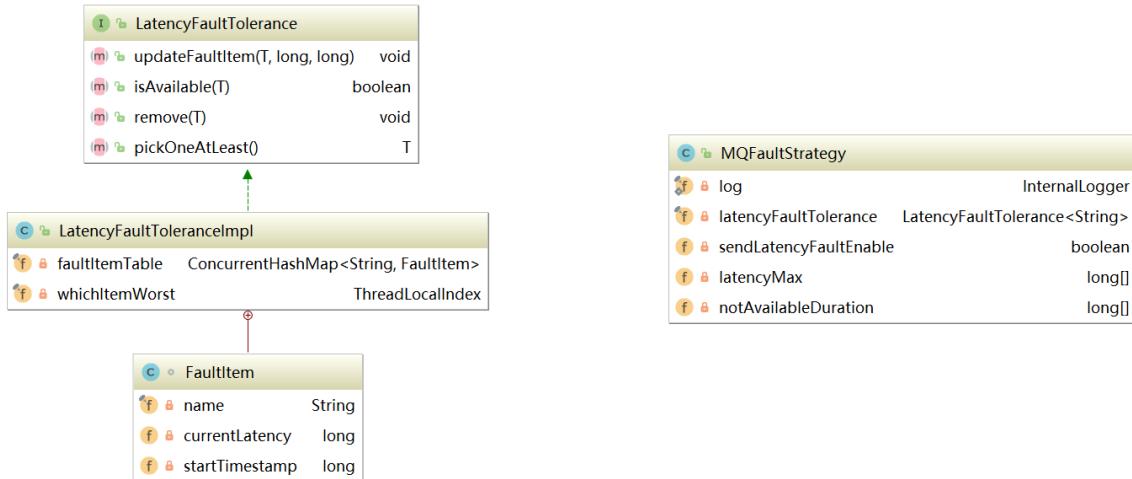
1 public MessageQueue selectOneMessageQueue(final TopicPublishInfo tpInfo,
2 final String lastBrokerName) {
3 //Broker故障延迟机制
4 if (this.sendLatencyFaultEnable) {
5 try {
6 //对sendwhichQueue自增
7 int index = tpInfo.getSendwhichQueue().getAndIncrement();
8 //对消息队列轮询获取一个队列
9 for (int i = 0; i < tpInfo.getMessageQueueList().size(); i++) {
10 int pos = Math.abs(index++) %
11 tpInfo.getMessageQueueList().size();
12 if (pos < 0)
13 pos = 0;
14 MessageQueue mq = tpInfo.getMessageQueueList().get(pos);
15 //验证该队列是否可用
16 if (latencyFaultTolerance.isAvailable(mq.getBrokerName()))
17 {
18 //可用
19 if (null == lastBrokerName ||
20 mq.getBrokerName().equals(lastBrokerName))
21 return mq;
22 }
23 }
24 //从规避的Broker中选择一个可用的Broker
25 final String notBestBroker =
26 latencyFaultTolerance.pickOneAtLeast();
27 //获得Broker的写队列集合
28 int writeQueueNums = tpInfo.getQueueIdByBroker(notBestBroker);
29 if (writeQueueNums > 0) {
30 //获得一个队列，指定broker和队列ID并返回
31 final MessageQueue mq = tpInfo.selectOneMessageQueue();
32 if (notBestBroker != null) {
33 mq.setBrokerName(notBestBroker);
34
35 mq.setQueueId(tpInfo.getSendwhichQueue().getAndIncrement() %
36 writeQueueNums);
37 }
38 return mq;
39 } else {
40 latencyFaultTolerance.remove(notBestBroker);
41 }
42 } catch (Exception e) {

```

```

36 log.error("Error occurred when selecting message queue", e);
37 }
38
39 return tpInfo.selectOneMessageQueue();
40 }
41
42 return tpInfo.selectOneMessageQueue(lastBrokerName);
43 }

```



- 延迟机制接口规范

```

1 public interface LatencyFaultTolerance<T> {
2 //更新失败条目
3 void updateFaultItem(final T name, final long currentLatency, final
4 long notAvailableDuration);
5 //判断Broker是否可用
6 boolean isAvailable(final T name);
7 //移除Fault条目
8 void remove(final T name);
9 //尝试从规避的Broker中选择一个可用的Broker
10 T pickOneAtLeast();
}

```

- FaultItem: 失败条目
- 如果向一个brokerName的broker发送消息失败，等待多长时间重试该broker

```

1 class FaultItem implements Comparable<FaultItem> {
2 //条目唯一键,这里为brokerName
3 private final String name;
4 //本次消息发送延迟
5 private volatile long currentLatency;
6 //故障规避开始时间
7 private volatile long startTimestamp;
8 }

```

- 消息失败策略

```
1 public class MQFaultStrategy {
2 //根据currentLatency本地消息发送延迟,从latencyMax尾部向前找到第一个比
3 //currentLatency小的索引,如果没有找到,返回0
4 private long[] latencyMax = {50L, 100L, 550L, 1000L, 2000L, 3000L,
5 15000L};
6 //根据这个索引从notAvailableDuration取出对应的时间,在该时长内,Broker设置为不可用
7 private long[] notAvailableDuration = {0L, 0L, 30000L, 60000L, 120000L,
8 180000L, 600000L};
9 }
```

## 原理分析

代码: `DefaultMQProducerImpl#sendDefaultImpl`

```
1 sendResult = this.sendKernelImpl(msg,
2 mq,
3 communicationMode,
4 sendCallback,
5 topicPublishInfo,
6 timeout - costTime);
7 endTimestamp = System.currentTimeMillis();
8 this.updateFaultItem(mq.getBrokerName(), endTimestamp - beginTimestampPrev,
9 false);
```

如果上述发送过程出现异常, 则调用 `DefaultMQProducerImpl#updateFaultItem`

```
1 public void updateFaultItem(final String brokerName, final long
2 currentLatency, boolean isolation) {
3 //参数一: broker名称
4 //参数二:本次消息发送延迟时间
5 //参数三:是否隔离
6 this.mqFaultStrategy.updateFaultItem(brokerName, currentLatency,
isolation);
7 }
```

代码: `MQFaultStrategy#updateFaultItem`

```
1 public void updateFaultItem(final String brokerName, final long
2 currentLatency, boolean isolation) {
3 if (this.sendLatencyFaultEnable) {
4 //计算broker规避的时长
5 long duration = computeNotAvailableDuration(isolation ? 30000 :
6 currentLatency);
7 //更新该FaultItem规避时长
8 this.latencyFaultTolerance.updateFaultItem(brokerName,
currentLatency, duration);
9 }
10 }
```

代码: `MQFaultStrategy#computeNotAvailableDuration`

```

1 private long computeNotAvailableDuration(final long currentLatency) {
2 //遍历latencyMax
3 for (int i = latencyMax.length - 1; i >= 0; i--) {
4 //找到第一个比currentLatency的latencyMax值
5 if (currentLatency >= latencyMax[i])
6 return this.notAvailableDuration[i];
7 }
8 //没有找到则返回0
9 return 0;
10 }

```

**代码:** *LatencyFaultToleranceImpl#updateFaultItem*

```

1 public void updateFaultItem(final String name, final long currentLatency,
2 final long notAvailableDuration) {
3 //获得原FaultItem
4 FaultItem old = this.faultItemTable.get(name);
5 //为空新建faultItem对象，设置规避时长和开始时间
6 if (null == old) {
7 final FaultItem faultItem = new FaultItem(name);
8 faultItem.setCurrentLatency(currentLatency);
9 faultItem.setStartTimestamp(System.currentTimeMillis() +
10 notAvailableDuration);
11
12 old = this.faultItemTable.putIfAbsent(name, faultItem);
13 if (old != null) {
14 old.setCurrentLatency(currentLatency);
15 old.setStartTimestamp(System.currentTimeMillis() +
16 notAvailableDuration);
17 }
18 } else {
19 //更新规避时长和开始时间
20 old.setCurrentLatency(currentLatency);
21 old.setStartTimestamp(System.currentTimeMillis() +
22 notAvailableDuration);
23 }
24 }

```

#### 5.3.3.4 发送消息

消息发送API核心入口*DefaultMQProducerImpl#sendKernelImpl*

```

1 private SendResult sendKernelImpl(
2 final Message msg, //待发送消息
3 final MessageQueue mq, //消息发送队列
4 final CommunicationMode communicationMode, //消息发送内模式
5 final SendCallback sendCallback, pp //异步消息回调函数
6 final TopicPublishInfo topicPublishInfo, //主题路由信息
7 final long timeout //超时时间
8)

```

**代码:** *DefaultMQProducerImpl#sendKernelImpl*

```
1 //获得broker网络地址信息
2 String brokerAddr =
3 this.mQClientFactory.findBrokerAddressInPublish(mq.getBrokerName());
4 if (null == brokerAddr) {
5 //没有找到从NameServer更新broker网络地址信息
6 tryToFindTopicPublishInfo(mq.getTopic());
7 brokerAddr =
8 this.mQClientFactory.findBrokerAddressInPublish(mq.getBrokerName());
9 }
```

```
1 //为消息分类唯一ID
2 if (!(msg instanceof MessageBatch)) {
3 MessageClientIDSetter.setUniqID(msg);
4 }
5
6 boolean topicwithNamespace = false;
7 if (null != this.mQClientFactory.getClientConfig().getNamespace()) {
8
9 msg.setInstanceId(this.mQClientFactory.getClientConfig().getNamespace());
10 topicwithNamespace = true;
11 }
12 //消息大小超过4K,启用消息压缩
13 int sysFlag = 0;
14 boolean msgBodyCompressed = false;
15 if (this.tryToCompressMessage(msg)) {
16 sysFlag |= MessageSysFlag.COMPRESSSED_FLAG;
17 msgBodyCompressed = true;
18 }
19 //如果是事务消息,设置消息标记MessageSysFlag.TRANSACTION_PREPARED_TYPE
20 final String tranMsg =
21 msg.getProperty(MessageConst.PROPERTY_TRANSACTION_PREPARED);
22 if (tranMsg != null && Boolean.parseBoolean(tranMsg)) {
23 sysFlag |= MessageSysFlag.TRANSACTION_PREPARED_TYPE;
24 }
```

```
1 //如果注册了消息发送钩子函数,在执行消息发送前的增强逻辑
2 if (this.hasSendMessageHook()) {
3 context = new SendMessageContext();
4 context.setProducer(this);
5 context.setProducerGroup(this.defaultMQProducer.getProducerGroup());
6 context.setCommunicationMode(communicationMode);
7 context.setBornHost(this.defaultMQProducer.getClientIP());
8 context.setBrokerAddr(brokerAddr);
9 context.setMessage(msg);
10 context.setMq(mq);
11 context.setNamespace(this.defaultMQProducer.getNamespace());
12 String isTrans =
13 msg.getProperty(MessageConst.PROPERTY_TRANSACTION_PREPARED);
14 if (isTrans != null && isTrans.equals("true")) {
15 context.setMsgType(MessageType.Trans_Msg_Half);
16 }
17 if (msg.getProperty("__STARTDELIVERTIME") != null ||
18 msg.getProperty(MessageConst.PROPERTY_DELAY_TIME_LEVEL) != null) {
19 context.setMsgType(MessageType.Delay_Msg);
20 }
```

```
20 this.executeSendMessageHookBefore(context);
21 }
```

**代码: SendMessageHook**

```
1 public interface SendMessageHook {
2 String hookName();
3
4 void sendMessageBefore(final SendMessageContext context);
5
6 void sendMessageAfter(final SendMessageContext context);
7 }
```

**代码: DefaultMQProducerImpl#sendKernelImpl**

```
1 //构建消息发送请求包
2 SendMessageRequestHeader requestHeader = new SendMessageRequestHeader();
3 //生产者组
4 requestHeader.setProducerGroup(this.defaultMQProducer.getProducerGroup());
5 //主题
6 requestHeader.setTopic(msg.getTopic());
7 //默认创建主题Key
8 requestHeader.setDefaultTopic(this.defaultMQProducer.getCreateTopicKey());
9 //该主题在单个Broker默认队列树
10 requestHeader.setDefaultTopicQueueNums(this.defaultMQProducer.getDefaultTopicQueueNums());
11 //队列ID
12 requestHeader.setQueueId(mq.getQueueId());
13 //消息系统标记
14 requestHeader.setSysFlag(sysFlag);
15 //消息发送时间
16 requestHeader.setBornTimestamp(System.currentTimeMillis());
17 //消息标记
18 requestHeader.setFlag(msg.getFlag());
19 //消息扩展信息
20 requestHeader.setProperties(MessageDecoder.messagePropertiesToString(msg.getProperties()));
21 //消息重试次数
22 requestHeader.setReconsumeTimes(0);
23 requestHeader.setUnitMode(this.isUnitMode());
24 //是否是批量消息等
25 requestHeader.setBatch(msg instanceof MessageBatch);
26 if (requestHeader.getTopic().startsWith(MixAll.RETRY_GROUP_TOPIC_PREFIX)) {
27 String reconsumeTimes = MessageAccessor.getReconsumeTime(msg);
28 if (reconsumeTimes != null) {
29 requestHeader.setReconsumeTimes(Integer.valueOf(reconsumeTimes));
30 MessageAccessor.clearProperty(msg,
31 MessageConst.PROPERTY_RECONSUME_TIME);
32 }
33 String maxReconsumeTimes = MessageAccessor.getMaxReconsumeTimes(msg);
34 if (maxReconsumeTimes != null) {
35 requestHeader.setMaxReconsumeTimes(Integer.valueOf(maxReconsumeTimes));
36 MessageAccessor.clearProperty(msg,
37 MessageConst.PROPERTY_MAX_RECONSUME_TIMES);
38 }
39 }
```

```
37 }
38 }
```

```
1 case ASYNC: //异步发送
2 Message tmpMessage = msg;
3 boolean messageCloned = false;
4 if (msgBodyCompressed) {
5 //If msg body was compressed, msgbody should be reset using
6 //prevBody.
7 //Clone new message using compressed message body and recover
8 //origin message.
9 //Fix bug:https://github.com/apache/rocketmq-externals/issues/66
10 tmpMessage = MessageAccessor.cloneMessage(msg);
11 messageCloned = true;
12 msg.setBody(prevBody);
13 }
14
15 if (topicwithNamespace) {
16 if (!messageCloned) {
17 tmpMessage = MessageAccessor.cloneMessage(msg);
18 messageCloned = true;
19 }
20 msg.setTopic(NamespaceUtil.withoutNamespace(msg.getTopic(),
21
22 this.defaultMQProducer.getNamespace()));
23 }
24
25 long costTimeAsync = System.currentTimeMillis() - beginStartTime;
26 if (timeout < costTimeAsync) {
27 throw new RemotingTooMuchRequestException("sendKernelImpl call
28 timeout");
29 }
30 sendResult = this.mQClientFactory.getMQClientAPIImpl().sendMessage(
31 brokerAddr,
32 mq.getBrokerName(),
33 tmpMessage,
34 requestHeader,
35 timeout - costTimeAsync,
36 communicationMode,
37 sendCallback,
38 topicPublishInfo,
39 this.mQClientFactory,
40
41 this.defaultMQProducer.getRetryTimesWhenSendAsyncFailed(),
42 context,
43 this);
44 break;
45
46 case ONEWAY:
47 case SYNC: //同步发送
48 long costTimeSync = System.currentTimeMillis() - beginStartTime;
49 if (timeout < costTimeSync) {
50 throw new RemotingTooMuchRequestException("sendKernelImpl call
51 timeout");
52 }
53 sendResult = this.mQClientFactory.getMQClientAPIImpl().sendMessage(
54 brokerAddr,
55 mq.getBrokerName(),
```

```

49 msg,
50 requestHeader,
51 timeout - costTimeSync,
52 communicationMode,
53 context,
54 this);
55 break;
56 default:
57 assert false;
58 break;
59 }

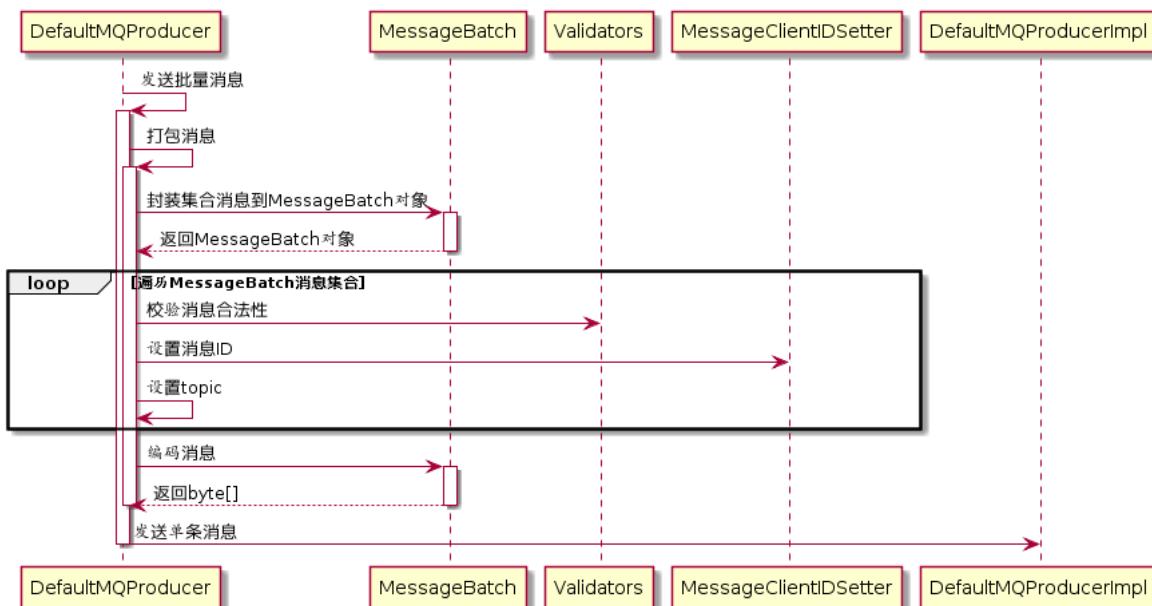
```

```

1 //如果注册了钩子函数，则发送完毕后执行钩子函数
2 if (this.hasSendMessageHook()) {
3 context.setSendResult(sendResult);
4 this.executeSendMessageHookAfter(context);
5 }

```

### 5.3.4 批量消息发送



批量消息发送是将同一个主题的多条消息一起打包发送到消息服务端，减少网络调用次数，提高网络传输效率。当然，并不是在同一批次中发送的消息数量越多越好，其判断依据是单条消息的长度，如果单条消息内容比较长，则打包多条消息发送会影响其他线程发送消息的响应时间，并且单批次消息总长度不能超过DefaultMQProducer#maxMessageSize。

批量消息发送要解决的问题是**如何将这些消息编码以便服务端能够正确解码出每条消息的内容**。

**代码: DefaultMQProducer#send**

```

1 public SendResult send(Collection<Message> msgs)
2 throws MQClientException, RemotingException, MQBrokerException,
3 InterruptedException {
4 //压缩消息集合成一条消息，然后发送出去
5 return this.defaultMQProducerImpl.send(batch(msgs));
6 }

```

**代码: DefaultMQProducer#batch**

```
1 private MessageBatch batch(Collection<Message> msgs) throws
2 MQClientException {
3 MessageBatch msgBatch;
4 try {
5 //将集合消息封装到MessageBatch
6 msgBatch = MessageBatch.generateFromList(msgs);
7 //遍历消息集合,检查消息合法性,设置消息ID,设置Topic
8 for (Message message : msgBatch) {
9 Validators.checkMessage(message, this);
10 MessageClientIDSetter.setUniqID(message);
11 message.setTopic(withNamespace(message.getTopic()));
12 }
13 //压缩消息,设置消息body
14 msgBatch.setBody(msgBatch.encode());
15 } catch (Exception e) {
16 throw new MQClientException("Failed to initiate the MessageBatch",
17 e);
18 }
19 //设置msgBatch的topic
20 msgBatch.setTopic(withNamespace(msgBatch.getTopic()));
21 return msgBatch;
22 }
```

## 5.4 消息存储

### 5.4.1 消息存储核心类

|          |                             |                                                             |
|----------|-----------------------------|-------------------------------------------------------------|
| <b>c</b> | <b>DefaultMessageStore</b>  |                                                             |
| .p       | OSPageCacheBusy             | boolean                                                     |
| .p       | minPhyOffset                | long                                                        |
| .p       | runningFlags                | RunningFlags                                                |
| .p       | accessRights                | RunningFlags                                                |
| .p       | storeStatsService           | StoreStatsService                                           |
| .p       | brokerStatsManager          | BrokerStatsManager                                          |
| .p       | allocateMappedFileService   | AllocateMappedFileService                                   |
| .p       | confirmOffset               | long                                                        |
| .p       | dispatcherList              | LinkedList<CommitLogDispatcher>                             |
| .p       | scheduleMessageService      | ScheduleMessageService                                      |
| .p       | transientStorePoolDeficient | boolean                                                     |
| .p       | consumeQueueTable           | ConcurrentMap<String, ConcurrentMap<Integer, ConsumeQueue>> |
| .p       | maxPhyOffset                | long                                                        |
| .p       | transientStorePool          | TransientStorePool                                          |
| .p       | runningDataInfo             | String                                                      |
| .p       | messageStoreConfig          | MessageStoreConfig                                          |
| .p       | systemClock                 | SystemClock                                                 |
| .p       | runtimeInfo                 | HashMap<String, String>                                     |
| .p       | storeCheckpoint             | StoreCheckpoint                                             |
| .p       | haService                   | HAService                                                   |
| .p       | commitLog                   | CommitLog                                                   |
| .p       | earliestMessageTime         | long                                                        |

Powered by yFiles

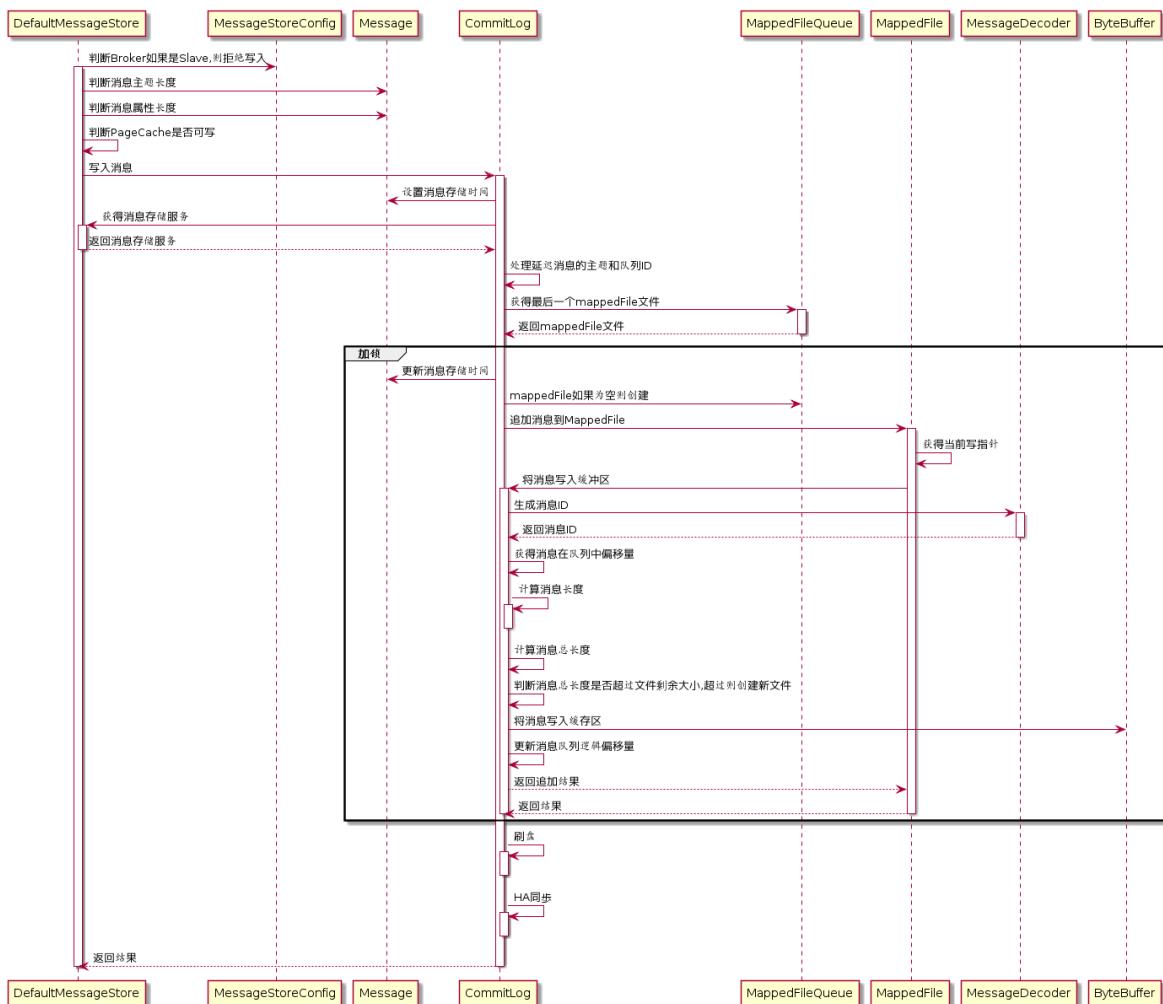
```

1 private final MessageStoreConfig messageStoreConfig; //消息配置属性
2 private final CommitLog commitLog; //CommitLog文件存储的实现类
3 private final ConcurrentMap<String/* topic */ , ConcurrentMap<Integer/* queueId */ , ConsumeQueue>> consumeQueueTable; //消息队列存储缓存表,按照消息主题分组
4 private final FlushConsumeQueueService flushConsumeQueueService; //消息队列文件刷盘线程
5 private final CleanCommitLogService cleanCommitLogService; //清除CommitLog文件服务
6 private final CleanConsumeQueueService cleanConsumeQueueService; //清除ConsumerQueue队列文件服务
7 private final IndexService indexService; //索引实现类
8 private final AllocateMappedFileService allocateMappedFileService; //MappedFile分配服务
9 private final ReputMessageService reputMessageService; //CommitLog消息分发,根据CommitLog文件构建ConsumerQueue、IndexFile文件
10 private final HAService haservice; //存储HA机制
11 private final ScheduleMessageService scheduleMessageService; //消息服务调度线程
12 private final StorestatsService storeStatsservice; //消息存储服务
13 private final TransientStorePool transientStorePool; //消息堆外内存缓存
14 private final BrokerStatsManager brokerStatsManager; //Broker状态管理器
15 private final MessageArrivingListener messageArrivingListener; //消息拉取长轮询模式消息达到监听器

```

```
16 private final BrokerConfig brokerConfig; //Broker配置类
17 private StoreCheckpoint storeCheckpoint; //文件刷盘监测点
18 private final LinkedList<CommitLogDispatcher> dispatcherList; //CommitLog
文件转发请求
```

## 5.4.2 消息存储流程



**消息存储入口: `DefaultMessageStore#putMessage`**

```
1 //判断Broker角色如果是从节点,则无需写入
2 if (BrokerRole.SLAVE == this.messageStoreConfig.getBrokerRole()) {
3 long value = this.printTimes.getAndIncrement();
4 if ((value % 50000) == 0) {
5 log.warn("message store is slave mode, so putMessage is
6 forbidden ");
7 }
8
9 return new PutMessageResult(PutMessageStatus.SERVICE_NOT_AVAILABLE,
10 null);
11
12 //判断当前写入状态如果是正在写入,则不能继续
13 if (!this.runningFlags.isWriteable()) {
14 long value = this.printTimes.getAndIncrement();
15 return new PutMessageResult(PutMessageStatus.SERVICE_NOT_AVAILABLE,
16 null);
17 } else {
```

```

16 this.printTimes.set(0);
17 }
18 //判断消息主题长度是否超过最大限制
19 if (msg.getTopic().length() > Byte.MAX_VALUE) {
20 log.warn("putMessage message topic length too long " +
21 msg.getTopic().length());
22 return new PutMessageResult(PutMessageStatus.MESSAGE_ILLEGAL, null);
23 }
24 //判断消息属性长度是否超过限制
25 if (msg.getPropertiesString() != null && msg.getPropertiesString().length() > Short.MAX_VALUE) {
26 log.warn("putMessage message properties length too long " +
27 msg.getPropertiesString().length());
28 return new PutMessageResult(PutMessageStatus.PROPERTIES_SIZE_EXCEEDED,
29 null);
30 }
31 //判断系统PageCache缓存去是否占用
32 if (this.isOSPageCacheBusy()) {
33 return new PutMessageResult(PutMessageStatus.OS_PAGECACHE_BUSY, null);
34 }
35 //将消息写入CommitLog文件
36 PutMessageResult result = this.commitLog.putMessage(msg);

```

#### 代码: CommitLog#putMessage

```

1 //记录消息存储时间
2 msg.setStoreTimestamp(beginLockTimestamp);
3
4 //判断如果mappedFile如果为空或者已满,创建新的mappedFile文件
5 if (null == mappedFile || mappedFile.isFull()) {
6 mappedFile = this.mappedFileQueue.getLastMappedFile(0);
7 }
8 //如果创建失败,直接返回
9 if (null == mappedFile) {
10 log.error("create mapped file1 error, topic: " + msg.getTopic() + " "
11 "clientAddr: " + msg.getBornHostString());
12 beginTimeInLock = 0;
13 return new PutMessageResult(PutMessageStatus.CREATE_MAPEDFILE_FAILED,
14 null);
15 }
16 //写入消息到mappedFile中
17 result = mappedFile.appendMessage(msg, this.appendMessageCallback);

```

#### 代码: MappedFile#appendMessagesInner

```

1 //获得文件的写入指针
2 int currentPos = this.wrotePosition.get();
3
4 //如果指针大于文件大小则直接返回
5 if (currentPos < this.filesize) {
6 //通过writeBuffer.slice()创建一个与MappedFile共享的内存区,并设置position为当前
7 //指针
8 ByteBuffer byteBuffer = writeBuffer != null ? writeBuffer.slice() :
9 this.mappedByteBuffer.slice();

```

```

8 byteBuffer.position(currentPos);
9 AppendMessageResult result = null;
10 if (messageExt instanceof MessageExtBrokerInner) {
11 //通过回调方法写入
12 result = cb.doAppend(this.getFileFromOffset(), byteBuffer,
13 this.filesize - currentPos, (MessageExtBrokerInner) messageExt);
14 } else if (messageExt instanceof MessageExtBatch) {
15 result = cb.doAppend(this.getFileFromOffset(), byteBuffer,
16 this.filesize - currentPos, (MessageExtBatch) messageExt);
17 } else {
18 return new AppendMessageResult(AppendMessageStatus.UNKNOWN_ERROR);
19 }
20 this.wrotePosition.addAndGet(result.getWroteBytes());
21 this.storeTimestamp = result.getStoreTimestamp();
22 return result;
23 }
```

**代码: CommitLog#doAppend**

```

1 //文件写入位置
2 long wroteOffset = fileFromOffset + byteBuffer.position();
3 //设置消息ID
4 this.resetByteBuffer(hostHolder, 8);
5 String msgId = MessageDecoder.createMessageId(this.msgIdMemory,
6 msgInner.getStoreHostBytes(hostHolder), wroteOffset);
7
8 //获得该消息在消息队列中的偏移量
9 keyBuilder.setLength(0);
10 keyBuilder.append(msgInner.getTopic());
11 keyBuilder.append('-');
12 keyBuilder.append(msgInner.getQueueId());
13 String key = keyBuilder.toString();
14 Long queueOffset = CommitLog.this.topicQueueTable.get(key);
15 if (null == queueOffset) {
16 queueOffset = 0L;
17 CommitLog.this.topicQueueTable.put(key, queueOffset);
18 }
19
20 //获得消息属性长度
21 final byte[] propertiesData =msgInner.getPropertiesString() == null ? null
22 : msgInner.getPropertiesString().getBytes(MessageDecoder.CHARSET_UTF8);
23
24 final int propertiesLength = propertiesData == null ? 0 :
25 propertiesData.length;
26
27 if (propertiesLength > Short.MAX_VALUE) {
28 log.warn("putMessage message properties length too long. length={}",
29 propertiesData.length);
30 return new
31 AppendMessageResult(AppendMessageStatus.PROPERTIES_SIZE_EXCEEDED);
32 }
33
34 //获得消息主题大小
35 final byte[] topicData =
36 msgInner.getTopic().getBytes(MessageDecoder.CHARSET_UTF8);
37 final int topicLength = topicData.length;
38 }
```

```

33 //获得消息体大小
34 final int bodyLength = msgInner.getBody() == null ? 0 :
msgInner.getBody().length;
35 //计算消息总长度
36 final int msgLen = calMsgLength(bodyLength, topicLength, propertiesLength);

```

**代码: CommitLog#calMsgLength**

```

1 protected static int calMsgLength(int bodyLength, int topicLength, int
propertiesLength) {
2 final int msgLen = 4 //TOTALSIZE
3 + 4 //MAGICCODE
4 + 4 //BODYCRC
5 + 4 //QUEUEID
6 + 4 //FLAG
7 + 8 //QUEUEOFFSET
8 + 8 //PHYSICALOFFSET
9 + 4 //SYSFLAG
10 + 8 //BORNTIMESTAMP
11 + 8 //BORNHOST
12 + 8 //STORETIMESTAMP
13 + 8 //STOREHOSTADDRESS
14 + 4 //RECONSUMETIMES
15 + 8 //Prepared Transaction offset
16 + 4 + (bodyLength > 0 ? bodyLength : 0) //BODY
17 + 1 + topicLength //TOPIC
18 + 2 + (propertiesLength > 0 ? propertiesLength : 0)
//propertiesLength
19 + 0;
20 return msgLen;
21 }

```

**代码: CommitLog#doAppend**

```

1 //消息长度不能超过4M
2 if (msgLen > this.maxMessageSize) {
3 CommitLog.log.warn("message size exceeded, msg total size: " + msgLen +
", msg body size: " + bodyLength
4 + ", maxMessageSize: " + this.maxMessageSize);
5 return new
AppendMessageResult(AppendMessageStatus.MESSAGE_SIZE_EXCEEDED);
6 }
7
8 //消息是如果没有足够的存储空间则新创建CommitLog文件
9 if ((msgLen + END_FILE_MIN_BLANK_LENGTH) > maxBlank) {
10 this.resetByteBuffer(this.msgStoreItemMemory, maxBlank);
11 // 1 TOTALSIZE
12 this.msgStoreItemMemory.putInt(maxBlank);
13 // 2 MAGICCODE
14 this.msgStoreItemMemory.putInt(CommitLog.BLANK_MAGIC_CODE);
15 // 3 The remaining space may be any value
16 // Here the length of the specially set maxBlank
17 final long beginTimeMills = CommitLog.this.defaultMessageStore.now();
18 byteBuffer.put(this.msgStoreItemMemory.array(), 0, maxBlank);
19 return new AppendMessageResult(AppendMessageStatus.END_OF_FILE,
wroteOffset, maxBlank, msgId, msgInner.getStoreTimestamp(),

```

```

20 queueOffset, CommitLog.this.defaultMessageStore.now() -
21 beginTimeMills);
22
23 //将消息存储到ByteBuffer中,返回AppendMessageResult
24 final long beginTimeMills = CommitLog.this.defaultMessageStore.now();
25 // write messages to the queue buffer
26 byteBuffer.put(this.msgStoreItemMemory.array(), 0, msgLen);
27 AppendMessageResult result = new
28 AppendMessageResult(AppendMessageStatus.PUT_OK, wroteOffset,
29 msgLen,
30 msgId, msgInner.getStoreTimestamp(),
31 queueoffset,
32 CommitLog.this.defaultMessageStore.now()
33 -beginTimeMills);
34 switch (tranType) {
35 case MessageSysFlag.TRANSACTION_PREPARED_TYPE:
36 case MessageSysFlag.TRANSACTION_ROLLBACK_TYPE:
37 break;
38 case MessageSysFlag.TRANSACTION_NOT_TYPE:
39 case MessageSysFlag.TRANSACTION_COMMIT_TYPE:
40 //更新消息队列偏移量
41 CommitLog.this.topicQueueTable.put(key, ++queueOffset);
42 break;
43 default:
44 break;
45 }

```

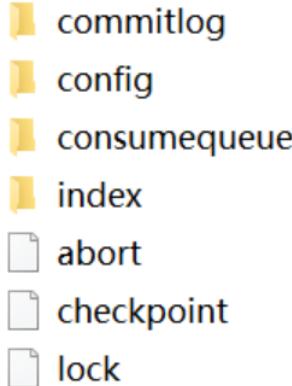
**代码: CommitLog#putMessage**

```

1 //释放锁
2 putMessageLock.unlock();
3 //刷盘
4 handleDiskFlush(result, putMessageResult, msg);
5 //执行HA主从同步
6 handleHA(result, putMessageResult, msg);

```

### 5.4.3 存储文件



- commitLog: 消息存储目录
- config: 运行期间一些配置信息

- consumerqueue: 消息消费队列存储目录
- index: 消息索引文件存储目录
- abort: 如果存在改文件寿命Broker非正常关闭
- checkpoint: 文件检查点, 存储CommitLog文件最后一次刷盘时间戳、consumerqueue最后一次刷盘时间, index索引文件最后一次刷盘时间戳。

#### 5.4.4 存储文件内存映射

MMap

RocketMQ通过使用内存映射文件提高IO访问性能, 无论是CommitLog、ConsumerQueue还是IndexFile, 单个文件都被设计为固定长度, 如果一个文件写满以后再创建一个新文件, 文件名就为该文件第一条消息对应的全局物理偏移量。

##### 5.4.4.1 MappedFileQueue

| C MappedFileQueue                                         |  |                           |
|-----------------------------------------------------------|--|---------------------------|
| f log                                                     |  | InternalLogger            |
| f LOG_ERROR                                               |  | InternalLogger            |
| f DELETE_FILES_BATCH_MAX                                  |  | int                       |
| f storePath                                               |  | String                    |
| f allocateMappedFileService                               |  | AllocateMappedFileService |
| m MappedFileQueue(String, int, AllocateMappedFileService) |  |                           |
| p lastMappedFile                                          |  | MappedFile                |
| p firstMappedFile                                         |  | MappedFile                |
| p storeTimestamp                                          |  | long                      |
| p mappedMemorySize                                        |  | long                      |
| p maxWrotePosition                                        |  | long                      |
| p mappedFileSize                                          |  | int                       |
| p maxOffset                                               |  | long                      |
| p flushedWhere                                            |  | long                      |
| p mappedFiles                                             |  | List<MappedFile>          |
| p committedWhere                                          |  | long                      |
| p minOffset                                               |  | long                      |

Powered by yFiles

```

1 string storePath; //存储目录
2 int mappedFileSize; // 单个文件大小
3 CopyOnwriteArrayList<MappedFile> mappedFiles; //MappedFile文件集合
4 AllocateMappedFileService allocateMappedFileService; //创建MapFile服务类
5 long flushedwhere = 0; //当前刷盘指针
6 long committedwhere = 0; //当前数据提交指针,内存中ByteBuffer当前的写指针,该值大于等于flushwhere

```

- 根据存储时间查询MappedFile

*代码: MappedFileQueue#getMappedFileByTime*

```

1 public MappedFile getMappedFileByTime(final long timestamp) {
2 Object[] mfs = this.copyMappedFiles(0);
3
4 if (null == mfs)
5 return null;
6 //遍历MappedFile文件数组
7 for (int i = 0; i < mfs.length; i++) {
8 MappedFile mappedFile = (MappedFile) mfs[i];
9 //MappedFile文件的最后修改时间大于指定时间戳则返回该文件
10 if (mappedFile.getLastModifiedTimestamp() >= timestamp) {
11 return mappedFile;
12 }
13 }
14
15 return (MappedFile) mfs[mfs.length - 1];
16 }

```

- 根据消息偏移量offset查找MappedFile

*代码: MappedFileQueue#findMappedFileByOffset*

```

1 public MappedFile findMappedFileByOffset(final long offset, final boolean
returnFirstOnNotFound) {
2 try {
3 //获得第一个MappedFile文件
4 MappedFile firstMappedFile = this.getFirstMappedFile();
5 //获得最后一个MappedFile文件
6 MappedFile lastMappedFile = this.getLastMappedFile();
7 //第一个文件和最后一个文件均不为空,则进行处理
8 if (firstMappedFile != null && lastMappedFile != null) {
9 if (offset < firstMappedFile.getFileFromOffset() ||
10 offset >= lastMappedFile.getFileFromOffset() +
this.mappedFileSize) {
11 } else {
12 //获得文件索引
13 int index = (int) ((offset / this.mappedFileSize)
14 - (firstMappedFile.getFileFromOffset() /
this.mappedFileSize));
15 MappedFile targetFile = null;
16 try {
17 //根据索引返回目标文件
18 targetFile = this.mappedFiles.get(index);
19 } catch (Exception ignored) {
20 }
21

```

```

22 if (targetFile != null && offset >=
targetFile.getFileFromOffset()
23 && offset < targetFile.getFileFromOffset() +
this.mappedFileSize) {
24 return targetFile;
25 }
26
27 for (MappedFile tmpMappedFile : this.mappedFiles) {
28 if (offset >= tmpMappedFile.getFileFromOffset()
29 && offset < tmpMappedFile.getFileFromOffset() +
this.mappedFileSize) {
30 return tmpMappedFile;
31 }
32 }
33 }
34
35 if (returnFirstOnNotFound) {
36 return firstMappedFile;
37 }
38 }
39 } catch (Exception e) {
40 log.error("findMappedFileByOffset Exception", e);
41 }
42
43 return null;
44 }
```

- 获取存储文件最小偏移量

*代码: MappedFileQueue#getMinOffset*

```

1 public long getMinOffset() {
2
3 if (!this.mappedFiles.isEmpty()) {
4 try {
5 return this.mappedFiles.get(0).getFileFromOffset();
6 } catch (IndexOutOfBoundsException e) {
7 //continue;
8 } catch (Exception e) {
9 log.error("getMinOffset has exception.", e);
10 }
11 }
12 return -1;
13 }
```

- 获取存储文件最大偏移量

```

1 public long getMaxOffset() {
2 MappedFile mappedFile = getLastMappedFile();
3 if (mappedFile != null) {
4 return mappedFile.getFileFromOffset() + mappedFile.getReadPosition();
5 }
6 return 0;
7 }
```

- 返回存储文件当前写指针

```
1 public long getMaxwrotePosition() {
2 MappedFile mappedFile = getLastMappedFile();
3 if (mappedFile != null) {
4 return mappedFile.getFileFromOffset() +
5 mappedFile.getWrotePosition();
6 }
7 }
```

#### 5.4.4.2 MappedFile

|                                                                                                                                                                                                                     |                    |                |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------|----------------|
|   MappedFile                                      |                    |                |
|   OS_PAGE_SIZE                                    |                    | int            |
|   log                                             |                    | InternalLogger |
|   TOTAL_MAPPED_VIRTUAL_MEMORY                     | AtomicLong         |                |
|   TOTAL_MAPPED_FILES                              | AtomicInteger      |                |
|   wrotePosition                                   | AtomicInteger      |                |
|   committedPosition                               | AtomicInteger      |                |
|   flushedPosition                                 | AtomicInteger      |                |
|   writeBuffer                                     | ByteBuffer         |                |
|   transientStorePool                              | TransientStorePool |                |
|   file                                            | File               |                |
|   MappedFile()                                    |                    |                |
|   MappedFile(String, int)                       |                    |                |
|   MappedFile(String, int, TransientStorePool) |                    |                |
|   wrotePosition                               | int                |                |
|   committedPosition                           | int                |                |
|   fileName                                    | String             |                |
|   fileFromOffset                              | long               |                |
|   readPosition                                | int                |                |
|   fileSize                                    | int                |                |
|   fileChannel                                 | FileChannel        |                |
|   flushedPosition                             | int                |                |
|   storeTimestamp                              | long               |                |
|   lastModifiedTimestamp                       | long               |                |
|   full                                        | boolean            |                |
|   firstCreateInQueue                          | boolean            |                |
|   mappedByteBuffer                            | MappedByteBuffer   |                |

```

1 int OS_PAGE_SIZE = 1024 * 4; //操作系统每页大小,默认4K
2 AtomicLong TOTAL_MAPPED_VIRTUAL_MEMORY = new AtomicLong(0); //当前JVM实例中
 MappedFile虚拟内存
3 AtomicInteger TOTAL_MAPPED_FILES = new AtomicInteger(0); //当前JVM实例中
 MappedFile对象个数
4 AtomicInteger wrotePosition = new AtomicInteger(0); //当前文件的写指针
5 AtomicInteger committedPosition = new AtomicInteger(0); //当前文件的提交指针
6 AtomicInteger flushedPosition = new AtomicInteger(0); //刷写到磁盘指针
7 int filesize; //文件大小
8 FileChannel fileChannel; //文件通道
9 ByteBuffer writeBuffer = null; //堆外内存ByteBuffer
10 TransientStorePool transientStorePool = null; //堆外内存池
11 String fileName; //文件名称
12 Long fileFromOffset; //该文件的处理偏移量
13 File file; //物理文件
14 MappedByteBuffer mappedByteBuffer; //物理文件对应的内存映射Buffer
15 volatile long storeTimestamp = 0; //文件最后一次内容写入时间
16 boolean firstCreateInQueue = false; //是否是MappedFileQueue队列中第一个文件

```

### MappedFile初始化

- 未开启 transientStorePoolEnable。transientStorePoolEnable=true 为 true 表示数据先存储到堆外内存，然后通过 Commit 线程将数据提交到内存映射 Buffer 中，再通过 Flush 线程将内存映射 Buffer 中数据持久化磁盘。

```

1 private void init(final String fileName, final int filesize) throws
 IOException {
2 this.fileName = fileName;
3 this.filesize = filesize;
4 this.file = new File(fileName);
5 this.fileFromOffset = Long.parseLong(this.file.getName());
6 boolean ok = false;
7
8 ensureDirOK(this.file.getParent());
9
10 try {
11 this.fileChannel = new RandomAccessFile(this.file,
12 "rw").getChannel();
13 this.mappedByteBuffer = this.fileChannel.map(MapMode.READ_WRITE, 0,
14 filesize);
15 TOTAL_MAPPED_VIRTUAL_MEMORY.addAndGet(filesize);
16 TOTAL_MAPPED_FILES.incrementAndGet();
17 ok = true;
18 } catch (FileNotFoundException e) {
19 log.error("create file channel " + this.fileName + " Failed. ", e);
20 throw e;
21 } catch (IOException e) {
22 log.error("map file " + this.fileName + " Failed. ", e);
23 throw e;
24 } finally {
25 if (!ok && this.fileChannel != null) {
26 this.fileChannel.close();
27 }
28 }
29 }

```

## 开启 transientStorePoolEnable

```
1 public void init(final String fileName, final int fileSize,
2 final TransientStorePool transientStorePool) throws IOException {
3 init(fileName, fileSize);
4 this.writeBuffer = transientStorePool.borrowBuffer(); //初始化
5 this.transientStorePool = transientStorePool;
6 }
```

## MappedFile提交

提交数据到FileChannel，commitLeastPages为本次提交最小的页数，如果待提交数据不满commitLeastPages，则不执行本次提交操作。如果writeBuffer如果为空，直接返回writePosition指针，无需执行commit操作，表名commit操作主体是writeBuffer。

```
1 public int commit(final int commitLeastPages) {
2 if (writeBuffer == null) {
3 //no need to commit data to file channel, so just regard
4 //wrotePosition as committedPosition.
5 return this.wrotePosition.get();
6 }
7 //判断是否满足提交条件
8 if (this.isAbleToCommit(commitLeastPages)) {
9 if (this.hold()) {
10 commit0(commitLeastPages);
11 this.release();
12 } else {
13 log.warn("in commit, hold failed, commit offset = " +
14 this.committedPosition.get());
15 }
16 }
17 // 所有数据提交后,清空缓冲区
18 if (writeBuffer != null && this.transientStorePool != null &&
19 this.fileSize == this.committedPosition.get()) {
20 this.transientStorePool.returnBuffer(writeBuffer);
21 this.writeBuffer = null;
22 }
23 return this.committedPosition.get();
24 }
```

## MappedFile#isAbleToCommit

判断是否执行commit操作，如果文件已满返回true；如果commitLeastPages大于0，则比较writePosition与上一次提交的指针committedPosition的差值，除以OS\_PAGE\_SIZE得到当前脏页的数量，如果大于commitLeastPages则返回true，如果commitLeastPages小于0表示只要存在脏页就提交。

```
1 protected boolean isAbleToCommit(final int commitLeastPages) {
2 //已经刷盘指针
3 int flush = this.committedPosition.get();
4 //文件写指针
5 int write = this.wrotePosition.get();
6 //写满刷盘
7 if (this.isFull()) {
```

```

8 return true;
9 }
10
11 if (commitLeastPages > 0) {
12 //文件内容达到commitLeastPages页数，则刷盘
13 return ((write / OS_PAGE_SIZE) - (flush / OS_PAGE_SIZE)) >=
commitLeastPages;
14 }
15
16 return write > flush;
17 }

```

### **MappedFile#commit0**

具体提交的实现，首先创建WriteBuffer区共享缓存区，然后将新创建的position回退到上一次提交的位置（commitPosition），设置limit为wrotePosition（当前最大有效数据指针），然后把commitPosition到wrotePosition的数据写入到FileChannel中，然后更新committedPosition指针为wrotePosition。commit的作用就是将MappedFile的writeBuffer中数据提交到文件通道FileChannel中。

```

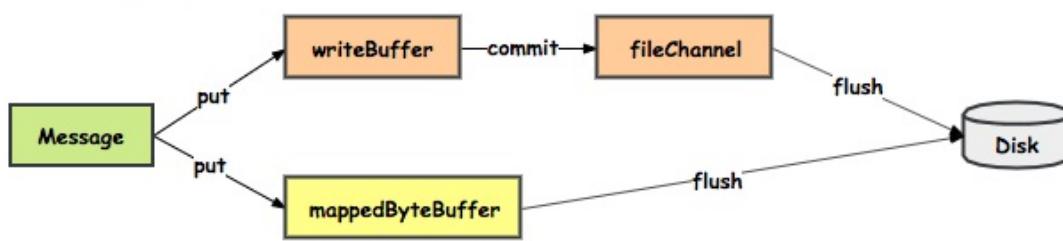
1 protected void commit0(final int commitLeastPages) {
2 //写指针
3 int writePos = this.wrotePosition.get();
4 //上次提交指针
5 int lastCommittedPosition = this.committedPosition.get();
6
7 if (writePos - this.committedPosition.get() > 0) {
8 try {
9 //复制共享内存区域
10 ByteBuffer byteBuffer = writeBuffer.slice();
11 //设置提交位置是上次提交位置
12 byteBuffer.position(lastCommittedPosition);
13 //最大提交数量
14 byteBuffer.limit(writePos);
15 //设置fileChannel位置为上次提交位置
16 this.fileChannel.position(lastCommittedPosition);
17 //将lastCommittedPosition到writePos的数据复制到filechannel中
18 this.fileChannel.write(byteBuffer);
19 //重置提交位置
20 this.committedPosition.set(writePos);
21 } catch (Throwable e) {
22 log.error("Error occurred when commit data to FileChannel.",
e);
23 }
24 }
25 }

```

### **MappedFile#flush**

刷写磁盘，直接调用MappedByteBuffer或fileChannel的force方法将内存中的数据持久化到磁盘，那么flushedPosition应该等于MappedByteBuffer中的写指针；如果writeBuffer不为空，则flushPosition应该等于上一次的commit指针；因为上一次提交的数据就是进入到MappedByteBuffer中的数据；如果writeBuffer为空，数据时直接进入到MappedByteBuffer，wrotePosition代表的是MappedByteBuffer中的指针，故设置flushPosition为wrotePosition。

### Persist message to disk



```

1 public int flush(final int flushLeastPages) {
2 //数据达到刷盘条件
3 if (this.isAbleToFlush(flushLeastPages)) {
4 //加锁，同步刷盘
5 if (this.hold()) {
6 //获得读指针
7 int value = getReadPosition();
8 try {
9 //数据从writeBuffer提交数据到fileChannel再刷新到磁盘
10 if (writeBuffer != null || this.filechannel.position() != 0) {
11 this.fileChannel.force(false);
12 } else {
13 //从mmap刷新数据到磁盘
14 this.mappedByteBuffer.force();
15 }
16 } catch (Throwable e) {
17 log.error("Error occurred when force data to disk.", e);
18 }
19 //更新刷盘位置
20 this.flushedPosition.set(value);
21 this.release();
22 } else {
23 log.warn("in flush, hold failed, flush offset = " +
24 this.flushedPosition.get());
25 this.flushedPosition.set(getReadPosition());
26 }
27 }
28 return this.getFlushedPosition();
}

```

### MappedFile#getReadPosition

获取当前文件最大可读指针。如果writeBuffer为空，则直接返回当前的写指针；如果writeBuffer不为空，则返回上一次提交的指针。在MappedFile设置中，只有提交了的数据（写入到MappedByteBuffer或FileChannel中的数据）才是安全的数据

```

1 public int getReadPosition() {
2 //如果writeBuffer为空，刷盘的位置就是应该等于上次commit的位置，如果为空则为mmap的写
3 //指针
4 return this.writeBuffer == null ? this.wrotePosition.get() :
5 this.committedPosition.get();
}

```

### MappedFile#selectMappedBuffer

查找pos到当前最大可读之间的数据，由于在整个写入期间都未曾改MappedByteBuffer的指针，如果mappedByteBuffer.slice()方法返回的共享缓存区空间为整个MappedFile，然后通过设置 ByteBuffer的position为待查找的值，读取字节长度当前可读最大长度，最终返回的ByteBuffer的limit为size。整个共享缓存区的容量为 (MappedFile#fileSize-pos) 。故在操作SelectMappedBufferResult不能对包含在里面的ByteBuffer调用flip方法。

```
1 public SelectMappedBufferResult selectMappedBuffer(int pos) {
2 //获得最大可读指针
3 int readPosition = getReadPosition();
4 //pos小于最大可读指针，并且大于0
5 if (pos < readPosition && pos >= 0) {
6 if (this.hold()) {
7 //复制mappedByteBuffer读共享区
8 ByteBuffer byteBuffer = this.mappedByteBuffer.slice();
9 //设置读指针位置
10 byteBuffer.position(pos);
11 //获得可读范围
12 int size = readPosition - pos;
13 //设置最大可读范围
14 ByteBuffer byteBufferNew = byteBuffer.slice();
15 byteBufferNew.limit(size);
16 return new SelectMappedBufferResult(this.fileFromOffset + pos,
17 byteBufferNew, size, this);
18 }
19 }
20 return null;
21 }
```

### MappedFile#shutdown

MappedFile文件销毁的实现方法为public boolean destory(long intervalForcibly), intervalForcibly表示拒绝被销毁的最大存活时间。

```
1 public void shutdown(final long intervalForcibly) {
2 if (this.available) {
3 //关闭MapedFile
4 this.available = false;
5 //设置当前关闭时间戳
6 this.firstShutdownTimestamp = System.currentTimeMillis();
7 //释放资源
8 this.release();
9 } else if (this.getRefCount() > 0) {
10 if ((System.currentTimeMillis() - this.firstShutdownTimestamp) >=
11 intervalForcibly) {
12 this.refCount.set(-1000 - this.getRefCount());
13 this.release();
14 }
15 }
16 }
```

### 5.4.4.3 TransientStorePool

短暂的存储池。RocketMQ单独创建一个MappedByteBuffer内存缓存池，用来临时存储数据，数据先写入该内存映射中，然后由commit线程定时将数据从该内存复制到与目标物理文件对应的内存映射中。RocketMQ引入该机制主要的原因是提供一种内存锁定，将当前堆外内存一直锁定在内存中，避免被进程将内存交换到磁盘。



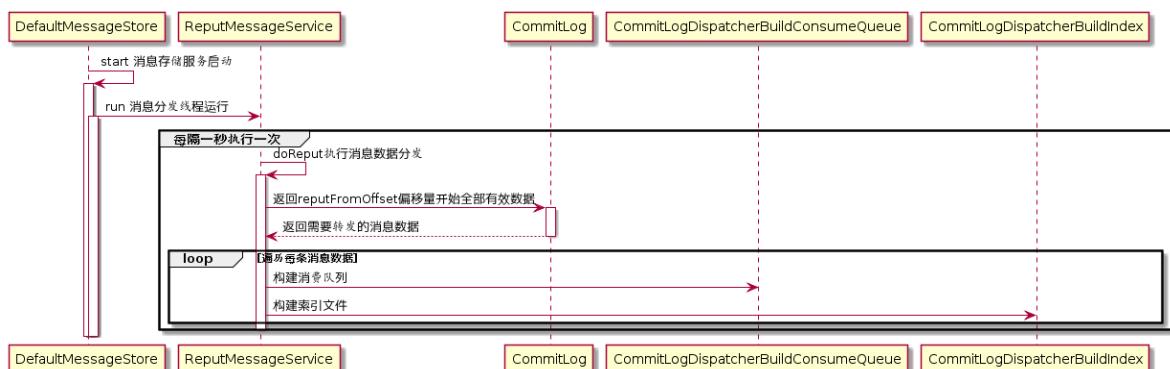
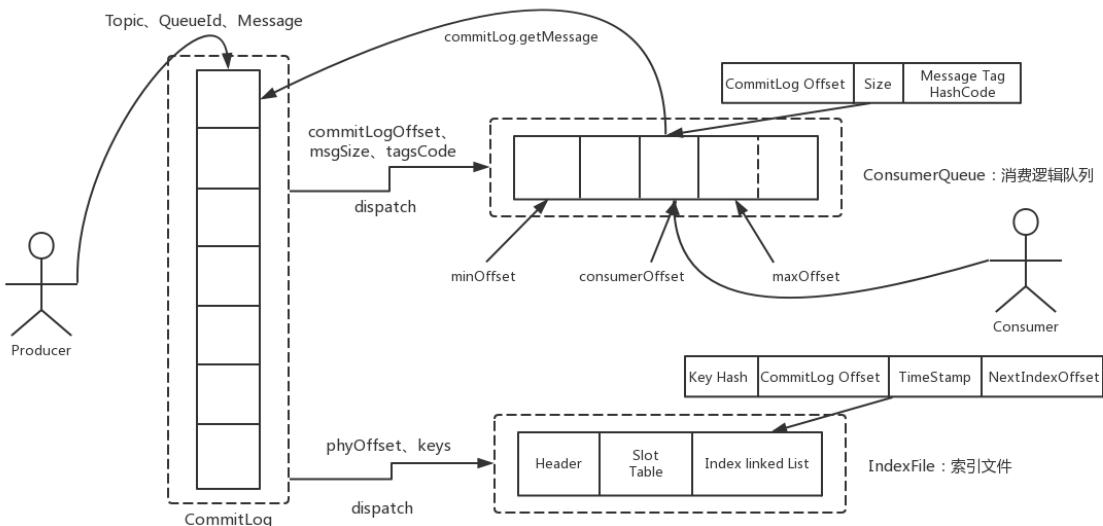
```
1 private final int poolsize; //availableBuffers个数
2 private final int filesize; //每隔ByteBuffer大小
3 private final Deque<ByteBuffer> availableBuffers; //ByteBuffer容器。双端队列
```

### 初始化

```
1 public void init() {
2 //创建poolsize个堆外内存
3 for (int i = 0; i < poolsize; i++) {
4 ByteBuffer byteBuffer = ByteBuffer.allocateDirect(filesize);
5 final long address = ((DirectBuffer) byteBuffer).address();
6 Pointer pointer = new Pointer(address);
7 //使用com.sun.jna.Library类库将该批内存锁定，避免被置换到交换区，提高存储性能
8 LibC.INSTANCE.mlock(pointer, new NativeLong(filesize));
9
10 availableBuffers.offer(byteBuffer);
11 }
12 }
```

## 5.4.5 实时更新消息消费队列与索引文件

消息消费队文件、消息属性索引文件都是基于CommitLog文件构建的，当消息生产者提交的消息存储在CommitLog文件中，ConsumerQueue、IndexFile需要及时更新，否则消息无法及时被消费，根据消息属性查找消息也会出现较大延迟。RocketMQ通过开启一个线程**ReputMessageService**来准时转发CommitLog文件更新事件，相应的任务处理器根据转发的消息及时更新**ConsumerQueue**、**IndexFile**文件。



### 代码: DefaultMessageStore: start

```

1 //设置CommitLog内存中最大偏移量
2 this.reputMessageService.setReputFromOffset(maxPhysicalPosInLogicQueue);
3 //启动
4 this.reputMessageService.start();

```

### 代码: DefaultMessageStore: run

```

1 public void run() {
2 DefaultMessageStore.log.info(this.getServiceName() + " service
3 started");
4 //每隔1毫秒就继续尝试推送消息到消息消费队列和索引文件
5 while (!this.isStopped()) {
6 try {
7 Thread.sleep(1);
8 this.doReput();
9 } catch (Exception e) {
10 DefaultMessageStore.log.warn(this.getServiceName() + " service
11 has exception. ", e);
12 }
13 }
14 }

```

### 代码: DefaultMessageStore: doReput

```

1 //从result中循环遍历消息，一次读一条，创建DispatcherRequest对象。
2 for (int readSize = 0; readSize < result.getSize() && doNext;) {
3 DispatchRequest dispatchRequest =
4 DefaultMessageStore.this.commitLog.checkMessageAndReturnSize(result.getByte
5 Buffer(), false, false);
6 int size = dispatchRequest.getBufferSize() == -1 ?
7 dispatchRequest.getMsgSize() : dispatchRequest.getBufferSize();
8
9 if (dispatchRequest.isSuccess()) {
10 if (size > 0) {
11 DefaultMessageStore.this.doDispatch(dispatchRequest);
12 }
13 }
14 }

```

### *DispatchRequest*

| c DispatchRequest |                                   |
|-------------------|-----------------------------------|
| • P               | propertiesMap Map<String, String> |
| • P               | preparedTransactionOffset long    |
| • P               | bitMap byte[]                     |
| • P               | commitLogOffset long              |
| • P               | uniqKey String                    |
| • P               | success boolean                   |
| • P               | keys String                       |
| • P               | storeTimestamp long               |
| • P               | sysFlag int                       |
| • P               | bufferSize int                    |
| • P               | consumeQueueOffset long           |
| • P               | msgSize int                       |
| • P               | queueId int                       |
| • P               | tagsCode long                     |
| • P               | topic String                      |

Powered by yFiles

```

1 string topic; //消息主题名称
2 int queueId; //消息队列ID
3 long commitLogOffset; //消息物理偏移量
4 int msgsize; //消息长度

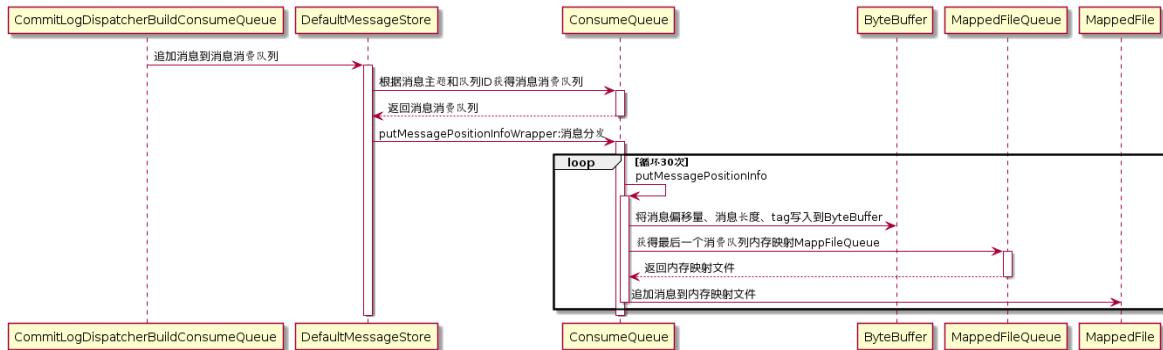
```

```

5 long tagsCode; //消息过滤tag hashCode
6 long storeTimestamp; //消息存储时间戳
7 long consumeQueueOffset; //消息队列偏移量
8 String keys; //消息索引key
9 boolean success; //是否成功解析到完整的消息
10 String uniqKey; //消息唯一键
11 int sysFlag; //消息系统标记
12 long preparedTransactionOffset; //消息预处理事务偏移量
13 Map<String, String> propertiesMap; //消息属性
14 byte[] bitMap; //位图

```

#### 5.4.5.1 转发到ConsumerQueue



```

1 class CommitLogDispatcherBuildConsumeQueue implements CommitLogDispatcher {
2 @Override
3 public void dispatch(DispatchRequest request) {
4 final int tranType =
5 MessageSysFlag.getTransactionValue(request.getSysFlag());
6 switch (tranType) {
7 case MessageSysFlag.TRANSACTION_NOT_TYPE:
8 case MessageSysFlag.TRANSACTION_COMMIT_TYPE:
9 //消息分发
10 DefaultMessageStore.this.putMessagePositionInfo(request);
11 break;
12 case MessageSysFlag.TRANSACTION_PREPARED_TYPE:
13 case MessageSysFlag.TRANSACTION_ROLLBACK_TYPE:
14 break;
15 }
16 }

```

代码: *DefaultMessageStore#putMessagePositionInfo*

```

1 public void putMessagePositionInfo(DispatchRequest dispatchRequest) {
2 //获得消费队列
3 ConsumeQueue cq = this.findConsumeQueue(dispatchRequest.getTopic(),
4 dispatchRequest.getQueueId());
4 //消费队列分发消息
5 cq.putMessagePositionInfoWrapper(dispatchRequest);
6 }

```

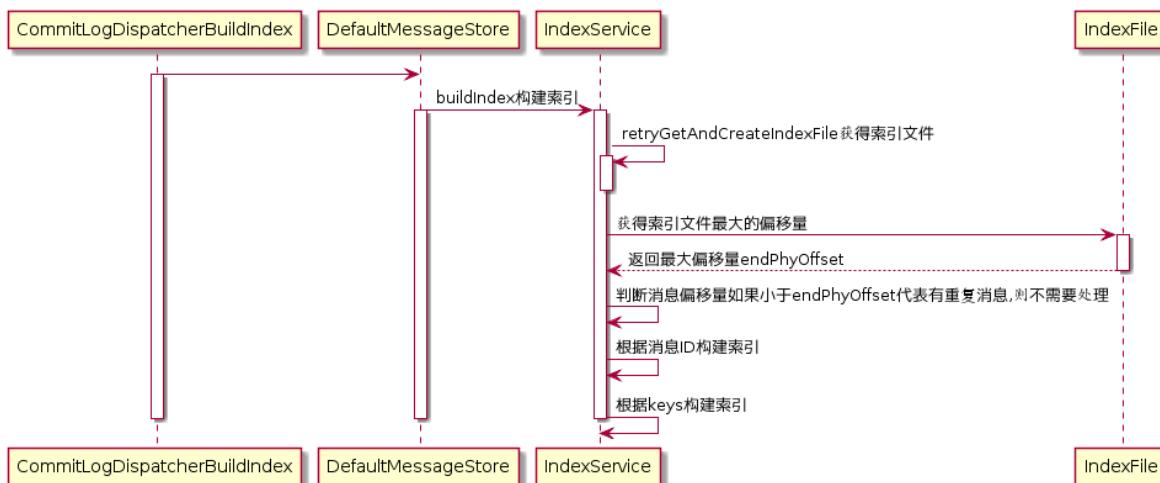
代码: *DefaultMessageStore#putMessagePositionInfo*

```

1 //依次将消息偏移量、消息长度、tag写入到ByteBuffer中
2 this.byteBufferIndex.flip();
3 this.byteBufferIndex.limit(CQ_STORE_UNIT_SIZE);
4 this.byteBufferIndex.putLong(offset);
5 this.byteBufferIndex.putInt(size);
6 this.byteBufferIndex.putLong(tagsCode);
7 //获得内存映射文件
8 MappedFile mappedFile =
9 this.mappedFileQueue.getLastMappedFile(expectLogicOffset);
10 if (mappedFile != null) {
11 //将消息追加到内存映射文件，异步写盘
12 return mappedFile.appendMessage(this.byteBufferIndex.array());
13 }

```

#### 5.4.5.2 转发到Index



```

1 class CommitLogDispatcherBuildIndex implements CommitLogDispatcher {
2
3 @Override
4 public void dispatch(DispatchRequest request) {
5 if
6 (DefaultMessageStore.this.messageStoreConfig.isMessageIndexEnable()) {
7 DefaultMessageStore.this.indexService.buildIndex(request);
8 }
9 }

```

代码: `DefaultMessageStore#buildIndex`

```

1 public void buildIndex(DispatchRequest req) {
2 //获得索引文件
3 IndexFile indexFile = retryGetAndCreateIndexFile();
4 if (indexFile != null) {
5 //获得文件最大物理偏移量
6 long endPhyoffset = indexFile.getEndPhyOffset();
7 DispatchRequest msg = req;
8 String topic = msg.getTopic();
9 String keys = msg.getKeys();
10 //如果该消息的物理偏移量小于索引文件中的最大物理偏移量，则说明是重复数据，忽略本次
11 //索引构建
12 if (msg.getCommitLogOffset() < endPhyoffset) {
13 return;
14 }
15 }
16 }

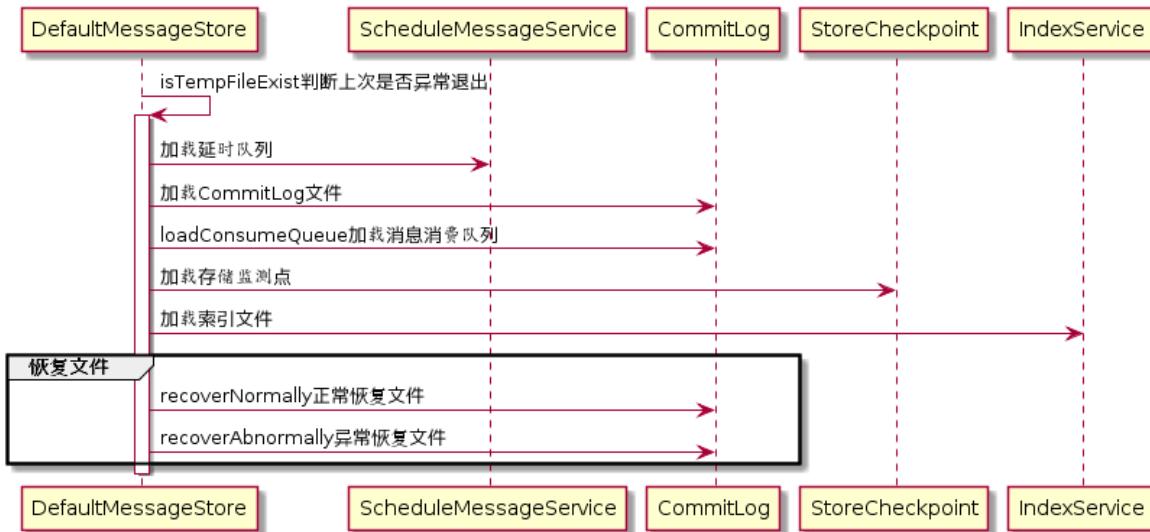
```

```

13 }
14
15 final int tranType =
16 MessageSysFlag.getTransactionValue(msg.getSysFlag());
17 switch (tranType) {
18 case MessageSysFlag.TRANSACTION_NOT_TYPE:
19 case MessageSysFlag.TRANSACTION_PREPARED_TYPE:
20 break;
21 case MessageSysFlag.TRANSACTION_COMMIT_TYPE:
22 return;
23 }
24
25 //如果消息ID不为空，则添加到Hash索引中
26 if (req.getUniqKey() != null) {
27 indexFile = putKey(indexFile, msg, buildKey(topic,
28 req.getUniqKey()));
29 if (indexFile == null) {
30 return;
31 }
32 //构建索引key，RocketMQ支持为同一个消息建立多个索引，多个索引键空格隔开。
33 if (keys != null && keys.length() > 0) {
34 String[] keyset = keys.split(MessageConst.KEY_SEPARATOR);
35 for (int i = 0; i < keyset.length; i++) {
36 String key = keyset[i];
37 if (key.length() > 0) {
38 indexFile = putKey(indexFile, msg, buildKey(topic,
39 key));
40 if (indexFile == null) {
41 return;
42 }
43 }
44 }
45 }
46 } else {
47 log.error("build index error, stop building index");
48 }
49 }
```

## 5.4.6 消息队列和索引文件恢复

由于RocketMQ存储首先将消息全量存储在CommitLog文件中，然后异步生成转发任务更新ConsumerQueue和Index文件。如果消息成功存储到CommitLog文件中，转发任务未成功执行，此时消息服务器Broker由于某个愿意宕机，导致CommitLog、ConsumerQueue、IndexFile文件数据不一致。如果不加以人工修复的话，会有一部分消息即便在CommitLog中文件中存在，但由于没有转发到ConsumerQueue，这部分消息将永远复发被消费者消费。



#### 5.4.6.1 存储文件加载

**代码: DefaultMessageStore#load**

判断上一次是否异常退出。实现机制是Broker在启动时创建abort文件，在退出时通过JVM钩子函数删除abort文件。如果下次启动时存在abort文件，说明Broker时异常退出的，CommitLog与ConsumerQueue数据有可能不一致，需要进行修复。

```

1 //判断临时文件是否存在
2 boolean lastExitOK = !this.isTempFileExist();
3 //根据临时文件判断当前Broker是否异常退出
4 private boolean isTempFileExist() {
5 String fileName = StorePathConfigHelper
6 .getAbortFile(this.messageStoreConfig.getStorePathRootDir());
7 File file = new File(fileName);
8 return file.exists();
9 }

```

**代码: DefaultMessageStore#load**

```

1 //加载延时队列
2 if (null != scheduleMessageService) {
3 result = result && this.scheduleMessageService.load();
4 }
5
6 // 加载CommitLog文件
7 result = result && this.commitLog.load();
8
9 // 加载消费队列文件
10 result = result && this.loadConsumeQueue();
11
12 if (result) {
13 //加载存储监测点，监测点主要记录CommitLog文件、ConsumerQueue文件、Index索引文件的
14 //刷盘点
15 this.storeCheckpoint =new
16 StoreCheckpoint(StorePathConfigHelper.getStoreCheckpoint(this.messageStoreC
17 onfig.getStorePathRootDir()));
18 //加载index文件
19 this.indexService.load(lastExitOK);
20 //根据Broker是否异常退出，执行不同的恢复策略
21 this.recover(lastExitOK);
22 }

```

19 }

#### 代码: *MappedFileQueue#load*

加载CommitLog到映射文件

```
1 //指向CommitLog文件目录
2 File dir = new File(this.storePath);
3 //获得文件数组
4 File[] files = dir.listFiles();
5 if (files != null) {
6 // 文件排序
7 Arrays.sort(files);
8 //遍历文件
9 for (File file : files) {
10 //如果文件大小和配置文件不一致,退出
11 if (file.length() != this.mappedFileSize) {
12 return false;
13 }
14 }
15
16 try {
17 //创建映射文件
18 MappedFile mappedFile = new MappedFile(file.getPath(),
19 mappedFileSize);
20 mappedFile.setWrotePosition(this.mappedFileSize);
21 mappedFile.setFlushedPosition(this.mappedFileSize);
22 mappedFile.setCommittedPosition(this.mappedFileSize);
23 //将映射文件添加到队列
24 this.mappedFiles.add(mappedFile);
25 log.info("load " + file.getPath() + " OK");
26 } catch (IOException e) {
27 log.error("load file " + file + " error", e);
28 return false;
29 }
30 }
31
32 return true;
```

#### 代码: *DefaultMessageStore#loadConsumeQueue*

加载消息消费队列

```
1 //执行消费队列目录
2 File dirLogic = new
3 File(StorePathConfigHelper.getStorePathConsumeQueue(this.messageStoreConfig
4 .getStorePathRootDir()));
5 //遍历消费队列目录
6 File[] fileTopicList = dirLogic.listFiles();
7 if (fileTopicList != null) {
8
9 for (File fileTopic : fileTopicList) {
10 //获得子目录名称,即topic名称
11 String topic = fileTopic.getName();
12 //遍历子目录下的消费队列文件
13 File[] fileQueueIdList = fileTopic.listFiles();
```

```
12 if (fileQueueIdList != null) {
13 //遍历文件
14 for (File fileQueueId : fileQueueIdList) {
15 //文件名称即队列ID
16 int queueId;
17 try {
18 queueId = Integer.parseInt(fileQueueId.getName());
19 } catch (NumberFormatException e) {
20 continue;
21 }
22 //创建消费队列并加载到内存
23 ConsumeQueue logic = new ConsumeQueue(
24 topic,
25 queueId,
26
27 StorePathConfigHelper.getStorePathConsumeQueue(this.messageStoreConfig.get
28 StorePathRootDir()),
29 this.getMessageStoreConfig().getMapedFileSizeConsumeQueue(),
30 this);
31 this.putConsumeQueue(topic, queueId, logic);
32 if (!logic.load()) {
33 return false;
34 }
35 }
36 }
37
38 log.info("Load logics queue all over, OK");
39
40 return true;
```

代码: *IndexService#load*

### 加载索引文件

```
1 public boolean load(final boolean lastExitOK) {
2 //索引文件目录
3 File dir = new File(this.storePath);
4 //遍历索引文件
5 File[] files = dir.listFiles();
6 if (files != null) {
7 //文件排序
8 Arrays.sort(files);
9 //遍历文件
10 for (File file : files) {
11 try {
12 //加载索引文件
13 IndexFile f = new IndexFile(file.getPath(),
this.hashSlotNum, this.indexNum, 0, 0);
14 f.load();
15
16 if (!lastExitOK) {
17 //索引文件上次的刷盘时间小于该索引文件的消息时间戳，该文件将立即删除
18 if (f.getEndTimestamp() >
this.defaultMessageStore.getStoreCheckpoint()
```

```

19 .getIndexMsgTimestamp()) {
20 f.destroy(0);
21 continue;
22 }
23 }
24 //将索引文件添加到队列
25 log.info("load index file ok, " + f.getFileName());
26 this.indexFileList.add(f);
27 } catch (IOException e) {
28 log.error("load file {} error", file, e);
29 return false;
30 } catch (NumberFormatException e) {
31 log.error("load file {} error", file, e);
32 }
33 }
34 }
35
36 return true;
37 }

```

#### **代码: DefaultMessageStore#recover**

文件恢复，根据Broker是否正常退出执行不同的恢复策略

```

1 private void recover(final boolean lastExitOK) {
2 //获得最大的物理便宜消费队列
3 long maxPhyOffsetOfConsumeQueue = this.recoverConsumeQueue();
4
5 if (lastExitOK) {
6 //正常恢复
7 this.commitLog.recoverNormally(maxPhyOffsetOfConsumeQueue);
8 } else {
9 //异常恢复
10 this.commitLog.recoverAbnormally(maxPhyOffsetOfConsumeQueue);
11 }
12 //在CommitLog中保存每个消息消费队列当前的存储逻辑偏移量
13 this.recoverTopicQueueTable();
14 }

```

#### **代码: DefaultMessageStore#recoverTopicQueueTable**

恢复ConsumerQueue后，将在CommitLog实例中保存每隔消息队列当前的存储逻辑偏移量，这也是消息中不仅存储主题、消息队列ID、还存储了消息队列的关键所在。

```

1 public void recoverTopicQueueTable() {
2 HashMap<String/* topic-queueid */, Long/* offset */> table = new
3 HashMap<String, Long>(1024);
4 //CommitLog最小偏移量
5 long minPhyOffset = this.commitLog.getMinOffset();
6 //遍历消费队列，将消费队列保存在CommitLog中
7 for (ConcurrentMap<Integer, ConsumeQueue> maps :
8 this.consumeQueueTable.values()) {
9 for (ConsumeQueue logic : maps.values()) {
10 String key = logic.getTopic() + "-" + logic.getQueueId();
11 table.put(key, logic.getMaxOffsetInQueue());
12 logic.correctMinOffset(minPhyOffset);
13 }
14 }
15 }

```

```
12 }
13 this.commitLog.setTopicQueueTable(table);
14 }
```

#### 5.4.6.2 正常恢复

代码: *CommitLog#recoverNormally*

```
1 public void recoverNormally(long maxPhyOffsetOfConsumeQueue) {
2
3 final List<MappedFile> mappedFiles =
4 this.mappedFileQueue.getMappedFiles();
5 if (!mappedFiles.isEmpty()) {
6 //Broker正常停止再重启时,从倒数第三个开始恢复,如果不足3个文件,则从第一个文件开始恢复。
7 int index = mappedFiles.size() - 3;
8 if (index < 0)
9 index = 0;
10 MappedFile mappedFile = mappedFiles.get(index);
11 ByteBuffer byteBuffer = mappedFile.sliceByteBuffer();
12 long processOffset = mappedFile.getFileFromOffset();
13 //代表当前已校验通过的offset
14 long mappedFileOffset = 0;
15 while (true) {
16 //查找消息
17 DispatchRequest dispatchRequest =
18 this.checkMessageAndReturnSize(byteBuffer, checkCRCOnRecover);
19 //消息长度
20 int size = dispatchRequest.getMsgSize();
21 //查找结果为true,并且消息长度大于0,表示消息正确.mappedFileOffset向前移动本消息长度
22 if (dispatchRequest.isSuccess() && size > 0) {
23 mappedFileOffset += size;
24 }
25 //如果查找结果为true且消息长度等于0,表示已到该文件末尾,如果还有下一个文件,则重置processOffset和MappedFileOffset重复查找下一个文件,否则跳出循环。
26 else if (dispatchRequest.isSuccess() && size == 0) {
27 index++;
28 if (index >= mappedFiles.size())
29 // current branch can not happen
30 break;
31 } else {
32 //取出每个文件
33 mappedFile = mappedFiles.get(index);
34 byteBuffer = mappedFile.sliceByteBuffer();
35 processOffset = mappedFile.getFileFromOffset();
36 mappedFileOffset = 0;
37 }
38 // 查找结果为false, 表明该文件未填满所有消息, 跳出循环, 结束循环
39 else if (!dispatchRequest.isSuccess()) {
40 log.info("recover physics file end, " +
41 mappedFile.getFileName());
42 break;
43 }
44 }
45 }
46 }
```

```

44 //更新MappedFileQueue的flushedwhere和committedwhere指针
45 processOffset += mappedFileOffset;
46 this.mappedFileQueue.setFlushedWhere(processOffset);
47 this.mappedFileQueue.setCommittedWhere(processOffset);
48 //删除offset之后的所有文件
49 this.mappedFileQueue.truncateDirtyFiles(processOffset);
50
51
52 if (maxPhyOffsetOfConsumeQueue >= processOffset) {
53
53 this.defaultMessageStore.truncateDirtyLogicFiles(processOffset);
54 }
55 } else {
56 this.mappedFileQueue.setFlushedWhere(0);
57 this.mappedFileQueue.setCommittedWhere(0);
58 this.defaultMessageStore.destroyLogics();
59 }
60 }
```

**代码: MappedFileQueue#truncateDirtyFiles**

```

1 public void truncateDirtyFiles(long offset) {
2 List<MappedFile> willRemoveFiles = new ArrayList<MappedFile>();
3 //遍历目录下文件
4 for (MappedFile file : this.mappedFiles) {
5 //文件尾部的偏移量
6 long fileTailOffset = file.getFileFromOffset() +
7 this.mappedFileSize;
8 //文件尾部的偏移量大于offset
9 if (fileTailOffset > offset) {
10 //offset大于文件的起始偏移量
11 if (offset >= file.getFileFromOffset()) {
12 //更新wrotePosition、committedPosition、flushedPosistion
13 file.setWrotePosition((int) (offset %
14 this.mappedFileSize));
15 file.setCommittedPosition((int) (offset %
16 this.mappedFileSize));
17 file.setFlushedPosition((int) (offset %
18 this.mappedFileSize));
19 } else {
20 //offset小于文件的起始偏移量,说明该文件是有效文件后面创建的,释放
21 //mappedFile占用内存,删除文件
22 file.destroy(1000);
23 willRemoveFiles.add(file);
24 }
25 }
26 }
27
28 this.deleteExpiredFile(willRemoveFiles);
29 }
```

### 5.4.6.3 异常恢复

Broker异常停止文件恢复的实现为CommitLog#recoverAbnormally。异常文件恢复步骤与正常停止文件恢复流程基本相同，其主要差别有两个。首先，正常停止默认从倒数第三个文件开始进行恢复，而异常停止则需要从最后一个文件往前走，找到第一个消息存储正常的文件。其次，如果CommitLog目录没有消息文件，如果消息消费队列目录下存在文件，则需要销毁。

代码: CommitLog#recoverAbnormally

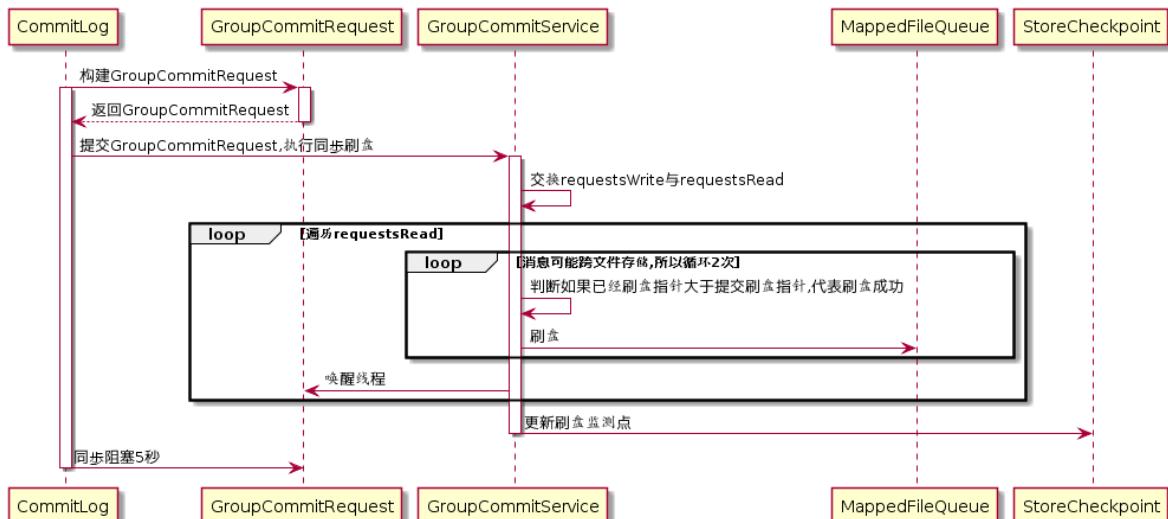
```
1 if (!mappedFiles.isEmpty()) {
2 // Looking beginning to recover from which file
3 int index = mappedFiles.size() - 1;
4 MappedFile mappedFile = null;
5 for (; index >= 0; index--) {
6 mappedFile = mappedFiles.get(index);
7 //判断消息文件是否是一个正确的文件
8 if (this.isMappedFileMatchedRecover(mappedFile)) {
9 log.info("recover from this mapped file " +
10 mappedFile.getFileName());
11 break;
12 }
13 }
14 //根据索引取出mappedFile文件
15 if (index < 0) {
16 index = 0;
17 mappedFile = mappedFiles.get(index);
18 }
19 //...验证消息的合法性，并将消息转发到消息消费队列和索引文件
20 }else{
21 //未找到mappedFile,重置flushWhere、committedWhere都为0，销毁消息队列文件
22 this.mappedFileQueue.setFlushedWhere(0);
23 this.mappedFileQueue.setCommittedWhere(0);
24 this.defaultMessageStore.destroyLogics();
25 }
```

## 5.4.7 刷盘机制

RocketMQ的存储是基于JDK NIO的内存映射机制（MappedByteBuffer）的，消息存储首先将消息追加到内存，再根据配置的刷盘策略在不同时间进行刷写磁盘。

### 5.4.7.1 同步刷盘

消息追加到内存后，立即将数据刷写到磁盘文件



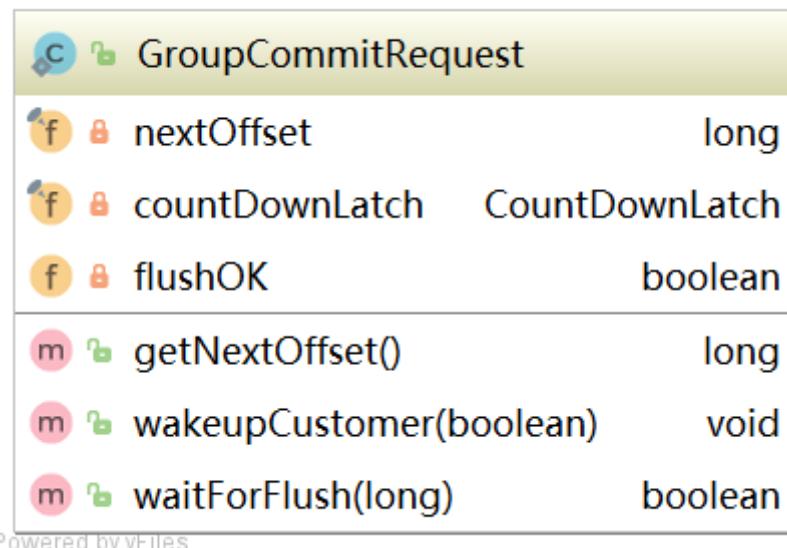
代码: CommitLog#handleDiskFlush

```

1 //刷盘服务
2 final GroupCommitService service = (GroupCommitService)
this.flushCommitLogService;
3 if (messageExt.isWaitStoreMsgOK()) {
4 //封装刷盘请求
5 GroupCommitRequest request = new
GroupCommitRequest(result.getWroteOffset() + result.getWroteBytes());
6 //提交刷盘请求
7 service.putRequest(request);
8 //线程阻塞5秒，等待刷盘结束
9 boolean flushOK =
request.waitForFlush(this.defaultMessageStore.getMessageStoreConfig().getSy
ncFlushTimeout());
10 if (!flushOK) {
11
putMessageResult.setPutMessageStatus(PutMessageStatus.FLUSH_DISK_TIMEOUT);
12 }

```

### *GroupCommitRequest*



Powered by yFiles

```

1 long nextOffset; //刷盘点偏移量
2 CountDownLatch countDownLatch = new CountDownLatch(1); //倒计时锁存器
3 volatile boolean flushOK = false; //刷盘结果;默认为false

```

### *代码: GroupCommitService#run*

```

1 public void run() {
2 CommitLog.log.info(this.getServiceName() + " service started");
3
4 while (!this.isStopped()) {
5 try {
6 //线程等待10ms
7 this.waitForRunning(10);
8 //执行提交
9 this.doCommit();
10 } catch (Exception e) {
11 CommitLog.log.warn(this.getServiceName() + " service has
exception. ", e);

```

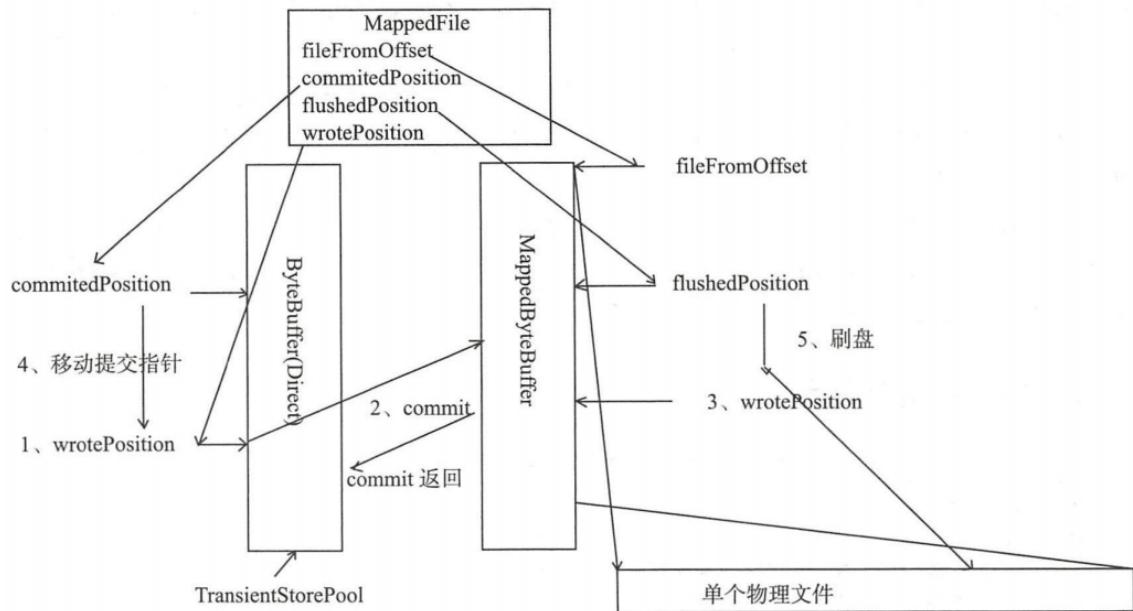
```
12 }
13 }
14 ...
15 }
```

代码: GroupCommitService#doCommit

```
1 private void doCommit() {
2 //加锁
3 synchronized (this.requestsRead) {
4 if (!this.requestsRead.isEmpty()) {
5 //遍历requestsRead
6 for (GroupCommitRequest req : this.requestsRead) {
7 // There may be a message in the next file, so a maximum of
8 // two times the flush
9 boolean flushOK = false;
10 for (int i = 0; i < 2 && !flushOK; i++) {
11 flushOK =
12 CommitLog.this.mappedFileQueue.getFlushedWhere() >= req.getNextOffset();
13 //刷盘
14 if (!flushOK) {
15 CommitLog.this.mappedFileQueue.flush(0);
16 }
17 //唤醒发送消息客户端
18 req.wakeupCustomer(flushOK);
19 }
20
21 //更新刷盘监测点
22 long storeTimestamp =
23 CommitLog.this.mappedFileQueue.getStoreTimestamp();
24 if (storeTimestamp > 0) {
25 CommitLog.this.defaultMessageStore.getStoreCheckpoint().setPhysicMsgTimestamp(storeTimestamp);
26
27 this.requestsRead.clear();
28 } else {
29 // Because of individual messages is set to not sync flush, it
30 // will come to this process
31 CommitLog.this.mappedFileQueue.flush(0);
32 }
33 }
34 }
35 }
36 }
```

#### 5.4.7.2 异步刷盘

在消息追加到内存后，立即返回给消息发送端。如果开启transientStorePoolEnable，RocketMQ会单独申请一个与目标物理文件（commitLog）同样大小的堆外内存，该堆外内存将使用内存锁定，确保不会被置换到虚拟内存中去，消息首先追加到堆外内存，然后提交到物理文件的内存映射中，然后刷写到磁盘。如果未开启transientStorePoolEnable，消息直接追加到物理文件直接映射文件中，然后刷写到磁盘中。



开启transientStorePoolEnable后异步刷盘步骤:

1. 将消息直接追加到ByteBuffer (堆外内存)
2. CommitRealTimeService线程每隔200ms将ByteBuffer新追加内容提交到 MappedByteBuffer中
3. MappedByteBuffer在内存中追加提交的内容, wrotePosition指针向后移动
4. commit操作成功返回, 将committedPosition位置恢复
5. FlushRealTimeService线程默认每500ms将MappedByteBuffer中新追加的内存刷写到磁盘

**代码: CommitLog\$CommitRealTimeService#run**

提交线程工作机制

```

1 //间隔时间,默认200ms
2 int interval =
3 CommitLog.this.defaultMessageStore.getMessageStoreConfig().getCommitInterval();
4
5 //一次提交的至少页数
6 int commitDataLeastPages =
7 CommitLog.this.defaultMessageStore.getMessageStoreConfig().getCommitCommitLogLeastPages();
8
9 //两次真实提交的最大间隔,默认200ms
10 int commitDataThoroughInterval =
11 CommitLog.this.defaultMessageStore.getMessageStoreConfig().getCommitCommitLogThoroughInterval();
12
13 //上次提交间隔超过commitDataThoroughInterval,则忽略提交
14 //commitDataThoroughInterval参数,直接提交
15 long begin = System.currentTimeMillis();
16 if (begin >= (this.lastCommitTimestamp + commitDataThoroughInterval)) {
17 this.lastCommitTimestamp = begin;
18 commitDataLeastPages = 0;
19 }
20
21 //执行提交操作,将待提交数据提交到物理文件的内存映射区
22 boolean result =
23 CommitLog.this.mappedFileQueue.commit(commitDataLeastPages);

```

```

20 long end = System.currentTimeMillis();
21 if (!result) {
22 this.lastCommitTimestamp = end; // result = false means some data
23 committed.
24 //now wake up flush thread.
25 //唤醒刷盘线程
26 flushCommitLogService.wakeup();
27 }
28 if (end - begin > 500) {
29 log.info("Commit data to file costs {} ms", end - begin);
30 }
31 this.waitForRunning(interval);

```

#### 代码: CommitLog\$FlushRealTimeService#run

刷盘线程工作机制

```

1 //表示await方法等待，默认false
2 boolean flushCommitLogTimed =
3 CommitLog.this.defaultMessageStore.getMessageStoreConfig().isFlushCommitLog
4 Timed();
5 //线程执行时间间隔
6 int interval =
7 CommitLog.this.defaultMessageStore.getMessageStoreConfig().getFlushInterval
8 CommitLog();
9 //一次刷写任务至少包含页数
10 int flushPhysicQueueLeastPages =
11 CommitLog.this.defaultMessageStore.getMessageStoreConfig().getFlushCommitLo
12 gLeastPages();
13 //两次真实刷写任务最大间隔
14 int flushPhysicQueueThoroughInterval =
15 CommitLog.this.defaultMessageStore.getMessageStoreConfig().getFlushCommitLo
16 gThoroughInterval();
17 ...
18 //距离上次提交间隔超过flushPhysicQueueThoroughInterval，则本次刷盘任务将忽略
19 flushPhysicQueueLeastPages，直接提交
20 long currentTimeMillis = System.currentTimeMillis();
21 if (currentTimeMillis >= (this.lastFlushTimestamp +
22 flushPhysicQueueThoroughInterval)) {
23 this.lastFlushTimestamp = currentTimeMillis;
24 flushPhysicQueueLeastPages = 0;
25 printFlushProgress = (printTimes++ % 10) == 0;
26 }
27 ...
28 //执行一次刷盘前，先等待指定时间间隔
29 if (flushCommitLogTimed) {
30 Thread.sleep(interval);
31 } else {
32 this.waitForRunning(interval);
33 }
34 ...
35 long begin = System.currentTimeMillis();
36 //刷写磁盘
37 CommitLog.this.mappedFileQueue.flush(flushPhysicQueueLeastPages);
38 long storeTimestamp = CommitLog.this.mappedFileQueue.getStoreTimestamp();
39 if (storeTimestamp > 0) {

```

```
31 //更新存储监测点文件的时间戳
32 CommitLog.this.defaultMessageStore.getStoreCheckpoint().setPhysicMsgTimestamp(storeTimestamp);
33
```

## 5.4.8 过期文件删除机制

由于RocketMQ操作CommitLog、ConsumerQueue文件是基于内存映射机制并在启动的时候回加载CommitLog、ConsumerQueue目录下的所有文件，为了避免内存与磁盘的浪费，不可能将消息永久存储在消息服务器上，所以要引入一种机制来删除已过期的文件。RocketMQ顺序写CommitLog、ConsumerQueue文件，所有写操作全部落在最后一个CommitLog或者ConsumerQueue文件上，之前的文件在下一个文件创建后将不会再被更新。RocketMQ清除过期文件的方法时：如果当前文件在在一定时间间隔内没有再次被消费，则认为是过期文件，可以被删除，RocketMQ不会关注这个文件上的消息是否全部被消费。默认每个文件的过期时间为72小时，通过在Broker配置文件中设置fileReservedTime来改变过期时间，单位为小时。

**代码:** *DefaultMessageStore#addScheduleTask*

```
1 private void addScheduleTask() {
2 //每隔10s调度一次清除文件
3 this.scheduledExecutorService.scheduleAtFixedRate(new Runnable() {
4 @Override
5 public void run() {
6 DefaultMessageStore.this.cleanFilesPeriodically();
7 }
8 }, 1000 * 60, this.messageStoreConfig.getCleanResourceInterval(),
9 TimeUnit.MILLISECONDS);
10 ...
11}
```

**代码:** *DefaultMessageStore#cleanFilesPeriodically*

```
1 private void cleanFilesPeriodically() {
2 //清除存储文件
3 this.cleanCommitLogService.run();
4 //清除消息消费队列文件
5 this.cleanConsumeQueueService.run();
6 }
```

**代码:** *DefaultMessageStore#deleteExpiredFiles*

```
1 private void deleteExpiredFiles() {
2 //删除的数量
3 int deleteCount = 0;
4 //文件保留的时间
5 long fileReservedTime =
DefaultMessageStore.this.getMessageStoreConfig().getFileReservedTime();
6 //删除物理文件的间隔
7 int deletePhysicFilesInterval =
DefaultMessageStore.this.getMessageStoreConfig().getDeleteCommitLogFileInterval();
8 //线程被占用，第一次拒绝删除后能保留的最大时间，超过该时间，文件将被强制删除
```

```

9 int destroyMapedFileIntervalForcibly =
10 DefaultMessageStore.this.getMessageStoreConfig().getDestroyMapedFileIntervalForcibly();
11
12 boolean timeup = this.isTimeToDelete();
13 boolean spacefull = this.isSpaceToDelete();
14 boolean manualDelete = this.manualDeleteFileSeveralTimes > 0;
15 if (timeup || spacefull || manualDelete) {
16 ...执行删除逻辑
17 } else{
18 ...无作为
19 }

```

### 删除文件操作的条件

1. 指定删除文件的时间点，RocketMQ通过deleteWhen设置一天的固定时间执行一次删除过期文件操作，默认4点
2. 磁盘空间如果不充足，删除过期文件
3. 预留，手工触发。

**代码: CleanCommitLogService#isSpaceToDelete**

当磁盘空间不足时执行删除过期文件

```

1 private boolean isSpaceToDelete() {
2 //磁盘分区的最大使用量
3 double ratio =
DefaultMessageStore.this.getMessageStoreConfig().getDiskMaxUsedSpaceRatio()
/ 100.0;
4 //是否需要立即执行删除过期文件操作
5 cleanImmediately = false;
6
7 {
8 String storePathPhysic =
DefaultMessageStore.this.getMessageStoreConfig().getStorePathCommitLog();
9 //当前CommitLog目录所在的磁盘分区的磁盘使用率
10 double physicRatio =
utilAll.getDiskPartitionSpaceUsedPercent(storePathPhysic);
11 //diskSpaceWarningLevelRatio:磁盘使用率警告阈值,默认0.90
12 if (physicRatio > diskSpaceWarningLevelRatio) {
13 boolean diskok =
DefaultMessageStore.this.runningFlags.getAndMakeDiskFull();
14 if (diskok) {
15 DefaultMessageStore.log.error("physic disk maybe full soon
" + physicRatio + ", so mark disk full");
16 }
17 //diskSpaceCleanForciblyRatio:强制清除阈值,默认0.85
18 cleanImmediately = true;
19 } else if (physicRatio > diskSpaceCleanForciblyRatio) {
20 cleanImmediately = true;
21 } else {
22 boolean diskok =
DefaultMessageStore.this.runningFlags.getAndMakeDiskOK();
23 if (!diskok) {
24 DefaultMessageStore.log.info("physic disk space OK " +
physicRatio + ", so mark disk ok");
25 }
26 }

```

```

27 if (physicRatio < 0 || physicRatio > ratio) {
28 DefaultMessageStore.log.info("physic disk maybe full soon, so
29 reclaim space, " + physicRatio);
30 return true;
31 }
32 }
```

#### 代码: MappedFileQueue#deleteExpiredFileByTime

执行文件销毁和删除

```

1 for (int i = 0; i < mfsLength; i++) {
2 //遍历每隔文件
3 MappedFile mappedFile = (MappedFile) mfs[i];
4 //计算文件存活时间
5 Long liveMaxTimestamp = mappedFile.getLastModifiedTimestamp() +
6 expiredTime;
7 //如果超过72小时,执行文件删除
8 if (System.currentTimeMillis() >= liveMaxTimestamp || cleanImmediately)
9 {
10 if (mappedFile.destroy(intervalForcibly)) {
11 files.add(mappedFile);
12 deleteCount++;
13
14 if (files.size() >= DELETE_FILES_BATCH_MAX) {
15 break;
16 }
17
18 if (deleteFilesInterval > 0 && (i + 1) < mfsLength) {
19 try {
20 Thread.sleep(deleteFilesInterval);
21 } catch (InterruptedException e) {
22 }
23 }
24 }
25 } else {
26 //avoid deleting files in the middle
27 break;
28 }
29 }
```

## 5.4.9 小结

RocketMQ的存储文件包括消息文件 (Commitlog) 、消息消费队列文件 (ConsumerQueue) 、Hash索引文件 (IndexFile) 、监测点文件 (checkPoint) 、abort (关闭异常文件) 。单个消息存储文件、消息消费队列文件、Hash索引文件长度固定以便使用内存映射机制进行文件的读写操作。RocketMQ组织文件以文件的起始偏移量来命名文件，这样根据偏移量能快速定位到真实的物理文件。RocketMQ基于内存映射文件机制提供了同步刷盘和异步刷盘两种机制，异步刷盘是指在消息存储时先追加到内存映射文件，然后启动专门的刷盘线程定时将内存中的文件数据刷写到磁盘。

CommitLog，消息存储文件，RocketMQ为了保证消息发送的高吞吐量，采用单一文件存储所有主题消息，保证消息存储是完全的顺序写，但这样给文件读取带来了不便，为此RocketMQ为了方便消息消费构建了消息消费队列文件，基于主题与队列进行组织，同时RocketMQ为消息实现了Hash索引，可以为消息设置索引键，根据所以能够快速从CommitLog文件中检索消息。

当消息达到CommitLog后，会通过ReputMessageService线程接近实时地将消息转发给消息消费队列文件与索引文件。为了安全起见，RocketMQ引入abort文件，记录Broker的停机是否是正常关闭还是异常关闭，在重启Broker时为了保证CommitLog文件，消息消费队列文件与Hash索引文件的正确性，分别采用不同策略来恢复文件。

RocketMQ不会永久存储消息文件、消息消费队列文件，而是启动文件过期机制并在磁盘空间不足或者默认凌晨4点删除过期文件，文件保存72小时并且在删除文件时并不会判断该消息文件上的消息是否被消费。

## 5.5 Consumer

### 5.5.1 消息消费概述

消息消费以组的模式开展，一个消费组内可以包含多个消费者，每一个消费者组可订阅多个主题，消费组内消费者之间有集群模式和广播模式两种消费模式。

集群模式，主题下的同一条消息只允许被其中一个消费者消费。

广播模式，主题下的同一条消息，将被集群内的所有消费者消费一次。

消息服务器与消费者之间的消息传递也有两种模式：推模式、拉模式。

所谓的拉模式，是消费端主动拉起拉消息请求，

而推模式是消息达到消息服务器后，推送给消息消费者。

RocketMQ消息推模式的实现**基于拉模式**，在拉模式上包装一层，一个拉取任务完成后开始下一个拉取任务。

集群模式下，多个消费者如何对消息队列进行负载呢？

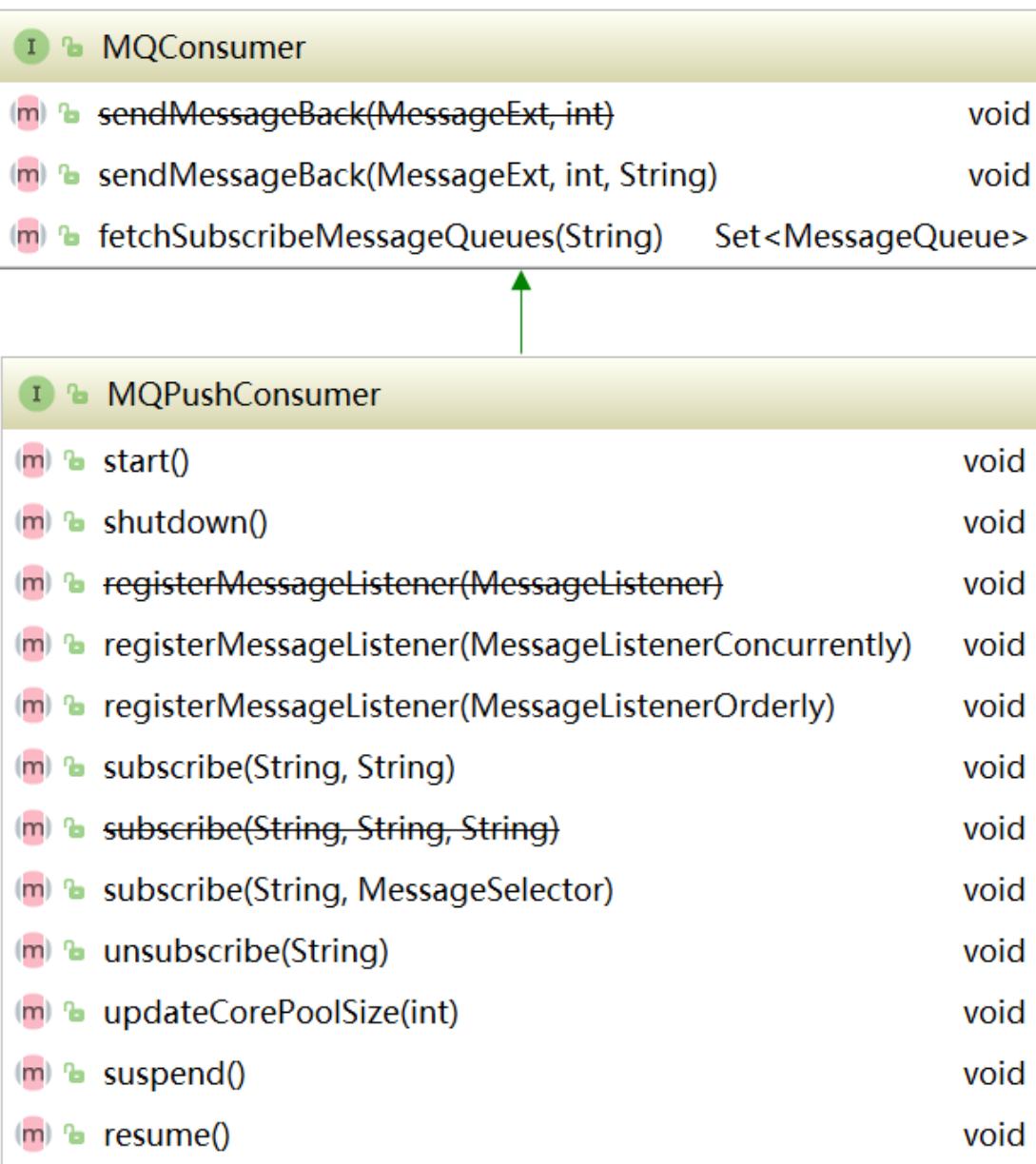
消息队列负载机制遵循一个通用思想：一个消息队列同一个时间只允许被一个消费者消费，一个消费者可以消费多个消息队列。

RocketMQ支持局部顺序消息消费，也就是保证同一个消息队列上的消息顺序消费。

**不支持消息全局顺序消费**，如果要实现某一个主题的全局顺序消费，可以将该主题的队列数设置为1，牺牲高可用性。

### 5.5.2 消息消费初探

#### 消息推送模式



Powered by yFiles

## 消息消费重要方法

- 1 `void sendMessageBack(final MessageExt msg, final int delayLevel, final String brokerName)`: 发送消息确认
- 2 `Set<MessageQueue> fetchSubscribeMessageQueues(final String topic)` : 获取消费者对主题分配了那些消息队列
- 3 `void registerMessageListener(final MessageListenerConcurrently messageListener)`: 注册并发事件监听器
- 4 `void registerMessageListener(final MessageListenerOrderly messageListener)`: 注册顺序消息事件监听器
- 5 `void subscribe(final String topic, final String subExpression)`: 基于主题订阅消息，消息过滤使用表达式
- 6 `void subscribe(final String topic, final String fullClassName, final String filterClassName)`: 基于主题订阅消息，消息过滤使用类模式
- 7 `void subscribe(final String topic, final MessageSelector selector)` : 订阅消息，并指定队列选择器
- 8 `void unsubscribe(final String topic)`: 取消消息订阅

## DefaultMQPushConsumer

| DefaultMQPushConsumer |                               |
|-----------------------|-------------------------------|
| f                     | log                           |
| f                     | defaultMQPushConsumerImpl     |
| f                     | consumerGroup                 |
| f                     | messageModel                  |
| f                     | consumeFromWhere              |
| f                     | consumeTimestamp              |
| f                     | allocateMessageQueueStrategy  |
| f                     | subscription                  |
| f                     | messageListener               |
| f                     | offsetStore                   |
| f                     | consumeThreadMin              |
| f                     | consumeThreadMax              |
| f                     | adjustThreadPoolNumsThreshold |
| f                     | consumeConcurrentlyMaxSpan    |
| f                     | pullThresholdForQueue         |
| f                     | pullThresholdSizeForQueue     |
| f                     | pullThresholdForTopic         |
| f                     | pullThresholdSizeForTopic     |
| f                     | pullInterval                  |
| f                     | consumeMessageBatchMaxSize    |
| f                     | pullBatchSize                 |
| f                     | postSubscriptionWhenPull      |
| f                     | unitMode                      |
| f                     | maxReconsumeTimes             |
| f                     | suspendCurrentQueueTimeMillis |
| f                     | consumeTimeout                |
| f                     | traceDispatcher               |

Powered by yFiles

```

1 //消费者组
2 private String consumerGroup;
3 //消息消费模式
4 private MessageModel messageModel = MessageModel.CLUSTERING;
5 //指定消费开始偏移量（最大偏移量、最小偏移量、启动时间戳）开始消费

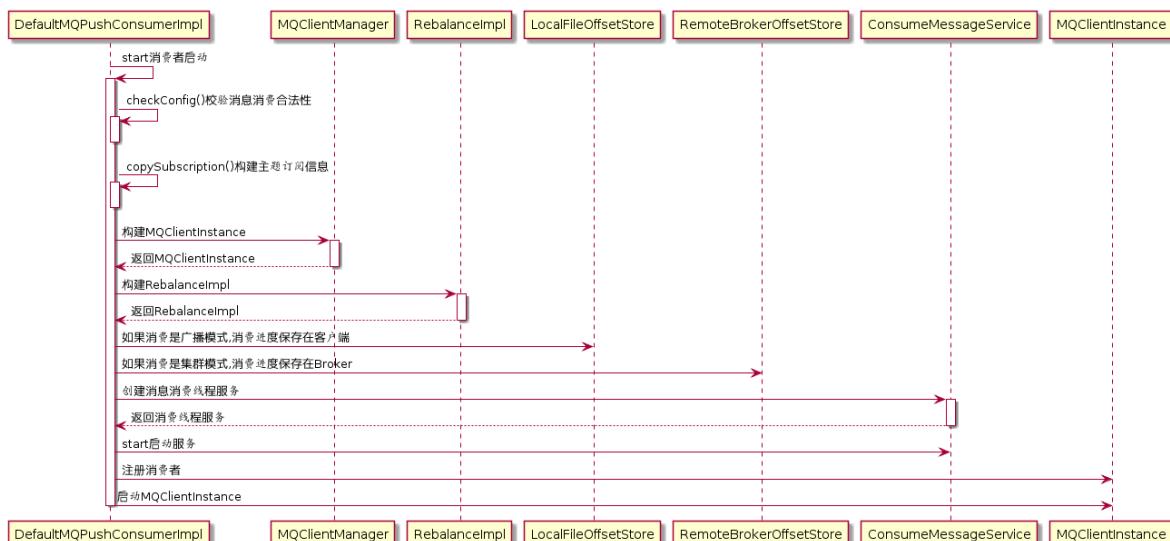
```

```

6 private ConsumeFromwhere consumeFromWhere =
7 ConsumeFromwhere.CONSUME_FROM_LAST_OFFSET;
8 private AllocateMessageQueueStrategy allocateMessageQueueStrategy;
9 //订阅信息
10 private Map<String /* topic */, String /* sub expression */> subscription =
11 new HashMap<String, String>();
12 private MessageListener messageListener;
13 //消息消费进度存储器
14 private OffsetStore offsetStore;
15 //消费者最小线程数量
16 private int consumeThreadMin = 20;
17 //消费者最大线程数量
18 private int consumeThreadMax = 20;
19 //并发消息消费时处理队列最大跨度
20 private int consumeConcurrentlyMaxSpan = 2000;
21 //每1000次流控后打印流控日志
22 private int pullThresholdForQueue = 1000;
23 //推模式下任务间隔时间
24 private long pullInterval = 0;
25 //推模式下任务拉取的条数,默认32条
26 private int pullBatchSize = 32;
27 //每次传入MessageListener#consumerMessage中消息的数量
28 private int consumeMessageBatchMaxsize = 1;
29 //是否每次拉取消息都订阅消息
30 private boolean postSubscriptionWhenPull = false;
31 //消息重试次数,-1代表16次
32 private int maxReconsumeTimes = -1;
33 //消息消费超时时间
34 private long consumeTimeout = 15;

```

### 5.5.3 消费者启动流程



代码: `DefaultMQPushConsumerImpl#start`

```

1 public synchronized void start() throws MQClientException {
2 switch (this.serviceState) {
3 case CREATE_JUST:
4

```

```
5 this.defaultMQPushConsumer.getMessageModel(),
6 this.defaultMQPushConsumer.isUnitMode());
7 this.serviceState = ServiceState.START_FAILED;
8 //检查消息者是否合法
9 this.checkConfig();
10 //构建主题订阅信息
11 this.copySubscription();
12 //设置消费者客户端实例名称为进程ID
13 if (this.defaultMQPushConsumer.getMessageModel() ==
14 MessageModel.CLUSTERING) {
15 this.defaultMQPushConsumer.changeInstanceNameToPID();
16 }
17 //创建MQClient实例
18 this.mQClientFactory =
19 MQClientManager.getInstance().getAndCreateMQClientInstance(this.defaultMQPushConsumer, this.rpcHook);
20 //构建rebalanceImpl
21
22 this.rebalanceImpl.setConsumerGroup(this.defaultMQPushConsumer.getConsumerGroup());
23
24 this.rebalanceImpl.setMessageModel(this.defaultMQPushConsumer.getMessageModel());
25
26 this.rebalanceImpl.setAllocateMessageQueueStrategy(this.defaultMQPushConsumer.getAllocateMessageQueueStrategy());
27 this.rebalanceImpl.setmQClientFactory(this.mQClientFactor
28 this.pullAPIWrapper = new PullAPIWrapper(
29 mQClientFactory,
30 this.defaultMQPushConsumer.getConsumerGroup(),
31 isUnitMode());
32
33 this.pullAPIWrapper.registerFilterMessageHook(filterMessageHookList);
34 if (this.defaultMQPushConsumer.getOffsetStore() != null) {
35 this.offsetStore =
36 this.defaultMQPushConsumer.getOffsetStore();
37 } else {
38 switch (this.defaultMQPushConsumer.getMessageModel()) {
39
40 case BROADCASTING: //消息消费广播模式,将消费进度保存在本地
41 this.offsetStore = new
42 LocalFileOffsetStore(this.mQClientFactory,
43 this.defaultMQPushConsumer.getConsumerGroup());
44 break;
45 case CLUSTERING: //消息消费集群模式,将消费进度保存在远端
46 Broker
47 this.offsetStore = new
48 RemoteBrokerOffsetStore(this.mQClientFactory,
49 this.defaultMQPushConsumer.getConsumerGroup());
50 break;
51 default:
52 break;
53 }
54
55 this.defaultMQPushConsumer.setOffsetStore(this.offsetStore);
56 }
57 this.offsetStore.load
58 //创建顺序消息消费服务
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
```

```

44 if (this.getMessageListenerInner() instanceof
45 MessageListenerOrderly) {
46 this.consumeOrderly = true;
47 this.consumeMessageService =
48 new ConsumeMessageOrderlyService(this,
49 (MessageListenerOrderly) this.getMessageListenerInner());
50 //创建并发消息消费服务
51 } else if (this.getMessageListenerInner() instanceof
52 MessageListenerConcurrently) {
53 this.consumeOrderly = false;
54 this.consumeMessageService =
55 new ConsumeMessageConcurrentlyService(this,
56 (MessageListenerConcurrently) this.getMessageListenerInner());
57 }
58 //消息消费服务启动
59 this.consumeMessageService.start();
60 //注册消费者实例
61 boolean registerOK =
62 mQClientFactory.registerConsumer(this.defaultMQPushConsumer.getConsumerGrou-
63 p(), this);
64
65 if (!registerOK) {
66 this.serviceState = ServiceState.CREATE_JUST;
67 this.consumeMessageService.shutdown();
68 throw new MQClientException("The consumer group[" +
69 this.defaultMQPushConsumer.getConsumerGroup()
70 + "] has been created before, specify another name
71 please." + FAQUrl.suggestTodo(FAQUrl.GROUP_NAME_DUPLICATE_URL),
72 null);
73 //启动消费者客户端
74 mQClientFactory.start();
75 log.info("the consumer [{}] start OK.",
76 this.defaultMQPushConsumer.getConsumerGroup());
77 this.serviceState = ServiceState.RUNNING;
78 break;
79 case RUNNING:
80 case START_FAILED:
81 case SHUTDOWN_ALREADY:
82 throw new MQClientException("The PushConsumer service state not
83 OK, maybe started once,
84 " + this.serviceState
85 + FAQUrl.suggestTodo(FAQUrl.CLIENT_SERVICE_NOT_OK),
86 null);
87 default:
88 break;
89 }
90
91 this.updateTopicSubscribeInfoWhenSubscriptionChanged();
92 this.mQClientFactory.checkClientInBroker();
93 this.mQClientFactory.sendHeartbeatToAllBrokerWithLock();
94 this.mQClientFactory.rebalanceImmediately();
95 }
```

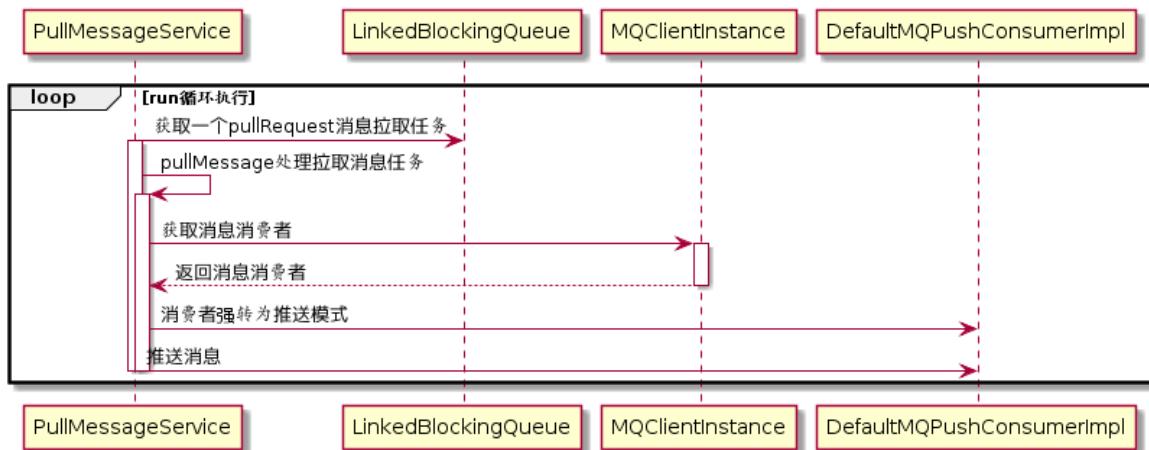
## 5.5.4 消息拉取

消息消费模式有两种模式：广播模式与集群模式。广播模式比较简单，每一个消费者需要拉取订阅主题下所有队列的消息。本文重点讲解**集群模式**。在集群模式下，同一个消费者组内有多个消息消费者，同一个主题存在多个消费队列，消费者通过负载均衡的方式消费消息。

消息队列负载均衡，通常的作法是一个消息队列在同一个时间只允许被一个消费消费者消费，一个消息消费者可以同时消费多个消息队列。

#### 5.5.4.1 PullMessageService实现机制

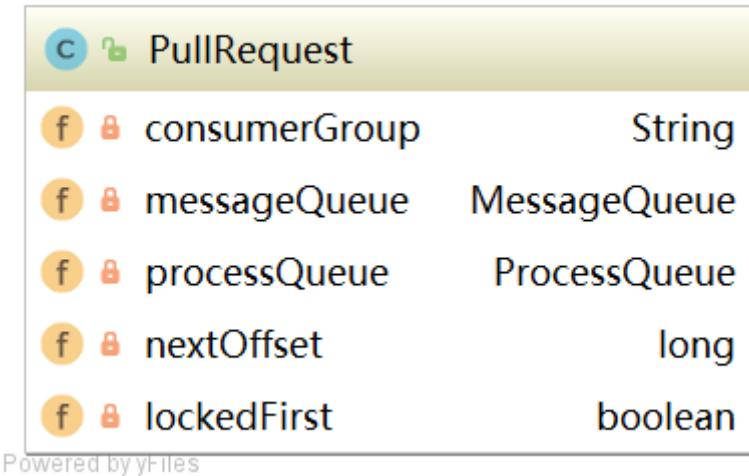
从MQClientInstance的启动流程中可以看出，RocketMQ使用一个单独的线程PullMessageService来负责消息的拉取。



代码: PullMessageService#run

```
1 public void run() {
2 log.info(this.getServiceName() + " service started");
3 //循环拉取消息
4 while (!this.isStopped()) {
5 try {
6 //从请求队列中获取拉取消息请求
7 PullRequest pullRequest = this.pullRequestQueue.take();
8 //拉取消息
9 this.pullMessage(pullRequest);
10 } catch (InterruptedException ignored) {
11 } catch (Exception e) {
12 log.error("Pull Message Service Run Method exception", e);
13 }
14 }
15
16 log.info(this.getServiceName() + " service end");
17 }
```

PullRequest



```

1 private String consumerGroup; //消费者组
2 private MessageQueue messageQueue; //待拉取消息队列
3 private ProcessQueue processQueue; //消息处理队列
4 private long nextOffset; //待拉取的MessageQueue偏移量
5 private boolean lockedFirst = false; //是否被锁定

```

#### 代码: PullMessageService#pullMessage

```

1 private void pullMessage(final PullRequest pullRequest) {
2 //获得消费者实例
3 final MQConsumerInner consumer =
4 this.mQClientFactory.selectConsumer(pullRequest.getConsumerGroup());
5 if (consumer != null) {
6 //强转为推送模式消费者
7 DefaultMQPushConsumerImpl impl = (DefaultMQPushConsumerImpl)
8 consumer;
9 //推送消息
10 impl.pullMessage(pullRequest);
11 } else {
12 log.warn("No matched consumer for the PullRequest {}, drop it",
13 pullRequest);
14 }
15 }

```

#### 5.5.4.2 ProcessQueue实现机制

ProcessQueue是MessageQueue在消费端的重现、快照。PullMessageService从消息服务器默认每次拉取32条消息，按照消息的队列偏移量顺序存放在ProcessQueue中，PullMessageService然后将消息提交到消费者消费线程池，消息成功消费后从ProcessQueue中移除。

|  |                                               |                           |
|--|-----------------------------------------------|---------------------------|
|  | <b>ProcessQueue</b>                           |                           |
|  | <b>REBALANCE_LOCK_MAX_LIVE_TIME</b>           | long                      |
|  | <b>REBALANCE_LOCK_INTERVAL</b>                | long                      |
|  | <b>PULL_MAX_IDLE_TIME</b>                     | long                      |
|  | <b>log</b>                                    | InternalLogger            |
|  | <b>lockTreeMap</b>                            | ReadWriteLock             |
|  | <b>msgTreeMap</b>                             | TreeMap<Long, MessageExt> |
|  | <b>msgCount</b>                               | AtomicLong                |
|  | <b>msgSize</b>                                | AtomicLong                |
|  | <b>lockConsume</b>                            | Lock                      |
|  | <b>consumingMsgOrderlyTreeMap</b>             | TreeMap<Long, MessageExt> |
|  | <b>tryUnlockTimes</b>                         | AtomicLong                |
|  | <b>queueOffsetMax</b>                         | long                      |
|  | <b>dropped</b>                                | boolean                   |
|  | <b>lastPullTimestamp</b>                      | long                      |
|  | <b>lastConsumeTimestamp</b>                   | long                      |
|  | <b>locked</b>                                 | boolean                   |
|  | <b>lastLockTimestamp</b>                      | long                      |
|  | <b>consuming</b>                              | boolean                   |
|  | <b>msgAccCnt</b>                              | long                      |
|  | <b>isLockExpired()</b>                        | boolean                   |
|  | <b>isPullExpired()</b>                        | boolean                   |
|  | <b>cleanExpiredMsg(DefaultMQPushConsumer)</b> | void                      |
|  | <b>putMessage(List&lt;MessageExt&gt;)</b>     | boolean                   |
|  | <b>getMaxSpan()</b>                           | long                      |
|  | <b>removeMessage(List&lt;MessageExt&gt;)</b>  | long                      |
|  | <b>getMsgTreeMap()</b>                        | TreeMap<Long, MessageExt> |
|  | <b>getMsgCount()</b>                          | AtomicLong                |
|  | <b>getMsgSize()</b>                           | AtomicLong                |
|  | <b>isDropped()</b>                            | boolean                   |
|  | <b>setDropped(boolean)</b>                    | void                      |
|  | <b>isLocked()</b>                             | boolean                   |

|   |                                            |                  |
|---|--------------------------------------------|------------------|
| J | setLocked(boolean)                         | void             |
| J | rollback()                                 | void             |
| J | commit()                                   | long             |
| J | makeMessageToCosomeAgain(List<MessageExt>) | void             |
| J | takeMessags(int)                           | List<MessageExt> |
| J | hasTempMessage()                           | boolean          |
| J | clear()                                    | void             |
| J | getLastLockTimestamp()                     | long             |
| J | setLastLockTimestamp(long)                 | void             |
| J | getLockConsume()                           | Lock             |
| J | getLastPullTimestamp()                     | long             |
| J | setLastPullTimestamp(long)                 | void             |
| J | getMsgAccCnt()                             | long             |
| J | setMsgAccCnt(long)                         | void             |
| J | getTryUnlockTimes()                        | long             |
| J | incTryUnlockTimes()                        | void             |
| J | fillProcessQueueInfo(ProcessQueueInfo)     | void             |
| J | getLastConsumeTimestamp()                  | long             |
| J | setLastConsumeTimestamp(long)              | void             |

Powered by yFiles

## 属性

```

1 //消息容器
2 private final TreeMap<Long, MessageExt> msgTreeMap = new TreeMap<Long,
MessageExt>();
3 //读写锁
4 private final ReadwriteLock lockTreeMap = new ReentrantReadWriteLock();
5 //ProcessQueue总消息数
6 private final AtomicLong msgCount = new AtomicLong();
7 //ProcessQueue队列最大偏移量
8 private volatile long queueOffsetMax = 0L;
9 //当前ProcessQueue是否被丢弃
10 private volatile boolean dropped = false;
11 //上一次拉取时间戳
12 private volatile long lastPullTimestamp = System.currentTimeMillis();
13 //上一次消费时间戳
14 private volatile long lastConsumeTimestamp = System.currentTimeMillis();

```

## 方法

```

1 //移除消费超时消息
2 public void cleanExpiredMsg(DefaultMQPushConsumer pushConsumer)

```

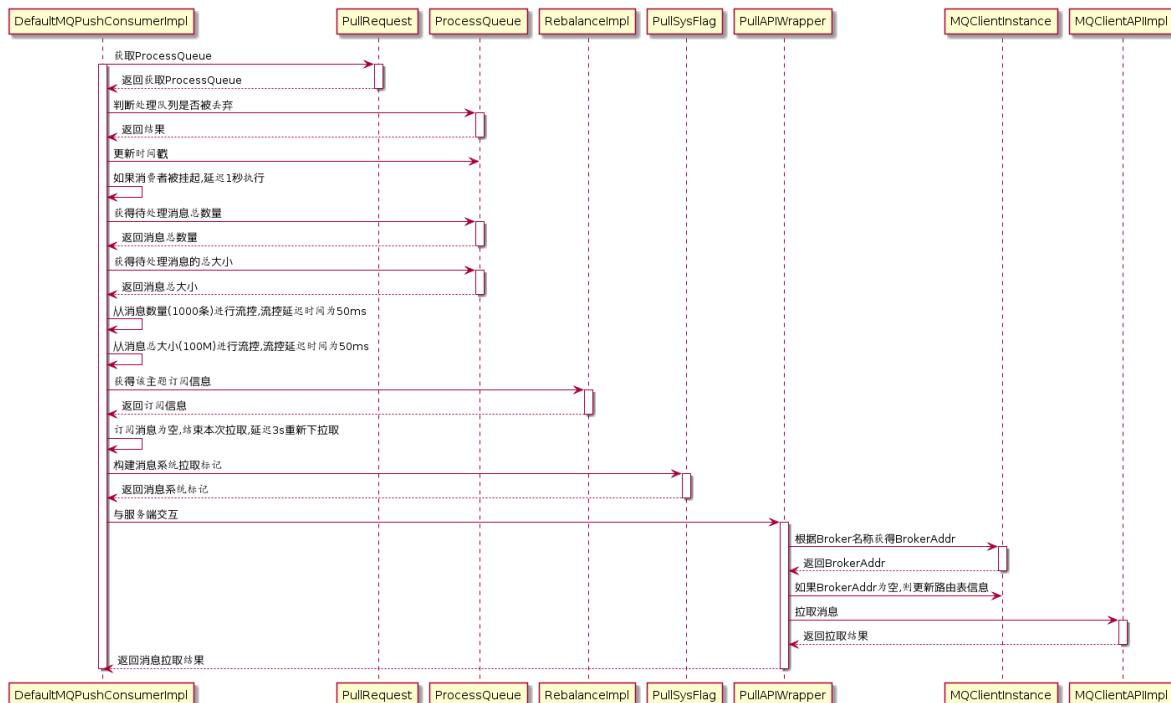
```

3 //添加消息
4 public boolean putMessage(final List<MessageExt> msgs)
5 //获取消息最大间隔
6 public long getMaxSpan()
7 //移除消息
8 public long removeMessage(final List<MessageExt> msgs)
9 //将consumingMsgOrderlyTreeMap中消息重新放在msgTreeMap，并清空
10 consumingMsgOrderlyTreeMap
11 public void rollback()
12 //将consumingMsgOrderlyTreeMap消息清除，表示成功处理该批消息
13 public long commit()
14 //重新处理该批消息
15 public void makeMessageToCosomeAgain(List<MessageExt> msgs)
16 //从processQueue中取出batchsize条消息
17 public List<MessageExt> takeMessages(final int batchsize)

```

### 5.5.4.3 消息拉取基本流程

#### 5.5.4.3.1.客户端发起拉取请求



代码: `DefaultMQPushConsumerImpl#pullMessage`

```

1 public void pullMessage(final PullRequest pullRequest) {
2 //从pullRequest获得ProcessQueue
3 final ProcessQueue processQueue = pullRequest.getProcessQueue();
4 //如果处理队列被丢弃,直接返回
5 if (processQueue.isDropped()) {
6 log.info("the pull request[{}] is dropped.",
7 pullRequest.toString());
8 return;
9 }
10 //如果处理队列未被丢弃,更新时间戳
11 pullRequest.getProcessQueue().setLastPullTimestamp(System.currentTimeMillis());
12
13 try {

```

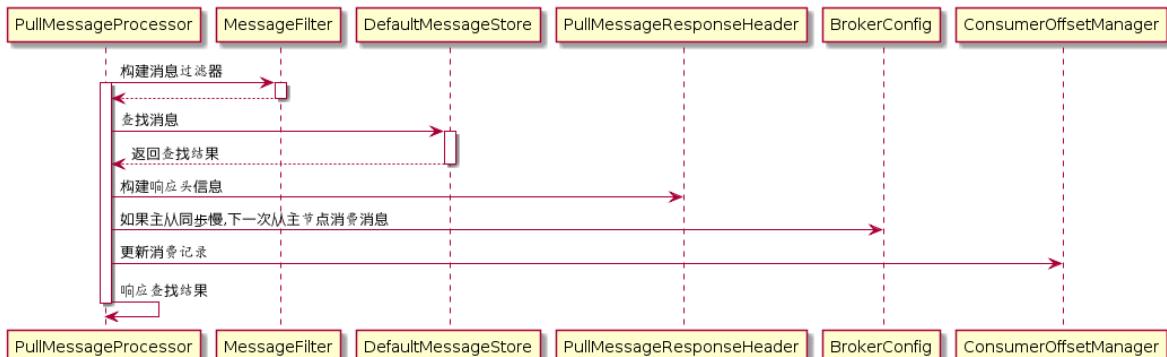
```
13 this.makeSureStateOK();
14 } catch (MQClientException e) {
15 log.warn("pullMessage exception, consumer state not ok", e);
16 this.executePullRequestLater(pullRequest,
17 PULL_TIME_DELAY_MILLS_WHEN_EXCEPTION);
18 return;
19 }
20 //如果处理队列被挂起,延迟1s后再执行
21 if (this.isPause()) {
22 log.warn("consumer was paused, execute pull request later.
23 instanceName={}, group={}", this.defaultMQPushConsumer.getInstanceName(),
24 this.defaultMQPushConsumer.getConsumerGroup());
25 this.executePullRequestLater(pullRequest,
26 PULL_TIME_DELAY_MILLS_WHEN_SUSPEND);
27 return;
28 }
29 //获得最大待处理消息数量
30 long cachedMessageCount = processQueue.getMsgCount().get();
31 //获得最大待处理消息大小
32 long cachedMessageSizeInMiB = processQueue.getMsgSize().get() / (1024 *
33 1024);
34 //从数量进行流控
35 if (cachedMessageCount >
36 this.defaultMQPushConsumer.getPullThresholdForQueue()) {
37 this.executePullRequestLater(pullRequest,
38 PULL_TIME_DELAY_MILLS_WHEN_FLOW_CONTROL);
39 if ((queueFlowControlTimes++ % 1000) == 0) {
40 log.warn(
41 "the cached message count exceeds the threshold {}, so do
42 flow control, minOffset={}, maxOffset={}, count={}, size={} MiB,
43 pullRequest={}, flowControlTimes={}",
44 this.defaultMQPushConsumer.getPullThresholdForQueue(),
45 processQueue.getMsgTreeMap().firstKey(),
46 processQueue.getMsgTreeMap().lastKey(), cachedMessageCount,
47 cachedMessageSizeInMiB, pullRequest, queueFlowControlTimes);
48 }
49 return;
50 }
51 //从消息大小进行流控
52 if (cachedMessageSizeInMiB >
53 this.defaultMQPushConsumer.getPullThresholdSizeForQueue()) {
54 this.executePullRequestLater(pullRequest,
55 PULL_TIME_DELAY_MILLS_WHEN_FLOW_CONTROL);
56 if ((queueFlowControlTimes++ % 1000) == 0) {
57 log.warn(
58 "the cached message size exceeds the threshold {} MiB, so
59 do flow control, minOffset={}, maxOffset={}, count={}, size={} MiB,
60 pullRequest={}, flowControlTimes={}",
61 this.defaultMQPushConsumer.getPullThresholdSizeForQueue(),
62 processQueue.getMsgTreeMap().firstKey(),
63 processQueue.getMsgTreeMap().lastKey(), cachedMessageCount,
64 cachedMessageSizeInMiB, pullRequest, queueFlowControlTimes);
65 }
66 return;
67 }
68 //获得订阅信息
```

```

50 final SubscriptionData subscriptionData =
51 this.rebalanceImpl.getSubscriptionInner().get(pullRequest.getMessageQueue()
52 .getTopic());
52 if (null == subscriptionData) {
53 this.executePullRequestLater(pullRequest,
54 PULL_TIME_DELAY_MILLS_WHEN_EXCEPTION);
55 log.warn("find the consumer's subscription failed, {}", pullRequest);
56 return;
57 }
58 //与服务端交互,获取消息
59 this.pullAPIWrapper.pullKernelImpl(
60 pullRequest.getMessageQueue(),
61 subExpression,
62 subscriptionData.getExpressionType(),
63 subscriptionData.getSubVersion(),
64 pullRequest.getNextOffset(),
65 this.defaultMQPushConsumer.getPullBatchSize(),
66 sysFlag,
67 commitOffsetValue,
68 BROKER_SUSPEND_MAX_TIME_MILLIS,
69 CONSUMER_TIMEOUT_MILLIS_WHEN_SUSPEND,
70 CommunicationMode.ASYNC,
71 pullCallback
72);
73 }

```

#### 5.5.4.3.2.消息服务端Broker组装消息



代码: *PullMessageProcessor#processRequest*

```

1 //构建消息过滤器
2 MessageFilter messageFilter;
3 if (this.brokerController.getBrokerConfig().isFilterSupportRetry()) {
4 messageFilter = new ExpressionForRetryMessageFilter(subscriptionData,
5 consumerFilterData,
6 this.brokerController.getConsumerFilterManager());
7 } else {
8 messageFilter = new ExpressionMessageFilter(subscriptionData,
9 consumerFilterData,
10 this.brokerController.getConsumerFilterManager());
11 }
12 //调用MessageStore.getMessage查找消息
13 final GetMessageResult getMessageResult =
14 this.brokerController.getMessageStore().getMessage(
15 requestHeader.getConsumerGroup(), //消费组名称
16 ...
17);

```

```

14 requestHeader.getTopic(), //主题名称
15 requestHeader.getQueueId(), //队列ID
16 requestHeader.getQueueOffset(), //待拉取偏移量
17 requestHeader.getMaxMsgNums(), //最大拉取消息条数
18 messageFilter //消息过滤器
19);

```

**代码: DefaultMessageStore#getMessage**

```

1 GetMessageStatus status = GetMessageStatus.NO_MESSAGE_IN_QUEUE;
2 long nextBeginOffset = offset; //查找下一次队列偏移量
3 long minOffset = 0; //当前消息队列最小偏移量
4 long maxOffset = 0; //当前消息队列最大偏移量
5 GetMessageResult getResult = new GetMessageResult();
6 final long maxOffsetPy = this.commitLog.getMaxOffset(); //当前commitLog最大偏移量
7 //根据主题名称和队列编号获取消息消费队列
8 ConsumeQueue consumeQueue = findConsumeQueue(topic, queueId);
9
10 ...
11 minOffset = consumeQueue.getMinOffsetInQueue();
12 maxOffset = consumeQueue.getMaxOffsetInQueue();
13 //消息偏移量异常情况校对下一次拉取偏移量
14 if (maxOffset == 0) { //表示当前消息队列中没有消息
15 status = GetMessageStatus.NO_MESSAGE_IN_QUEUE;
16 nextBeginOffset = nextOffsetCorrection(offset, 0);
17 } else if (offset < minOffset) { //待拉取消息的偏移量小于队列的其实偏移量
18 status = GetMessageStatus.OFFSET_TOO_SMALL;
19 nextBeginOffset = nextOffsetCorrection(offset, minOffset);
20 } else if (offset == maxOffset) { //待拉取偏移量为队列最大偏移量
21 status = GetMessageStatus.OFFSET_OVERFLOW_ONE;
22 nextBeginOffset = nextOffsetCorrection(offset, offset);
23 } else if (offset > maxOffset) { //偏移量越界
24 status = GetMessageStatus.OFFSET_OVERFLOW_BADLY;
25 if (0 == minOffset) {
26 nextBeginOffset = nextOffsetCorrection(offset, minOffset);
27 } else {
28 nextBeginOffset = nextOffsetCorrection(offset, maxOffset);
29 }
30 }
31 ...
32 //根据偏移量从CommitLog中拉取32条消息
33 SelectMappedBufferResult selectResult = this.commitLog.getMessage(offsetPy,
sizePy);

```

**代码: PullMessageProcessor#processRequest**

```

1 //根据拉取结果填充responseHeader
2 response.setRemark(getMessageResult.getStatus().name());
3 responseHeader.setNextBeginOffset(getMessageResult.getNextBeginOffset());
4 responseHeader.setMinOffset(getMessageResult.getMinOffset());
5 responseHeader.setMaxOffset(getMessageResult.getMaxOffset());
6
7 //判断如果存在主从同步慢,设置下一次拉取任务的ID为主节点
8 switch (this.brokerController.getMessageStoreConfig().getBrokerRole()) {
9 case ASYNC_MASTER:

```

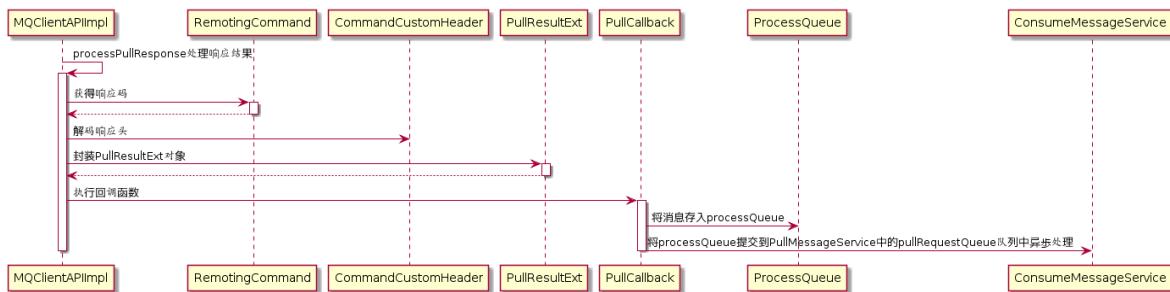
```
10 case SYNC_MASTER:
11 break;
12 case SLAVE:
13 if (!this.brokerController.getBrokerConfig().isSlaveReadEnable()) {
14 response.setCode(ResponseCode.PULL_RETRY_IMMEDIATELY);
15 responseHeader.setSuggestWhichBrokerId(MixAll.MASTER_ID);
16 }
17 break;
18 }
19 ...
20 //GetMessageResult与Response的Code转换
21 switch (getResultSet.getStatus()) {
22 case FOUND: //成功
23 response.setCode(ResponseCode.SUCCESS);
24 break;
25 case MESSAGE_WAS_REMOVING: //消息存放在下一个commitLog中
26 response.setCode(ResponseCode.PULL_RETRY_IMMEDIATELY); //消息重试
27 break;
28 case NO_MATCHED_LOGIC_QUEUE: //未找到队列
29 case NO_MESSAGE_IN_QUEUE: //队列中未包含消息
30 if (0 != requestHeader.getQueueOffset()) {
31 response.setCode(ResponseCode.PULL_OFFSET_MOVED);
32 requestHeader.getQueueOffset(),
33 getResultSet.getNextBeginOffset(),
34 requestHeader.getTopic(),
35 requestHeader.getQueueId(),
36 requestHeader.getConsumerGroup()
37);
38 } else {
39 response.setCode(ResponseCode.PULL_NOT_FOUND);
40 }
41 break;
42 case NO_MATCHED_MESSAGE: //未找到消息
43 response.setCode(ResponseCode.PULL_RETRY_IMMEDIATELY);
44 break;
45 case OFFSET_FOUND_NULL: //消息物理偏移量为空
46 response.setCode(ResponseCode.PULL_NOT_FOUND);
47 break;
48 case OFFSET_OVERFLOW_BADLY: //offset越界
49 response.setCode(ResponseCode.PULL_OFFSET_MOVED);
50 // XXX: warn and notify me
51 log.info("the request offset: {} over flow badly, broker max
52 offset: {}, consumer: {}",
53 requestHeader.getQueueOffset(),
54 getResultSet.getMaxOffset(), channel.remoteAddress());
55 break;
56 case OFFSET_OVERFLOW_ONE: //offset在队列中未找到
57 response.setCode(ResponseCode.PULL_NOT_FOUND);
58 break;
59 case OFFSET_TOO_SMALL: //offset未在队列中
60 response.setCode(ResponseCode.PULL_OFFSET_MOVED);
61 requestHeader.getConsumerGroup(),
62 requestHeader.getTopic(),
63 requestHeader.getQueueOffset(),
64 getResultSet.getMinOffset(), channel.remoteAddress());
65 break;
66 default:
67 assert false;
```

```

66 break;
67 }
68 ...
69 //如果CommitLog标记可用，并且当前Broker为主节点，则更新消息消费进度
70 boolean storeOffsetEnable = brokerAllowSuspend;
71 storeOffsetEnable = storeOffsetEnable && hasCommitoffsetFlag;
72 storeOffsetEnable = storeOffsetEnable
73 && this.brokerController.getMessageStoreConfig().getBrokerRole() != BrokerRole.SLAVE;
74 if (storeOffsetEnable) {
75
76 this.brokerController.getConsumerOffsetManager().commitoffset(RemotingHelper
77 .parseChannelRemoteAddr(channel),
78 requestHeader.getConsumerGroup(), requestHeader.getTopic(),
79 requestHeader.getQueueId(), requestHeader.getCommitoffset());
80 }

```

#### 5.5.4.3.3.消息拉取客户端处理消息



代码: *MQClientAPIMpl#processPullResponse*

```

1 private PullResult processPullResponse(
2 final RemotingCommand response) throws MQBrokerException,
3 RemotingCommandException {
4 PullStatus pullStatus = PullStatus.NO_NEW_MSG;
5 //判断响应结果
6 switch (response.getCode()) {
7 case ResponseCode.SUCCESS:
8 pullStatus = PullStatus.FOUND;
9 break;
10 case ResponseCode.PULL_NOT_FOUND:
11 pullStatus = PullStatus.NO_NEW_MSG;
12 break;
13 case ResponseCode.PULL_RETRY_IMMEDIATELY:
14 pullStatus = PullStatus.NO_MATCHED_MSG;
15 break;
16 case ResponseCode.PULL_OFFSET_MOVED:
17 pullStatus = PullStatus.OFFSET_ILLEGAL;
18 break;
19
20 default:
21 throw new MQBrokerException(response.getCode(),
22 response.getRemark());
23 }
24 //解码响应头
25 PullMessageResponseHeader responseHeader =
26 (PullMessageResponseHeader)
27 response.decodeCommandCustomHeader(PullMessageResponseHeader.class);
28 //封装PullResultExt返回

```

```

26 return new PullResultExt(pullStatus,
27 responseHeader.getNextBeginOffset(), responseHeader.getMinOffset(),
28 responseHeader.getMaxOffset(), null,
29 responseHeader.getSuggestWhichBrokerId(), response.getBody());
30 }

```

### PullResult类

```

1 private final PullStatus pullStatus; //拉取结果
2 private final long nextBeginOffset; //下次拉取偏移量
3 private final long minOffset; //消息队列最小偏移量
4 private final long maxOffset; //消息队列最大偏移量
5 private List<MessageExt> msgFoundList; //拉取的消息列表

```



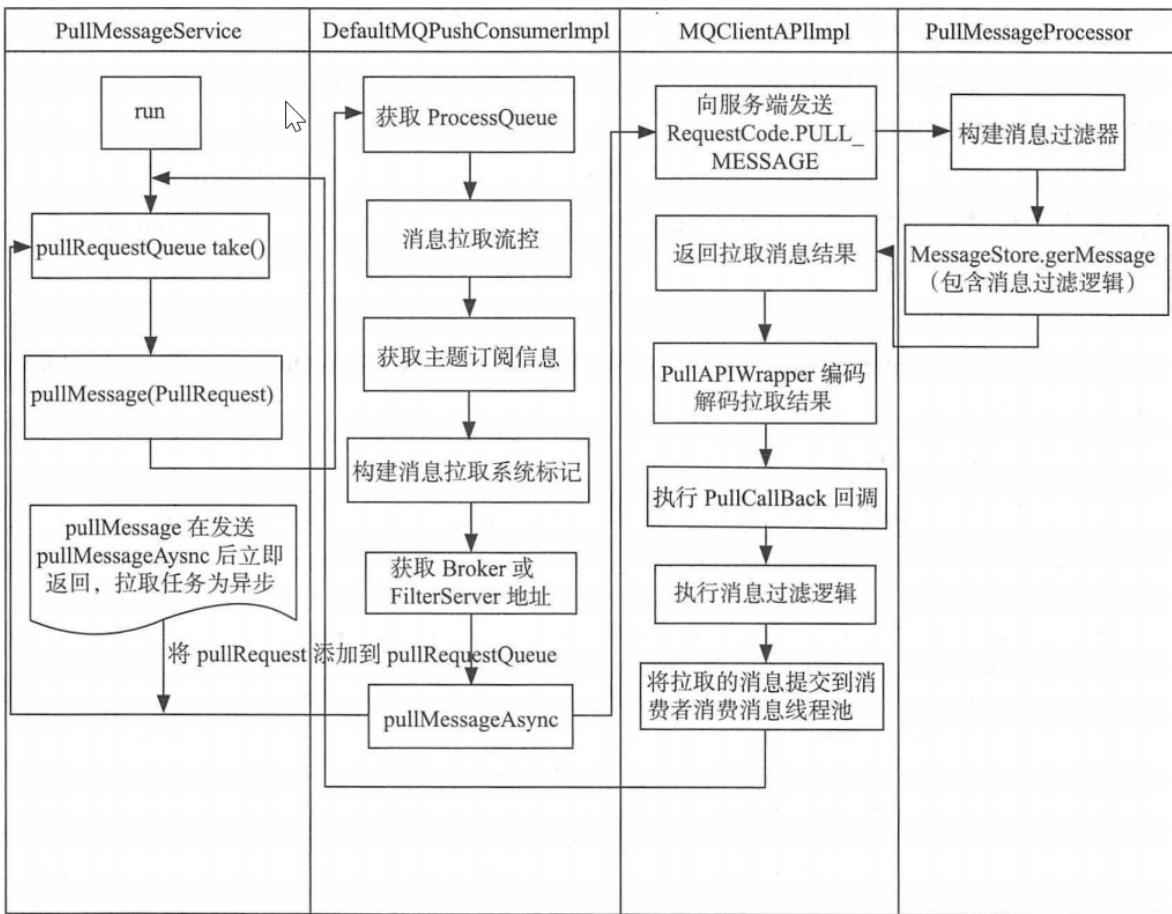
**代码: DefaultMQPushConsumerImpl\$PullCallback#OnSuccess**

```

1 //将拉取到的消息存入processQueue
2 boolean dispatchToConsume =
3 processQueue.putMessage(pullResult.getMsgFoundList());
4 //将processQueue提交到consumeMessageService中供消费者消费
5 DefaultMQPushConsumerImpl.this.consumeMessageService.submitConsumeRequest(
6 pullResult.getMsgFoundList(),
7 processQueue,
8 pullRequest.getMessageQueue(),
9 dispatchToConsume);
10 //如果pullInterval大于0，则等待pullInterval毫秒后将pullRequest对象放入到
11 //PullMessageService中的pullRequestQueue队列中
12 if (DefaultMQPushConsumerImpl.this.defaultMQPushConsumer.getPullInterval() > 0) {
13 DefaultMQPushConsumerImpl.this.executePullRequestLater(pullRequest,
14
15 DefaultMQPushConsumerImpl.this.defaultMQPushConsumer.getPullInterval());
16 } else {
17
18 DefaultMQPushConsumerImpl.this.executePullRequestImmediately(pullRequest);
19 }

```

#### 5.5.4.3.4.消息拉取总结



#### 5.5.4.4 消息拉取长轮询机制分析

RocketMQ未真正实现消息推模式，而是消费者主动向消息服务器拉取消息，RocketMQ推模式是循环向消息服务端发起消息拉取请求，如果消息消费者向RocketMQ拉取消息时，消息未到达消费队列时，如果不启用长轮询机制，则会在服务端等待shortPollingTimeMills时间后（挂起）再去判断消息是否已经到达指定消息队列，如果消息仍未到达则提示拉取消息客户端PULL—NOT—FOUND（消息不存在）；如果开启长轮询模式，RocketMQ一方面会每隔5s轮询检查一次消息是否可达，同时一有消息达到后立马通知挂起线程再次验证消息是否是自己感兴趣的消息，如果是则从CommitLog文件中提取消息返回给消息拉取客户端，否则直到挂起超时，超时时间由消息拉取方在消息拉取是封装在请求参数中，PUSH模式为15s，PULL模式通过DefaultMQPullConsumer#setBrokerSuspendMaxTimeMillis设置。RocketMQ通过在Broker客户端配置longPollingEnable为true来开启长轮询模式。

**代码: PullMessageProcessor#processRequest**

```

1 //当没有拉取到消息时，通过长轮询方式继续拉取消息
2 case ResponseCode.PULL_NOT_FOUND:
3 if (brokerAllowSuspend && hasSuspendFlag) {
4 long pollingTimeMills = suspendTimeoutMillisLong;
5 if (!this.brokerController.getBrokerConfig().isLongPollingEnable())
6 {
7 pollingTimeMills =
8 this.brokerController.getBrokerConfig().getShortPollingTimeMills();
9 }
10
11 String topic = requestHeader.getTopic();
12 long offset = requestHeader.getQueueOffset();
13 int queueId = requestHeader.getQueueId();
14 //构建拉取请求对象
15 PullRequest pullRequest = new PullRequest(request, channel,
16 pollingTimeMills,
17 ...
18);
19 ...
20 }
21 }

```

```

14 this.brokerController.getMessageStore().now(), offset,
15 subscriptionData, messageFilter);
16 //处理拉取请求
17
18 this.brokerController.getPullRequestHoldService().suspendPullRequest(topic
19 , queueId, pullRequest);
20 response = null;
21 break;
22 }

```

### PullRequestHoldService方式实现长轮询

**代码: PullRequestHoldService#suspendPullRequest**

```

1 //将拉取消息请求，放置在ManyPullRequest集合中
2 public void suspendPullRequest(final String topic, final int queueId, final
3 PullRequest pullRequest) {
4 String key = this.buildKey(topic, queueId);
5 ManyPullRequest mpr = this.pullRequestTable.get(key);
6 if (null == mpr) {
7 mpr = new ManyPullRequest();
8 ManyPullRequest prev = this.pullRequestTable.putIfAbsent(key, mpr);
9 if (prev != null) {
10 mpr = prev;
11 }
12 }
13 mpr.addPullRequest(pullRequest);
14 }

```

**代码: PullRequestHoldService#run**

```

1 public void run() {
2 log.info("{} service started", this.getServiceName());
3 while (!this.isStopped()) {
4 try {
5 //如果开启长轮询每隔5秒判断消息是否到达
6 if
7 (this.brokerController.getBrokerConfig().isLongPollingEnable()) {
8 this.waitForRunning(5 * 1000);
9 } else {
10 //没有开启长轮询，每隔1s再次尝试
11 }
12
13 long beginLockTimestamp = this.systemClock.now();
14 this.checkHoldRequest();
15 long costTime = this.systemClock.now() - beginLockTimestamp;
16 if (costTime > 5 * 1000) {
17 log.info("[NOTIFYME] check hold request cost {} ms.",
18 costTime);
19 }
20 } catch (Throwable e) {
21 log.warn(this.getServiceName() + " service has exception. ",
22 e);
23 }

```

```

21 }
22 }
23
24 log.info("{} service end", this.getServiceName());
25 }
```

**代码: PullRequestHoldService#checkHoldRequest**

```

1 //遍历拉取任务
2 private void checkHoldRequest() {
3 for (String key : this.pullRequestTable.keySet()) {
4 String[] kArray = key.split(TOPIC_QUEUEID_SEPARATOR);
5 if (2 == kArray.length) {
6 String topic = kArray[0];
7 int queueId = Integer.parseInt(kArray[1]);
8 //获得消息偏移量
9 final long offset =
this.brokerController.getMessageStore().getMaxOffsetInQueue(topic,
queueId);
10 try {
11 //通知有消息达到
12 this.notifyMessageArriving(topic, queueId, offset);
13 } catch (Throwable e) {
14 log.error("check hold request failed. topic={}, queueId=
{}", topic, queueId, e);
15 }
16 }
17 }
18 }
```

**代码: PullRequestHoldService#notifyMessageArriving**

```

1 //如果拉取消息偏移大于请求偏移量,如果消息匹配调用executeRequestWhenwakeup处理消息
2 if (newestOffset > request.getPullFromThisOffset()) {
3 boolean match =
request.getMessageFilter().isMatchedByConsumeQueue(tagsCode,
4 new ConsumeQueueExt.CqExtUnit(tagsCode, msgStoreTime,
filterBitMap));
5 // match by bit map, need eval again when properties is not null.
6 if (match && properties != null) {
7 match = request.getMessageFilter().isMatchedByCommitLog(null,
properties);
8 }
9
10 if (match) {
11 try {
12
this.brokerController.getPullMessageProcessor().executeRequestWhenwakeup(r
equest.getClientChannel(),
13 request.getRequestCommand());
14 } catch (Throwable e) {
15 log.error("execute request when wakeup failed.", e);
16 }
17 continue;
18 }
19 }
```

```

20 //如果过期时间超时，则不继续等待将直接返回给客户端消息未找到
21 if (System.currentTimeMillis() >= (request.getSuspendTimestamp() +
request.getTimeoutMillis())) {
22 try {
23
24 this.brokerController.getPullMessageProcessor().executeRequestWhenWakeup(r
equest.getClientChannel(),
25 request.getRequestCommand());
26 } catch (Throwable e) {
27 log.error("execute request when wakeup failed.", e);
28 }
29 continue;
}

```

如果开启了长轮询机制，PullRequestHoldService会每隔5s被唤醒去尝试检测是否有新的消息的到来才给客户端响应，或者直到超时才给客户端进行响应，消息实时性比较差，为了避免这种情况，RocketMQ引入另外一种机制：当消息到达时唤醒挂起线程触发一次检查。

### DefaultMessageStore\$ReputMessageService机制

**代码:** *DefaultMessageStore#start*

```

1 //长轮询入口
2 this.reputMessageService.setReputFromOffset(maxPhysicalPosInLogicQueue);
3 this.reputMessageService.start();

```

**代码:** *DefaultMessageStore\$ReputMessageService#run*

```

1 public void run() {
2 DefaultMessageStore.log.info(this.getServiceName() + " service
started");
3
4 while (!this.isStopped()) {
5 try {
6 Thread.sleep(1);
7 //长轮询核心逻辑代码入口
8 this.doReput();
9 } catch (Exception e) {
10 DefaultMessageStore.log.warn(this.getServiceName() + " service
has exception. ", e);
11 }
12 }
13
14 DefaultMessageStore.log.info(this.getServiceName() + " service end");
15 }

```

**代码:** *DefaultMessageStore\$ReputMessageService#deReput*

```
1 //当新消息达到是,进行通知监听器进行处理
2 if (BrokerRole.SLAVE != DefaultMessageStore.this.getMessageStoreConfig().getBrokerRole()
3 && DefaultMessageStore.this.brokerConfig.isLongPollingEnable()) {
4
5 DefaultMessageStore.this.messageArrivingListener.arriving(dispatchRequest.getTopic(),
6 dispatchRequest.getQueueId(), dispatchRequest.getConsumeQueueOffset()
7 + 1,
8 dispatchRequest.getTagsCode(), dispatchRequest.getStoreTimestamp(),
9 dispatchRequest.getBitMap(), dispatchRequest.getPropertiesMap());
10 }
```

**代码: NotifyMessageArrivingListener#arriving**

```
1 public void arriving(String topic, int queueId, long logicOffset, long
2 tagsCode,
3 long msgStoreTime, byte[] filterBitMap, Map<String, String> properties) {
4 this.pullRequestHoldService.notifyMessageArriving(topic, queueId,
5 logicOffset, tagsCode,
6 msgStoreTime, filterBitMap, properties);
7 }
```

## 5.5.5 消息队列负载与重新分布机制

RocketMQ消息队列重新分配是由RebalanceService线程来实现。一个MQClientInstance持有一个RebalanceService实现，并随着MQClientInstance的启动而启动。

**代码: RebalanceService#run**

```
1 public void run() {
2 log.info(this.getServiceName() + " service started");
3 //RebalanceService线程默认每隔20s执行一次mqClientFactory.doRebalance方法
4 while (!this.isStopped()) {
5 this.waitForRunning(waitInterval);
6 this.mqClientFactory.doRebalance();
7 }
8
9 log.info(this.getServiceName() + " service end");
10 }
```

**代码: MQClientInstance#doRebalance**

```

1 public void doRebalance() {
2 //MQClientInstance遍历以注册的消费者,对消费者执行doRebalance()方法
3 for (Map.Entry<String, MQConsumerInner> entry :
4 this.consumerTable.entrySet()) {
5 MQConsumerInner impl = entry.getValue();
6 if (impl != null) {
7 try {
8 impl.doRebalance();
9 } catch (Throwable e) {
10 log.error("doRebalance exception", e);
11 }
12 }
13 }

```

**代码: RebalanceImpl#doRebalance**

```

1 //遍历订阅消息对每个主题的订阅的队列进行重新负载
2 public void doRebalance(final boolean isOrder) {
3 Map<String, SubscriptionData> subTable = this.getSubscriptionInner();
4 if (subTable != null) {
5 for (final Map.Entry<String, SubscriptionData> entry :
6 subTable.entrySet()) {
7 final String topic = entry.getKey();
8 try {
9 this.rebalanceByTopic(topic, isOrder);
10 } catch (Throwable e) {
11 if (!topic.startsWith(MixAll.RETRY_GROUP_TOPIC_PREFIX)) {
12 log.warn("rebalanceByTopic Exception", e);
13 }
14 }
15 }
16 }
17 this.truncateMessageQueueNotMyTopic();
18 }

```

**代码: RebalanceImpl#rebalanceByTopic**

```

1 //从主题订阅消息缓存表中获取主题的队列信息
2 Set<MessageQueue> mqSet = this.topicSubscribeInfoTable.get(topic);
3 //查找该主题订阅组所有的消费者ID
4 List<String> cidAll = this.mqClientFactory.findConsumerIdList(topic,
5 consumerGroup);
6
7 //给消费者重新分配队列
8 if (mqSet != null && cidAll != null) {
9 List<MessageQueue> mqAll = new ArrayList<MessageQueue>();
10 mqAll.addAll(mqSet);
11
12 Collections.sort(mqAll);
13 Collections.sort(cidAll);
14
15 AllocateMessageQueueStrategy strategy =
this.allocateMessageQueueStrategy;

```

```

16 List<MessageQueue> allocateResult = null;
17 try {
18 allocateResult = strategy.allocate(
19 this.consumerGroup,
20 this.mQClientFactory.getClientId(),
21 mqAll,
22 cidAll);
23 } catch (Throwable e) {
24 log.error("AllocateMessageQueueStrategy.allocate Exception.
allocateMessageQueueStrategyName={}", strategy.getName(),
25 e);
26 return;
27 }

```

RocketMQ默认提供5中负载均衡分配算法

```

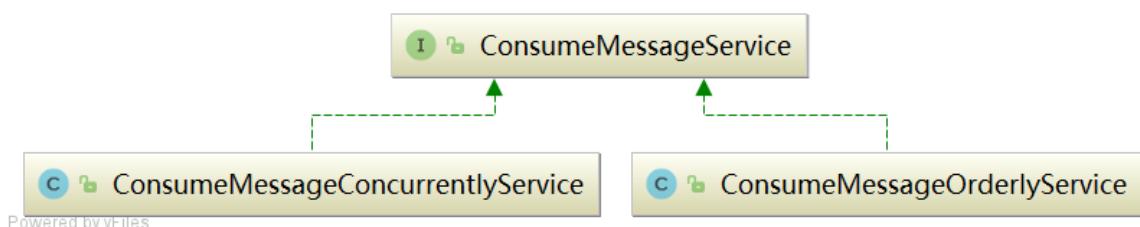
1 AllocateMessageQueueAveragely:平均分配
2 举例:8个队列q1,q2,q3,q4,q5,a6,q7,q8,消费者3个:c1,c2,c3
3 分配如下:
4 c1:q1,q2,q3
5 c2:q4,q5,a6
6 c3:q7,q8
7 AllocateMessageQueueAveragelyByCircle:平均轮询分配
8 举例:8个队列q1,q2,q3,q4,q5,a6,q7,q8,消费者3个:c1,c2,c3
9 分配如下:
10 c1:q1,q4,q7
11 c2:q2,q5,a8
12 c3:q3,q6

```

注意：消息队列的分配遵循一个消费者可以分配到多个队列，但同一个消息队列只会分配给一个消费者，故如果出现消费者个数大于消息队列数量，则有些消费者无法消费消息。

## 5.5.6 消息消费过程

PullMessageService负责对消息队列进行消息拉取，从远端服务器拉取消息后将消息存储 ProcessQueue消息队列处理队列中，然后调用ConsumeMessageService#submitConsumeRequest方法进行消息消费，使用线程池来消费消息，确保了消息拉取与消息消费的解耦。 ConsumeMessageService支持顺序消息和并发消息，核心类图如下：



### 并发消息消费

代码: *ConsumeMessageConcurrentlyService#submitConsumeRequest*

```

1 //消息批次单次
2 final int consumeBatchsize =
3 this.defaultMQPushConsumer.getConsumeMessageBatchMaxSize();
4 //msgs.size()默认最多为32条。
//如果msgs.size()小于consumeBatchsize，则直接将拉取到的消息放入到consumeRequest，然后将consumeRequest提交到消费者线程池中

```

```

5 if (msgs.size() <= consumeBatchSize) {
6 ConsumeRequest consumeRequest = new ConsumeRequest(msgs, processQueue,
7 messageQueue);
8 try {
9 this.consumeExecutor.submit(consumeRequest);
10 } catch (RejectedExecutionException e) {
11 this.submitConsumeRequestLater(consumeRequest);
12 }
13 }else{ //如果拉取的消息条数大于consumeBatchsize,则对拉取消息进行分页
14 for (int total = 0; total < msgs.size();) {
15 List<MessageExt> msgThis = new ArrayList<MessageExt>
16 (consumeBatchSize);
17 for (int i = 0; i < consumeBatchSize; i++, total++) {
18 if (total < msgs.size()) {
19 msgThis.add(msgs.get(total));
20 } else {
21 break;
22 }
23
24 ConsumeRequest consumeRequest = new ConsumeRequest(msgThis,
25 processQueue, messageQueue);
26 try {
27 this.consumeExecutor.submit(consumeRequest);
28 } catch (RejectedExecutionException e) {
29 for (; total < msgs.size(); total++) {
30 msgThis.add(msgs.get(total));
31
32 this.submitConsumeRequestLater(consumeRequest);
33 }
34 }
35 }
```

**代码: ConsumeMessageConcurrentlyService\$ConsumeRequest#run**

```

1 //检查processQueue的dropped,如果为true,则停止该队列消费。
2 if (this.processQueue.isDropped()) {
3 log.info("the message queue not be able to consume, because it's
4 dropped. group={} {}",
5 ConsumeMessageConcurrentlyService.this.consumerGroup, this.messageQueue);
6 return;
7 }
8 ...
9 //执行消息处理的钩子函数
10 if
11 (ConsumeMessageConcurrentlyService.this.defaultMQPushConsumerImpl.hasHook()
12) {
13 consumeMessageContext = new ConsumeMessageContext();
14
15 consumeMessageContext.setNamespace(defaultMQPushConsumer.getNamespace());
16
17 consumeMessageContext.setConsumerGroup(defaultMQPushConsumer.getConsumerGr
18 oup());
19 consumeMessageContext.setProps(new HashMap<String, String>());
20 consumeMessageContext.setMq(messageQueue);
21 consumeMessageContext.setMsgList(msgs);
22 consumeMessageContext.setSuccess(false);
23 }
```

```

17 ConsumeMessageConcurrentlyService.this.defaultMQPushConsumerImpl.executeHo
okBefore(consumeMessageContext);
18 }
19 ...
20 //调用应用程序消息监听器的consumeMessage方法,进入到具体的消息消费业务处理逻辑
21 status = listener.consumeMessage(collections.unmodifiableList(msgs),
context);
22
23 //执行消息处理后的钩子函数
24 if
25 (ConsumeMessageConcurrentlyService.this.defaultMQPushConsumerImpl.hasHook())
{
26 consumeMessageContext.setStatus(status.toString());
27
28 consumeMessageContext.setSuccess(ConsumeConcurrentlyStatus.CONSUME_SUCCESS
== status);
29
30 ConsumeMessageConcurrentlyService.this.defaultMQPushConsumerImpl.executeHo
okAfter(consumeMessageContext);
31 }

```

## 5.5.7 定时消息机制

定时消息是消息发送到Broker后，并不立即被消费者消费而是要等到特定的时间后才能被消费，RocketMQ并不支持任意的时间精度，如果要支持任意时间精度定时调度，不可避免地需要在Broker层做消息排序，再加上持久化方面的考量，将不可避免的带来巨大的性能消耗，所以RocketMQ只支持特定级别的延迟消息。消息延迟级别在Broker端通过messageDelayLevel配置，默认为“1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m 30m 1h 2h”，delayLevel=1表示延迟消息1s,delayLevel=2表示延迟5s,依次类推。

RocketMQ定时消息实现类为ScheduleMessageService，该类在DefaultMessageStore中创建。通过在DefaultMessageStore中调用load方法加载该类并调用start方法启动。

### 代码: ScheduleMessageService#load

```

1 //加载延迟消息消费进度的加载与delayLevelTable的构造。延迟消息的进度默认存储路径
2 为/store/config/delayoffset.json
3 public boolean load() {
4 boolean result = super.load();
5 result = result && this.parseDelayLevel();
6 return result;
7 }

```

### 代码: ScheduleMessageService#start

```

1 //遍历延迟队列创建定时任务,遍历延迟级别,根据延迟级别level从offsetTable中获取消费队列的
2 消费进度。如果不存在,则使用0
3 for (Map.Entry<Integer, Long> entry : this.delayLevelTable.entrySet()) {
4 Integer level = entry.getKey();
5 Long timeDelay = entry.getValue();
6 Long offset = this.offsetTable.get(level);
7 if (null == offset) {
8 offset = 0L;
9 }

```

```

10 if (timeDelay != null) {
11 this.timer.schedule(new DeliverDelayedMessageTimerTask(level,
12 offset), FIRST_DELAY_TIME);
13 }
14
15 //每隔10s持久化一次延迟队列的消息消费进度
16 this.timer.scheduleAtFixedRate(new TimerTask() {
17
18 @Override
19 public void run() {
20 try {
21 if (started.get()) ScheduleMessageService.this.persist();
22 } catch (Throwable e) {
23 log.error("scheduleAtFixedRate flush exception", e);
24 }
25 }
26 }, 10000,
27 this.defaultMessageStore.getMessageStoreConfig().getFlushDelayOffsetInterval());

```

## 调度机制

ScheduleMessageService的start方法启动后，会为每一个延迟级别创建一个调度任务，每一个延迟级别对应SCHEDULE\_TOPIC\_XXXX主题下的一个消息消费队列。定时调度任务的实现类为DeliverDelayedMessageTimerTask，核心实现方法为executeOnTimeup

**代码: ScheduleMessageService\$DeliverDelayedMessageTimerTask#executeOnTimeup**

```

1 //根据队列ID与延迟主题查找消息消费队列
2 ConsumeQueue cq =
3
4 ScheduleMessageService.this.defaultMessageStore.findConsumeQueue(SCHEDULE_
5 TOPIC,
6 delayLevel2QueueId(delayLevel));
7 ...
8
9 //根据偏移量从消息消费队列中获取当前队列中所有有效的消息
10 SelectMappedBufferResult bufferCQ = cq.getIndexBuffer(this.offset);
11 ...
12 //遍历ConsumeQueue，解析消息队列中消息
13 for (; i < bufferCQ.getSize(); i += ConsumeQueue.CQ_STORE_UNIT_SIZE) {
14 long offsetPy = bufferCQ.getByteBuffer().getLong();
15 int sizePy = bufferCQ.getByteBuffer().getInt();
16 long tagsCode = bufferCQ.getByteBuffer().getLong();
17
18 if (cq.isExtAddr(tagsCode)) {
19 if (cq.getExt(tagsCode, cqExtUnit)) {
20 tagsCode = cqExtUnit.getTagsCode();
21 } else {
22 //can't find ext content.So re compute tags code.
23 log.error("[BUG] can't find consume queue extend file
24 content!addr={}, offsetPy={}, sizePy={}",
25 tagsCode, offsetPy, sizePy);
26 }
27 }
28
29 defaultMessageStore.getCommitLog().pickupStoreTimestamp(offsetPy, sizePy);

```

```

24 tagsCode = computeDeliverTimestamp(delayLevel, msgStoreTime);
25 }
26 }
27
28 long now = System.currentTimeMillis();
29 long deliverTimestamp = this.correctDeliverTimestamp(now, tagsCode);
30
31 ...
32 //根据消息偏移量与消息大小,从CommitLog中查找消息.
33 MessageExt msgExt =
34 ScheduleMessageService.this.defaultMessageStore.lookMessageByOffset(
35 offsetPy, sizePy);
36 }
```

## 5.5.8 顺序消息

顺序消息实现类是  
org.apache.rocketmq.client.impl.consumer.ConsumeMessageOrderlyService

**代码: ConsumeMessageOrderlyService#start**

```

1 public void start() {
2 //如果消息模式为集群模式, 启动定时任务, 默认每隔20s执行一次锁定分配给自己的消息消费队
列
3 if
4 (MessageMode1.CLUSTERING.equals(ConsumeMessageOrderlyService.this.defaultMQ
PushConsumerImpl.messageMode1())) {
5 this.scheduledExecutorService.scheduleAtFixedRate(new Runnable() {
6 @Override
7 public void run() {
8 ConsumeMessageOrderlyService.this.lockMQPeriodically();
9 }
10 }, 1000 * 1, ProcessQueue.REBALANCE_LOCK_INTERVAL,
11 TimeUnit.MILLISECONDS);
12 }
13 }
```

**代码: ConsumeMessageOrderlyService#submitConsumeRequest**

```

1 //构建消息任务,并提交消费线程池中
2 public void submitConsumeRequest(
3 final List<MessageExt> msgs,
4 final ProcessQueue processQueue,
5 final MessageQueue messageQueue,
6 final boolean dispathToConsume) {
7 if (dispathToConsume) {
8 ConsumeRequest consumeRequest = new ConsumeRequest(processQueue,
9 messageQueue);
10 this.consumeExecutor.submit(consumeRequest);
11 }
12 }
```

**代码: ConsumeMessageOrderlyService\$ConsumeRequest#run**

```
1 //如果消息队列为丢弃，则停止本次消费任务
2 if (this.processQueue.isDropped()) {
3 log.warn("run, the message queue not be able to consume, because it's
dropped. {}", this.messageQueue);
4 return;
5 }
6 //从消息队列中获取一个对象。然后消费消息时先申请独占objLock锁。顺序消息一个消息消费队列同一时刻只会被一个消费线程池处理
7 final Object objLock = messageQueueLock.fetchLockObject(this.messageQueue);
8 synchronized (objLock) {
9 ...
10 }
```

## 5.5.9 小结

RocketMQ消息消费方式分别为集群模式、广播模式。

消息队列负载由RebalanceService线程默认每隔20s进行一次消息队列负载，根据当前消费者组内消费者个数与主题队列数量按照某一种负载算法进行队列分配，分配原则为同一个消费者可以分配多个消息消费队列，同一个消息消费队列同一个时间只会分配给一个消费者。

消息拉取由PullMessageService线程根据RebalanceService线程创建的拉取任务进行拉取，默认每次拉取32条消息，提交给消费者消费线程后继续下一次消息拉取。如果消息消费过慢产生消息堆积会触发消息消费拉取流控。

并发消息消费指消费线程池中的线程可以并发对同一个消息队列的消息进行消费，消费成功后，取出消息队列中最小的消息偏移量作为消息消费进度偏移量存储在消息消费进度存储文件中，集群模式消息消费进度存储在Broker（消息服务器），广播模式消息消费进度存储在消费者端。

RocketMQ不支持任意精度的定时调度消息，只支持自定义的消息延迟级别，例如1s、2s、5s等，可通过在broker配置文件中设置messageDelayLevel。

顺序消息一般使用集群模式，是指对消息消费者内的线程池中的线程对消息消费队列只能串行消费。并并发消息消费最本质的区别是消息消费时必须成功锁定消息消费队列，在Broker端会存储消息消费队列的锁占用情况。