阿里云开发者社区 ALIBABA CLOUD DEVELOPER COMMUNITY | 阿里巴巴云原生 | RocketMQ

Apache RocketMQ 从入门到实战

藏经阁

阿里云开发者电子书系列

扫一扫加入作者公众号
中间件兴趣圈

扫一扫关注
RocketMQ 官微

扫一扫关注【阿里巴巴云原生】公众号
获取第一手技术干货

阿里云开发者“藏经阁”
海量免费电子书下载

作者简介

作者简介

丁威，《 RocketMQ 技术内幕 》作者，RocketMQ 官方社区优秀布道师，荣获 CSDN2020 博客之星亚军；担任中通快递研发中心资深架构师，维护『 中间件兴趣圈 』公众号，主打成体系剖析 Java 主流中间件，尝试从源码分析、架构设计、实战、故障分析等维度深刻揭晓中间件技术，已覆盖 RocketMQ、Dubbo、Sentienl、Kafka、Canal、MyCat、ElasticJob、ElasticSearch 等。

推荐人及推荐序

推荐人

杜恒，Apache RocketMQ PMC Member/committer，Linux OpenMessaging TSC Member，目前负责 RocketMQ 专有云商业化以及开源技术生态构建。具有多年分布式系统、中间件研究及工程经验。目前对分布式中间件、K8s、微服务、物联网、Serverless 感兴趣。

推荐序

Apache RocketMQ 作为一款高吞吐，抗万亿消息堆积的云原生消息平台，目前已经被国内 75% 以上互联网、金融等公司所采用，逐渐成为企业 IT 架构的核心基础设施。

丁威老师作为资深架构师，在分布式架构、存储方面功底深厚，目前在企业内部负责着日均千亿级消息流转的 RocketMQ 集群。本书不仅由浅入深的介绍了 RocketMQ 的架构与实现，而且包含了多年线上超大规模集群开发运维经验的总结，通过本书不仅能够掌握分布式消息平台的设计原理，对线上疑难问题排查分析、性能调优与架构设计也大有帮助。

目录

开篇：我的另一种参与 RocketMQ 开源社区的方式

很荣幸在 2019 年获得了 RocketMQ 开源社区的授予我优秀布道师荣誉称号。

说到参与开源项目，很多人都理解为成为一名 Committer 才能算式参与到开源社区的建设？但其实这个就是参与开源项目有代码层面的贡献，也有非代码贡献层面的如技术布道、社区运营（线上直播、线下活动、文档编辑）等。如何参与一个开源项目，容我慢慢道来。

一、与 RocketMQ 相识、相知到“在一起”

在 2017 年听到阿里巴巴将 RocketMQ 捐赠给 Apache 基金会成为 Apache 的顶级项目，我内心是无比激动，因为终于可以一睹一款高性能的消息中间件的实现原理。

通过阅读了 RocketMQ 官方，以下几个特别的点更是吸引了我的注意，让我下定决心深入研究一番。

- RocketMQ 为什么性能高效，到底运用了什么“厉害”的技术?
- RocketMQ 如何实现刷盘（可以类比一下数据库方面的刷盘、redo、undo 日志）?
- RocketMQ 文件存储设计理念、基于文件的 Hash 索引是怎么实现的?
- 定时消息、消息过滤等实现原理。
- 如何进行网络编程（Netty 实战）?

下定决心后便开始了我的源码分析 RocketMQ 之旅，大概在 4 个多月的时间中连续发表了 30 余篇文章，从 Nameserver、消息发送高可用设计、消息存储、消息消费、消息过滤、事务消息等各个方面对其进行了体系化的剖析，边写边分享，边分享边传播，终于得到了机械工业出版社华章分社的杨福川老师的认可，邀请我出书。

在杨老师和张工的帮助与指点下，经过将近半年的努力，书稿基本完稿。由于我当时是一位名不经传的新人，按照出版行业的惯例，需要找一些该领域内专家大牛帮忙做序或写写推荐语。当时我也是初生牛犊不怕虎，蹦出了一个非常大胆的想法，是不是可以联系到 RocketMQ 官方的一些大佬，最终我直接锁定了 RocketMQ 创始人冯嘉大神，希望他能帮我作序推荐，令人惊喜的是冯嘉大神非常平易见人，得知我的来意后，他说了这样一句话：“我是非常愿意为写书的朋友作序，但需要评估一下书稿的质量，如果质量 OK，非常愿意效劳”。我备受鼓舞，在和出版社初步沟通后，将试读稿件再加上消息存储整章的内容发给冯嘉大神后，经冯嘉大神认真审稿后，决定帮忙推荐作序，真的非常受鼓舞。

随着《RocketMQ 技术内幕》一书的正式出版上市，并得到广大读者朋友的认可，与官方的联系也越来越多，后面在 RocketMQ 中国社区负责人青峰大佬的筹备下，我还参与了 RocketMQ 官方社区的源码解析直播活动、官方文档审稿等工作，并在社区得到了不错的反响。

说到这里大家是不是觉得非常奇怪，是不是都认为你只是在写文章，写书，没有真正参与开源社区呀，没有贡献代码，这个算哪门子参与开源社区?

其实我一开始连我自己也没有意识到我正在参与一个开源项目，直到我在冯嘉大神为我写的序言中给了我一个新的称号：RocketMQ 布道师，从而才真正了解到参与开源的另外一种方式：做一个开源项目的传播者，让更多人更容易的应用它，即降低大众对它的使用门槛。

有了新的称号，那就得更加努力，朝着优秀努力，在 2019 年我又陆续发表了 20 几篇关于 RocketMQ 相关的文章，这些文章含金量极高，不仅及时跟进了 RocketMQ4.3.0 之后的新特性：消息轨迹、ACL、主从切换等机制，更是发表了数篇实战类文章，详细指出在生产环境下一些使用误区，更是输出了几篇生产环境真实故障与解决方案。最终于 2019 年 RocketMQ 官方社区授予我优秀布道师荣誉称号。

RocketMQ 成就了我，我也会继续努力，为传播 RocketMQ 尽一份力所能及的力量。2020 年，继续努力。

二、如何成为开源项目的 Committer

有一些粉丝在问我，您对 RocketMQ 研究的这么深入，为什么不考虑贡献代码，成为一名 Committer 呢？这是因为参与开源项目需要具备一些基本条件，当下我的实际情况不符合，那成为一个开源项目的 Committer 有些什么条件呢？

1. 扎实的 Java 基础功底

一个开源项目的底层都会涉及到存储，这就要求具备一定的数据结构基础，JAVA 集合框架中的类自然成为了我们突破数据结构最好的老师，其次是 java 并发，即多线程、并发容器、锁等课题，这方面可以好好学习一下 JUC 框架。最后最好是具备一些网络方面的知识，例如 NIO、Netty。

2. 持续输出能力

成为一个开源项目的 contributions 非常容易，提交一个 PR 并被通过即可，甚至于提交一个文档被接受也同样可以，难的是持续贡献，最终被开源项目的 PMC 认为对该项目有着突出贡献。

我比较“苦逼”，在带娃方面我的资源只有我老婆，父母在老家无法分身，故下班后我没有连续的空闲时间专心投入一项任务中，而开源最需要的是精益求精，不只是需要完成功能，而是要编写结构优良的代码，设计所占据的时间比代码开发时间要多的多，故我个人认为我暂时不方便走代码贡献这条道路。但我零碎时间还是充足的，故现阶段我会好好利用这些零碎时间，继续通过写文章的方式为开源项目贡献自己的一份力量。

接下来我们回到本节的主题，那如何参与一个开源项目呢?

在参与一个开源项目之前，我觉得第一个最基本的步骤还是要打牢基础，这里的基础至少要包括 JAVA 集合、JAVA 并发（JUC)这两项，只是最最基本的，至少要阅读其源码，理解其设计理念，至于 NIO，Netty 这些可以后续在需要使用时再去专门学习，有针对性的学习，有使用需求，或许学习动力更强劲，学习效率更高效。

当具备一定的基础后，如何从零开始参与进开源项目呢？通常有如下几个方法：

- 看看官方文档，特别是设计手册，从整体上把握其设计理念。
- 写写源码分析类文章，从整体上把控这个框架，这个花费时间较多，如果框架正在起步。阶段，不建议该方法；如果框架比较成熟，非常建议采用该方法。
- 尝试看看开源项目中的 issues，看能不能解决，从问题入手，快速融入该项目。
- 尝试谢谢单元测试用例，测试驱动开发，借此学习该框架。

后面的事情就是坚持不懈，朝着目标不断前进，中途可以放慢速度，但千万别放弃，因为只有坚持，才能胜利，只要前进，就离目标更近。

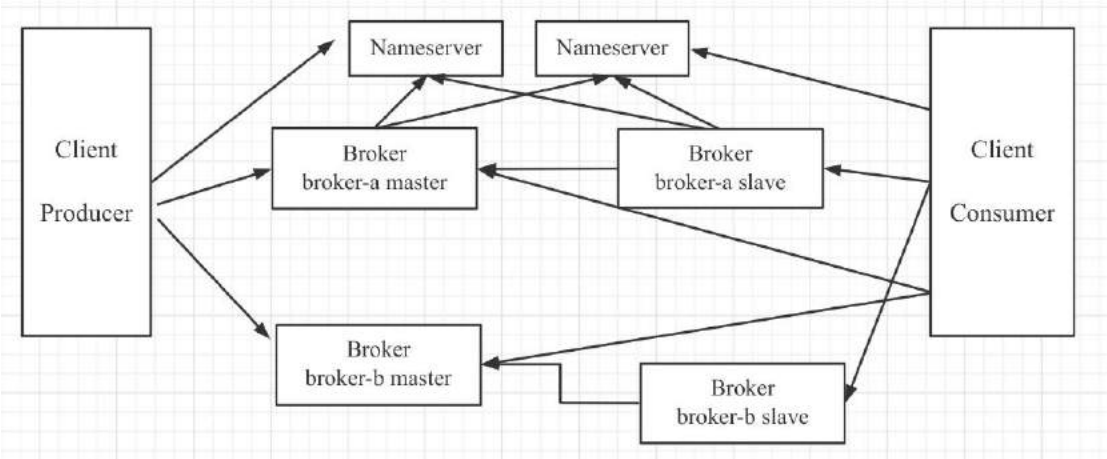
参与开源，一个最基本的条件是拥有大量的连续时间，想要成为一个开源框架的 Committer ,唯有坚持不懈，持续投入，持续产出。

最后再次感谢 RocketMQ 社区对我的认可，我会尽努力做出更大的贡献，也希望广大读者朋友们，积极参与开源社区，贡献一份自己的力量，同事打造自身影响力，助力职场步步高升。

1.1 RocketMQ 核心概念扫盲篇

在正式进入 RocketMQ 的学习之前，我觉得有必要梳理一下 RocketMQ 核心概念，为大家学习 RocketMQ 打下牢固的基础。

一、RocketMQ 部署架构



在 RocketMQ 主要的组件如下：

1. Nameserver

Nameserver 集群，topic 的路由注册中心，为客户端根据 Topic 提供路由服务，从而引导客户端向 Broker 发送消息。Nameserver 之间的节点不通信。路由信息在 Nameserver 集群中数据一致性采取的最终一致性。

2. Broker

消息存储服务器，分为两种角色：Master 与 Slave，上图中呈现的就是 2 主 2 从的部署架构，在 RocketMQ 中，主服务承担读写操作，从服务器作为一个备份，当主服务器存在压力时，从服务器可以承担读服务（消息消费）。所有 Broker，包含 Slave 服务器每隔 30s 会向 Nameserver 发送心跳包，心跳包中会包含存在在 Broker 上所有的 topic 的路由信息。

3. Client

消息客户端，包括 Producer(消息发送者)和 Consumer(消费消费者). 客户端在同一时间只会连接一台 nameserver，只有在连接出现异常时才会向尝试连接另外一台。客户端每隔 30s 向 Nameserver 发起 topic 的路由信息查询。

温馨提示：Nameserver 是在内存中存储 Topic 的路由信息，持久化 Topic 路由信息的地方是在 Broker 中，即${ ROCKETMQ_HOME}/store/config/topics.json。

在 RocketMQ4.5.0 版本后引入了多副本机制，即一个复制组（m-s）可以演变为基于 raft 协议的复制组，复制组内部使用 raft 协议保证 broker 节点数据的强一致性，该部署架构在金融行业用的比较多。

二、消息订阅模型

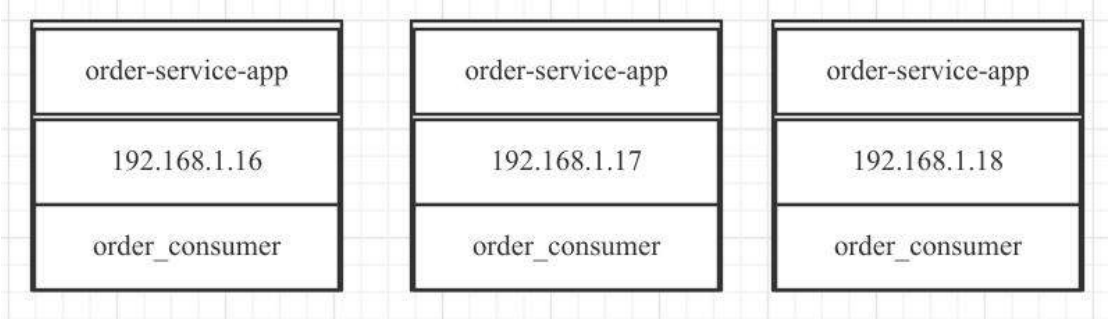
在 RocketMQ 的消息消费模式采用的是发布与订阅模式。

topic：一类消息的集合，消息发送者将一类消息发送到一个主题中，例如订单模块将订单发送到 order_topic 中，而用户登录时，将登录事件发送到 user_login_topic 中。

consumegroup：消息消费组，一个消费单位的“群体”，消费组首先在启动时需要订阅需要消费的 topic。一个 topic 可以被多个消费组订阅，同样一个消费组也可以订阅多个主题。一个消费组拥有多个消费者。

术语解释起来有点枯燥晦涩，接下来我举例来阐述。

例如我们在开发一个订单系统，其中有一个子系统：order-service-app，在该项目中会创建一个消费组 order_consumer 来订阅 order_topic，并且基于分布式部署，order-service-app 的部署情况如下：



即 order-service-app 部署了 3 台服务器，每一个 jvm 进程可以看做是消费组 order_consumer 消费组的其中一个消费者。

1. 消费模式

那这三个消费者如何来分工来共同消费 order_topic 中的消息呢?

在 RocketMQ 中支持广播模式与集群模式。

广播模式：一个消费组内的所有消费者每一个都会处理 topic 中的每一条消息，通常用于刷新内存缓存。

集群模式：一个消费组内的所有消费者共同消费一个 topic 中的消息，即分工协作，一个消费者消费一部分数据，启动负载均衡，

集群模式是非常普遍的模式，符合分布式架构的基本理念，即横向扩容，当前消费者如果无法快速及时处理消息时，可以通过增加消费者的个数横向扩容，快速提高消费能力，及时处理挤压的消息。

2. 消费队列负载算法与重平衡机制

那集群模式下，消费者是如何来分配消息的呢?

例如上面实例中 order_topic 有 16 个队列，那一个拥有 3 个消费者的消费组如何来分配队列中。

在 MQ 领域有一个不成文的约定：同一个消费者同一时间可以分配多个队列，但一个队列同一时间只会分配给一个消费者。

RocketMQ 提供了众多的队列负载算法，其中最常用的两种平均分配算法。

- AllocateMessageQueueAveragely
  - 平均分配

- AllocateMessageQueueAveragelyByCircle
  - 轮流平均分配

为了说明这两种分配算法的分配规则，现在对 16 个队列，进行编号，用 q0~q15 表示，消费者用 c0～c2 表示。

AllocateMessageQueueAveragely 分配算法的队列负载机制如下：

c0：q0 q1 q2 q3 q4 q5

c1: q6 q7 q8 q9 q10

c2: q11 q12 q13 q14 q15

其算法的特点是用总数除以消费者个数，余数按消费者顺序分配给消费者，故 c0 会多分配一个队列，而且队列分配是连续的。

AllocateMessageQueueAveragelyByCircle 分配算法的队列负载机制如下：

c0：q0 q3 q6 q9 q12 q15

c1: q1 q4 q7 q10 q13

c2: q2 q5 q8 q11 q14

该分配算法的特点就是轮流一个一个分配。

温馨提示：如果 topic 的队列个数小于消费者的个数，那有些消费者无法分配到消息。在 RocketMQ 中一个 topic 的队列数直接决定了最大消费者的个数，但 topic 队列个数的增加对 RocketMQ 的性能不会产生影响。

在实际过程中，对主题进行扩容(增加队列个数)或者对消费者进行扩容、缩容是一件非常寻常的事情，那如果新增一个消费者，该消费者消费哪些队列呢？这就涉及到消息消费队列的重新分配，即**消费队列重平衡机制**。

在 RocketMQ 客户端中会每隔 20s 去查询当前 topic 的所有队列、消费者的个数，运用队列负载算法进行重新分配，然后与上一次的分配结果进行对比，如果发生了变化，则进行队列重新分配；如果没有发生变化，则忽略。

例如采取的分配算法如下图所示，现在增加一个消费者 c3，那队列的分布情况是怎样的呢？

AllocateMessageQueueAveragely分配算法的队列负载机制如下：

c0：q0 q1 q2 q3 q4 q5

c1：q6 q7 q8 q9 q10

c2：q11 q12 q13 q14 q15

根据新的分配算法，其队列最终的情况如下：

c0：q0 q1 q2 q3

c1：q4 q5 q6 q7

c2：q8 q9 q10 q11

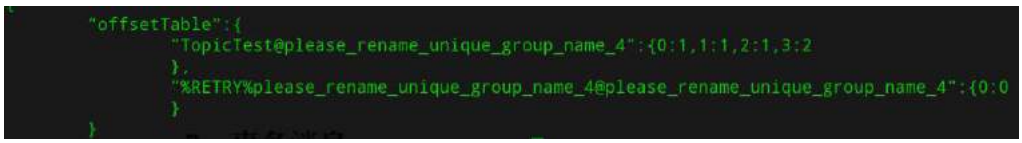
c3：q12 q13 q14 q15

上述整个过程无需应用程序干预，由 RocketMQ 完成。大概的做法就是将将原先分配给自己但这次不属于的队列进行丢弃，新分配的队列则创建新的拉取任务。

3. 消费进度

消费者消费一条消息后需要记录消费的位置，这样在消费端重启的时候，继续从上一次消费的位点开始进行处理新的消息。在 RocketMQ 中，消息消费位点的存储是以消费组为单位的。

集群模式下，消息消费进度存储在 broker 端，${ROCKETMQ_HOME}/store/config/consumerOffset.json 是其具体的存储文件，其中内容截图如下：



可见消费进度的 Key 为：topic@consumeGroup，然后每一个队列一个偏移量。

广播模式的消费进度文件存储在用户的主目录，默认文件全路劲名：${USER_HOME}/.rocketmq_offsets。

4. 消费模型

RocketMQ 提供了并发消费、顺序消费两种消费模型。

并发消费：对一个队列中消息，每一个消费者内部都会创建一个线程池，对队列中的消息多线程处理，即偏移量大的消息比偏移量小的消息有可能先消费。

顺序消费：在某一项场景，例如 MySQL binlog 场景，需要消息按顺序进行消费。在 RocketMQ 中提供了基于队列的顺序消费模型，即尽管一个消费组中的消费者会创建一个多线程，但针对同一个 Queue，会加锁。

温馨提示：并发消费模型中，消息消费失败默认会重试 16 次，每一次的间隔时间不一样；而顺序消费，如果一条消息消费失败，则会一直消费，直到消费成功。故在顺序消费的

使用过程中，应用程序需要区分系统异常、业务异常，如果是不符合业务规则导致的异常，则重试多少次都无法消费成功，这个时候一定要告警机制，及时进行人为干预，否则消费会积压。

三、事务消息

事务消息并不是为了解决分布式事务，而是提供消息发送与业务落库的一致性，其实现原理就是一次分布式事务的具体运用，请看如下示例：

```
public static void main(String[] args) {

    OrderService orderService;
    DefaultMQProducer mqProducer;

    //  存储订单  opertaor1
    orderService.saveOrCreateOrder(null);

    // 发送消息  opertaor2
    mqProducer.send( msg: null);

}
```

上述伪代码中，将订单存储关系型数据库中和将消息发送到 MQ 这是两个不同介质的两个操作，如果能保证消息发送、数据库存储这两个操作要么同时成功，要么同时失败，RocketMQ 为了解决该问题引入了事务消息。

温馨提示，本节主要的目的是让大家知晓各个术语的概念，由于事务消息的使用，将在该专栏的后续文章中详细介绍。

四、定时消息

开源版本的 RocketMQ 目前并不支持任意精度的定时消息。所谓的定时消息就是将消息发送到 Broker，但消费端不会立即消费，而是要到指定延迟时间后才能被消费端消费。

RocketMQ 目前支持指定级别的延迟，其延迟级别如下：

1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m 30m 1h 2h

五、消息过滤

消息过滤是指消费端可以根据某些条件对一个 topic 中的消息进行过滤，即只消费一个主题下满足过滤条件的消息。

RocketMQ 目前主要的过滤机制是基于 tag 的过滤与基于消息属性的过滤，基于消息属性的过滤支持 SQL92 表达式，对消息进行过滤。

六、小结

本文的主要目的是介绍 RocketMQ 常见的术语，例如 nameserver、broker、主题、消费组、消费者、队列负载算法、队列重平衡机制、并发消费、顺序消费、消费进度存储、定时消息、事务消息、消息过滤等基本概念，为后续的实战系列打下坚实基础。

从下一篇开始，将正式开始 RocketMQ 之旅，开始学习消息发送。

1.2 生产环境中，autoCreateTopicEnable 为什么不能设置为 true

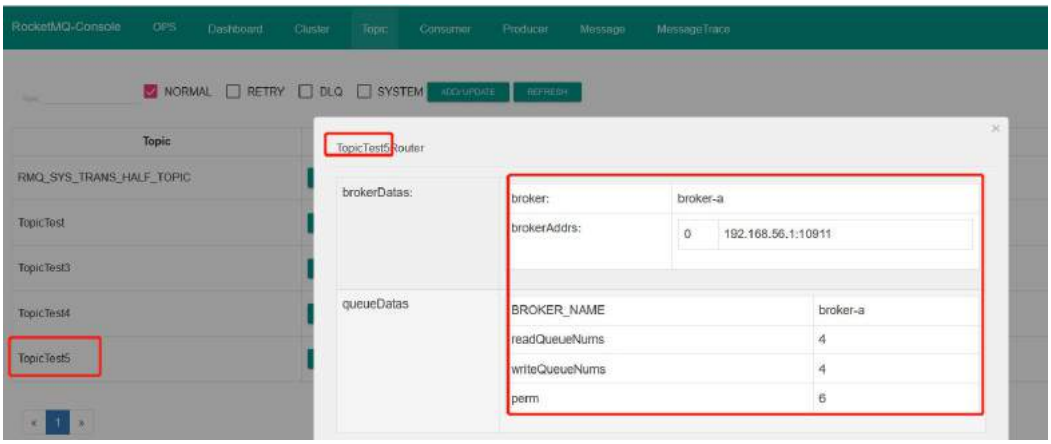
一、现象

很多网友会问，为什么明明集群中有多台 Broker 服务器，autoCreateTopicEnable 设置为 true，表示开启 Topic 自动创建，但新创建的 Topic 的路由信息只包含在其中一台 Broker 服务器上，这是为什么呢?

期望值：为了消息发送的高可用，希望新创建的 Topic 在集群中的每台 Broker 上创建对应的队列，避免 Broker 的单节点故障。

现象截图如下：

Broker 集群信息

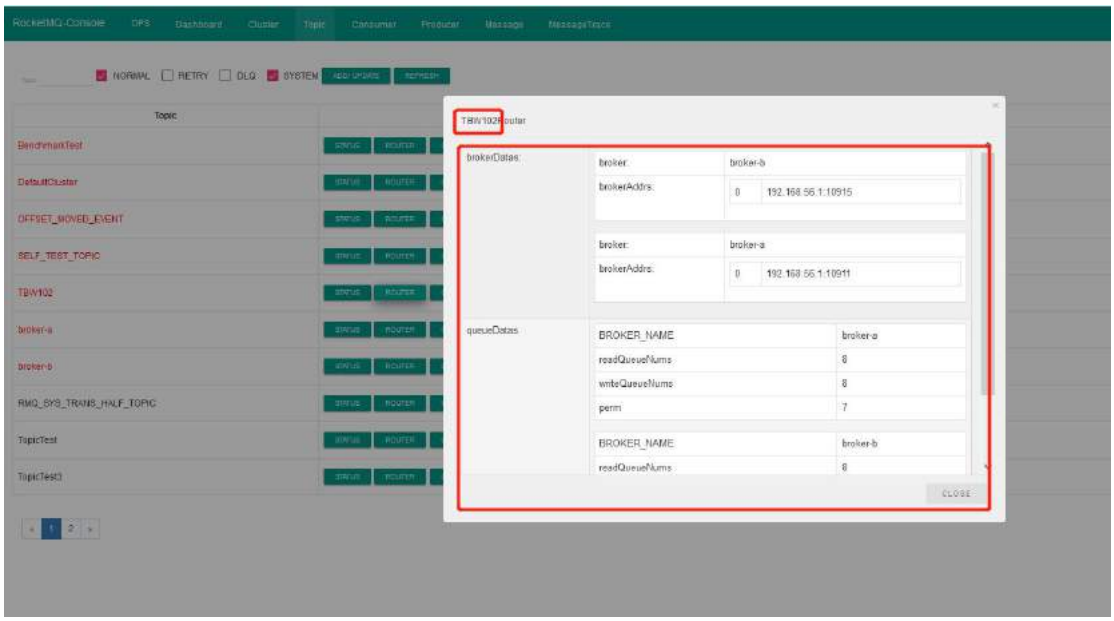


自动创建的 topicTest5 的路由信息：

topicTest5 只在 broker-a 服务器上创建了队列，并没有在 broker-b 服务器创建队列，不符合期望。

默认读写队列的个数为 4。

我们再来看一下 RocketMQ 默认 topic 的路由信息截图如下：



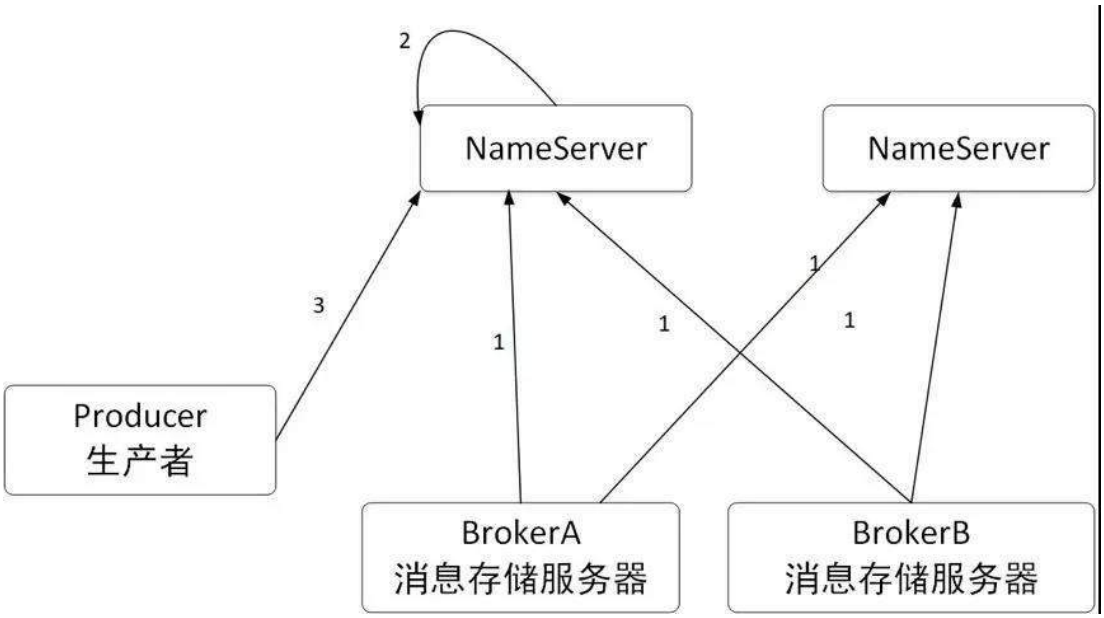
从图中可以默认 Topic 的路由信息为 broker-a、broker-b 上各 8 个队列。

二、思考

- 默认 Topic 的路由信息是如何创建的?
- Topic 的路由信息是存储在哪里? Nameserver? broker?
- RocketMQ Topic 默认队列个数。

三、原理

1. RocketMQ 基本路由规则



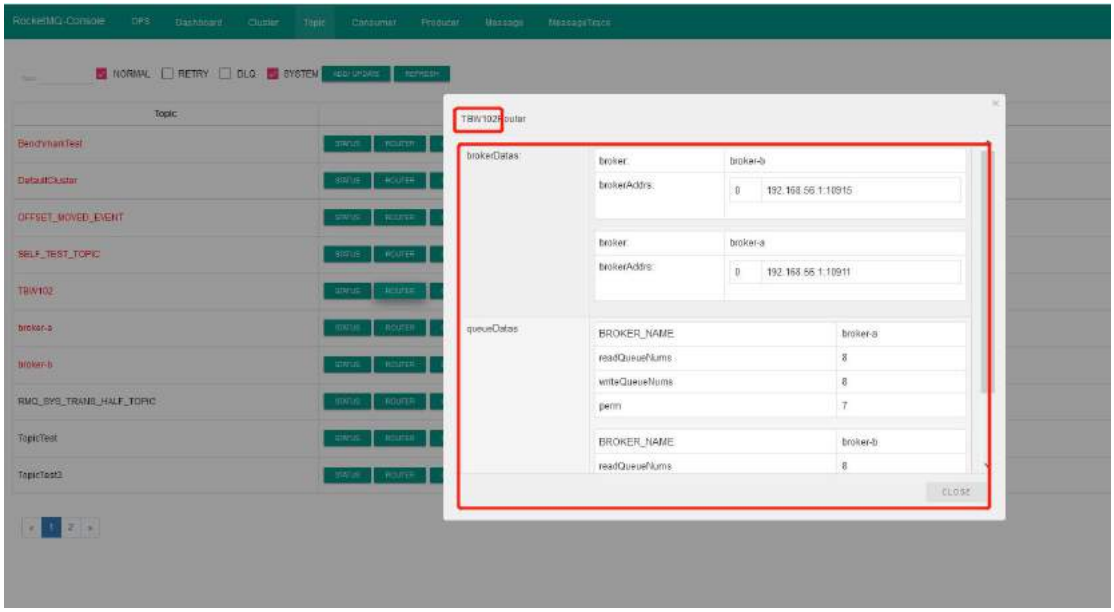
Broker 在启动时向 Nameserver 注册存储在该服务器上的路由信息，并每隔 30s 向 Nameserver 发送心跳包，并更新路由信息。

Nameserver 每隔 10s 扫描路由表，如果检测到 Broker 服务宕机，则移除对应的路由信息。

消息生产者每隔 30s 会从 Nameserver 重新拉取 Topic 的路由信息并更新本地路由表；在消息发送之前，如果本地路由表中不存在对应主题的路由消息时，会主动向 Nameserver 拉取该主题的消息。

回到本文的主题：autoCreateTopicEnable，开启自动创建主题，试想一下，如果生产者向一个不存在的主题发送消息时，上面的任何一个步骤都无法获取一个不存在的主题的路由信息，那该如何处理这种情况呢？

在 RocketMQ 中，如果 autoCreateTopicEnable 设置为 true，消息发送者向 NameServer 查询主题的路由消息返回空时，会尝试用一个系统默认的主题名称(MixAll.AUTO_CREATE_TOPIC_KEY_TOPIC)，此时消息发送者得到的路由信息为：



但问题就来了，默认 Topic 在集群的每一台 Broker 上创建 8 个队列，那问题来了，为啥新创建的 Topic 只在一个 Broker 上创建 4 个队列?

2. 探究 autoCreateTopicEnable 机制

默认 Topic 路由创建时机

温馨提示：本文不会详细跟踪整个整个源码创建过程，只会点出代码的关键入口点，如想详细了解 NameServer 路由消息、消息发送高可用的实现原理，建议查阅笔者的书籍《RocketMQ 技术内幕》第二、三章。

Step1：在 Broker 启动流程中，会构建 TopicConfigManager 对象，其构造方法中首先会判断是否开启了允许自动创建主题，如果启用了自动创建主题，则向 topicConfigTable 中添加默认主题的路由信息。

TopicConfigManager 构造方法：

```
{
    // MixAll.AUTO_CREATE_TOPIC_KEY_TOPIC
    if (this.brokerController.getBrokerConfig().isAutoCreateTopicEnable()) {
        String topic = MixAll.AUTO_CREATE_TOPIC_KEY_TOPIC;
        TopicConfig topicConfig = new TopicConfig(topic);
        this.systemTopicList.add(topic);
        topicConfig.setReadQueueNums(this.brokerController.getBrokerConfig()
            .getDefaultTopicQueueNums());
        topicConfig.setWriteQueueNums(this.brokerController.getBrokerConfig()
            .getDefaultTopicQueueNums());
        int perm = PermName.PERM_INHERIT | PermName.PERM_READ | PermName.PERM_WRITE;
        topicConfig.setPerm(perm);
        this.topicConfigTable.put(topicConfig.getTopicName(), topicConfig);
    }
}
```

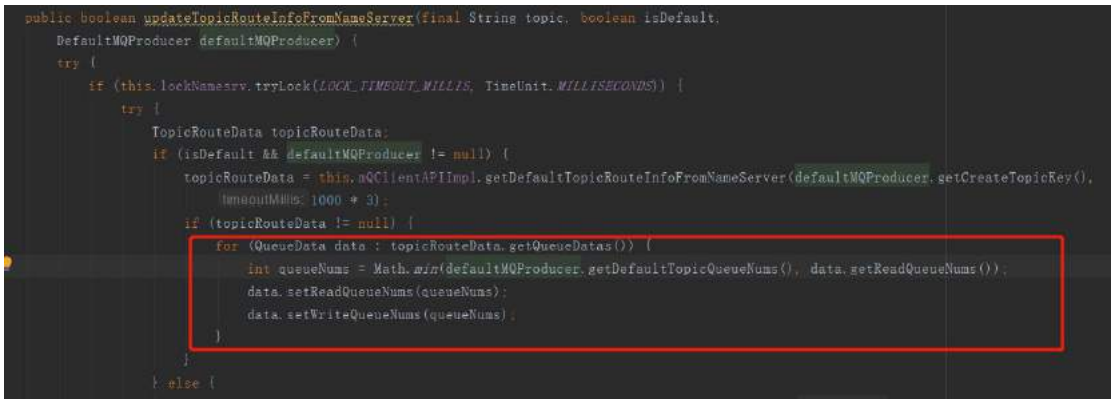
备注：该 topicConfigTable 中所有的路由信息，会随着 Broker 向 Nameserver 发送心跳包中，Nameserver 收到这些信息后，更新对应 Topic 的路由信息表。

注意：BrokerConfig 的 defaultTopicQueueNum 默认为 8。两台 Broker 服务器都会运行上面的过程，故最终 Nameserver 中关于默认主题的路由信息中，会包含两个 Broker 分别各 8 个队列信息。

Step2：生产者寻找路由信息

生产者首先向 NameServer 查询路由信息，由于是一个不存在的主题，故此时返回的路由信息为空，RocketMQ 会使用默认的主题再次寻找，由于开启了自动创建路由信息，NameServer 会向生产者返回默认主题的路由信息。然后从返回的路由信息中选择一个队列（默认轮询）。消息发送者从 Nameserver 获取到默认的 Topic 的队列信息后，队列的个数会改变吗？答案是会的，其代码如下：

MQClientInstance#updateTopicRouteInfoFromNameServer



温馨提示：消息发送者在到默认路由信息时，其队列数量，会选择 DefaultMQProducer#defaultTopicQueueNums 与 Nameserver 返回的的队列数取最小值，DefaultMQProducer#defaultTopicQueueNums 默认值为 4，故自动创建的主题，其队列数量默认为 4。

Step3：发送消息

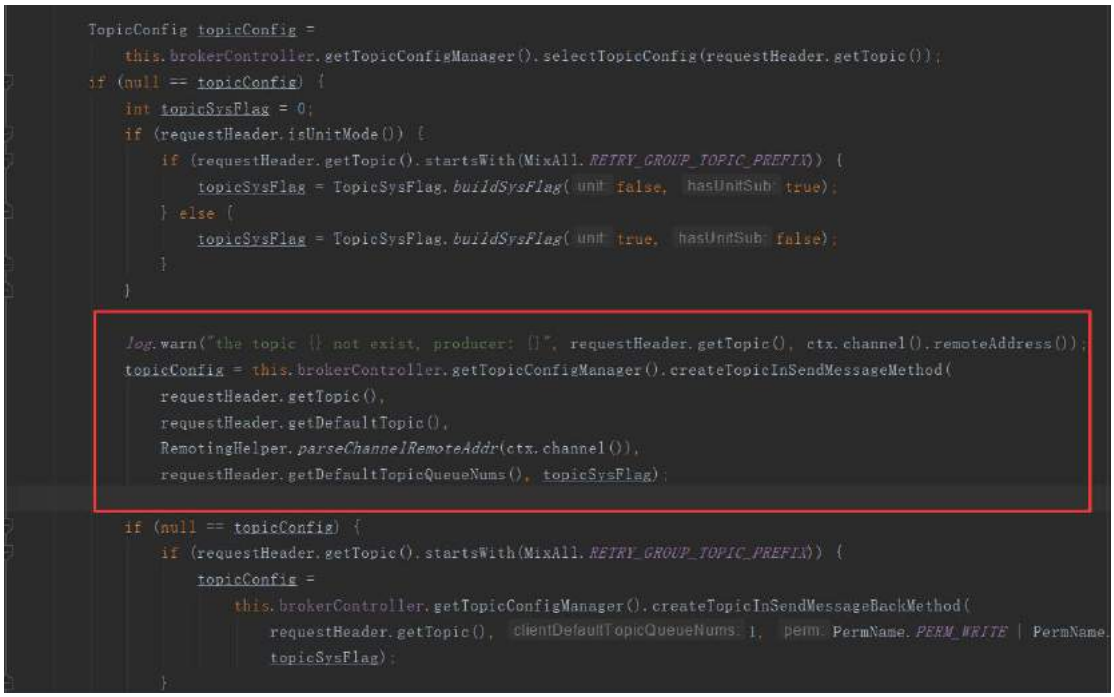
DefaultMQProducerImpl#sendKernelImpl

在消息发送时的请求报文中，设置默认 topic 名称，消息发送 topic 名称，使用的队列数量为 DefaultMQProducer#defaultTopicQueueNums，即默认为 4。

Step4：Broker 端收到消息后的处理流程

服务端收到消息发送的处理器为：SendMessageProcessor，在处理消息发送时，会调用 super.msgCheck 方法：

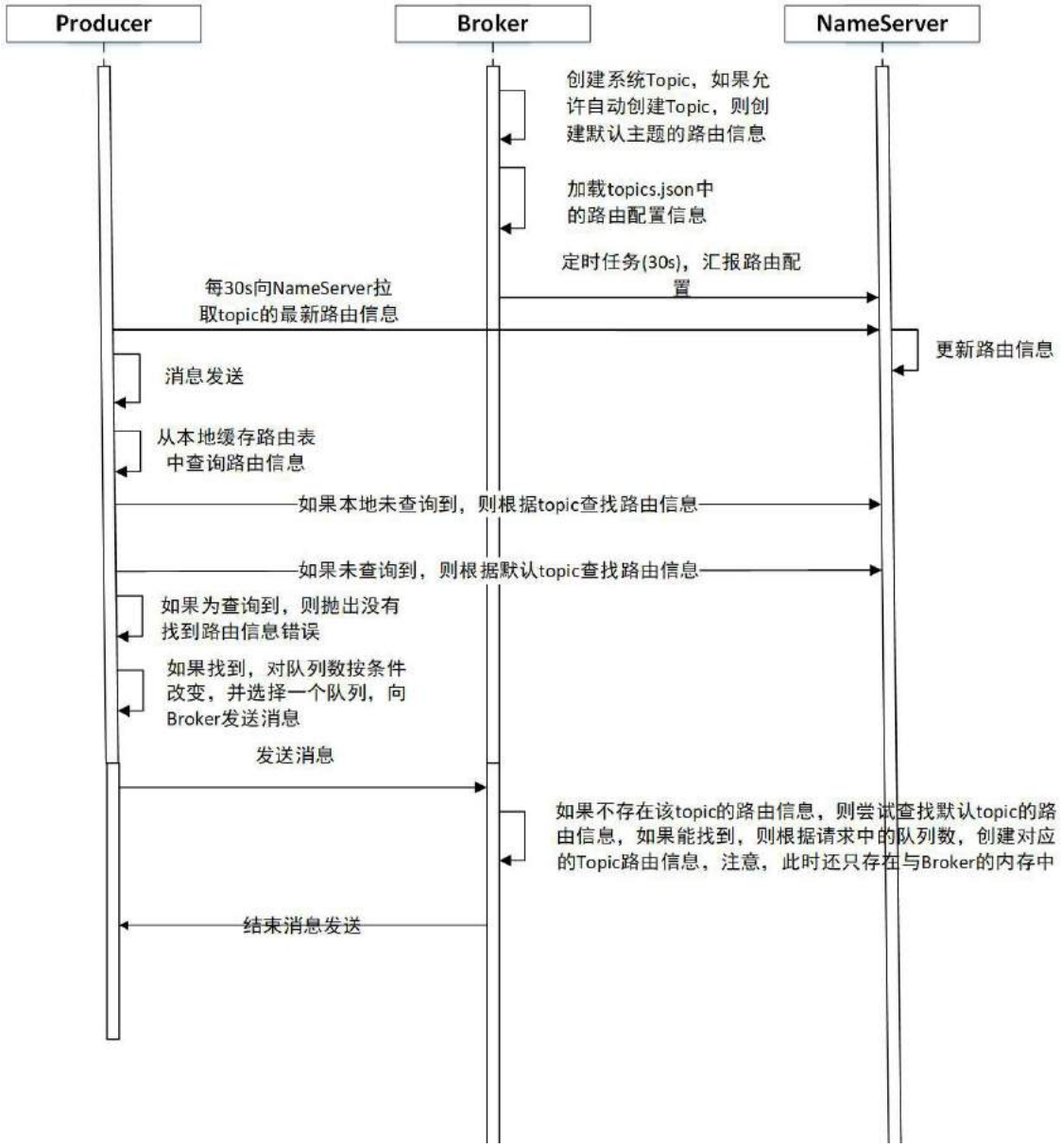
AbstractSendMessageProcessor#msgCheck



在 Broker 端，首先会使用 TopicConfigManager 根据 topic 查询路由信息，如果 Broker 端不存在该主题的路由配置(路由信息),此时如果 Broker 中存在默认主题的路由配置信息，则根据消息发送请求中的队列数量，在 Broker 创建新 Topic 的路由信息。这样 Broker 服务端就会存在主题的路由信息。

在 Broker 端的 topic 配置管理器中存在的路由信息，一会向 Nameserver 发送心跳包，汇报到 Nameserver，另一方面会有一个定时任务，定时存储在 broker 端，具体路径为${ROCKET_HOME}/store/config/topics.json 中，这样在 Broker 关闭后再重启，并不会丢失路由信息。

广大读者朋友，跟踪到这一步的时候，大家应该对启用自动创建主题机制时，新主题是的路由信息是如何创建的，为了方便理解，给出创建主题序列图：



现象分析

经过上面自动创建路由机制的创建流程，我们可以比较容易的分析得出如下结论：

因为开启了自动创建路由信息，消息发送者根据 Topic 去 NameServer 无法得到路由信息，但接下来根据默认 Topic 从 NameServer 是能拿到路由信息(在每个 Broker 中，存在 8 个队列)，因为两个 Broker 在启动时都会向 NameServer 汇报路由信息。此时消息发送者缓存的路由信息是 2 个 Broker，每个 Broker 默认 4 个队列（原因见 3.2.1: Step2 的分析）。消息发送者然后按照轮询机制，发送第一条消息选择(broker-a 的

messageQueue:0)，向 Broker 发送消息，Broker 服务器在处理消息时，首先会查看自己的路由配置管理器(TopicConfigManager)中的路由信息，此时不存在对应的路由信息，然后尝试查询是否存在默认 Topic 的路由信息，如果存在，说明启用了 autoCreateTopicEnable，则在 TopicConfigManager 中创建新 Topic 的路由信息，此时存在与 Broker 服务端的内存中，然后本次消息发送结束。此时，在 NameServer 中还不存在新创建的 Topic 的路由信息。

这里有三个关键点：

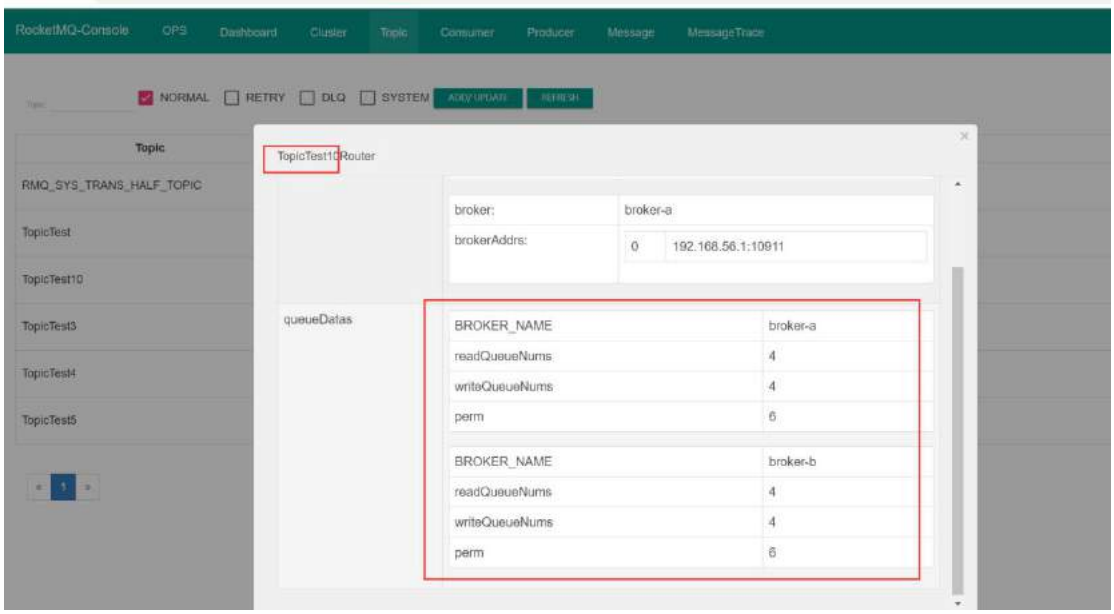
1. 启用 autoCreateTopicEnable 创建主题时，在 Broker 端创建主题的时机为，消息生产者往 Broker 端发送消息时才会创建。
2. 然后 Broker 端会在一个心跳包周期内，将新创建的路由信息发送到 NameServer，于此同时，Broker 端还会有一个定时任务，定时将内存中的路由信息，持久化到 Broker 端的磁盘上。
3. 消息发送者会每隔 30s 向 NameServer 更新路由信息，如果消息发送端一段时间内未发送消息，就不会有消息发送集群内的第二台 Broker，那么 NameServer 中新创建的 Topic 的路由信息只会包含 Broker-a，然后消息发送者会向 NameServer 拉取最新的路由信息，此时就会消息发送者原本缓存了 2 个 broker 的路由信息，将会变为一个 Broker 的路由信息，则该 Topic 的消息永远不会发送到另外一个 Broker，就出现了上述现象。

原因就分析到这里了，现在我们还可以的大胆假设，开启 autoCreateTopicEnable 机制，什么情况会在两个 Broker 上都创建队列，其实，我们只需要连续快速的发送 9 条消息，就有可能在 2 个 Broker 上都创建队列，验证代码如下：

```
    public static void main(String[] args) throws MQClientException, InterruptedException
{
        DefaultMQProducer producer = new DefaultMQProducer("please_rename_unique_
group_name");
        producer.setNamesrvAddr("127.0.0.1:9876");
        producer.start();
        for (int i = 0; i < 9; i++) {
            try {
                Message msg = new Message("TopicTest10" ,"TagA" , ("Hello RocketMQ
" + i).getBytes(RemotingHelper.DEFAULT_CHARSET));
```

```
                SendResult sendResult = producer.send(msg);
                System.out.printf("%s%n", sendResult);
            } catch (Exception e) {
                e.printStackTrace();
                Thread.sleep(1000);
            }
        }
        producer.shutdown();
    }
```

其路由信息如下，符合预期。



本文就分析到这里，如果喜欢这篇文章，希望大家帮忙点赞，转发，谢谢你们，同时大家也可以给作者留言在使用 RocketMQ 的过程中遇到的疑难杂症，与作者互动。

1.3 实战：RocketMQ 学习环境搭建指南篇

本文主要分如下几个部分展开：

- Linux 服务器安装 RocketMQ、RocketMQ-Console
- IDEA 中搭建可调试环境

一、Linux 安装 RocketMQ、RocketMQ-Console

1. 安装 RocketMQ

Step1：从如下地址下载 RocketMQ 安装包

```
cd /opt/application
wget https://mirrors.tuna.tsinghua.edu.cn/apache/rocketmq/4.7.1/rocketmq-all-4.7.1-bin-release.zip
```

Step2：解压安装包

```
unzip rocketmq-all-4.7.1-bin-release.zip
ls -l
```

解压后的文件如下图所示：

```
dingwpmz@dingwpmz-PC:/opt/application/rocketmq-all-4.7.1-bin-release$ ls -l
总用量 48
drwxr-xr-x 2 dingwpmz dingwpmz  4096 6月  24 14:50 benchmark
drwxr-xr-x 3 dingwpmz dingwpmz  4096 6月  24 14:02 bin
drwxr-xr-x 6 dingwpmz dingwpmz  4096 6月   2 14:09 conf
drwxr-xr-x 2 dingwpmz dingwpmz  4096 6月  24 14:50 lib
-rw-r--r-- 1 dingwpmz dingwpmz 17336 6月   2 14:09 LICENSE
-rw-r--r-- 1 dingwpmz dingwpmz  1338 6月   2 14:09 NOTICE
-rw-r--r-- 1 dingwpmz dingwpmz  5069 6月  24 14:02 README.md
dingwpmz@dingwpmz-PC:/opt/application/rocketmq-all-4.7.1-bin-release$
```

其中 conf 文件夹存放的是 RocketMQ 的配置文件，提供了各种部署结构的示例配置。例如 2m-2s-async 是 2 主 2 从异步复制的配置示例；2m-noslave 是 2 主的示例配置。由于本文主要是搭建一个学习环境，故采取的部署架构为 1 主的部署架构，关于生产环境下如何搭建 RocketMQ 集群、如何调优参数将在该专栏的后续文章中专门介绍。

Step3：修改 Nameserver jvm 参数

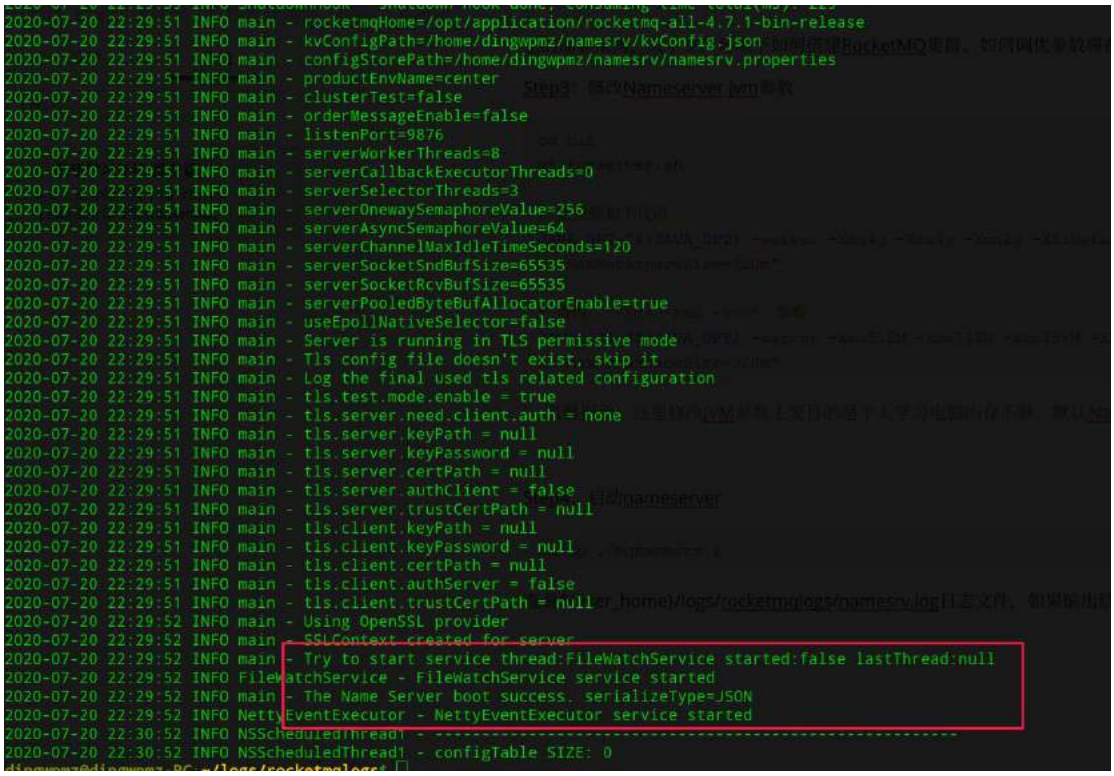
```
cd bin
vi runserver.sh

# 定位到如下代码
JAVA_OPT="${JAVA_OPT} -server -Xms4g -Xmx4g -Xmn2g -XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=320m"

# 修改 "-Xms -Xmx -Xmn" 参数
JAVA_OPT="${JAVA_OPT} -server -Xms512M -Xmx512M -Xmn256M -XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=320m"
```

温馨提示：这里修改 JVM 参数主要目的是个人学习电脑内存不够，默认 NameServer 会占用 4G。

Step4：启动 nameserver

```
nohup ./mqnamesrv &
```

查看${user_home}/logs/rocketmqlogs/namesrv.log 日志文件，如果输出结果如下图所示即表示启动成功。



Step5：修改 broker 的配置文件

```
vi conf/broker.conf
# 使用如下配置文件
brokerClusterName = DefaultCluster
brokerName = broker-a
brokerId = 0
deleteWhen = 04
fileReservedTime = 48
brokerRole = ASYNC_MASTER
flushDiskType = ASYNC_FLUSH
storePathRootDir=/data/rocketmq/store
storePathCommitLog=/data/rocketmq/store/commitlog
namesrvAddr=127.0.0.1:9876
brokerIP1=192.168.3.10
brokerIP2=192.168.3.10
autoCreateTopicEnable=false
```

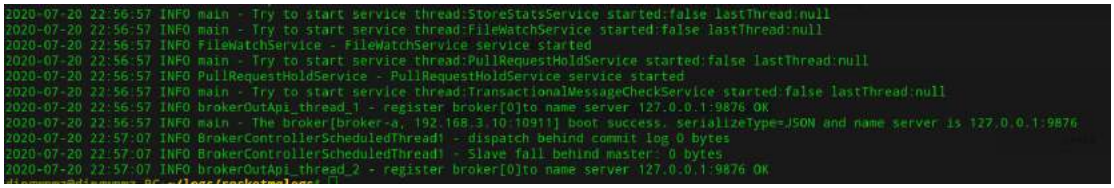
Step6：修改 broker jvm 参数。

```
cd bin
vi runbroker.sh
#修改如下配置(配置前)
JAVA_OPT="${JAVA_OPT} -server -Xms8g -Xmx8g -Xmn4g"
#配置后
JAVA_OPT="${JAVA_OPT} -server -Xms1g -Xmx1g -Xmn512m"
```

Step7：启动 broker

```
cd bin
nohup ./mqbroker -c ../conf/broker.conf &
```

查看${user_home}/logs/rocketmqlogs/broker.log，如果输出结果如下图所示表示启动成功。



经过上面的步骤，就成功在 Linux 环境上安装了 RocketMQ Nameserver 服务器与 Broker 服务器。

温馨提示：如果上面在安装过程中发生了错误，大家可以查看{user_home}为用户主目录。

该目录下会有众多的日志文件，如果一开始对这些文件的含义不了解也没关系，大家可以通过 ls -l 命令，逐一查看文件大小不为 0 的文件，从而寻找错误日志，便于快速解决问题。

RocketMQ 提供了众多的运维命令来查看 RocketMQ 集群的运行状态，在这里我先简单使用 clusterList 命令来查看集群的状态，用于验证一下集群的状态。

```
sh ./mqadmin clusterList -n 127.0.0.1:9876
```

其运行结果如下图所示：

2. 安装 RocketMQ-Console

使用运维命令不太直观，学习成本较大，为此 RocketMQ 官方提供了一个运维管理界面 RokcetMQ-Console，用于对 RocketMQ 集群提供常用的运维功能，故本节主要讲解如何在 Linux 环境安装 rocketmq-console。

RocketMQ 官方并未提供 rocketmq-console 的安装包，故需要通过源码进行编译。

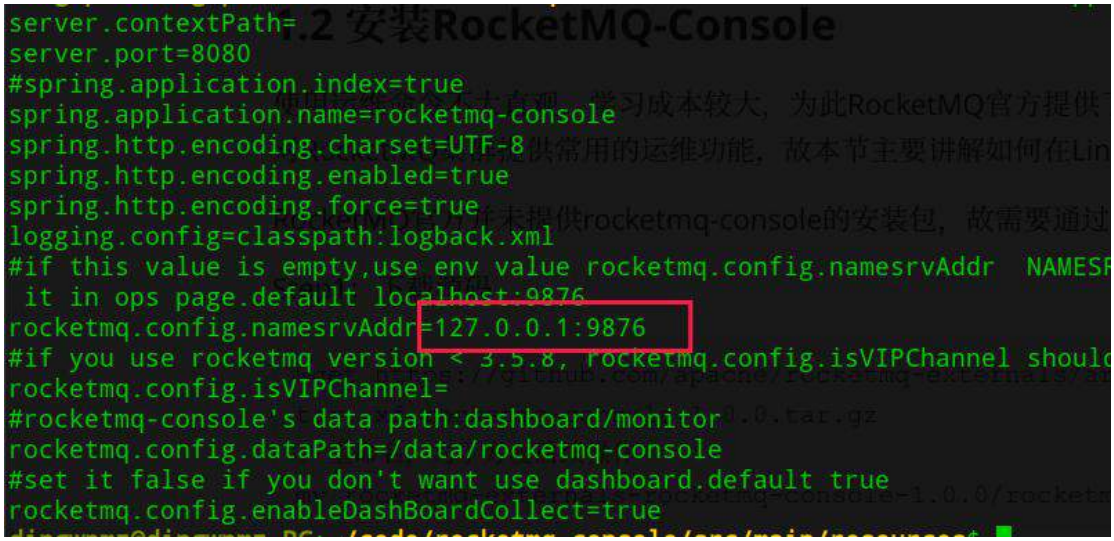
Step1：下载源码

```
wget https://github.com/apache/rocketmq-externals/archive/rocketmq-console-1.0.0.tar.gz
tar -xf rocketmq-console-1.0.0.tar.gz
# 重命名，为了方便后续操作
mv rocketmq-externals-rocketmq-console-1.0.0/rocketmq-console rocketmq-console
```

Step2：修改配置文件

```
cd rocketmq-console
vi src/main/resources/applications.properties
```

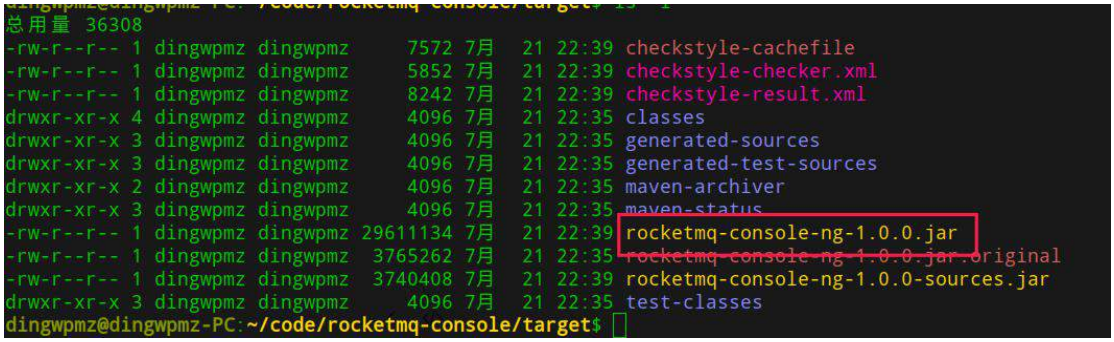
主要是修改指向的 nameserver 地址，修改结果如下图所示：



Step3：使用 maven 命令编译源代码。

```
mvn clean  package -DskipTests
```

编译后在 target 目录下会生成可运行的 jar 包，如下图所示：



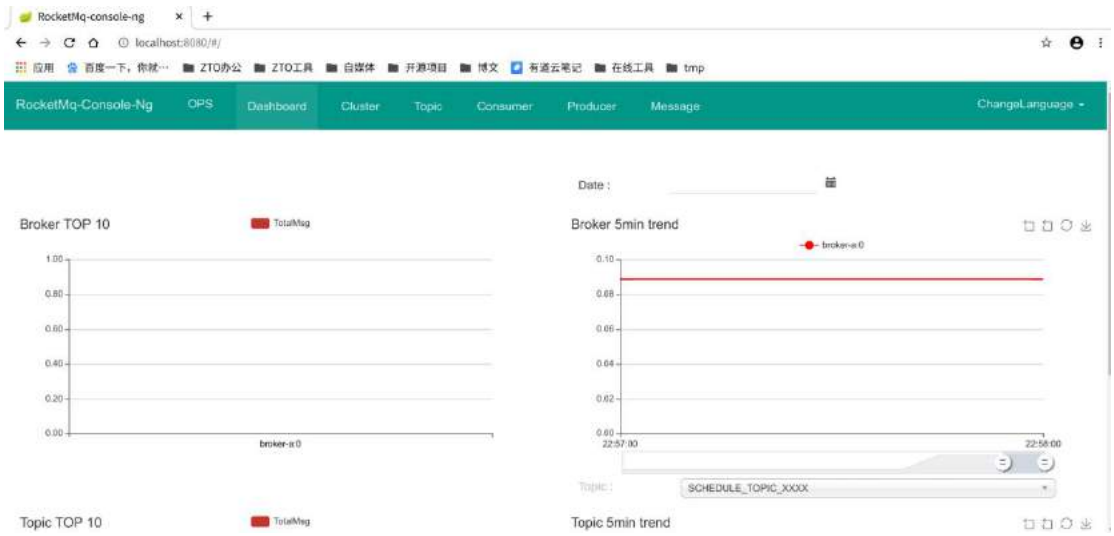
Step4：我们可以将该包复制到自己常用的软件安装目录，例如笔者喜欢将其放在/opt/application 下。

```
cp rocketmq-console-ng-1.0.0.jar /opt/application/
```

Step5：启动 rocketmq-conolse

```
nohup java -jar rocketmq-console-ng-1.0.0.jar &
```

在浏览器中输入:http://localhost:8080 查看是否安装成功，如果出现如下图则表示安装成功。



3. 异常分析与解决思路

如果在安装过程中出现意想不到的错误，别慌，通过查看相关的日志文件，寻找错误日志，根据错误日志进行思考或百度，相信能够轻易将其解决。

例如使用的 baseuser 启动的 rocketmq，rocketmq-console，那相关的日志路径如下：

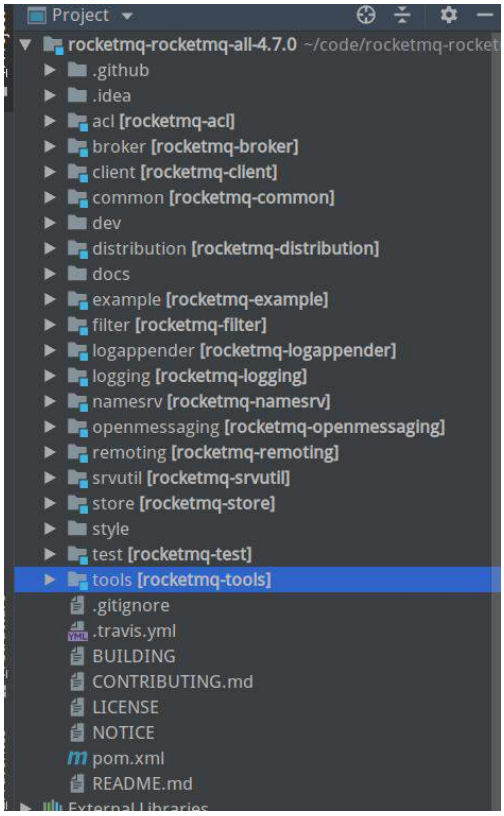
rocketmq：/home/baseuser/logs/rocketmqlogs/

rocketmq-console：/home/baseuser/logs/consolelogs

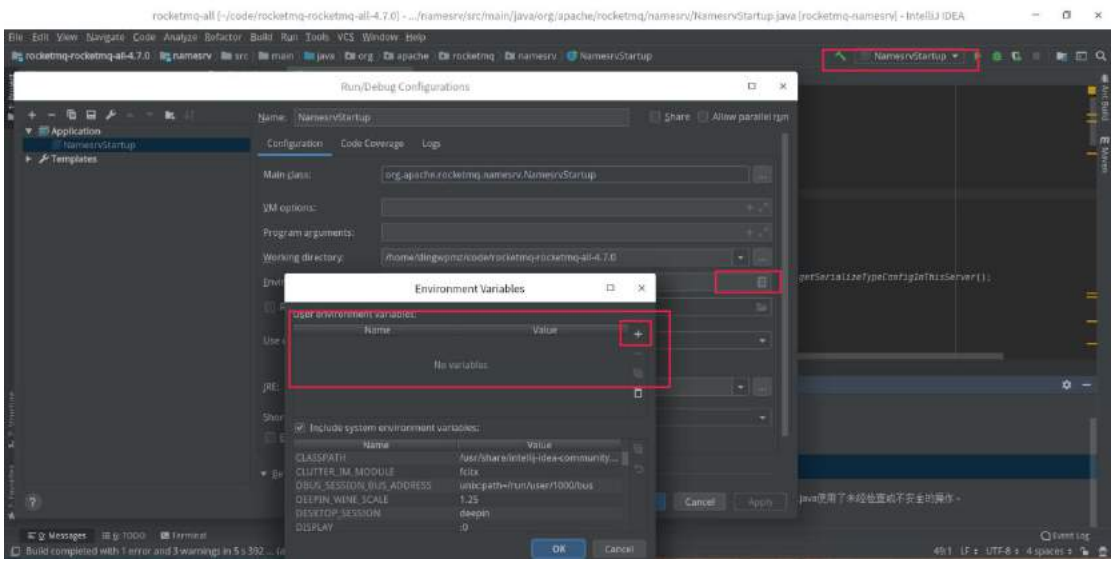
二、IDEA 中安装 RocketMQ

绝大数的程序员最信赖的开发调试工具基本都是 DEBUG，那能在 IDEA 中 debug RocketMQ 的源码吗？答案当然是可以的。本节就来演示如何在 IDEA 中运行 RocketMQ 的 Nameserver、Broker 组件，并进行 Debug。

Setp1：从 github 上下载 RocketMQ 源码，并将其导入到 IEDA 中，其截图如下：



Step2：namesrv/src/main/java/org/apache/rocketmq/namesrv/NamesrvStartup 设置环境变量 ROCKETMQ_HOME，操作步骤如下图所示：



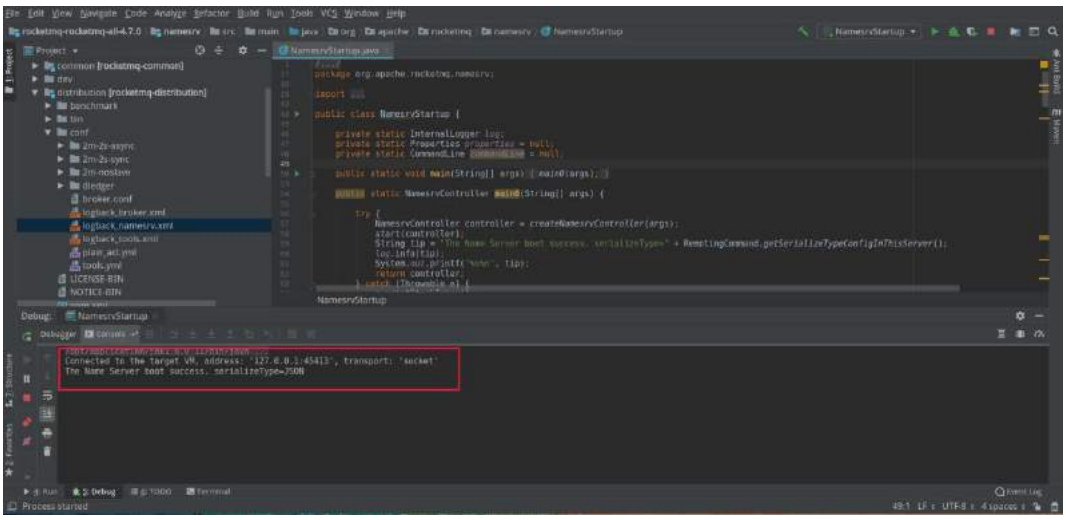
设置环境变量名称：ROCKETMQ_HOME，其值用于指定 RocketMQ 运行的主目录，笔者设置的路径为：/home/dingwpmz/tmp/rocketmq。

Step3：将 distribution/conf/logback_namesrv.xml 文件拷贝到【Step2】中设置的主目录下,执行后的效果如下图所示：

```
dingwpmz@dingwpmz-PC:~/tmp/rocketmq/conf$ ls
logback_namesrv.xml
dingwpmz@dingwpmz-PC:~/tmp/rocketmq/conf$ pwd
/home/dingwpmz/tmp/rocketmq/conf
dingwpmz@dingwpmz-PC:~/tmp/rocketmq/conf$
```

温馨提示：该文件为 nameserver 的日志路劲，可以手动修改 logback_namesrv.xml 文件中的日志目录，由于这是 logback 的基础知识，这里就不再详细介绍 logback 的配置方法。

Step4：以 debug 方法运行 NamesrvStartup，执行效果如下图所示,表示启动成功。



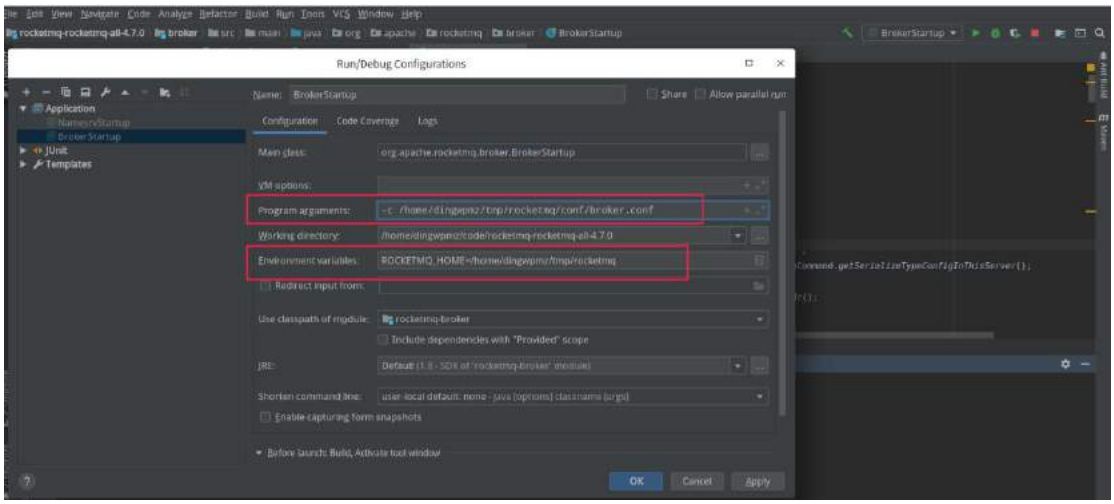
Step5：将 distribution/conf/logback_brokerxml、broker.conf 文件拷贝到【Step2】中设置的主目录下,执行后的效果如下图所示：

```
dingwpmz@dingwpmz-PC:~/tmp/rocketmq/conf$ pwd
/home/dingwpmz/tmp/rocketmq/conf
dingwpmz@dingwpmz-PC:~/tmp/rocketmq/conf$ ls
broker.conf  logback_broker.xml  logback_namesrv.xml
dingwpmz@dingwpmz-PC:~/tmp/rocketmq/conf$
```

Step6：修改 broker.conf 中的配置，主要设置 nameserver 的地址，broker 的名称等相关属性。

```
vi broker.conf
# 使用如下配置文件
brokerClusterName = DefaultCluster
brokerName = broker-a
brokerId = 0
deleteWhen = 04
fileReservedTime = 48
brokerRole = ASYNC_MASTER
flushDiskType = ASYNC_FLUSH
storePathRootDir=/home/dingwpmz/tmp/rocketmq/store
storePathCommitLog=/home/dingwpmz/tmp/rocketmq/store/commitlog
namesrvAddr=127.0.0.1:9876
brokerIP1=192.168.3.10
brokerIP2=192.168.3.10
autoCreateTopicEnable=true
```

Step7：broker/src/main/java/org/apache/rocketmq/broker/BrokerStartup 设置环境变量 ROCKETMQ_HOME，操作步骤如下图所示：



Step8：以 Debug 模式运行 BrokerStartup，其运行结果如下图所示：

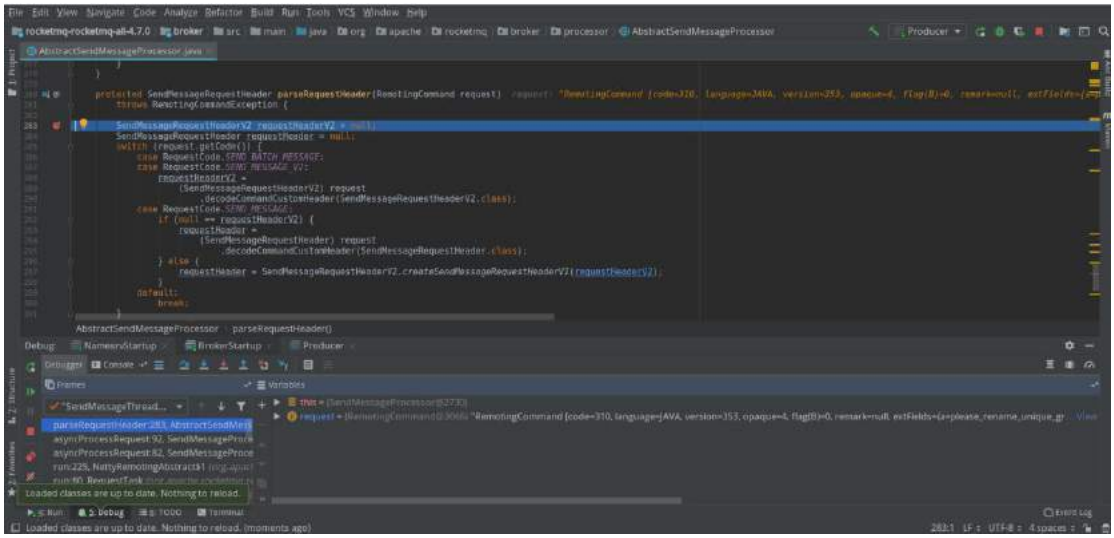
```
/opt/application/jdk1.8.0_11/bin/java ...
Connected to the target VM, address: '127.0.0.1:45233', transport: 'socket'
The broker[broker-a, 192.168.3.10:10911] boot success. serializeType=JSON
```

看到这样的提示就表示大功告成。

接下来简单来做一个验证。

首先先在 AbstractSendMessageProcessor 类的 parseRequestHeader 方法中打上一个断点。

然后运行 example 中 org/apache/rocketmq/example/quickstart/Producer，看是否能进入到断点中，运行结果如下图所示，已进入到 Debug 模式。



三、小结

本篇作为 RocketMQ 实战系列的第一篇文章，其目的就是构建一个研究 RocketMQ 的学习环境，故从两个方面进行展开：

1. 在 Linux 环境安装 RocketMQ、RocketMQ-Console。
2. 在 IDEA 中运行 RocketMQ，构建一个可以调试 RocketMQ 的环境。

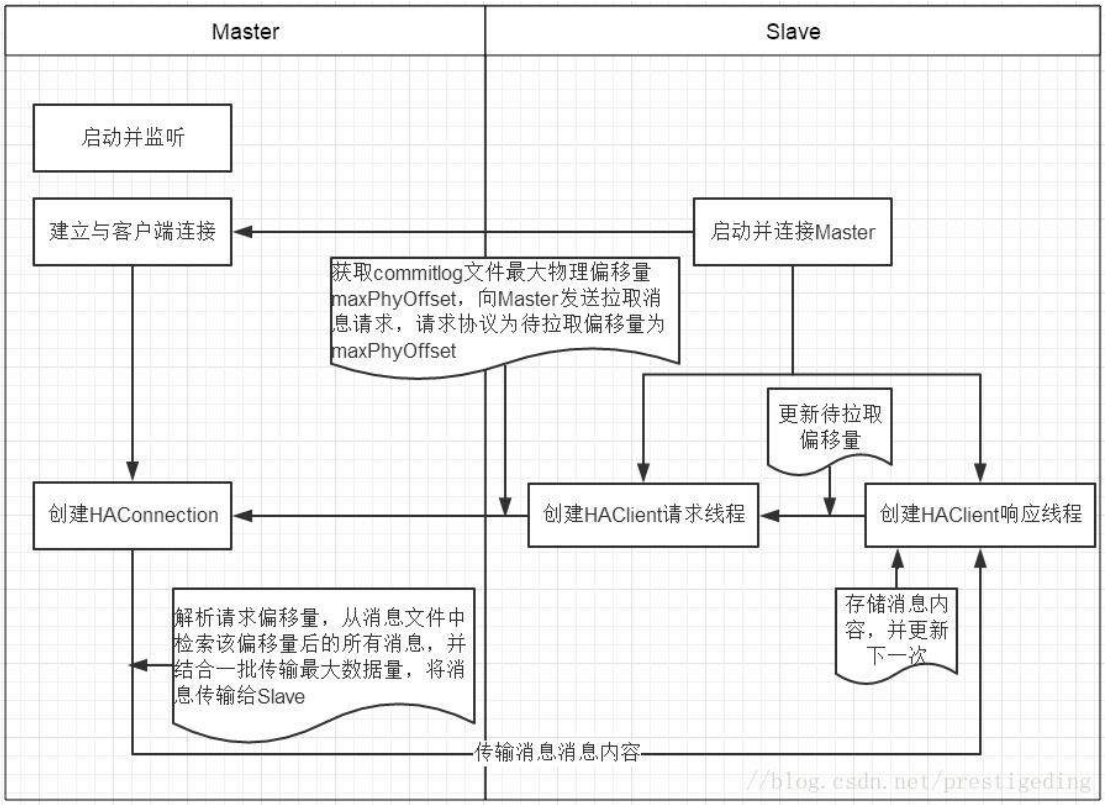
温馨提示：搭建一个可调试的环境，但绝不是学习 RocketMQ 源码，就从 Debug 一步异步跟踪，这样会陷入其中而不可自拔，DEBUG 只是一种辅助，应该用在无法理解某一端代码时，使用 DEBUG,借助运行时的一些数据，使之更容易理解。

1.4 RocketMQ HA 核心工作机制

温馨提示：建议参考代码 RocketMQ4.4 版本，4.5 版本引入了多副本机制，实现了主从自动切换，本文并不关心主从切换功能。

一、初识主从同步

主从同步基本实现过程如下图所示：



RocketMQ 的主从同步机制如下：

- 首先启动 Master 并在指定端口监听；
- 客户端启动，主动连接 Master，建立 TCP 连接；
- 客户端以每隔 5s 的间隔时间向服务端拉取消息，如果是第一次拉取的话，先获取本地 commitlog 文件中最大的偏移量，以该偏移量向服务端拉取消息；
- 服务端解析请求，并返回一批数据给客户端；

- 客户端收到一批消息后，将消息写入本地 commitlog 文件中，然后向 Master 汇报拉取进度，并更新下一次待拉取偏移量；
- 然后重复第 3 步；

RocketMQ 主从同步一个重要的特征：主从同步不具备主从切换功能，即当主节点宕机后，从不会接管消息发送，但可以提供消息读取。

本文并不会详细分析 RocketMQ 主从同步的实现细节，如大家对其感兴趣，可以查阅笔者所著的《 RocketMQ 技术内幕 》或查看笔者博文：https://blog.csdn.net/prestigeding/article/details/79600792

二、提出问题

主，从服务器都在运行过程中，消息消费者是从主拉取消息还是从从拉取?

RocketMQ 主从同步架构中，如果主服务器宕机，从服务器会接管消息消费，此时消息消费进度如何保持，当主服务器恢复后，消息消费者是从主拉取消息还是从从服务器拉取，主从服务器之间的消息消费进度如何同步?

三、原理探究

1. RocketMQ 主从读写分离机制

RocketMQ 的主从同步，在默认情况下 RocketMQ 会优先选择从主服务器进行拉取消息，并不是通常意义的上的读写分离，那什么时候会从拉取呢?

温馨提示：本节同样不会详细整个流程，只会点出其关键点，如果想详细了解消息拉取、消息消费等核心流程，建议大家查阅笔者所著的《 RocketMQ 技术内幕 》。

在 RocketMQ 中判断是从主拉取，还是从从拉取的核心代码如下：

```
DefaultMessageStore#getMessage
long diff = maxOffsetPy - maxPhyOffsetPulling;  // @1
long memory = (long) (StoreUtil.TOTAL_PHYSICAL_MEMORY_SIZE
                        * (this.messageStoreConfig.getAccessMessageInMemoryM
```

```
axRatio() / 100.0));   // @2
getResult.setSuggestPullingFromSlave(diff > memory);    // @3
```

代码@1：首先介绍一下几个局部变量的含义：

- maxOffsetPy

  当前最大的物理偏移量。返回的偏移量为已存入到操作系统的 PageCache 中的内容。

- maxPhyOffsetPulling

  本次消息拉取最大物理偏移量，按照消息顺序拉取的基本原则，可以基本预测下次开始拉取的物理偏移量将大于该值，并且就在其附近。

- diff

  maxOffsetPy 与 maxPhyOffsetPulling 之间的间隔，getMessage 通常用于消息消费时，即这个间隔可以理解为目前未处理的消息总大小。

代码@2：获取 RocketMQ 消息存储在 PageCache 中的总大小，如果当 RocketMQ 容量超过该阈值，将会将被置换出内存，如果要访问不在 PageCache 中的消息，则需要从磁盘读取。

- StoreUtil.TOTAL_PHYSICAL_MEMORY_SIZE

  返回当前系统的总物理内存。参数

- accessMessageInMemoryMaxRatio

  设置消息存储在内存中的阀值，默认为 40。

结合代码@2 这两个参数的含义，算出 RocketMQ 消息能映射到内存中最大值为 40% * (机器物理内存)。

代码@3：设置下次拉起是否从从拉取标记，触发下次从从服务器拉取的条件为：当前所有可用消息数据(所有 commitlog)文件的大小已经超过了其阈值，默认为物理内存的 40%。

那 GetResult 的 suggestPullingFromSlave 属性在哪里使用呢?

```
    PullMessageProcessor#processRequest
    if (getMessageResult.isSuggestPullingFromSlave()) {         // @1
    responseHeader.setSuggestWhichBrokerId(subscriptionGroupConfig.getWhichBrokerW
henConsumeSlowly());
    } else {
            responseHeader.setSuggestWhichBrokerId(MixAll.MASTER_ID);
    }
    switch (this.brokerController.getMessageStoreConfig().getBrokerRole()) {         // @2
            case ASYNC_MASTER:
            case SYNC_MASTER:
                    break;
            case SLAVE:
                    if (!this.brokerController.getBrokerConfig().isSlaveReadEnable()) {
                            response.setCode(ResponseCode.PULL_RETRY_IMMEDIATE
LY);
                            responseHeader.setSuggestWhichBrokerId(MixAll.MASTER_I
D);
                    }
                    break;
     }

    if (this.brokerController.getBrokerConfig().isSlaveReadEnable()) { // @3
                    // consume too slow ,redirect to another machine
                    if (getMessageResult.isSuggestPullingFromSlave()) {
                            responseHeader.setSuggestWhichBrokerId(subscriptionGroupConfig.g
etWhichBrokerWhenConsumeSlowly());
                    }
                    // consume ok
                    else {
                            responseHeader.setSuggestWhichBrokerId(subscriptionGroupConfig.ge
tBrokerId());
                    }
            } else {
                    responseHeader.setSuggestWhichBrokerId(MixAll.MASTER_ID);
            }
```

代码@1：如果从 commitlog 文件查找消息时，发现消息堆积太多，默认超过物理内存的 40%后，会建议从从服务器读取。

代码@2：如果当前服务器的角色为从服务器:并且 slaveReadEnable=true，则忽略代码@1 设置的值，下次拉取切换为从主拉取。

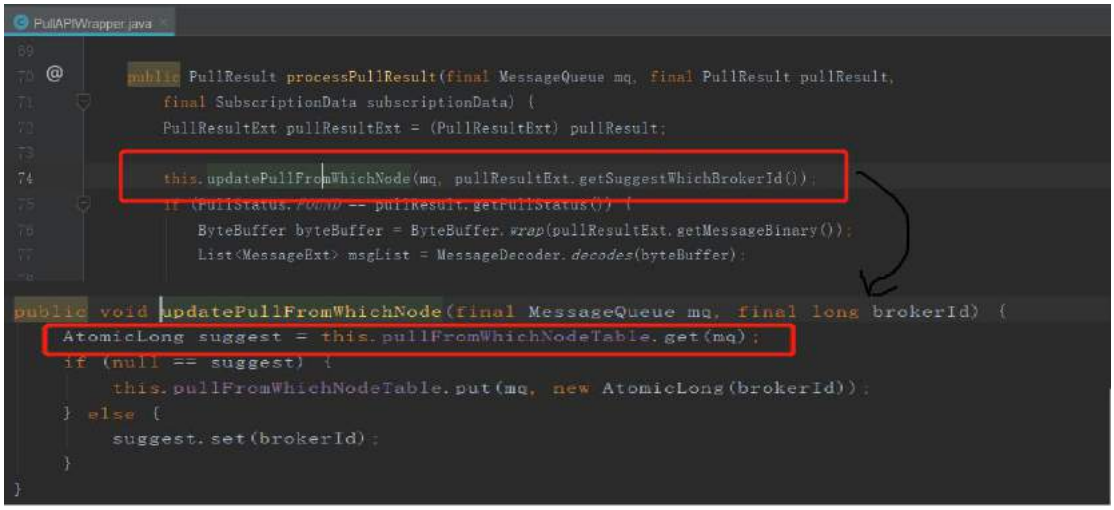
代码@3：如果 slaveReadEnable=true(从允许读)，并且建议从从服务器读取，则从消息消费组建议当消息消费缓慢时建议的拉取 brokerId，由订阅组配置属性 whichBrokerWhenConsumeSlowly 决定；如果消息消费速度正常，则使用订阅组建议的 brokerId 拉取消息进行消费，默认为主服务器。如果不允许从可读，则固定使用从主拉取。

温馨提示：请注意 broker 服务参数 slaveReadEnable，与订阅组配置信息：whichBrokerWhenConsumeSlowly、brokerId 的值，在生产环境中，可以通过 updateSubGroup 命令动态改变订阅组的配置信息。

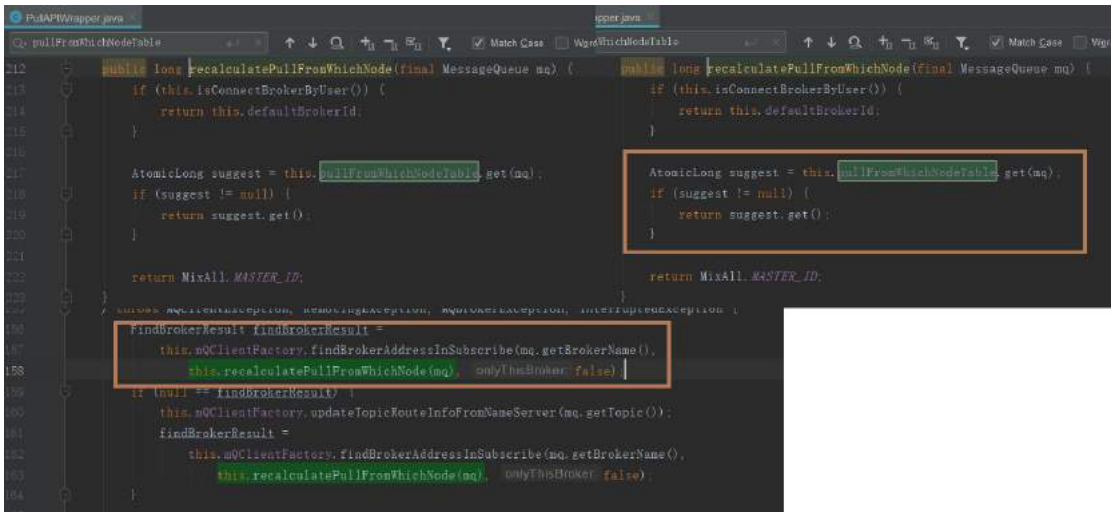
如果订阅组的配置保持默认值的话，拉取消息请求发送到从服务器后，下一次消息拉取，无论是否开启 slaveReadEnable，下一次拉取，还是会发往主服务器。

上面的步骤，在消息拉取命令的返回字段中，会将下次建议拉取 Broker 返回给客户端，根据其值从指定的 broker 拉取。

消息拉取实现 PullAPIWrapper 在处理拉取结果时会将服务端建议的 brokerId 更新到 broker 拉取缓存表中。



在发起拉取请求之前，首先根据如下代码，选择待拉取消息的 Broker。



2. 消息消费进度同步机制

从上面内容可知，主从同步引入的主要目的就是消息堆积的内容默认超过物理内存的40%，则消息读取则由从服务器来接管，实现消息的读写分离，避免主服务 IO 抖动严重。那问题来了，主服务器宕机后，从服务器接管消息消费后，那消息消费进度存储在哪里？当主服务器恢复正常后，消息是从主服务器拉取还是从从服务器拉取？主服务器如何得知最新的消息消费进度呢？

RocketMQ 消息消费进度管理（集群模式）：

集群模式下消息消费进度存储文件位于服务端${ROCKETMQ_HOME}/store/config/consumerOffset.json。消息消费者从服务器拉取一批消息后提交到消费组特定的线程池中处理消息，当消息消费成功后会向 Broker 发送 ACK 消息，告知消费端已成功消费到哪条消息，Broker 收到消息消费进度反馈后，首先存储在内存中，然后定时持久化到 consumeOffset.json 文件中。备注：关于消息消费进度管理更多的实现细节，建议查阅笔者所著的《 RocketMQ 技术内幕 》。

我们先看一下客户端向服务端反馈消息消费进度时如何选择 Broker。

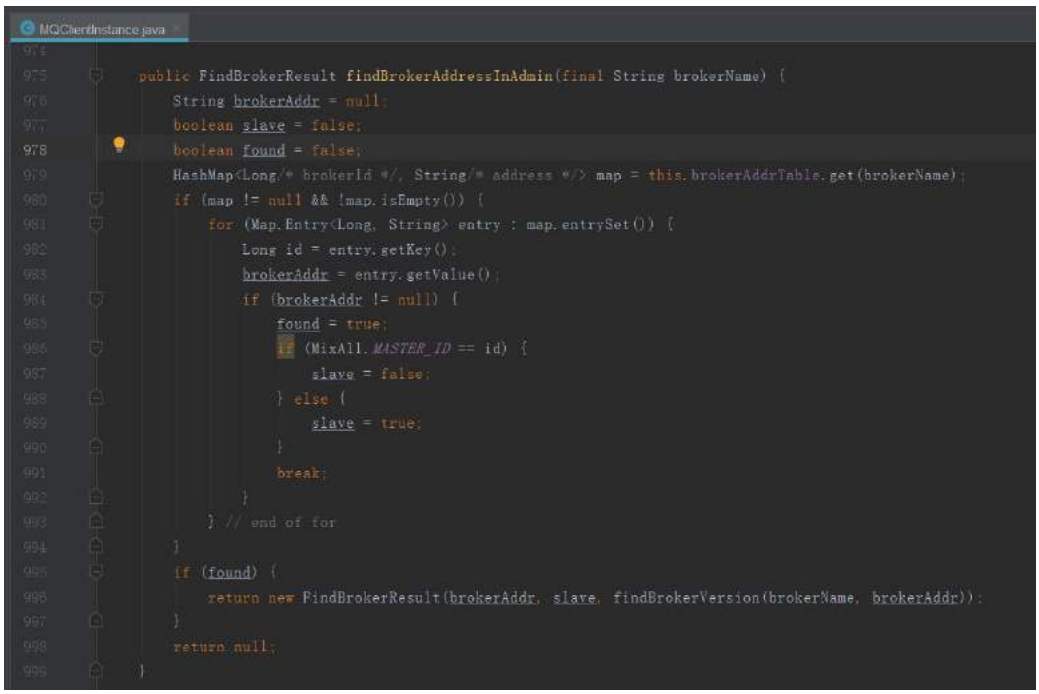
因为主服务的 brokerId 为 0，默认情况下当主服务器存活的时候，优先会选择主服务器，只有当主服务器宕机的情况下，才会选择从服务器。

既然集群模式下消息消费进度存储在 Broker 端，当主服务器正常时，消息消费进度文件存储在主服务器，那提出如下两个问题：

1. 消息消费端在主服务器存活的情况下，会优先向主服务器反馈消息消费进度，那从服务器是如何同步消息消费进度的。
2. 当主服务器宕机后则消息消费端会向从服务器反馈消息消费进度，此时消息消费进度如何存储，当主服务器恢复正常后，主服务器如何得知最新的消息消费进度。

为了解开上述两个疑问，我们优先来看一下 Broker 服务器在收到提交消息消费进度反馈命令后的处理逻辑：

客户端定时向 Broker 端发送更新消息消费进度的请求，其入口为：RemoteBrokerOffsetStore#updateConsumeOffsetToBroker，该方法中一个非常关键的点是：选择 broker 的逻辑，如下所示：



如果主服务器存活，则选择主服务器，如果主服务器宕机，则选择从服务器。也就是说，不管消息是从主服务器拉取的还是从从服务器拉取的，提交消息消费进度请求，优先选择主服务器。服务端就是接收其偏移量，更新到服务端的内存中，然后定时持久化到${ROCKETMQ_HOME}/store/config/consumerOffset.json。

经过上面的分析，我们来讨论一下这个场景：

消息消费者首先从主服务器拉取消息，并向其提交消息消费进度，如果当主服务器宕机后，从服务器会接管消息拉取服务，此时消息消费进度存储在从服务器，主从服务器的消息消费进度会出现不一致？那当主服务器恢复正常后，两者之间的消息消费进度如何同步？

从服务定时同步主服务器进度

```
public class SlaveSynchronize {
    private static final InternalLogger log = InternalLoggerFactory.getLogger(LoggerName.BROKER_LOGGER_NAME);
    private final BrokerController brokerController;
    private volatile String masterAddr = null;

    public SlaveSynchronize(BrokerController brokerController) { this.brokerController = brokerController; }

    public String getMasterAddr() { return masterAddr; }

    public void setMasterAddr(String masterAddr) { this.masterAddr = masterAddr; }

    public void syncAll() {
        this.syncTopicConfig();
        this.syncConsumerOffset();
        this.syncDelayOffset();
        this.syncSubscriptionGroupConfig();
    }
```

如果 Broker 角色为从服务器，会通过定时任务调用 syncAll，从主服务器定时同步 topic 路由信息、消息消费进度、延迟队列处理进度、消费组订阅信息。

那问题来了，如果主服务器启动后，从服务器马上从主服务器同步消息消息进度，那岂不是又要重新消费？

其实在绝大部分情况下，就算从服务从主服务器同步了很久之前的消费进度，只要消息者没有重新启动，就不需要重新消费，在这种情况下，RocketMQ 提供了两种机制来确保不丢失消息消费进度。

第一种，消息消费者在内存中存在最新的消息消费进度，继续以该进度去服务器拉取消息后，消息处理完后，会定时向 Broker 服务器反馈消息消费进度，在上面也提到过，在反馈消息消费进度时，会优先选择主服务器，此时主服务器的消息消费进度就立马更新了，从服务器此时只需定时同步主服务器的消息消费进度即可。

第二种是，消息消费者在向主服务器拉取消息时，如果是是主服务器，在处理消息拉取时，也会更新消息消费进度。

主服务器消息拉取时更新消息消费进度

主服务器在处理消息拉取命令时，会触发消息消费进度的更新，其代码入口为：

```
PullMessageProcessor#processRequest
boolean storeOffsetEnable = brokerAllowSuspend;  // @1
storeOffsetEnable = storeOffsetEnable && hasCommitOffsetFlag;
storeOffsetEnable = storeOffsetEnable
                && this.brokerController.getMessageStoreConfig().getBrokerRole() != BrokerRole.SLAVE;  // @2
if (storeOffsetEnable) {
                this.brokerController.getConsumerOffsetManager().commitOffset(RemotingHelper.parseChannelRemoteAddr(channel),
                        requestHeader.getConsumerGroup(), requestHeader.getTopic(), requestHeader.getQueueId(), requestHeader.getCommitOffset());
    }
```

代码@1：首先介绍几个局部变量的含义：

brokerAllowSuspend：broker 是否允许挂起，在消息拉取时，该值默认为 true。

hasCommitOffsetFlag：消息消费者在内存中是否缓存了消息消费进度，如果缓存了，该标记设置为 true。

如果 Broker 的角色为主服务器，并且上面两个变量都为 true，则首先使用 commitOffset 更新消息消费进度。

看到这里，主从同步消息消费进度的相关问题，应该就有了答案了。

四、总结

上述实现原理的讲解有点枯燥无味，我们先来回答如下几个问题：

问：主，从服务器都在运行过程中，消息消费者是从主拉取消息还是从从拉取?

答：默认情况下，RocketMQ 消息消费者从主服务器拉取，当主服务器积压的消息超过了物理内存的 40%，则建议从从服务器拉取。但如果 slaveReadEnable 为 false，表示从服务器不可读，从服务器也不会接管消息拉取。

问：当消息消费者向从服务器拉取消息后，会一直从从服务器拉取?

答：不是的。分如下情况：

1. 如果从服务器的 slaveReadEnable 设置为 false，则下次拉取，从主服务器拉取。
2. 如果从服务器允许读取并且从服务器积压的消息未超过其物理内存的 40%，下次拉取使用的 Broker 为订阅组的 brokerId 指定的 Broker 服务器，该值默认为 0，代表主服务器。
3. 如果从服务器允许读取并且从服务器积压的消息超过了其物理内存的 40%，下次拉取使用的 Broker 为订阅组的 whichBrokerWhenConsumeSlowly 指定的 Broker 服务器，该值默认为 1，代表从服务器。

问：主从服务消息消费进是如何同步的?

答：消息消费进度的同步时单向的，从服务器开启一个定时任务，定时从主服务器同步消息消费进度；无论消息消费者是从主服务器拉的消息还是从从服务器拉取的消息，在向 Broker 反馈消息消费进度时，优先向主服务器汇报；消息消费者向主服务器拉取消息时，如果消息消费者内存中存在消息消费进度时，主会尝试跟新消息消费进度。

读写分离的正确使用姿势：

1. 主从 Broker 服务器的 slaveReadEnable 设置为 true。
2. 通过 updateSubGroup 命令更新消息组 whichBrokerWhenConsumeSlowly、brokerId，特别是其 brokerId 不要设置为 0，不然从从服务器拉取一次后，下一次拉取就会从主去拉取。

1.5 踩坑记：rocketmq-console 消费 TPS 为 0，但消息积压数却在降低是个什么“鬼”

一、背景

上周六的 19:00，接到项目反馈，他们的项目从昨天的 23:00 就停止消费了，而整个集群没有出现异常，故此种情况更多的是因为项目组的原因，由于业务已积压将近一天，由于项目在昨天 20：00 发过变更，故为了快速恢复业务，项目组首先决定将版本进行回退，回退后通过 rocketmq-console 查看消费组的消费 TPS，却显示为 0，如图所示：

乍一看，第一时间得出应用还未恢复，就开始去查看相关的启动日志,通常查看的是应用服务器的 /home/baseuser/logs/rockemqlogs/rocketmq_client.logs，碰巧又看到如下的错误日志：

```
RebalanceService - [BUG] ConsumerGroup: consumer-grouptest The consumerId: consumer-client-id-clusterA-192.168.3.122@21932 not in cidAll: [consumer-client-id-clusterA-192.168.3.123@22164]
```

上面的日志显示在队列负载时候，当前节点竟然不属于 consumer-grouptest 消费组的活跃连接，导致一大片的报错：

```
2019-11-02 19:29:17 WARN NettyClientPublicExecutor_1 - execute the pull request exception
org.apache.rocketmq.client.exception.MQBrokerException: CODE: 25  DESC: the consumer's subscription not latest
For more information, please visit the url, http://rocketmq.apache.org/docs/faq/
 at org.apache.rocketmq.client.impl.MQClientAPIImpl.processPullResponse(MQClientAPIImpl.java:639)
 at org.apache.rocketmq.client.impl.MQClientAPIImpl.access$200(MQClientAPIImpl.java:156)
 at org.apache.rocketmq.client.impl.MQClientAPIImpl$2.operationComplete(MQClientAPIImpl.java:592)
 at org.apache.rocketmq.remoting.netty.ResponseFuture.executeInvokeCallback(ResponseFuture.java:51)
 at org.apache.rocketmq.remoting.netty.NettyRemotingAbstract$2.run(NettyRemotingAbstract.java:275)
 at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:511)
 at java.util.concurrent.FutureTask.run(FutureTask.java:266)
 at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
 at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
 at java.lang.Thread.run(Thread.java:745)
```

乍一看确实是 rocketmq 相关的问题，导致上述 消费 TPS 为 0，经过半个小时的日志分析，发现这是 RocketMQ 这是一种正常现象，最终会自动恢复，经过日志分析得出 rocketmq 没问题，故后面去查看消息积压，发现消息积压明显在减少，那这就奇了怪了，咋消息积压在快速减少，但为啥消费 TPS 还是为 0 呢?

接下来将该问题进行探讨。

温馨提示：在问题分析部分，作者没有直接给出答案，而是一步一步探寻答案，因此会通过追踪源码来寻求答案，如果大家想急于答案，可以跳过问题分析，直接查看本文末尾的问题解答部分。

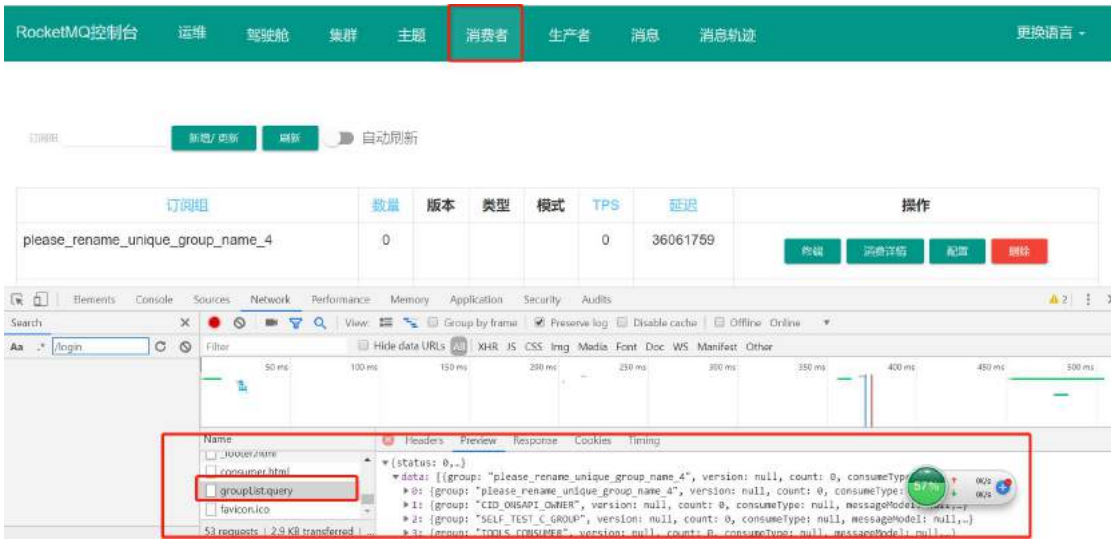
通过本文的阅读，您将获得如下信息：

1. RocketMQ 消费 TPS 的收集与计算逻辑。
2. RocketMQ 监控指标的设计思路。
3. RocketMQ 主从同步，消费者从主服务器拉取还是从从服务器拉取的判断逻辑。

二、问题分析

1. rocketmq-console 数据获获取逻辑探讨

要解开消费 TPS 显示为 0 的问题，我们首先要来看一下 rocketmq-console 这个页面的展示逻辑，即通过阅读 rocketmq-console 的源码来解开其采集逻辑。



得知，【消费者】界面查询各个消费组的基本信息的接口为 /consumer/groupList.query，那接下来，我们首先从源码的角度来分析该接口的实现逻辑。其入口如下：

```
org.apache.rocketmq.console.controller.ConsumerController#list
@RequestMapping(value = "/groupList.query")
@ResponseBody
public Object list() {
    return consumerService.queryGroupList();
}
```

就是调用消费服务处理类的 queryGroupList 方法，其实现代码如下：

```
ConsumerServiceImpl#queryGroupList
public List<GroupConsumeInfo> queryGroupList() {
    Set<String> consumerGroupSet = Sets.newHashSet();
    try {
```

```
            ClusterInfo clusterInfo = mqAdminExt.examineBrokerClusterInfo();      // @1
            for (BrokerData brokerData : clusterInfo.getBrokerAddrTable().values()) {    // @2
                SubscriptionGroupWrapper subscriptionGroupWrapper = mqAdminExt.getAllSubscriptionGroup(brokerData.selectBrokerAddr(), 3000L);  // @3
                consumerGroupSet.addAll(subscriptionGroupWrapper.getSubscriptionGroupTable().keySet());
            }
        } catch (Exception err) {
            throw Throwables.propagate(err);
        }
        List<GroupConsumeInfo> groupConsumeInfoList = Lists.newArrayList();
        for (String consumerGroup : consumerGroupSet) {
            // @4
            groupConsumeInfoList.add(queryGroup(consumerGroup));

        }
        Collections.sort(groupConsumeInfoList);
        return groupConsumeInfoList;
    }
```

代码@1：获取集群的 broker 信息，主要是通过向 NameServer 发送 GET_BROKER_CLUSTER_INFO 请求，NameServer 返回集群包含的所有 broker 信息，包含从节点的信息，返回的格式如下：

```
    "clusterInfo": {
        "brokerAddrTable": {
        "broker-a": {
            "cluster": "DefaultCluster",
              "brokerName": "broker-a",
              "brokerAddrs": {
                    "0": "192.168.0.168:10911",
                    "1": "192.168.0.169:10911"
              }
         },
            "broker-b": {
            "cluster": "DefaultCluster",
              "brokerName": "broker-b",
              "brokerAddrs": {
```

```
                "0": "192.168.0.170:10911",
                "1": "192.168.1.171:10911"
            }
        }
    },
    "clusterAddrTable": {
        "DefaultCluster": ["broker-a","broker-b"]
    }
}
```

代码@2：遍历集群中的 brokerAddrTable 数据结构，即存储了 broker 的地址信息的 Map 。

代码@3：分别向集群中的主节点(brokerData.selectBrokerAddr()) 获取所有的订阅关系（即消费组的订阅信息）。然后将所有的消费者组名称存入 consumerGroupSet。

代码@4：遍历代码@3 收集到的消费组，调用 queryGroup 依次请求消费组的运行时信息，后面接下来详细分析。

接下来将重点分析 queryGroup 方法的实现细节。

```
ConsumerServiceImpl#queryGroup
public GroupConsumeInfo queryGroup(String consumerGroup) {
    GroupConsumeInfo groupConsumeInfo = new GroupConsumeInfo();
    try {
        ConsumeStats consumeStats = null;
        try {
            consumeStats = mqAdminExt.examineConsumeStats(consumerGroup);  //
@1
        } catch (Exception e) {
            logger.warn("examineConsumeStats exception, " + consumerGroup, e);
        }
        ConsumerConnection consumerConnection = null;
        try {
            consumerConnection = mqAdminExt.examineConsumerConnectionInfo(con
sumerGroup);
        } catch (Exception e) {
```

```
                logger.warn("examineConsumerConnectionInfo exception, " + consumerGr
oup, e);
            }
            groupConsumeInfo.setGroup(consumerGroup);

            if (consumeStats != null) {
                groupConsumeInfo.setConsumeTps((int)consumeStats.getConsumeTps());
   // @2
                groupConsumeInfo.setDiffTotal(consumeStats.computeTotalDiff());
          // @3
            }

            if (consumerConnection != null) {
                groupConsumeInfo.setCount(consumerConnection.getConnectionSet().size
());
                groupConsumeInfo.setMessageModel(consumerConnection.getMessageMo
del());
                groupConsumeInfo.setConsumeType(consumerConnection.getConsumeTy
pe());
                groupConsumeInfo.setVersion(MQVersion.getVersionDesc(consumerConne
ction.computeMinVersion()));
            }
        } catch (Exception e) {
            logger.warn("examineConsumeStats or examineConsumerConnectionInfo exce
ption, "
                    + consumerGroup, e);
        }
        return groupConsumeInfo;
    }
```

从上面@1，@2，@3 这三处代码可以得知，rocketmq-console 相关界面上的消费 TPS 主要来自 examineConsumeStats 方法，该方法我就不再继续深入，我们只需找到该方法向 broker 发送的请求编码，然后根据该请求编码找到 broker 的处理逻辑即可，最后跟踪发送的请求编码为：RequestCode.GET_CONSUME_STATS。

GET_CONSUME_STATS 命令在 broker 的处理逻辑如下：

```
    AdminBrokerProcessor#getConsumeStats
    private RemotingCommand getConsumeStats(ChannelHandlerContext ctx, RemotingC
ommand request) throws RemotingCommandException {
        final RemotingCommand response = RemotingCommand.createResponseCom
mand(null);
        final GetConsumeStatsRequestHeader requestHeader =
            (GetConsumeStatsRequestHeader) request.decodeCommandCustomHead
er(GetConsumeStatsRequestHeader.class);
        ConsumeStats consumeStats = new ConsumeStats();
        Set<String> topics = new HashSet<String>();
        if (UtilAll.isBlank(requestHeader.getTopic())) {
            topics = this.brokerController.getConsumerOffsetManager().whichTopicByC
onsumer(requestHeader.getConsumerGroup());
        } else {
            topics.add(requestHeader.getTopic());
        }
        for (String topic : topics) {    // @1
            TopicConfig topicConfig = this.brokerController.getTopicConfigManager().s
electTopicConfig(topic);
            if (null == topicConfig) {  // @2
                log.warn("consumeStats, topic config not exist, {}", topic);
                continue;
            }
            {
                SubscriptionData findSubscriptionData =
                    this.brokerController.getConsumerManager().findSubscriptionData(r
equestHeader.getConsumerGroup(), topic);    // @3
                if (null == findSubscriptionData //
                    && this.brokerController.getConsumerManager().findSubscriptionDa
taCount(requestHeader.getConsumerGroup()) > 0) {
                    log.warn("consumeStats, the consumer group[{}], topic[{}] not exist
", requestHeader.getConsumerGroup(), topic);
                    continue;
                }
            }
            for (int i = 0; i < topicConfig.getReadQueueNums(); i++) {    // @4
                MessageQueue mq = new MessageQueue();
                mq.setTopic(topic);
```

```
        mq.setBrokerName(this.brokerController.getBrokerConfig().getBrokerName());
                        mq.setQueueId(i);
                        OffsetWrapper offsetWrapper = new OffsetWrapper();
                        long brokerOffset = this.brokerController.getMessageStore().getMaxOff
setInQueue(topic, i);
                        if (brokerOffset < 0)
                            brokerOffset = 0;
                        long consumerOffset = this.brokerController.getConsumerOffsetManag
er().queryOffset(//
                            requestHeader.getConsumerGroup(), //
                            topic, //
                            i);
                        if (consumerOffset < 0)
                            consumerOffset = 0;
                        offsetWrapper.setBrokerOffset(brokerOffset);
            // @5
                        offsetWrapper.setConsumerOffset(consumerOffset);
    // @6
                        long timeOffset = consumerOffset - 1;
                        if (timeOffset >= 0) {
                            long lastTimestamp = this.brokerController.getMessageStore().get
MessageStoreTimeStamp(topic, i, timeOffset);
                            if (lastTimestamp > 0) {
                                offsetWrapper.setLastTimestamp(lastTimestamp);
    // @7
                            }
                        }
                        consumeStats.getOffsetTable().put(mq, offsetWrapper);       // @8
                    }
                    double consumeTps = this.brokerController.getBrokerStatsManager().tpsGr
oupGetNums(requestHeader.getConsumerGroup(), topic); // @9
                    consumeTps += consumeStats.getConsumeTps(); // @10
                    consumeStats.setConsumeTps(consumeTps);
                }
                byte[] body = consumeStats.encode();
                response.setBody(body);
                response.setCode(ResponseCode.SUCCESS);
                response.setRemark(null);
                return response;}
```

该方法比较长，重点关注如下关键点：

- 代码@1：遍历该消费组订阅的所有主题。消费 TPS 将是所有主题消费 TPS 的总和，其他的信息按主题、队列信息单独存放。
- 代码@2：如果 topic 的元信息不存在，则跳过该主题。
- 代码@3：如果消费组的订阅信息不存在，则跳过该订阅关系。
- 代码@4：收集该主题所有的读队列，以 messagequeue 为键，OffsetWrapper 为值存储在 consumeStats.getOffsetTable() ，见代码@8。
- 代码@5：设置该队列的最新偏移量。
- 代码@6：设置该消费组对该队列的消费进度，设置为 consumeOffset。
- 代码@7：lastTimestamp 上一次消费的消息的存储时间，实现逻辑为：取消费组对于队列的消息消费进度 -1 的消息，存储在 broker 的时间，如果对应的消息已过期被删除，则在界面上显示的时间就会为 1970-01-01 08:00:00。
- 代码@9：通过 BrokerStatsManager 的 tpsGroupGetNums 方法从统计数据中获取该消费组针对该队列的消费 TPS。
- 代码@10：累积消费 TPS，并最终作为该消费组的总 TPS。

上面这个方法非常关键，是返回给前段页面核心的数据组装逻辑，以队列、消费组为纬度给出 brokerOffset、consumeOffset、lastTimestamp。然后将数据返回给前段页面进行展示。

接下将聚焦到消费组消费 TPS 的统计处理，其入口为 tpsGroupGetNums 。

2. rocketmq 消费 TPS 统计实现原理

消费 TPS 计算逻辑

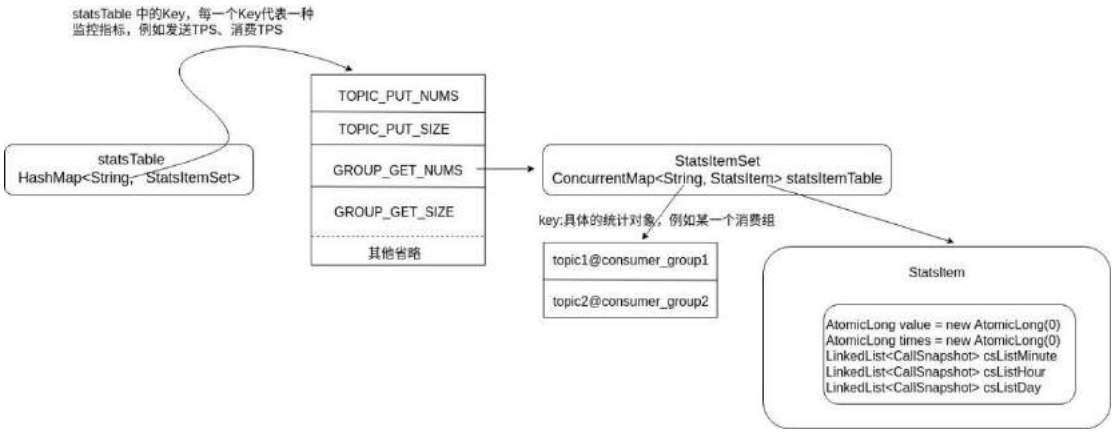
首先我们还是从 tpsGroupGetNums 方法入手，探究一下 tps 的获取逻辑，然后再探究数据的采集原理（这也是 rocketmq 监控相关）。

```
BrokerStatsManager#tpsGroupGetNums
public double tpsGroupGetNums(final String group, final String topic) {
    final String statsKey = buildStatsKey(topic, group); // @1
```

```
        return this.statsTable.get(GROUP_GET_NUMS).getStatsDataInMinute(statsKey).get
Tps(); // @2
    }
```

代码@1：构建统计 key，其逻辑为：其键为：topic@consumerGroup，即消息主题@消费组名。

要读懂 代码@2 的代码，先来看一下 rocketmq 监控指标的存储数据结构，如下图所示：



正如上图所示：RocketMQ 使用 HashMap<String, StatusItemSet> 来存储监控收集的数据，其中 Key 为监控指标的类型，例如 topic 发送消息数量、topic 发送消息大小、消费组获取消息个数等信息，每一项使用 StatsItemSet 存储，该存储结构内部又维护一个 HashMap：ConcurrentMap，key 代表某一个具体的统计目标，例如记录消费组拉取消息的数量监控指标，那其统计的对象即 topic@consumer_group，最终数据的载体是 StatsItem，使用如下几个关键字段来记录统计信息：

- AtomicLong value = new AtomicLong(0)

总数量，统计指标 TOPIC_GET_NUMS 指标为例，记录的是消息拉取的总条数，例如一次消息拉取操作获取了 32 条消息，则该数量增加 32。

- AtomicLong times = new AtomicLong(0)

改变上述 value 的次数，还是以统计指标 TOPIC_GET_NUMS 指标为例，记录的是增加 value 的次数。

- LinkedList<CallSnapshot> csListMinute

一分钟的快照信息，该 List 只会存储 6 个元素，每 10s 记录一次调用快照，超过 6 条，则移除第一条，这个将在下文介绍。

- LinkedList<CallSnapshot> csListHour

一小时的快照信息，该 List 只会存储 6 个元素，每 10 分钟记录一次快照，超过 6 条，则移除第一条。

- LinkedList<CallSnapshot> csListDay

一天的快照新，该 List 只会存储 24 个元素，每 1 小时记录一次快照，超过 24 条，则移除第一条。

了解了上述存储结构后，代码@2，最终其实调用的就是 StatsItemSet 的 getStatsDataInMinute 方法。

```
StatsItemSet#getStatsDataInMinute
public StatsSnapshot getStatsDataInMinute(final String statsKey) {
    StatsItem statsItem = this.statsItemTable.get(statsKey);
    if (null != statsItem) {
        return statsItem.getStatsDataInMinute();
    }
    return new StatsSnapshot();
}
```

从代码上最终调用 StatesItem 的 getStatsDataInMinute 方法。

```
StatesItem#getStatsDataInMinute
public StatsSnapshot getStatsDataInMinute() {
    return computeStatsData(this.csListMinute);
}
private static StatsSnapshot computeStatsData(final LinkedList<CallSnapshot> csList)
{
    StatsSnapshot statsSnapshot = new StatsSnapshot();
    synchronized (csList) {
        double tps = 0;
        double avgpt = 0;
```

```
            long sum = 0;
            if (!csList.isEmpty()) {
                CallSnapshot first = csList.getFirst();    // @1
                CallSnapshot last = csList.getLast();      // @2
                sum = last.getValue() - first.getValue();  // @3
                tps = (sum * 1000.0d) / (last.getTimestamp() - first.getTimestamp());    //
@4
                long timesDiff = last.getTimes() - first.getTimes();
                if (timesDiff > 0) {
                            // @5
                    avgpt = (sum * 1.0d) / timesDiff;
                }
            }

            statsSnapshot.setSum(sum);
            statsSnapshot.setTps(tps);
            statsSnapshot.setAvgpt(avgpt);

        }
        return statsSnapshot;
    }
```

代码@1：首先取快照中的第一条消息。

代码@2：取快照列表中的最后一条消息。

代码@3：计算这两个时间点 value 的差值，即这段时间内新增的总数。

代码@4：计算这段时间内的 tps，即每秒处理的消息条数。

代码@5：计算 avgpt ，即平均一次操作新增的消息条数（即平均一次操作，value 新增的个数）。

消费组的消费 TPS 的计算逻辑就介绍到这里了，那还有一个疑问，即 StatsItem 中 csListMinute 中的数据从哪来呢?

如何采集消费 TPS 原始数据

```
StatsItem#init
public void init() {
  this.scheduledExecutorService.scheduleAtFixedRate(new Runnable() {
            @Override
            public void run() {
                try {
                    samplingInSeconds();
                } catch (Throwable ignored) {
                }
            }
        }, 0, 10, TimeUnit.SECONDS);
    // 省略其他代码
}
```

原来在创建一个新的 StatsItem 的时候，就会启动一个定时任务，每隔 10s 调用 samplingInSeconds 方法进行抽样，那我们简单看一下这个方法：

```
StatsItem#samplingInSeconds
public void samplingInSeconds() {
    synchronized (this.csListMinute) {
        this.csListMinute.add(new CallSnapshot(System.currentTimeMillis(), this.times.get(), this.value
                .get()));
        if (this.csListMinute.size() > 7) {
            this.csListMinute.removeFirst();
        }
    }
}
```

就是将当前 StatsItem 中的 value 与 变更次数(time) 存入封装成 CallSnapshot ，然后存储在快照列表中。这里的关键是 times values 这些值在什么情况下会改变呢?

接着往下看，源码在消息拉取的时候，会将本次拉取的信息加入到统计信息中，其入口为：

```
PullMessageProcessor#processRequest
switch (response.getCode()) {
    case ResponseCode.SUCCESS:
        this.brokerController.getBrokerStatsManager().incGroupGetNums(requestHeade
r.getConsumerGroup(), requestHeader.getTopic(),
                getMessageResult.getMessageCount());
        this.brokerController.getBrokerStatsManager().incGroupGetSize(requestHeader.
getConsumerGroup(), requestHeader.getTopic(),
                getMessageResult.getBufferTotalSize());
        this.brokerController.getBrokerStatsManager().incBrokerGetNums(getMessage
Result.getMessageCount());

        // 省略其他代码
}
```

该方法会最终更新 StatsItem 中的 values ，而 times 是 每调用一次，加 1。

理论基础讲解完毕后，接下来我们来回答一下题目中的现象。

三、问题解答

按照上面的讲解，通过 rocketmq-console 发起查看消费组的 TPS 时，Broker 会根据过去一分钟内采集的快照数据进行计算。快照信息的采集机制是 broker 端会每 10s 会记录一下消费组对应的拉取消息数量与拉取次数。

那既然消息延迟(堆积数量在不断减少)，说明消费端正在消费，按道理来说，通过上述机制进行计算，TPS 不可能会是 0? 那又是什么原因呢? 如果 TPS 为 0，可以说明消费端并没有向 broker 拉取消息，因为一旦从 broker 拉取消息，有关 StatsItem 的 拉取消息总数(value) 与 拉取次数(times) 再两次采集国产中肯定不会相等，只要两者有差距，其 TPS 就不可能为 0，那消费组在消费消息，但又不从主节点上拉取消息，这种情况会出现吗?

答案是会的，在 RocketMQ 主从同步架构中，如果需要访问的消息偏移量与当前 commitlog 最大偏移的之间的差距超过了内存的 40%，消息消费将由从节点接管，故此时消费的拉取不会去主节点拉取，故上面返回的 TPS 就会为 0。这样就能完美解答了。

经过上面的分析，我相信大家已经非常认可这个原因了，其实我们还有一个重要的论据，大家可以分别去查看 Rocketmq 主从节点 /home/{username}/logs/rocketmqlogs/stats.log，里面会每隔 1 分钟在日志中打印各个消费组的消费 TPS，日志如下：

从服务器(rocketmq-slave)对应的日志如下：

```
INFO - [GROUP_GET_NUMS] [orderCenterOrder@bjjdOrderCenterOrderConsumer] Stats In One Minute, SUM: 785717 TPS: 15714.34 AVGPT: 8.14
INFO - [GROUP_GET_NUMS] [orderCenterOrder@bjjdOrderCenterOrderConsumer] Stats In One Minute, SUM: 940522 TPS: 15675.37 AVGPT: 8.06
```

主服务器(rocketmq-master)对应的日志如下：

```
INFO - [GROUP_GET_NUMS] [orderCenterOrder@bjjdOrderCenterOrderConsumer] Stats In One Minute, SUM: 0 TPS: 0.00 AVGPT: 0.00
INFO - [GROUP_GET_NUMS] [orderCenterOrder@bjjdOrderCenterOrderConsumer] Stats In One Minute, SUM: 0 TPS: 0.00 AVGPT: 0.00
```

主服务器上的 TPS 一定会 0 吗？不一定，其实也不一定。这里借着这波日志，再来总结一下 RocketMQ 主从同步时的切换逻辑。

1. 如果消费端请求的消息物理偏移量与 broker 当前最新的物理偏移量之间的差距查过内存的 40%，下一次拉取会往从节点发送（当然前提是 slaveReadEnable = true）。
2. 当从节点开始接管消息消费时，下一次拉取请求一定会往从节点发送码？答案也是不一定：

- 如果待拉取的消息偏移量与从节点最新的物理偏移量之间的差距超过内存的 30%，下一次拉取请求还是会发往从节点。
- 如果待拉取的消息偏移量与从节点最新的物理偏移量之际的差距少于内存的 30%，下一次拉取请求将发送到主节点。

关于 RocketMQ 主从同步若干问题答疑，可以参考笔者的另外一篇文章：https://blog.csdn.net/prestigeding/article/details/93672079 。

1.6 RocketMQ 一个新的消费组初次启动时从何处开始消费呢?

概要：CONSUME_FROM_MAX_OFFSET 可能不是你认为的那样哦？问题驱动、原理分析、提出方案。

一、抛出问题

一个新的消费组订阅一个已存在的 Topic 主题时，消费组是从该 Topic 的哪条消息开始消费呢?

首先翻阅 DefaultMQPushConsumer 的 API 时，setConsumeFromWhere(ConsumeFromWhere consumeFromWhere)API 映入眼帘，从字面意思来看是设置消费者从哪里开始消费，正是解开该问题的”钥匙“。ConsumeFromWhere 枚举类图如下：

- CONSUME_FROM_MAX_OFFSET
  从消费队列最大的偏移量开始消费。

- CONSUME_FROM_FIRST_OFFSET
  从消费队列最小偏移量开始消费。

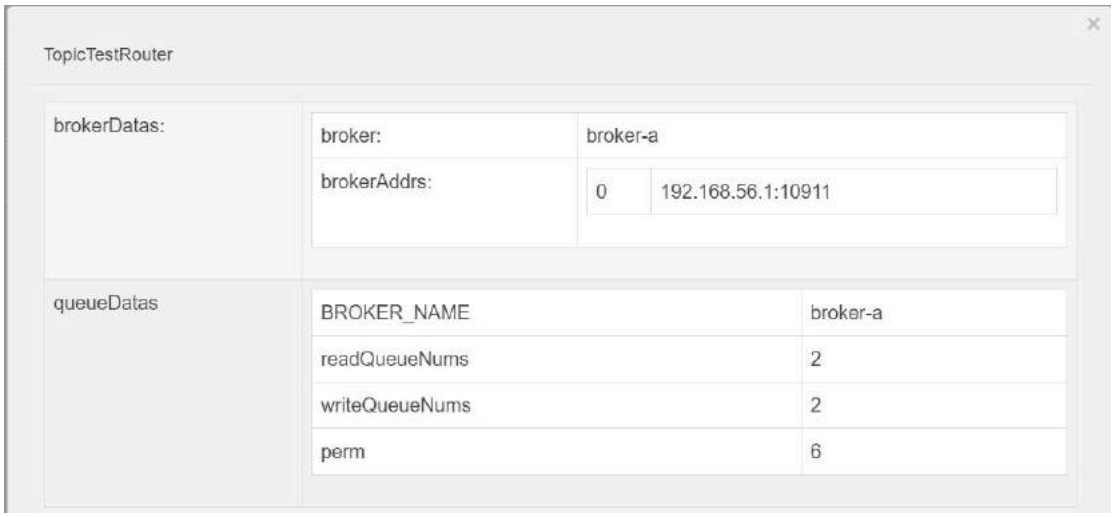
- CONSUME_FROM_TIMESTAMP
  从指定的时间戳开始消费，默认为消费者启动之前的 30 分钟处开始消费。可以通过 DefaultMQPushConsumer#setConsumeTimestamp。

是不是点小激动，还不快试试。

需求：新的消费组启动时，从队列最后开始消费，即只消费启动后发送到消息服务器后的最新消息。

1. 环境准备

本示例所用到的 Topic 路由信息如下：



Broker 的配置如下(broker.conf)：

```
brokerClusterName = DefaultCluster
brokerName = broker-a
brokerId = 0
deleteWhen = 04
fileReservedTime = 48
brokerRole = ASYNC_MASTER
flushDiskType = ASYNC_FLUSH

storePathRootDir=E:/SH2019/tmp/rocketmq_home/rocketmq4.5_simple/store
storePathCommitLog=E:/SH2019/tmp/rocketmq_home/rocketmq4.5_simple/store/commitlog
namesrvAddr=127.0.0.1:9876
autoCreateTopicEnable=false
```

```
mapedFileSizeCommitLog=10240
mapedFileSizeConsumeQueue=2000
```

其中重点修改了如下两个参数:

mapedFileSizeCommitLog

单个 commitlog 文件的大小，这里使用 10M，方便测试用。

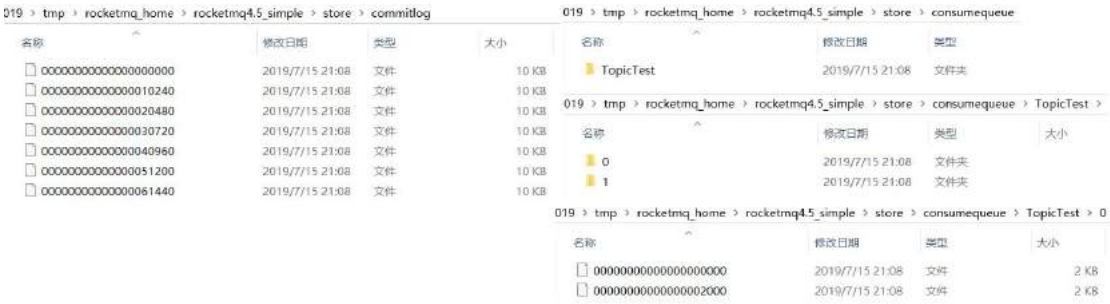
mapedFileSizeConsumeQueue

单个 consumequeue 队列长度，这里使用 1000，表示一个 consumequeue 文件中包含 1000 个条目。

2. 消息发送者代码

```
public static void main(String[] args) throws MQClientException, InterruptedException {
        DefaultMQProducer producer = new DefaultMQProducer("please_rename_unique_group_name");
        producer.setNamesrvAddr("127.0.0.1:9876");
        producer.start();
        for (int i = 0; i < 300; i++) {
            try {
                Message msg = new Message("TopicTest" ,"TagA" , ("Hello RocketMQ " + i).getBytes(RemotingHelper.DEFAULT_CHARSET));
                SendResult sendResult = producer.send(msg);
                System.out.printf("%s%n", sendResult);
            } catch (Exception e) {
                e.printStackTrace();
                Thread.sleep(1000);
            }
        }
        producer.shutdown();
    }
```

通过上述，往 TopicTest 发送 300 条消息，发送完毕后，RocketMQ Broker 存储结构如下:



3. 消费端验证代码

```
    public static void main(String[] args) throws InterruptedException, MQClientException
{
        DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("my_consu
mer_01");
        consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_LAST
_OFFSET);
        consumer.subscribe("TopicTest", "*");
        consumer.setNamesrvAddr("127.0.0.1:9876");
        consumer.registerMessageListener(new MessageListenerConcurrently() {
            @Override
            public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msg
s,
                    ConsumeConcurrentlyContext context) {
                    System.out.printf("%s Receive New Messages: %s %n", Thread.currentTh
read().getName(), msgs);
                    return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
                }
        });
        consumer.start();
        System.out.printf("Consumer Started.%n");
    }
```

执行上述代码后，按照期望，应该是不会消费任何消息，只有等生产者再发送消息后，才会对消息进行消费，事实是这样吗？执行效果如图所示：

令人意外的是，竟然从队列的最小偏移量开始消费了，这就“尴尬”了。难不成是 RocketMQ 的 Bug。带着这个疑问，从源码的角度尝试来解读该问题，并指导我们实践。

二、探究 CONSUME_FROM_MAX_OFFSET 实现原理

对于一个新的消费组，无论是集群模式还是广播模式都不会存储该消费组的消费进度，可以理解为-1,此时就需要根据 DefaultMQPushConsumer#consumeFromWhere 属性来决定其从何处开始消费，首先我们需要找到其对应的处理入口。我们知道，消息消费者从 Broker 服务器拉取消息时，需要进行消费队列的负载，即 RebalanceImpl。

温馨提示：本文不会详细介绍 RocketMQ 消息队列负载、消息拉取、消息消费逻辑，只会展示出通往该问题的简短流程，如想详细了解消息消费具体细节，建议购买笔者出版的《RocketMQ 技术内幕》书籍。

```
    RebalancePushImpl#computePullFromWhere
    public long computePullFromWhere(MessageQueue mq) {
            long result = -1;

  // @1
            final ConsumeFromWhere consumeFromWhere = this.defaultMQPushConsum
erImpl.getDefaultMQPushConsumer().getConsumeFromWhere();
            final OffsetStore offsetStore = this.defaultMQPushConsumerImpl.getOffsetStor
e();
            switch (consumeFromWhere) {
                case CONSUME_FROM_LAST_OFFSET_AND_FROM_MIN_WHEN_BOO
T_FIRST:
                case CONSUME_FROM_MIN_OFFSET:
                case CONSUME_FROM_MAX_OFFSET:
                case CONSUME_FROM_LAST_OFFSET: {
```

```
// @2
                    // 省略部分代码
                    break;
                }
                case CONSUME_FROM_FIRST_OFFSET: {

// @3
                    // 省略部分代码
                    break;
                }
                case CONSUME_FROM_TIMESTAMP: {

//@4
                    // 省略部分代码
                    break;
                }
                default:
                    break;
            }
            return result;

 // @5
    }
```

代码@1：先解释几个局部变量。

- result

  最终的返回结果，默认为-1。

- consumeFromWhere

  消息消费者开始消费的策略，即 CONSUME_FROM_LAST_OFFSET 等。

- offsetStore

  offset 存储器，消费组消息偏移量存储实现器。

代码@2：CONSUME_FROM_LAST_OFFSET(从队列的最大偏移量开始消费)的处理逻辑，下文会详细介绍。

代码@3：CONSUME_FROM_FIRST_OFFSET(从队列最小偏移量开始消费)的处理逻辑，下文会详细介绍。

代码@4：CONSUME_FROM_TIMESTAMP(从指定时间戳开始消费)的处理逻辑，下文会详细介绍。

代码@5：返回最后计算的偏移量，从该偏移量出开始消费。

1. CONSUME_FROM_LAST_OFFSET 计算逻辑

```
    case CONSUME_FROM_LAST_OFFSET: {
        long lastOffset = offsetStore.readOffset(mq, ReadOffsetType.READ_FROM_STOR
E);    // @1
        if (lastOffset >= 0) {
                                                    // @2
            result = lastOffset;
        }
        // First start,no offset
        else if (-1 == lastOffset) {
                                                // @3
            if (mq.getTopic().startsWith(MixAll.RETRY_GROUP_TOPIC_PREFIX)) {

                result = 0L;
            } else {
                try {
                    result = this.mQClientFactory.getMQAdminImpl().maxOffset(mq);

                } catch (MQClientException e) {
                                        // @4
                    result = -1;
                }
            }
        } else {
            result = -1;
        }
     break;
    }
```

代码@1：使用 offsetStore 从消息消费进度文件中读取消费消费进度，本文将以集群模式为例展开。稍后详细分析。

代码@2：如果返回的偏移量大于等于 0，则直接使用该 offset，这个也能理解，大于等于 0，表示查询到有效的消息消费进度，从该有效进度开始消费，但我们要特别留意 lastOffset 为 0 是什么场景，因为返回 0，并不会执行 CONSUME_FROM_LAST_OFFSET(语义)。

代码@3：如果 lastOffset 为-1,表示当前并未存储其有效偏移量，可以理解为第一次消费，如果是消费组重试主题，从重试队列偏移量为 0 开始消费；如果是普通主题，则从队列当前的最大的有效偏移量开始消费，即 CONSUME_FROM_LAST_OFFSET 语义的实现。

代码@4：如果从远程服务拉取最大偏移量拉取异常或其他情况，则使用-1 作为第一次拉取偏移量。

分析，上述执行的现象，虽然设置的是 CONSUME_FROM_LAST_OFFSET，但现象是从队列的第一条消息开始消费，根据上述源码的分析，只有从消费组消费进度存储文件中取到的消息偏移量为 0 时，才会从第一条消息开始消费，故接下来重点分析消息消费进度存储器(OffsetStore)在什么情况下会返回 0。

接下来我们将以集群模式来查看一下消息消费进度的查询逻辑,集群模式的消息进度存储管理器实现为：RemoteBrokerOffsetStore,最终 Broker 端的命令处理类为：ConsumerManageProcessor。

```
    ConsumerManageProcessor#queryConsumerOffset
    private RemotingCommand queryConsumerOffset(ChannelHandlerContext ctx, Remoti
ngCommand request) throws RemotingCommandException {
        final RemotingCommand response =
            RemotingCommand.createResponseCommand(QueryConsumerOffsetRespons
eHeader.class);
        final QueryConsumerOffsetResponseHeader responseHeader =
            (QueryConsumerOffsetResponseHeader) response.readCustomHeader();
        final QueryConsumerOffsetRequestHeader requestHeader =
            (QueryConsumerOffsetRequestHeader) request
```

```
                .decodeCommandCustomHeader(QueryConsumerOffsetRequestHeader.cla
ss);

        long offset =
            this.brokerController.getConsumerOffsetManager().queryOffset(
                requestHeader.getConsumerGroup(), requestHeader.getTopic(), requestHe
ader.getQueueId());      // @1

        if (offset >= 0) {
                                                                    // @2
            responseHeader.setOffset(offset);
            response.setCode(ResponseCode.SUCCESS);
            response.setRemark(null);
        } else {
                                                                        // @3
            long minOffset =
                this.brokerController.getMessageStore().getMinOffsetInQueue(requestHead
er.getTopic(),
                    requestHeader.getQueueId());
                                                                // @4
            if (minOffset <= 0
                && !this.brokerController.getMessageStore().checkInDiskByConsumeOffset
(                                   // @5
                requestHeader.getTopic(), requestHeader.getQueueId(), 0)) {
                responseHeader.setOffset(0L);
                response.setCode(ResponseCode.SUCCESS);
                response.setRemark(null);
            } else {
                                                                    // @6
                response.setCode(ResponseCode.QUERY_NOT_FOUND);
                response.setRemark("Not found, V3_0_6_SNAPSHOT maybe this group
consumer boot first");
            }
        }
        return response;
    }
```

代码@1：从消费消息进度文件中查询消息消费进度。

代码@2：如果消息消费进度文件中存储该队列的消息进度，其返回的 offset 必然会大于等于 0，则直接返回该偏移量该客户端，客户端从该偏移量开始消费。

代码@3：如果未从消息消费进度文件中查询到其进度，offset 为-1。则首先获取该主题、消息队列当前在 Broker 服务器中的最小偏移量(@4)。如果小于等于 0(返回 0 则表示该队列的文件还未曾删除过)并且其最小偏移量对应的消息存储在内存中而不是存在磁盘中，则返回偏移量 0，这就意味着 ConsumeFromWhere 中定义的三种枚举类型都不会生效，直接从 0 开始消费，到这里就能解开其谜团了(@5)。

代码@6：如果偏移量小于等于 0，但其消息已经存储在磁盘中，此时返回未找到，最终 RebalancePushImpl#computePullFromWhere 中得到的偏移量为-1。

看到这里，大家应该能回答文章开头处提到的问题了吧?

看到这里，大家应该明白了，为什么设置的 CONSUME_FROM_LAST_OFFSET，但消费组是从消息队列的开始处消费了吧,原因就是消息消费进度文件中并没有找到其消息消费进度，并且该队列在 Broker 端的最小偏移量为 0，说的更直白点，consumequeue/topicName/queueNum 的第一个消息消费队列文件为 00000000000000000000,并且消息其对应的消息缓存在 Broker 端的内存中(pageCache)，其返回给消费端的偏移量为 0，故会从 0 开始消费，而不是从队列的最大偏移量处开始消费。

为了知识体系的完备性，我们顺便来看一下其他两种策略的计算逻辑。

2. CONSUME_FROM_FIRST_OFFSET

```
    case CONSUME_FROM_FIRST_OFFSET: {
        long lastOffset = offsetStore.readOffset(mq, ReadOffsetType.READ_FROM_STOR
E);    // @1
        if (lastOffset >= 0) {     // @2
            result = lastOffset;
        } else if (-1 == lastOffset) {  // @3
            result = 0L;
        } else {
            result = -1;                          // @4
        }
```

```
        break;
    }
```

从队列的开始偏移量开始消费，其计算逻辑如下：

- 代码@1：首先通过偏移量存储器查询消费队列的消费进度。
- 代码@2：如果大于等于 0，则从当前该偏移量开始消费。
- 代码@3：如果远程返回-1，表示并没有存储该队列的消息消费进度，从 0 开始。
- 代码@4：否则从-1 开始消费。

4. CONSUME_FROM_TIMESTAMP

从指定时戳后的消息开始消费。

```
    case CONSUME_FROM_TIMESTAMP: {
        ong lastOffset = offsetStore.readOffset(mq, ReadOffsetType.READ_FROM_STOR
E);    // @1
        if (lastOffset >= 0) {
                                                    // @2
            result = lastOffset;
        } else if (-1 == lastOffset) {
                                                  // @3
            if (mq.getTopic().startsWith(MixAll.RETRY_GROUP_TOPIC_PREFIX)) {
                try {
                    result = this.mQClientFactory.getMQAdminImpl().maxOffset(mq);
                } catch (MQClientException e) {
                    result = -1;
                }
            } else {
                try {
                    long timestamp = UtilAll.parseDate(this.defaultMQPushConsumerImpl.g
etDefaultMQPushConsumer().getConsumeTimestamp(),
                            UtilAll.YYYYMMDDHHMMSS).getTime();
                    result = this.mQClientFactory.getMQAdminImpl().searchOffset(mq, time
stamp);
                } catch (MQClientException e) {
                    result = -1;
```

```
                }
            }
        } else {
            result = -1;
        }
        break;
    }
```

其基本套路与 CONSUME_FROM_LAST_OFFSET 一样：

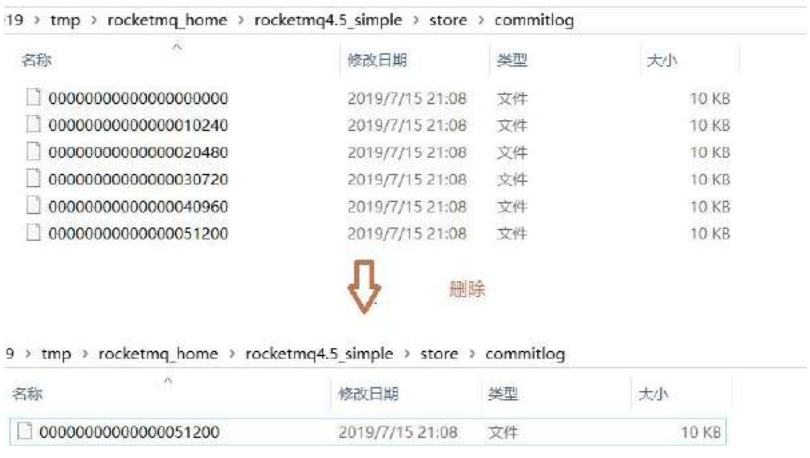
- 代码@1：首先通过偏移量存储器查询消费队列的消费进度。
- 代码@2：如果大于等于 0，则从当前该偏移量开始消费。
- 代码@3：如果远程返回-1，表示并没有存储该队列的消息消费进度，如果是重试主题，则从当前队列的最大偏移量开始消费，如果是普通主题，则根据时间戳去 Broker 端查询，根据查询到的偏移量开始消费。

原理就介绍到这里，

三、猜想与验证

根据上述理论分析我们得知设置 CONSUME_FROM_LAST_OFFSET 但并不是从消息队列的最大偏移量开始消费的“罪魁祸首”是因为消息消费队列的最小偏移量为 0，如果不为 0，则就会符合预期，我们来验证一下这个猜想。

首先我们删除 commitlog 目录下的文件，如图所示：

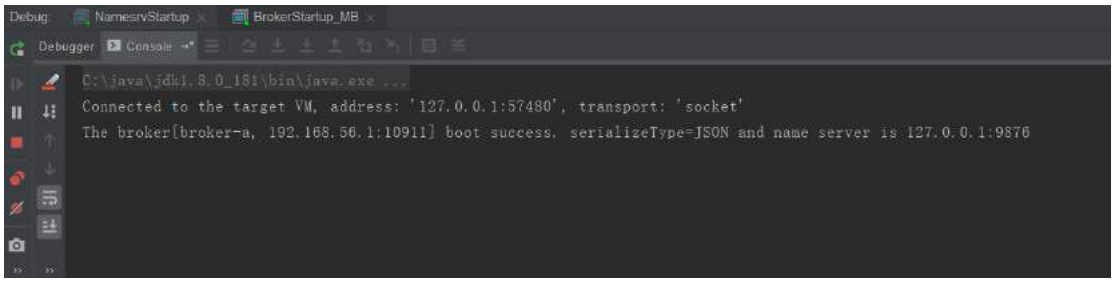


其消费队列截图如下：

消费端的验证代码如下：

```
    public static void main(String[] args) throws InterruptedException, MQClientException
{
        DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("my_consu
mer_02");
        consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_LAST
_OFFSET);
        consumer.subscribe("TopicTest", "*");
        consumer.setNamesrvAddr("127.0.0.1:9876");
        consumer.registerMessageListener(new MessageListenerConcurrently() {
            @Override
            public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msg
s,
                    ConsumeConcurrentlyContext context) {
                    System.out.printf("%s Receive New Messages: %s %n", Thread.currentTh
read().getName(), msgs);
                    return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
                }
        });
        consumer.start();
        System.out.printf("Consumer Started.%n");
    }
```

运行结果如下：



并没有消息存在的消息，符合预期。

四、解决方案

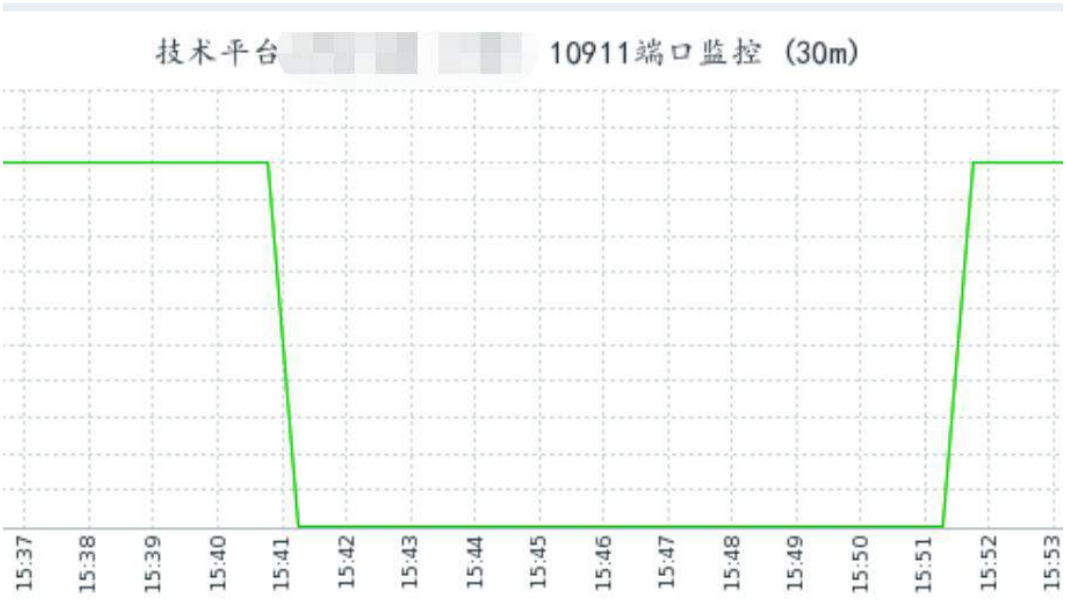
如果在生产环境下，一个新的消费组订阅一个已经存在比较久的 topic，设置 CONSUME_FROM_MAX_OFFSET 是符合预期的，即该主题的 consumequeue/{queueNum}/fileName，fileName 通常不会是 00000000000000000000，如是上述文件名，想要实现从队列的最后开始消费，该如何做呢？那就走自动创建消费组的路子，执行如下命令：

```
./mqadmin updateSubGroup -n 127.0.0.1:9876 -c DefaultCluster -g my_consumer_05

//克隆一个订阅了该 topic 的消费组消费进度
./mqadmin cloneGroupOffset -n 127.0.0.1:9876 -s my_consumer_01 -d my_consumer_05 -t TopicTest

//重置消费进度到当前队列的最大值
./mqadmin resetOffsetByTime -n 127.0.0.1:9876 -g my_consumer_05 -t TopicTest -s -1

// 最后就启动消费者，从队列最大偏移量开始消费。
```

1.7 一次 RocketMQ 进程自动退出排查经验分享

一、背景

公司一个 RocketMQ 集群由 4 主 4 从组成，突然其中 3 台服务器“竟然”在同一时间下线，其监控显示如下：



三台机器的图形，时间戳几乎完美“吻合”。

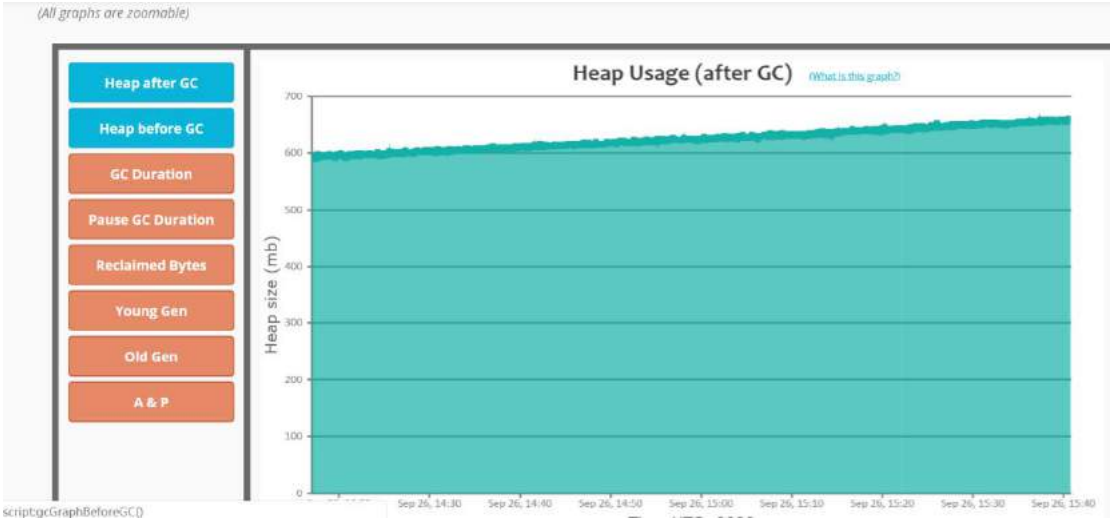
二、故障分析

出现问题，先二话不说，马上重启各服务器，尽快恢复集群，降低对业务的影响，接下来开始对日志进行分析。

Java 进程自动退出(rocketmq 本身就是一个 java 进程)，一种最常见的问题是由于内存溢出或由于内存泄漏导致进程发送 Crash 等。由于我们的启动参数中未配置-XX:

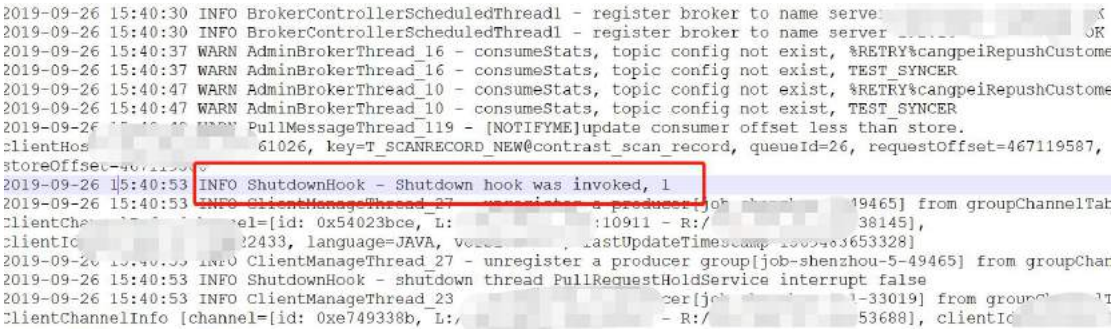
+HeapDumpOnOutOfMemoryError
-XX:HeapDumpPath=/opt/jvmdump

这两个参数，不能直接根据 是否生成 dump 文件，那退而求其次去查看其 GC 日志，将 GC 日志下载到本地，然后可以使用一个在线 gc 日志分析工具：https://gceasy.io/ ，将 gc 日志上传后会给出图形化的展示，其图如下：



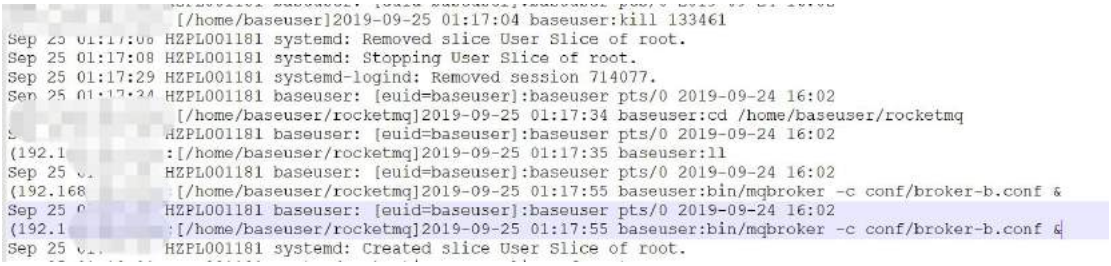
发现垃圾回收很正常。

既然 Java 进程不是由于内存溢出等问题导致的退出，那又会是什么原因呢？那我们来看一下那个点的 broker 的日志，其关键日志截图如下：



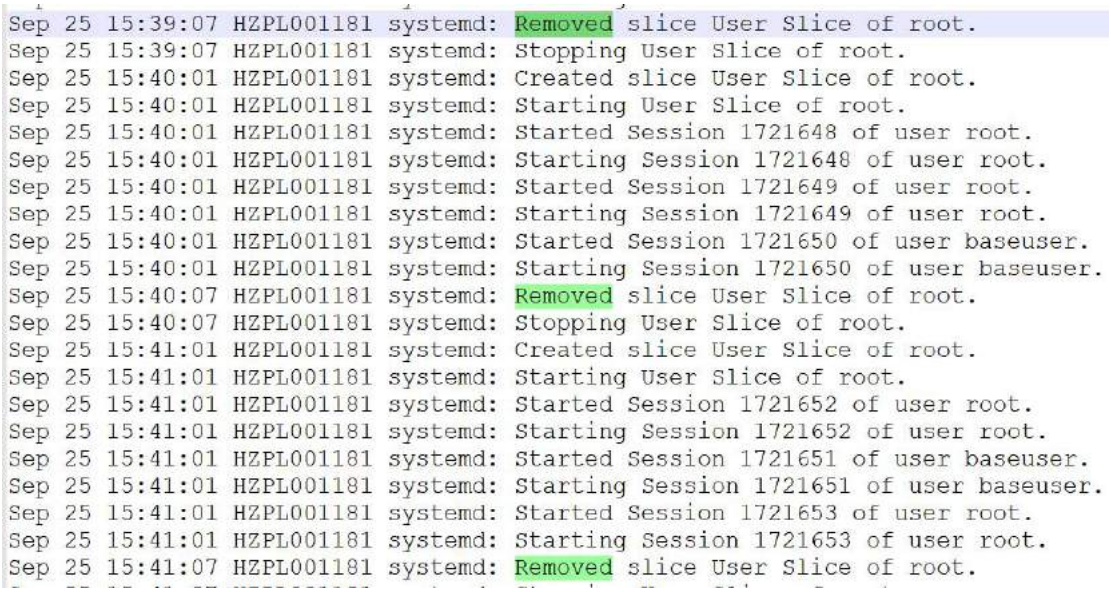
发现 broker 日志中有打印出 shutdownHook，表示在进程退出之前执行了启动时注册时的退出钩子函数，说明是 broker 是正常停止的，并且也不可能是 kill -9 命令，肯定是显示的执行了 kill 命令，于是立马使用 history 命令 查看历史命令，都未在指定时间执行过该命令，并且切换到 root 命令后，同样使用 history 命令，并未发现端倪。

但我始终相信，肯定是执行了手动执行了 kill 命令导致进程退出的，经过网上查找查，得知可以通过查阅系统日志/var/log/messages 来查看系统命令的调用，于是乎把日志文件下载到本地，开始搜索 kill 关键字，发现如下日志：



发现最近一次 kill 命令是在 25 号的凌晨 1 点多，停止 rocketmq 集群，并使用 bin/mqbroker -c conf/broker-b.conf & 进行了重新启动。

这个命令是有问题的，没有使用 nohup ，如果会话失效，该进程就会被退出，为了验证，我们再查一下进程退出时的日志：



发现在故障发生点确实有 Removed 相关的日志。

故障原因基本分析到位了，运维在启动的时候没有使用 nohup 来启动，故马上排查刚启动的集群的方式，重新重启刚启动的 Broker。

RocketMQ 优雅重启小建议：

首先将 broker 的写权限关闭，命令如下：

```
bin/mqadmin updateBrokerConfig -b 192.168.x.x:10911 -n 192.168.x.x:9876 -k brokerPermission -v 4
```

通过 rocketmq-console 查看 该 broker 的写入 TPS，当写入 TPS 降为 0 后，再使用 kill pid 关闭 rocketmq 进程。温馨提示：将 broker 的写权限关闭后，非顺序消息不会立马拒绝，而是需要等客户端路由信息更新后，不会在往该 broker 上发送消息，故这个过程需要等待。

三、启动 rocketmq

```
nohup bin/mqbroker -c conf/broker-a.conf /dev/null 2>&1 &
```

注意：nohup。

四、恢复该节点的写权限

```
bin/mqadmin updateBrokerConfig -b 192.168.x.x:10911 -n 192.168.x.x:9876 -k brokerPermission -v 6
```

1.8 RocketMQ 主题扩分片后遇到的坑

推荐语：RocketMQ 分片扩容后部分队列中的数据无法消费?

消息组 接到某项目组反馈，topic 在扩容后出现部分队列无法被消费者，导致消息积压，影响线上业务?

考虑到该问题是发送在真实的线上环境，为了避免泄密，本文先在笔者的虚拟机中来重现问题。

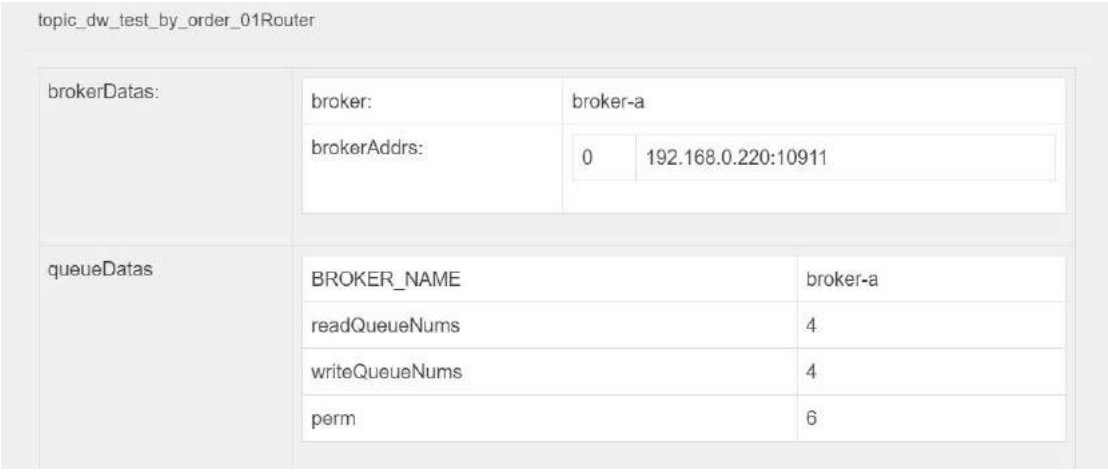
一、案情回顾

1. 集群现状

集群信息如下：

Broker	NO.	Address	Version	Produce Massage TPS	Consumer Massage TPS	Yesterday Produce Count	Yesterday Consume Count	Today Produce Count	Today Consume Count	Operation
broker-b	0(master)	192.168.0.221:10911	V4_5_2	0.00	0.00	0	0	0	0	STATUS CONFIG
broker-a	0(master)	192.168.0.220:10911	V4_5_2	0.00	0.00	0	0	100	100	STATUS CONFIG

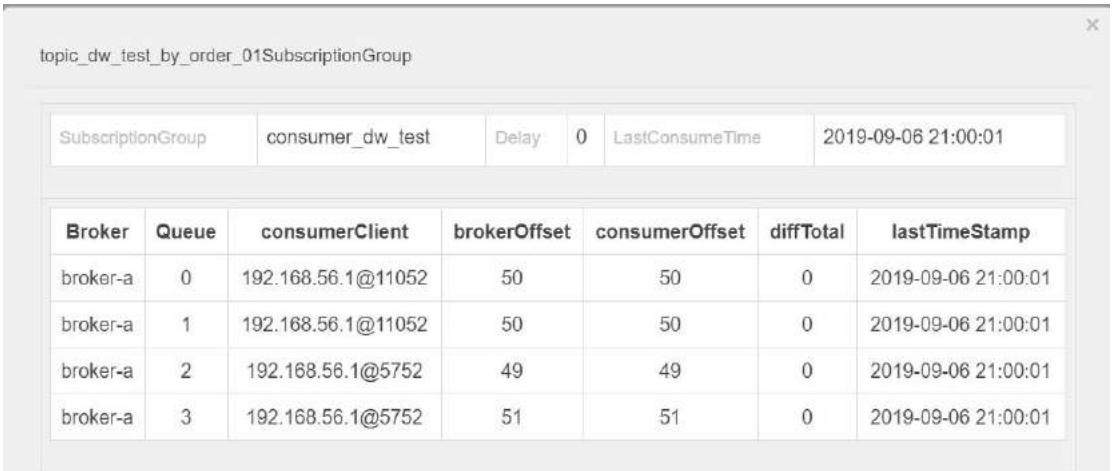
例如业务主体名 topic_dw_test_by_order_01 的路由信息如图所示：



topic_dw_test_by_order_01Router

brokerDatas:	broker:	broker-a
	brokerAddrs:	0 192.168.0.220:10911
queueDatas	BROKER_NAME	broker-a
	readQueueNums	4
	writeQueueNums	4
	perm	6

当前的消费者信息：



topic_dw_test_by_order_01SubscriptionGroup

SubscriptionGroup	consumer_dw_test	Delay	0	LastConsumeTime	2019-09-06 21:00:01

Broker	Queue	consumerClient	brokerOffset	consumerOffset	diffTotal	lastTimeStamp
broker-a	0	192.168.56.1@11052	50	50	0	2019-09-06 21:00:01
broker-a	1	192.168.56.1@11052	50	50	0	2019-09-06 21:00:01
broker-a	2	192.168.56.1@5752	49	49	0	2019-09-06 21:00:01
broker-a	3	192.168.56.1@5752	51	51	0	2019-09-06 21:00:01

broker 的配置信息如下：

```
brokerClusterName = DefaultCluster
brokerName = broker-a
brokerId = 0
deleteWhen = 04
fileReservedTime = 48
brokerRole = ASYNC_MASTER
flushDiskType = ASYNC_FLUSH
brokerIP1=192.168.0.220
brokerIP2-192.168.0.220
namesrvAddr=192.168.0.221:9876;192.168.0.220:9876
storePathRootDir=/opt/application/rocketmq-all-4.5.2-bin-release/store
storePathCommitLog=/opt/application/rocketmq-all-4.5.2-bin-release/store/commitlog
autoCreateTopicEnable=false
autoCreateSubscriptionGroup=false
```

备注：公司对 topic、消费组进行了严格的管控，项目组需要使用时需要向运维人员申请，故 broker 集群不允许自动创建主题与自动创建消费组。

由于该业务量稳步提升，项目组觉得该主题的队列数太少，不利于增加消费者来提高其消费能力，故向运维人员提出增加队列的需求。

2. RocketMQ 在线扩容队列

运维通过公司自研的消息运维平台，直接以指定集群的方式为 topic 扩容，该运维平台底层其实使用了 RocketMQ 提供的 updateTopic 命令，其命令说明如下：

9.7.2.1 创建或更新主题(UpdateTopic)

1、实现类：org.apache.rocketmq.tools.command.topic.UpdateTopicSubCommand

2、参数一览表

参数名称	是否必填	说明
-n	是	nameserver 地址。
-h	否	打印命令帮助信息。
-b	-b、-c 必须一个不为空	broker 地址，表示主题只在指定的 Broker 服务器上创建。
-c	-b、-c 必须一个不为空	broker 集群名称，如果-b 不为空该参数不生效。会依次从 NameServer 根据集群名称获取集群下所有 Master。
-t	是	主题名称。
-r	否	读队列个数，默认 4 个。
-w	否	写队列个数，默认 4 个。
-p	否	队列权限，默认为 6，表示可读可写。
-o	否	是否是顺序消息主题，默认为 false。

表 9-1 updateTopic 命令参数一览表

3、实现概述

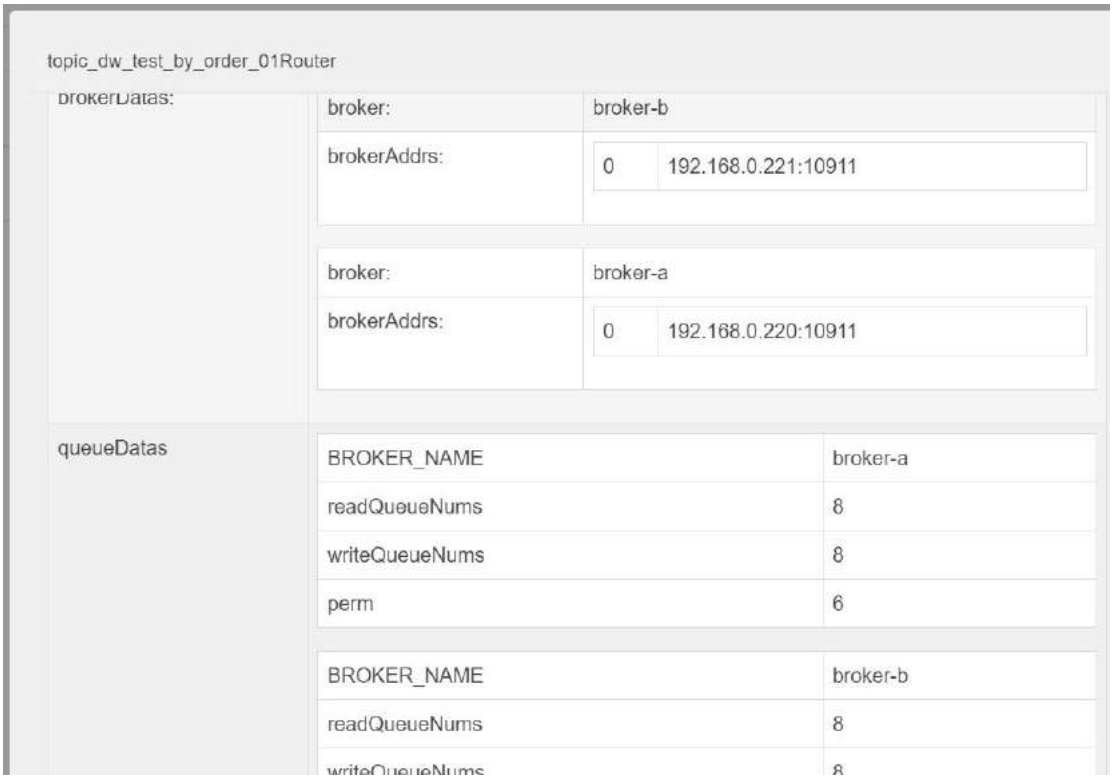
从上图可以得知可以通过 -c 命令来指定在集群中所有的 broker 上创建队列，在本例中，将队列数从 4 设置为 8，具体命令如下：

```
sh ./mqadmin upateTopic -n 192.168.0.220:9876 -c DefaultCluster -t topic_dw_test_by_order_01 -r 8 -w 8
```

执行效果如图所示，表示更新成功。

```
[root@localhost bin]# sh ./mqadmin updateTopic -n 192.168.0.220:9876 -c DefaultCluster -t topic_dw_test_by_order_01 -r 8 -w 8
RocketMQLog:WARN No appenders could be found for logger (io.netty.util.internal.PlatformDependent0).
RocketMQLog:WARN Please initialize the logger system properly.
create topic to 192.168.0.220:10911 success.
create topic to 192.168.0.221:10911 success.
TopicConfig [topicName=topic_dw_test_by_order_01, readQueueNums=8, writeQueueNums=8, perm=RW-, topicFilterType=SINGLE_TAG, topicSysFlag=0, order=false]
```

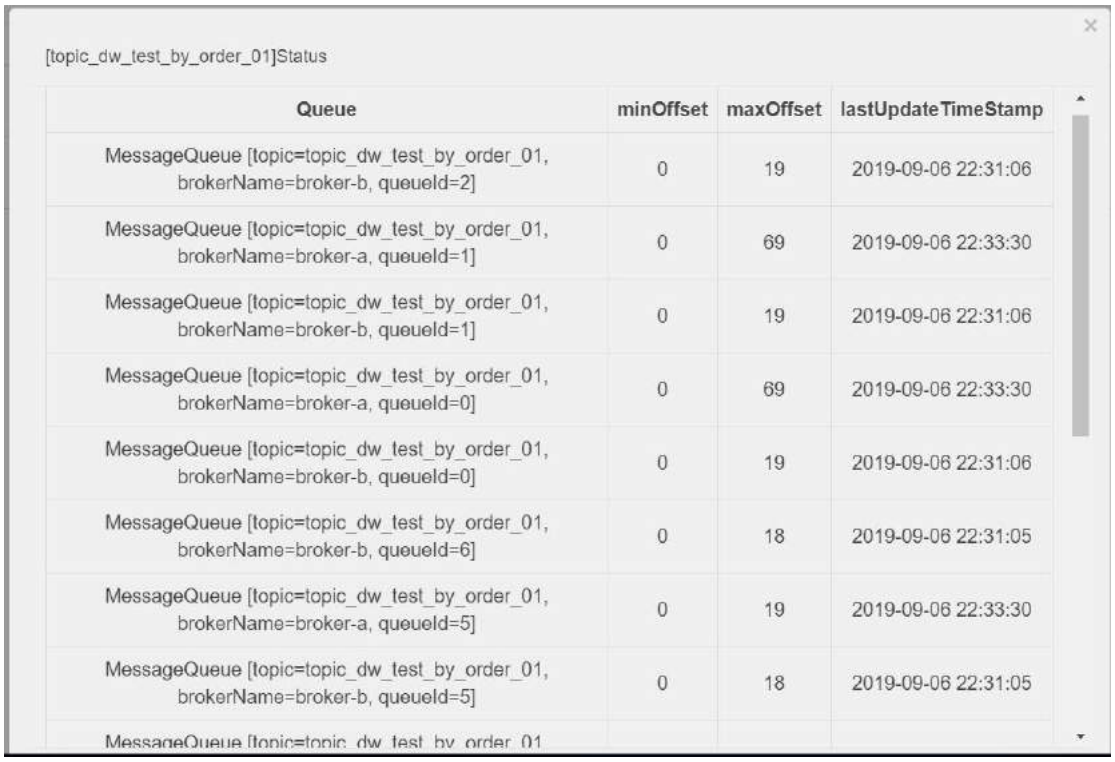
我们再来从 rocketmq-console 中来看命令执行后的效果：



从上图可以得知，主题的队列数已经扩容到了 8 个，并且在集群的两台 broker 上都创建了队列。

3. 消息发送

从 RocketMQ 系列可知，RocketMQ 是支持在线 topic 在线扩容机制的，故无需重启 消息发送者、消息消费者，随着时间的推移，我们可以查看 topic 的所有队列都参与到了消息的负载中，如图所示：



[topic_dw_test_by_order_01]Status

Queue	minOffset	maxOffset	lastUpdateTimeStamp
MessageQueue [topic=topic_dw_test_by_order_01, brokerName=broker-b, queueId=2]	0	19	2019-09-06 22:31:06
MessageQueue [topic=topic_dw_test_by_order_01, brokerName=broker-a, queueId=1]	0	69	2019-09-06 22:33:30
MessageQueue [topic=topic_dw_test_by_order_01, brokerName=broker-b, queueId=1]	0	19	2019-09-06 22:31:06
MessageQueue [topic=topic_dw_test_by_order_01, brokerName=broker-a, queueId=0]	0	69	2019-09-06 22:33:30
MessageQueue [topic=topic_dw_test_by_order_01, brokerName=broker-b, queueId=0]	0	19	2019-09-06 22:31:06
MessageQueue [topic=topic_dw_test_by_order_01, brokerName=broker-b, queueId=6]	0	18	2019-09-06 22:31:05
MessageQueue [topic=topic_dw_test_by_order_01, brokerName=broker-a, queueId=5]	0	19	2019-09-06 22:33:30
MessageQueue [topic=topic_dw_test_by_order_01, brokerName=broker-b, queueId=5]	0	18	2019-09-06 22:31:05
MessageQueue [topic=topic_dw_test_by_order_01			

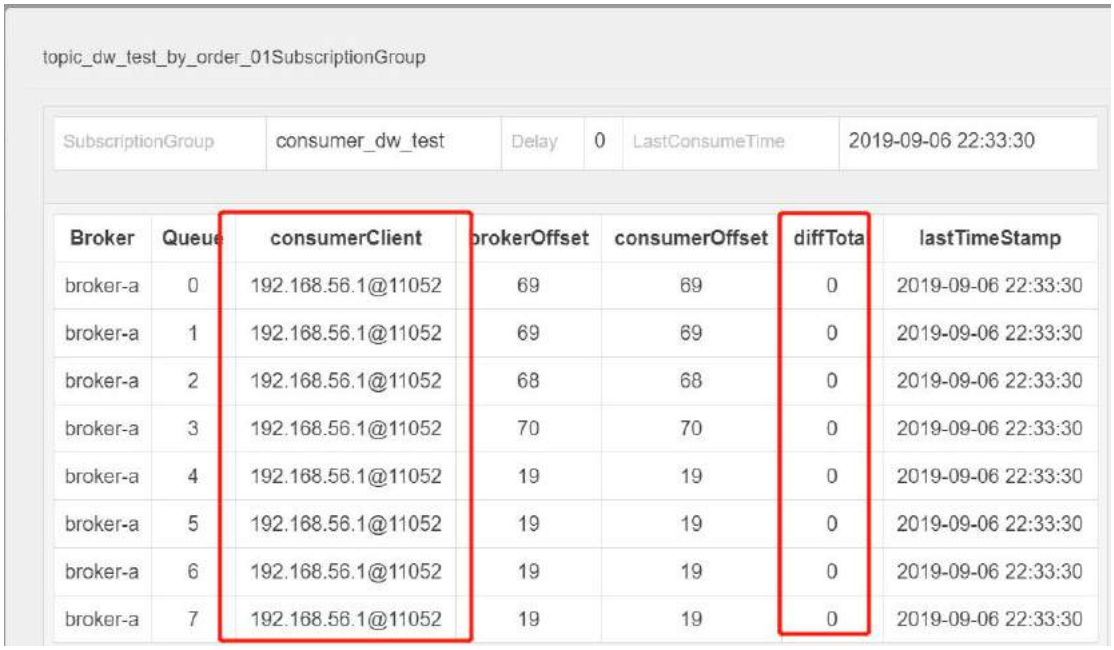
我们可以清晰的看到，所有的 16 个队列(每个 broker 8 个队列)都参与到了消息发送的，运维小哥愉快的完成了 topic 的扩容。

二、问题暴露

该 topic 被 5 个消费组所订阅，突然接到通知，其中有两个消费组反馈，部分队列的消息没有被消费，导致下游系统并没有及时推动项目。

三、问题分析

当时到项目组提交到消息组时，我第一反应是先看消费者的队列，打开该主题的消费情况，如图所示：



topic_dw_test_by_order_01SubscriptionGroup

SubscriptionGroup	consumer_dw_test	Delay	0	LastConsumeTime	2019-09-06 22:33:30

Broker	Queue	consumerClient	brokerOffset	consumerOffset	diffTota	lastTimeStamp
broker-a	0	192.168.56.1@11052	69	69	0	2019-09-06 22:33:30
broker-a	1	192.168.56.1@11052	69	69	0	2019-09-06 22:33:30
broker-a	2	192.168.56.1@11052	68	68	0	2019-09-06 22:33:30
broker-a	3	192.168.56.1@11052	70	70	0	2019-09-06 22:33:30
broker-a	4	192.168.56.1@11052	19	19	0	2019-09-06 22:33:30
broker-a	5	192.168.56.1@11052	19	19	0	2019-09-06 22:33:30
broker-a	6	192.168.56.1@11052	19	19	0	2019-09-06 22:33:30
broker-a	7	192.168.56.1@11052	19	19	0	2019-09-06 22:33:30

发现队列数并没有积压，备注（由于生产是 4 主 4 从，每一个 broker 上 8 个队列，故总共 32 个队列），当时由于比较急，并没有第一时间发现这个界面，竟然只包含一个消费者，觉得并没有消息积压，又由于同一个集群，其他消费组没有问题，只有两个消费组有问题，怀疑是应用的问题，就采取了重启，打印线程栈等“老路”？

事后诸葛亮：其实这完成是错误的，为什么这样说呢？因为项目组（业务方）已经告知一部分业务未处理，说明肯定有队列的消息积压，当根据自己的知识，结合看到的监控页面做出的判断与业务方反馈的出现冲突时，一定是自己的判断出了问题。

正在我们“如火如荼”的认定是项目有问题时，这时我的领导肖工提出了自己的观点，原来在得到业务方反馈时，他得知同一个主题，被 5 个消费组订阅，只有其中两个有问题，那他通过 rocketmq-console 来找两者的区别，找到区别，找到规律，就离解决问题的路近了。

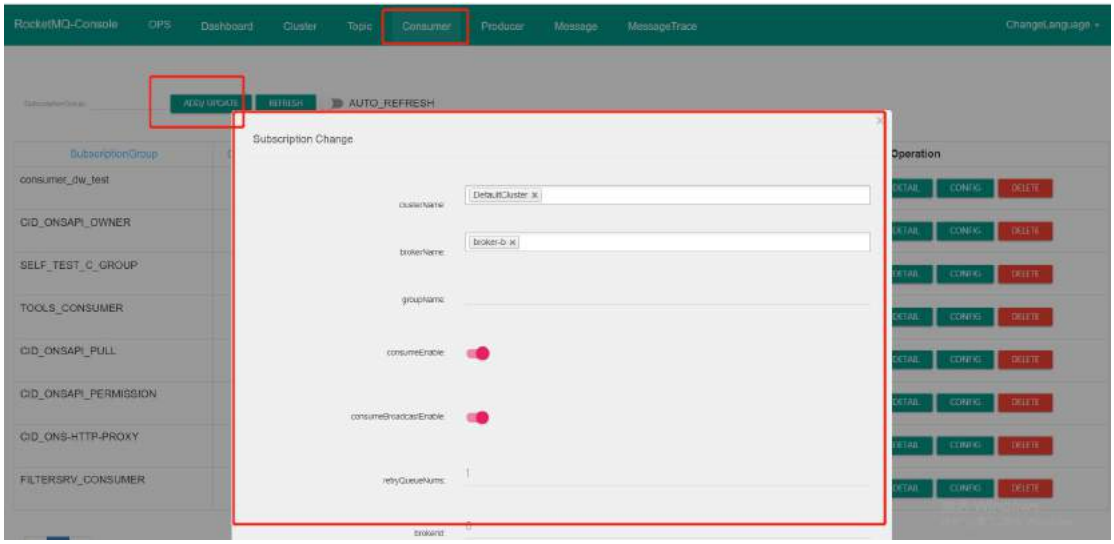
他通过对比发现，出问题的消费组只有两个客户端在消费（通常生产环境是 4 节点消费），而没有出现问题的发现有 4 个进程都在处理，即发现现象：出错的消费组，并没有全员参与到消费。正如上面的图所示：只有其中一个进程在处理 8 个队列，另外 8 个队列并没有在消费。

那现在就是要分析为啥 topic 共有 16 个队列，但这里只有 1 个消费者队列在消费，另外一个消费者不作为?

首先根据 RocketMQ 消息队列负载机制，2 个消费者，只有 1 个消费者在消费，并且一个有一个明显的特点是，只有 broker-a 上的队列在消费，broker-b 上的队列一个也没消费。

正在思考为啥会出现这种现象时，我的领导肖工又在思考是不是集群是不是 broker-b (对应我们生产环境是 broker-c、broker-d 上的队列都未消费)是新扩容的机器? 扩容的时候是不是没有把订阅关系在新的集群上创建? 提出了疑问，接下来肖工就开始验证猜想，通过查阅 broker-c、broker-d 在我们系统中创建的时间是 2018-7 月的时候，就基本得出结论，扩容时并没有在新集群上创建订阅消息，故无法消费消息。

然后运维小哥，根据肖工的建议，创建订阅组，创建方法如图所示:



创建好消费组后，再去查看 topic 的消费情况时，另外一个消费组也开始处理消息了，如下图所示:

SubscriptionGroup	consumer_dw_test	Delay	0	LastConsumeTime	2019-09-06 22:33:30

Broker	Queue	consumerClient	brokerOffset	consumerOffset	diffTotal	lastTimeStamp
broker-a	0	192.168.56.1@11052	69	69	0	2019-09-06 22:33:30
broker-a	1	192.168.56.1@11052	69	69	0	2019-09-06 22:33:30
broker-a	2	192.168.56.1@11052	68	68	0	2019-09-06 22:33:30
broker-a	3	192.168.56.1@11052	70	70	0	2019-09-06 22:33:30
broker-a	4	192.168.56.1@11052	19	19	0	2019-09-06 22:33:30
broker-a	5	192.168.56.1@11052	19	19	0	2019-09-06 22:33:30
broker-a	6	192.168.56.1@11052	19	19	0	2019-09-06 22:33:30
broker-a	7	192.168.56.1@11052	19	19	0	2019-09-06 22:33:30
broker-b	0	192.168.56.1@5752	19	19	0	2019-09-06 22:31:06
broker-b	1	192.168.56.1@5752	19	19	0	2019-09-06 22:31:06
broker-b	2	192.168.56.1@5752	19	19	0	2019-09-06 22:31:06
broker-b	3	192.168.56.1@5752	19	19	0	2019-09-06 22:31:06
broker-b	4	192.168.56.1@5752	18	18	0	2019-09-06 22:31:05
broker-b	5	192.168.56.1@5752	18	18	0	2019-09-06 22:31:05
broker-b	6	192.168.56.1@5752	18	18	0	2019-09-06 22:31:05
broker-b	7	192.168.56.1@5752	18	18	0	2019-09-06 22:31:05

四、问题复盘

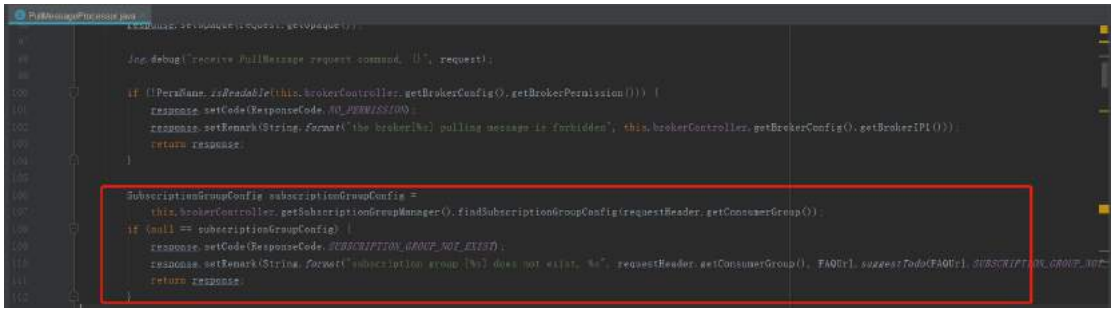
潜在原因：DefaultCluster 集群进行过一次集群扩容，从原来的一台消息服务器(broker-a)额外增加一台 broker 服务器(broker-b)，但扩容的时候并没有把原先的存在于 broker-a 上的主题、消费组扩容到 broker-b 服务器。

触发原因：接到项目组的扩容需求，将集群队列数从 4 个扩容到 8 个，这样该 topic 就在集群的 a、b 都会存在 8 个队列，但 Broker 不允许自动创建消费组（订阅关系），消费者无法从 broker-b 上队列上拉取消息，导致在 broker-b 队列上的消息堆积，无法被消费。

解决办法：运维通过命令，在 broker-b 上创建对应的订阅消息，问题解决。

经验教训：集群扩容时，需要同步在集群上的 topic.json、subscriptionGroup.json 文件。

RocketMQ 理论基础，消费者向 Broker 发起消息拉取请求时，如果 broker 上并没有存在该消费组的订阅消息时，如果不允许自动创建(autoCreateSubscriptionGroup 设置为 false)，默认为 true，则不会返回消息给客户端，其代码如下：



问题解决后，我们的领导也分享了一下他在本次排查问题的思路：出现问题的规律、推断问题、 然后验证问题。规律可以是问题本身的规律　也可以是和正常对比的差。

1.9 RocketMQ 消息发送 system busy、broker busy 原因分析与解决方案坑

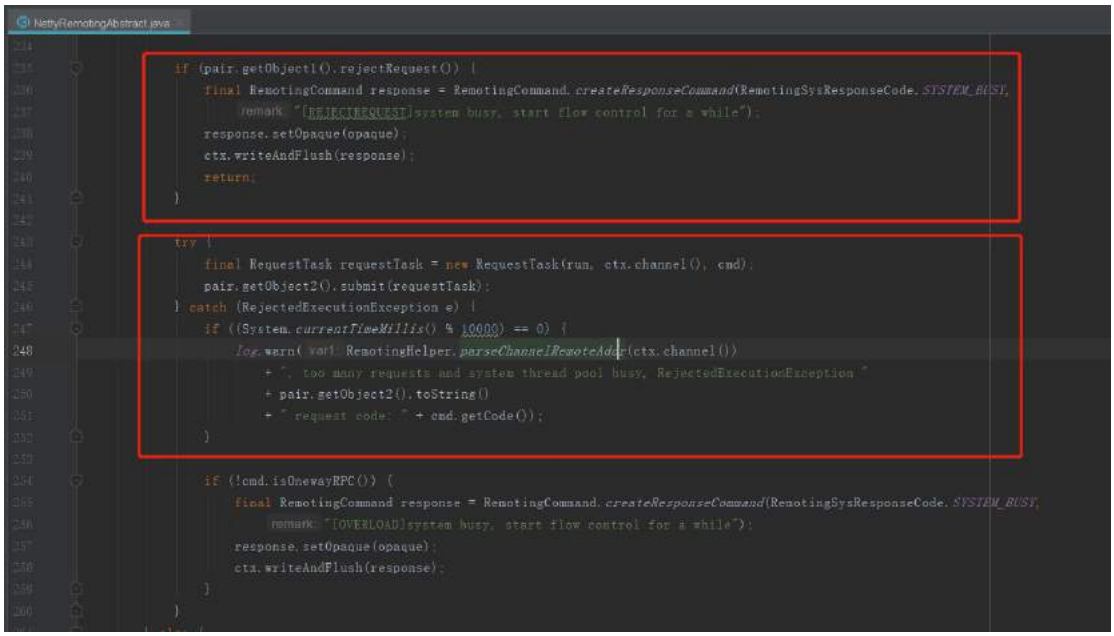
一、现象

最近收到很多 RocketMQ 使用者，反馈生产环境中在消息发送过程中偶尔会出现如下 4 个错误信息之一：

- [REJECTREQUEST]system busy, start flow control for a while
- too many requests and system thread pool busy, RejectedExecutionException
- [PC_SYNCHRONIZED]broker busy, start flow control for a while
- [PCBUSY_CLEAN_QUEUE]broker busy, start flow control for a while, period in queue: %sms, size of queue: %d

二、原理解读

在进行消息中间件的选型时，如果待选中间件在功能上、性能上都能满足业务的情况下，我各个建议把中间件的实现语言这个因素也考虑进去，毕竟选择一门用自己擅长的语言实现的中间件会更具掌控性。在出现异常的情况下，我们可以根据自己的经验提取错误信息关键字 system busy，在 RocketMQ 源码中直接搜索，得到抛出上述错误信息的代码如下：



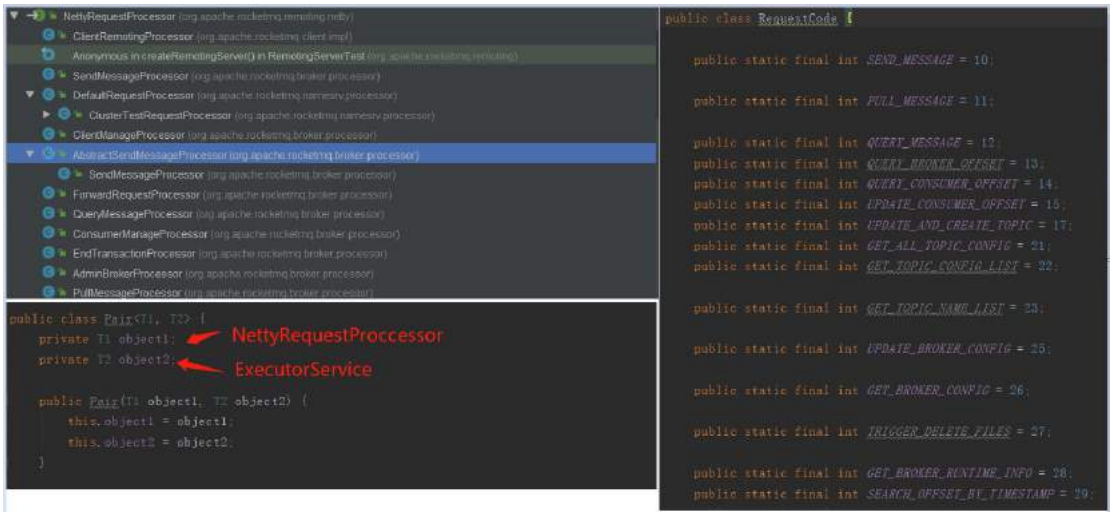
其代码入口为：org.apache.rocketmq.remoting.netty.NettyRemotingAbstract#processRequestCommand。从图中可以看出，抛出上述错误的关键原因是：pair.getObject1().rejectRequest()和抛出 RejectedExecutionException 异常。

备注：本文偏实战，源码只是作为分析的重点证据，故本文只会点出关键源码，并不会详细跟踪其整个实现流程，如果想详细了解其实现，可以查阅笔者编著的《RocketMQ 技术内幕》。

1. RocketMQ 网络处理机制概述

RocketMQ 的网络设计非常值得我们学习与借鉴，首先在客户端端将不同的请求定义不同的请求命令 CODE，服务端会将客户端请求进行分类，每个命令或每类请求命令定义一个处理器(NettyRequestProcessor)，然后每一个 NettyRequestProcessor 绑定到一个单独的线程池，进行命令处理，不同类型的请求将使用不同的线程池进行处理，实现线程隔离。

为了方便下文的描述，我们先简单的认识一下 NettyRequestProcessor、Pair、RequestCode。其核心关键点如下：



- NettyRequestProcessor

RocketMQ 服务端请求处理器，例如 SendMessageProcessor 是消息发送处理器、PullMessageProcessor 是消息拉取命令处理器。

- RequestCode

请求 CODE，用来区分请求的类型，例如 SEND_MESSAGE：表示该请求为消息发送，PULL_MESSAGE:消息拉取请求。

- Pair

用来封装 NettyRequestProcessor 与 ExecuteService 的绑定关系。在 RocketMQ 的网络处理模型中，会为每一个 NettyRequestProcessor 与特定的线程池绑定，所有该 NettyRequestProcessor 的处理逻辑都在该线程池中运行。

2. pair.getObject1().rejectRequest()

由于读者朋友提出的问题，都是发生在消息发送过程中，故本文重点关注 SendMessageProcessor#rejectRequest 方法。

```
SendMessageProcessor#rejectRequest
public boolean rejectRequest() {
    return this.brokerController.getMessageStore().isOSPageCacheBusy() ||
// @1
```

```
        this.brokerController.getMessageStore().isTransientStorePoolDeficient();          // @2
    }
```

拒绝请求的条件有两个，只要其中任意一个满足，则返回 true。

代码@1：Os PageCache busy，判断操作系统 PageCache 是否繁忙，如果忙，则返回 true。想必看到这里大家肯定与我一样好奇，RocketMQ 是如何判断 pageCache 是否繁忙呢？下面会重点分析。

代码@2：transientStorePool 是否不足。

isOSPageCacheBusy()

```
    DefaultMessageStore#isOSPageCacheBusy()
    public boolean isOSPageCacheBusy() {
        long begin = this.getCommitLog().getBeginTimeInLock();  // @1 start
        long diff = this.systemClock.now() - begin;                                    // @1  end

        return diff < 10000000
                    && diff > this.messageStoreConfig.getOsPageCacheBusyTimeOutMills
();      // @2
    }
```

代码@1：先重点解释 begin、diff 两个局部变量的含义：

- begin

通俗的一点讲，就是将消息写入 Commitlog 文件所持有锁的时间，精确说是将消息体追加到内存映射文件(DirectByteBuffer)或 pageCache(FileChannel#map)该过程中开始持有锁的时间戳，具体的代码请参考：CommitLog#putMessage。

- diff

一次消息追加过程中持有锁的总时长，即往内存映射文件或 pageCache 追加一条消息所耗时间。

代码@2：如果一次消息追加过程的时间超过了 Broker 配置文件 osPageCacheBusyTimeOutMills，则认为 pageCache 繁忙，osPageCacheBusyTimeOutMills 默认值为 1000，表示 1s。

isTransientStorePoolDeficient()

```
DefaultMessageStore#isTransientStorePoolDeficient
public boolean isTransientStorePoolDeficient() {
    return remainTransientStoreBufferNumbs() == 0;
}
public int remainTransientStoreBufferNumbs() {
    return this.transientStorePool.remainBufferNumbs();
}
```

最终调用 TransientStorePool#remainBufferNumbs 方法。

```
public int remainBufferNumbs() {
        if (storeConfig.isTransientStorePoolEnable()) {
            return availableBuffers.size();
        }
        return Integer.MAX_VALUE;
}
```

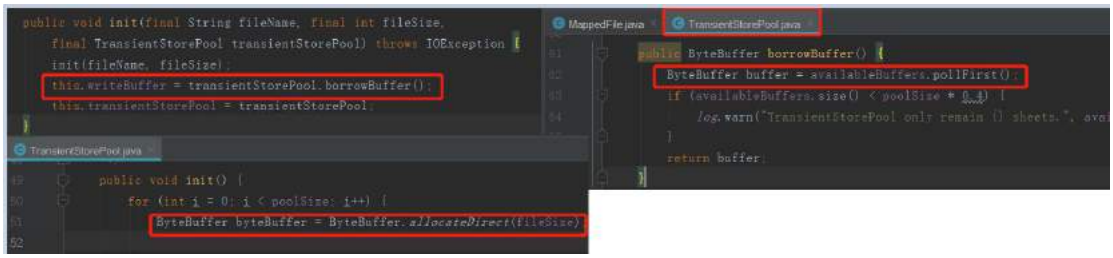
如果启用 transientStorePoolEnable 机制，返回当前可用的 ByteBuffer 个数，即整个 isTransientStorePoolDeficient 方法的用意是是否还存在可用的 ByteBuffer，如果不存在，即表示 pageCache 繁忙。那什么是 transientStorePoolEnable 机制呢?

3. 漫谈 transientStorePoolEnable 机制

Java NIO 的内存映射机制，提供了将文件系统中的文件映射到内存机制，实现对文件的操作转换对内存地址的操作，极大的提高了 IO 特性，但这部分内存并不是常驻内存，可以被置换到交换内存(虚拟内存)，RocketMQ 为了提高消息发送的性能，引入了内存锁定机制，即将最近需要操作的 commitlog 文件映射到内存，并提供内存锁定功能，确保这些文件始终存在内存中，该机制的控制参数就是 transientStorePoolEnable。

MappedFile

重点关注 MappedFile 的 ByteBuffer writeBuffer、MappedByteBuffer mapped ByteBuffer 这两个属性的初始化，因为这两个方法是写消息与查消息操作的直接数据结构。



两个关键点如下：

- ByteBuffer writeBuffer

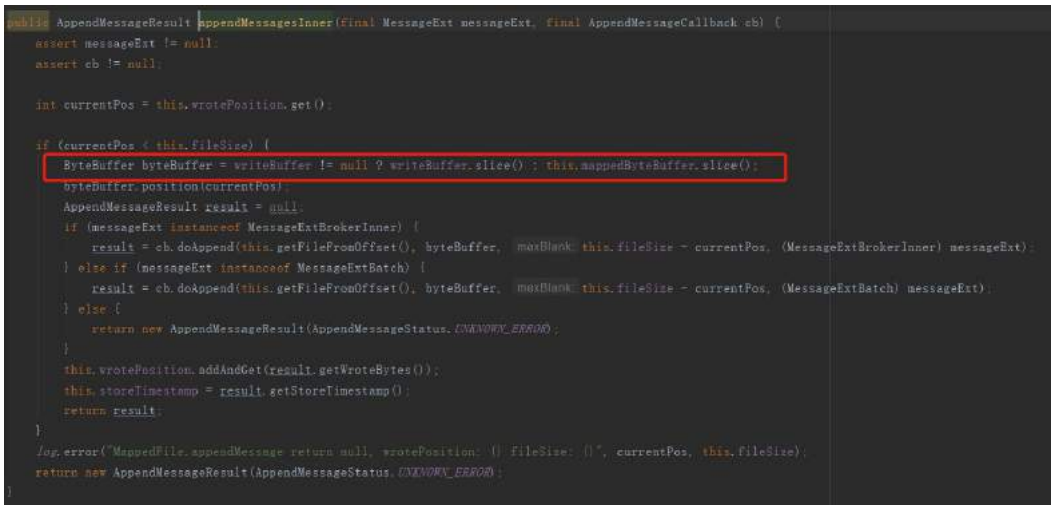
如果开启了 transientStorePoolEnable,则使用 ByteBuffer.allocateDirect(fileSize),创建(java.nio 的内存映射机制)。如果未开启，则为空。

- MappedByteBuffer mappedByteBuffer

使用 FileChannel#map 方法创建，即真正意义上的 PageCache。

消息写入时：

MappedFile#appendMessagesInner



从中可见，在消息写入时，如果 writerBuffer 不为空，说明开启了 transientStorePoolEnable 机制，则消息首先写入 writerBuffer 中，如果其为空，则写入 mappedByteBuffer 中。

消息拉取(读消息)：

MappedFile#selectMappedBuffer

```
public SelectMappedBufferResult selectMappedBuffer(int pos) {
    int readPosition = getReadPosition();
    if (pos < readPosition && pos >= 0) {
        if (this.hold()) {
            ByteBuffer byteBuffer = this.mappedByteBuffer.slice();
            byteBuffer.position(pos);
            int size = readPosition - pos;
            ByteBuffer byteBufferNew = byteBuffer.slice();
            byteBufferNew.limit(size);
            return new SelectMappedBufferResult( startOffset: this.fileFromOffset + pos, byteBufferNew, size,  mappedFile: this);
        }
    }

    return null;
}
```

消息读取时，是从 mappedByteBuffer 中读(pageCache)。

大家是不是发现了一个有趣的点，如果开启 transientStorePoolEnable 机制，是不是有了读写分离的效果，先写入 writerBuffer 中，读却是从 mappedByteBuffer 中读取。

为了对 transientStorePoolEnable 引入意图阐述的更加明白，这里我引入 Rocketmq 社区贡献者胡宗棠关于此问题的见解：

一般有两种，有两种方式进行读写：

第一种，Mmap+PageCache 的方式，读写消息都走的是 pageCache，这样子读写都在 pagecache 里面不可避免会有锁的问题，在并发的读写操作情况下，会出现缺页中断降低，内存加锁，污染页的回写。

第二种，DirectByteBuffer(堆外内存)+PageCache 的两层架构方式，这样子可以实现读写消息分离，写入消息时候写到的是 DirectByteBuffer——堆外内存中,读消息走的是 PageCache(对于,DirectByteBuffer 是两步刷盘，一步是刷到 PageCache，还有一步

是刷到磁盘文件中)，带来的好处就是，避免了内存操作的很多容易堵的地方，降低了时延，比如说缺页中断降低，内存加锁，污染页的回写。

温馨提示：如果想与胡宗棠大神进一步沟通交流，可以关注他的 github 账号：https://github.com/zongtanghu

不知道大家会不会有另外一个担忧，如果开启了 transientStorePoolEnable，内存锁定机制，那是不是随着 commitlog 文件的不断增加，最终导致内存溢出?

TransientStorePool 初始化

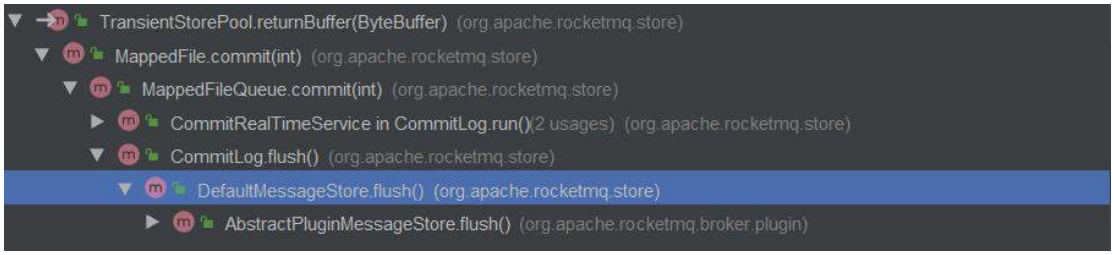
```
public TransientStorePool(final MessageStoreConfig storeConfig) {
    this.storeConfig = storeConfig;
    this.poolSize = storeConfig.getTransientStorePoolSize();
    this.fileSize = storeConfig.getMapedFileSizeCommitLog();
    this.availableBuffers = new ConcurrentLinkedDeque<>();
}

/**
 * It's a heavy init method.
 */
public void init() {
    for (int i = 0; i < poolSize; i++) {
        ByteBuffer byteBuffer = ByteBuffer.allocateDirect(fileSize);

        final long address = ((DirectBuffer) byteBuffer).address();
        Pointer pointer = new Pointer(address);
        LibC.INSTANCE.mlock(pointer, new NativeLong(fileSize));

        availableBuffers.offer(byteBuffer);
    }
}
```

从这里可以看出，TransientStorePool 默认会初始化 5 个 DirectByteBuffer(对外内存)，并提供内存锁定功能，即这部分内存不会被置换，可通过 transientStorePoolSize 参数控制。在消息写入消息时，首先从池子中获取一个 DirectByteBuffer 进行消息的追加，那当 5 个 DirectByteBuffer 全部写满消息后，该如何处理呢? 从 RocketMQ 的设计中来看，同一时间，只会对一个 commitlog 文件进行顺序写，写完一个后，继续创建一个

新的 commitlog 文件。故 TransientStorePool 的设计思想是循环利用这 5 个 DirectByteBuffer，只需要写入到 DirectByteBuffer 的内容被提交到 PageCache 后，即可重复利用。对应的代码如下：

```
TransientStorePool#returnBuffer
public void returnBuffer(ByteBuffer byteBuffer) {
    byteBuffer.position(0);
    byteBuffer.limit(fileSize);
    this.availableBuffers.offerFirst(byteBuffer);
}
```

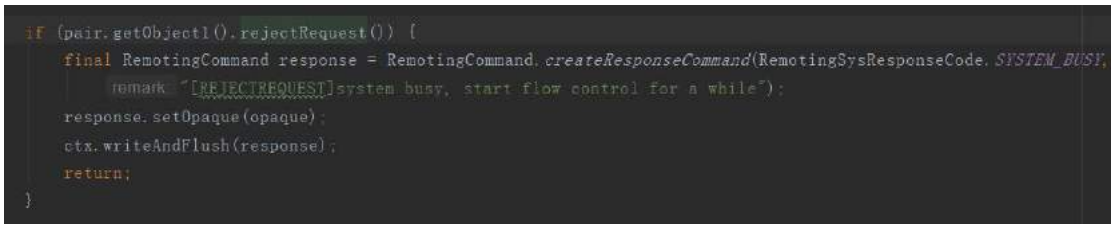
其调用栈如下：



从上面的分析看来，并不会随着消息的不断写入而导致内存溢出。

三、现象解答

1. [REJECTREQUEST]system busy, start flow control for a while



其抛出的源码入口点：NettyRemotingAbstract#processRequestCommand，上面的原理分析部分已经详细介绍其实现原理，总结如下。

在不开启 transientStorePoolEnable 机制时，如果 Broker PageCache 繁忙时则抛出上述错误，判断 PageCache 繁忙的依据就是向 PageCache 追加消息时，如果持有锁的时间超过 1s，则会抛出该错误；在开启 transientStorePoolEnable 机制时，其判断依据是如果 TransientStorePool 中不存在可用的堆外内存时抛出该错误。

2. too many requests and system thread pool busy, RejectedExecutionException

```
if (pair.getObject1().rejectRequest()) {
    final RemotingCommand response = RemotingCommand.createResponseCommand(RemotingSysResponseCode.SYSTEM_BUSY,
        remark: "[REJECTREQUEST]system busy, start flow control for a while");
    response.setOpaque(opaque);
    ctx.writeAndFlush(response);
    return;
}

try {
    final RequestTask requestTask = new RequestTask(run, ctx.channel(), cmd);
    pair.getObject2().submit(requestTask);
} catch (RejectedExecutionException e) {
    if ((System.currentTimeMillis() % 10000) == 0) {
        log.warn( var1: RemotingHelper.parseChannelRemoteAddr(ctx.channel())
            + ", too many requests and system thread pool busy, RejectedExecutionException "
            + pair.getObject2().toString()
            + " request code: " + cmd.getCode());
    }
```

其抛出的源码入口点：NettyRemotingAbstract#processRequestCommand，其调用地方紧跟 3.1,是在向线程池执行任务时，被线程池拒绝执行时抛出的，我们可以顺便看看 Broker 消息处理发送的线程信息：

BrokerController#registerProcessor

```
public void registerProcessor() {
    /**
     * SendMessageProcessor
     */
    SendMessageProcessor sendProcessor = new SendMessageProcessor( brokerController: this);
    sendProcessor.registerSendMessageHook(sendMessageHookList);
    sendProcessor.registerConsumeMessageHook(consumeMessageHookList);

    this.remotingServer.registerProcessor(RequestCode.SEND_MESSAGE, sendProcessor, this.sendMessageExecutor);
    this.remotingServer.registerProcessor(RequestCode.SEND_MESSAGE_V2, sendProcessor, this.sendMessageExecutor);
    this.remotingServer.registerProcessor(RequestCode.SEND_BATCH_MESSAGE, sendProcessor, this.sendMessageExecutor);
    this.remotingServer.registerProcessor(RequestCode.CONSUMER_SEND_MSG_BACK, sendProcessor, this.sendMessageExecutor);
    this.fastRemotingServer.registerProcessor(RequestCode.SEND_MESSAGE, sendProcessor, this.sendMessageExecutor);
    this.fastRemotingServer.registerProcessor(RequestCode.SEND_MESSAGE_V2, sendProcessor, this.sendMessageExecutor);
    this.fastRemotingServer.registerProcessor(RequestCode.SEND_BATCH_MESSAGE, sendProcessor, this.sendMessageExecutor);
    this.fastRemotingServer.registerProcessor(RequestCode.CONSUMER_SEND_MSG_BACK, sendProcessor, this.sendMessageExecutor);
```

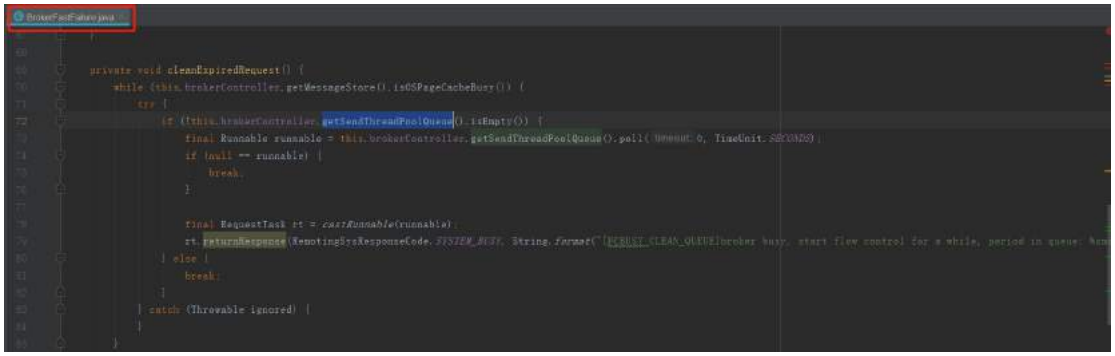
该线程池的队列长度默认为 10000，我们可以通过 sendThreadPoolQueueCapacity 来改变默认值。

3. [PC_SYNCHRONIZED]broker busy, start flow control for a while

```
if (this.isOSPageCacheBusy()) {
    return new PutMessageResult(PutMessageStatus.OS_PAGECACHE_BUSY, appendMessageResult: null);
}
```

其抛出的源码入口点：DefaultMessageStore#putMessage，在进行消息追加时，再一次判断 PageCache 是否繁忙，如果繁忙，则抛出上述错误。

4. broker busy, start flow control for a while, period in queue: %sms, size of queue: %d



其抛出源码的入口点：BrokerFastFailure#cleanExpiredRequest。该方法的调用频率为每隔 10s 中执行一次，不过有一个执行条件，就是 Broker 端要开启快速失败，默认为开启，可以通过参数 brokerFastFailureEnable 来设置。该方法的实现要点是每隔 10s，检测一次，如果检测到 PageCache 繁忙，并且发送队列中还有排队的任务，则直接不再等待，直接抛出系统繁忙错误，使正在排队的线程快速失败，结束等待。

四、实践建议

经过上面的原理讲解与现象分析，消息发送时抛出 system busy、broker busy 的原因都是 PageCache 繁忙，那是不是可以通过调整上述提到的某些参数来避免抛出错误呢?

- osPageCacheBusyTimeOutMills

设置 PageCache 系统超时的时间，默认为 1000，表示 1s，那是不是可以把增加这个值，例如设置为 2000 或 3000。作者观点：非常不可取。

- sendThreadPoolQueueCapacity

Broker 服务器处理的排队队列，默认为 10000，如果队列中积压了 10000 个请求，则会抛出 RejectExecutionException。作者观点：不可取。

- brokerFastFailureEnable

是否启用快速失败，默认为 true，表示当如果发现 Broker 服务器的 PageCache 繁忙，如果发现 sendThreadPoolQueue 队列中不为空，表示还有排队的发送请求在排队等待执行，则直接结束等待，返回 broker busy。那如果不开启快速失败，则同样可以避免抛出这个错误。作者观点：非常不可取。

修改上述参数，都不可取，原因是出现 system busy、broker busy 这个错误，其本质是系统的 PageCache 繁忙，通俗一点讲就是向 PageCache 追加消息时，单个消息发送占用的时间超过 1s 了，如果继续往该 Broker 服务器发送消息并等待，其 TPS 根本无法满足，哪还是高性能的消息中间了呀。故才会采用快速失败机制，直接给消息发送者返回错误，消息发送者默认情况会重试 2 次，将消息发往其他 Broker，保证其高可用。

方案 1：开启 transientStorePoolEnable

在 broker.config 中将 transientStorePoolEnable=true。

方案依据：启用“读写”分离，消息发送时消息先追加到 DirectByteBuffer(堆外内存)中，然后在异步刷盘机制下，会将 DirectByteBuffer 中的内容提交到 PageCache，然后刷写到磁盘。消息拉取时，直接从 PageCache 中拉取，实现了读写分离，减轻了 PageCaceh 的压力，能从根本上解决该问题。

方案缺点：会增加数据丢失的可能性，如果 Broker JVM 进程异常退出，提交到 PageCache 中的消息是不会丢失的，但存在堆外内存(DirectByteBuffer)中但还未提交到 PageCache 中的这部分消息，将会丢失。但通常情况下，RocketMQ 进程退出的可能性不大。

方案 2：扩容 Broker 服务器

方案依据：当 Broker 服务器自身比较忙的时候，快速失败，并且在接下来的一段时间内会规避该 Broker，这样该 Broker 恢复提供了时间保证，Broker 本身的架构是支持分布式水平扩容的，增加 Topic 的队列数，降低单台 Broker 服务器的负载，从而避免出现 PageCache。

温馨提示：在 Broker 扩容时候，可以复制集群中任意一台 Broker 服务下${ROCKETMQ_HOME}/store/config/topics.json 到新 Broker 服务器指定目录，避免在新 Broker 服务器上为 Broker 创建队列，然后消息发送者、消息消费者都能动态获取 Topic 的路由信息。

与之扩容对应的，也可以通过对原有 Broker 进行升配，例如增加内存、把机械盘换成 SSD，但这种情况，通常需要重启 Broekr 服务器，没有扩容来的方便。

本文就介绍到这里了，如果大家觉得文章对自己有用的话，麻烦帮忙点赞、转发，谢谢。亲爱的读者朋友，还有更好的方案没？欢迎留言与作者互动，共同探讨。

1.10 再谈 RocketMQ broker busy

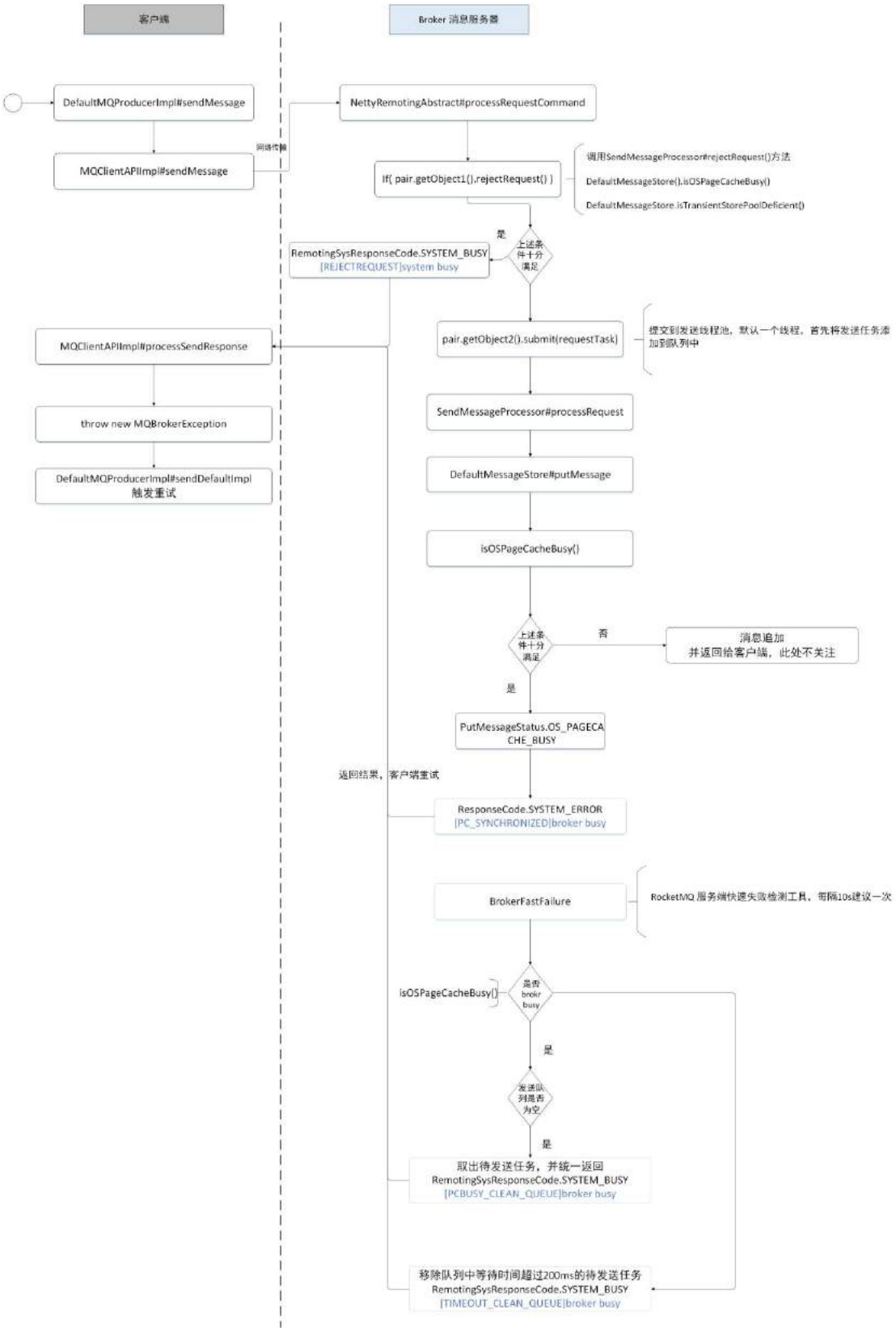
本文将在 这篇的文章（https://blog.csdn.net/prestigeding/article/details/92800672）的基础上，结合生产上的日志尝试再次理解 broker busy 以及探讨解决方案。

首先，broker busy 相关的日志关键字如下：

- [REJECTREQUEST]system busy
- too many requests and system thread pool busy
- [PC_SYNCHRONIZED]broker busy
- [PCBUSY_CLEAN_QUEUE]broker busy
- [TIMEOUT_CLEAN_QUEUE]broker busy

上述 4 个关键字在上篇文章中已详细介绍，本文先对出现上述错误进行一个总结，具体的分析过程请查阅上篇文章。

本文先给出一张流程图，展示上述 5 种 broker busy 分别会在消息发送的什么时候发生。



针对前 4 种 broker busy 出现的问题已经在上篇文章中详细介绍，主要是由于 Broker 在追加消息时持有的锁时间超过了设置的 1s，Broker 为了自我保护，会抛出错误，客户端会选择其他 broker 服务器进行重试。如果对不是金融级服务，建议将 transientStorePoolEnable = true，可以有效避免前面 4 种 broker ，因为开启这个参数，消息首先会存储在 堆外内存中，并 RocketMQ 提供了内存锁定的功能，其追加性能能得到一定的保障，这样可以做到在内存使用层面的读写分离，即写消息是直接写入堆外内存，消费消息直接从 pagecache 中读，然后定时将堆外内存的消息写入 pagecache。但这种方案随之带来的就是可能存在消息丢失，如果对消息非常严谨的话，建议扩容集群，或迁移 topic 到新的集群。

同时在做 Broker 服务器巡检的时候，可以通过去通过如下命令去查看 broker 一次消息追加是否会超过 500 ms。

在这个图中我们看到在设置了 transientStorePoolEnable 为 true 的情况下，虽然一天只有一条超过 500ms 的消息，但也值得警惕了，由于对系统内核参数掌握程度不够，这种情况，估计只能走集群扩容的路子了。但如果一天消息量巨大而且出现频率不高的情况，由于有重试机制，倒不会带来太大的问题。如果出现太多的错误，建议集群扩容。

本文接下来想重点探讨一下 [TIMEOUT_CLEAN_QUEUE]broker busy 这种情况。

```
    BrokerFastFailure#cleanExpiredRequest
    while (true) {
        try {
            if (!this.brokerController.getSendThreadPoolQueue().isEmpty()) {
                final Runnable runnable = this.brokerController.getSendThreadPoolQueue
().peek();
                if (null == runnable) {
                    break;
                }
                final RequestTask rt = castRunnable(runnable);
                if (rt == null || rt.isStopRun()) {
```

```
                    break;
                }

                final long behind = System.currentTimeMillis() - rt.getCreateTimestamp();
                if (behind >= this.brokerController.getBrokerConfig().getWaitTimeMillsInSen
dQueue()) {
                    if (this.brokerController.getSendThreadPoolQueue().remove(runnable))
{
                        rt.setStopRun(true);
                        rt.returnResponse(RemotingSysResponseCode.SYSTEM_BUSY, S
tring.format("[TIMEOUT_CLEAN_QUEUE]broker busy, start flow control for a while, perio
d in queue: %sms, size of queue: %d", behind, this.brokerController.getSendThreadPool
Queue().size()));
                    }
                } else {
                    break;
                }
            } else {
                break;
            }
        } catch (Throwable ignored) {
        }
    }
```

可以看出来，抛出这种错误，在 broker 还没有发送“严重”的 pagecache 繁忙，即消息追加到内存中的最大时延没有超过 1s，通常追加是很快的，绝大部分都会低于 1ms，但可能会由于出现一个超过 200ms 的追加时间，导致排队中的任务等待时间超过了 200ms，则此时会触发 broker 端的快速失败，让请求快速失败，便于客户端快速重试。但是这种请求并不是实时的，而是每隔 10s 检查一遍。

值得注意的是，一旦出现 TIMEOUT_CLEAN_QUEUE，可能在一个点会有多个这样的错误信息，具体多少与当前积压在待发送队列中的个数有关。

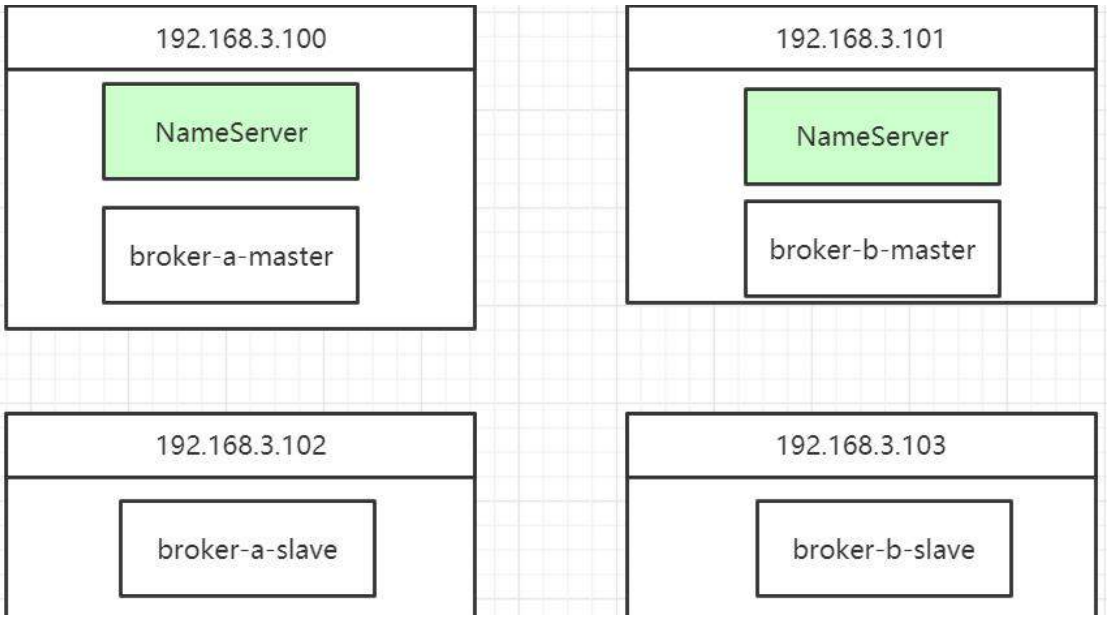
关于 [TIMEOUT_CLEAN_QUEUE]broker busy 我们也可以适当调整 waitTimeMillsInSendQueue，默认值为 200ms，可以适当调整到 400ms。

1.11 从年末生产故障解锁 RocketMQ 集群部署的最佳实践

笔者比较“悲催”，临近年末由笔者维护的生产 MQ 集群中的一台物理机内存故障导致操作系统异常重启，持续 10 分钟中出现众多的应用发送客户端出现发送消息络超时，导致事故并定性为 S1，笔者的“年终奖”。。。

一、故障描述

RocketMQ 集群采取的部署架构为 2 主 2 从，其部署架构如下图所示：



其部署架构中一个非常明显的特点是一台物理机上分别部署了 nameserver，broker 两个进程。

其中一台机器(192.168.3.100)的内存出现故障，导致机器重启，但 Linux 操作系统由于重启需要自检等因素，整个重启过程竟然持续了将近 10 分钟，客户端的发送超时持续 10 分钟，这显然是不能接受的！！！

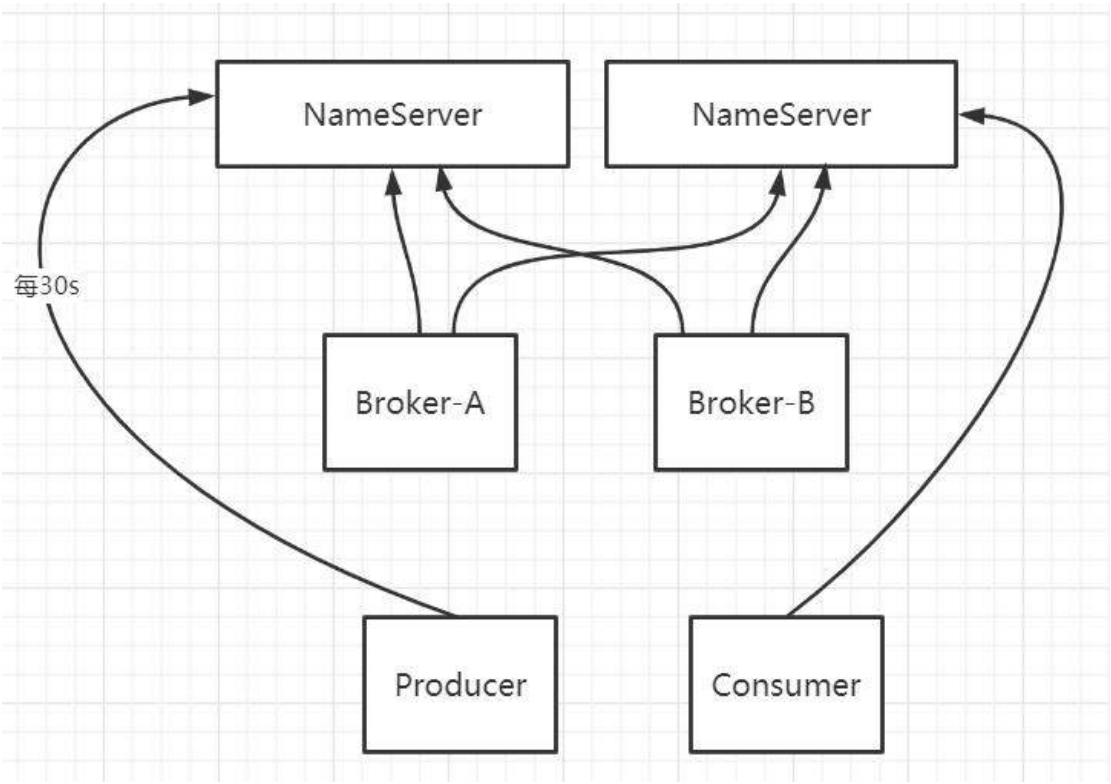
RocketMQ 的高可用设计何在？接下来我们将详细介绍其分析过程。

二、故障分析

当得知一台机器故障导致故障持续 10 分钟，我的第一反应是不应该呀，因为 RocketMQ 集群是分布式部署架构，天然支持故障发现与故障恢复，消息发送客户端能自动感知 Broker 异常的的时间绝对不会超过 10 分钟，那故障又是怎么发生的呢?

首先我们先来回顾一下 RocketMQ 的路由注册与发现机制。

1. RocketMQ 路由注册与剔除机制



其路由注册、剔除机制说明如下:

- 集群中所有 Broker 每隔 30s 向集群中所有的 NameServer 发送心跳包，注册 Topic 路由信息。
- NameServer 在收到 Broker 端的心跳包时首先会更新路由表，并记录收到心跳包的时间。

- NameServer 会启动一个定时任务每 10s 会扫描 Broker，如果 Nameserver 连续 120s 未收到 Broker 的心跳包，会判定 Broker 已下线，将从路由表中将该 Broker 移除。
- 如果 Nameserver 与 Broker 端的长连接断开，NameServer 会立即感知 Broker 下线并从路由表中将该 Broker 移除。
- 消息客户端(消息发送者、消息消费者)在任意时刻只会和其中一台 NameServer 建立连接，并每隔 30s 向 NameServer 查询路由信息，如果查询到结果会更新发送者的本地路由信息。

从上述的路由注册、剔除机制来看，当一台 Broker 服务器宕机，消息发送者感知路由信息发生变化需要的时间是多长呢?

分如下两种情况分别讨论:

- NameServer 与 Broker 服务器 TCP 连接断开，此时 NameServer 能立即感知路由信息变化，将其从路由表中移除，从而消息发送端应该在 30s 左右就能感知路由发送变化，在此 30s 内在发送端会出现消息发送失败,但结合发送规避机制，并不会对发送方带来重大故障，可接受。
- 如果 NameServer 与 Broker 服务器的 TCP 连接未断开，但 Broker 已无法提供服务(例如假死)，此时 NameServer 需要 120s 才能感知 Broker 宕机，此时消息发送端最多需要 150s 才能感知其路由信息的变化。

但问题来了，为什么在生产实际过程中一台 Broker 由于内存故障重启,10 分钟后重启成功后业务才恢复，即业务才真正感知 Broker 宕机呢?

既然出现了，我们就需要对其进行分析，给出解决方案，避免不会在生产环境出现同类型的错误。

2. 故障排查经过

先查询客户端的日志(/home/{user}/logs/rocketmqlogs/rocketmq_client.log)，从中可以看到从客户端第一次报消息发送超时的时间是 14:44，其日志输出如下:

```
2020-  14:44:19 WARN MQClientFactoryScheduledThread - updateTopicRouteInfoFromNameServer Exception
org.apache.rocketmq.remoting.exception.RemotingTimeoutException: wait response on the channel  :9876> timeout, 30(
    at org.apache.rocketmq.remoting.netty.NettyRemotingAbstract.invokeSyncImpl(NettyRemotingAbstract.java:369)
    at org.apache.rocketmq.remoting.netty.NettyRemotingClient.invokeSync(NettyRemotingClient.java:340)
    at org.apache.rocketmq.client.impl.MQClientAPIImpl.getTopicRouteInfoFromNameServer(MQClientAPIImpl.java:1214)
    at org.apache.rocketmq.client.impl.factory.MQClientInstance.updateTopicRouteInfoFromNameServer(MQClientInstance.java:603)
    at org.apache.rocketmq.client.impl.factory.MQClientInstance.updateTopicRouteInfoFromNameServer(MQClientInstance.java:491)
    at org.apache.rocketmq.client.impl.factory.MQClientInstance.updateTopicRouteInfoFromNameServer(MQClientInstance.java:460)
    at org.apache.rocketmq.client.impl.factory.MQClientInstance$3.run(MQClientInstance.java:277)
    at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:511)
    at java.util.concurrent.FutureTask.runAndReset(FutureTask.java:308)
    at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.access$301(ScheduledThreadPoolExecutor.java:180)
    at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.run(ScheduledThreadPoolExecutor.java:294)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
```

由于 192.168.3.100 机器内存故障，故首先去查看该集群中其他 nameserver 中的日志，看正常机器中的 NameServer 感知 broker-a 故障的时长，其日志如下所示：

```
2020-12-  14:46:09 INFO NettyServerCodecThread_7 - NETTY SERVER PIPELINE: channelUnregistered, the channel[
2020-12-  14:46:09 WARN NettyServerCodecThread_7 - NETTY SERVER PIPELINE: IDLE exception  :159
2020-12-  14:46:09 INFO NettyEventExecuter - the broker's channel destroyed,  :10911, clean it'
2020-12-  14:46:09 INFO NettyServerNIOSelector_3_3 - closeChannel: close the connection to remote address[1
2020-12-  14:46:09 INFO NettyEventExecuter - remove brokerAddr[0,  :10911] from brokerAddrTable
2020-12-  14:46:09 INFO NettyServerCodecThread_7 - NETTY SERVER PIPELINE: channelInactive, the channel[192.
2020-12-  14:46:09 INFO NettyServerCodecThread_7 - NETTY SERVER PIPELINE: channelUnregistered, the channel[
2020-12-  14:46:10 INFO NettyServerCodecThread_2 - NETTY SERVER PIPELINE: channelRegistered  :4
```

从中可以看出 192.138.3.101 的 nameserver 基本在 2 分钟左右才感知其宕机，即虽然机器在重启，但可能由于操作系统要做硬件自检等其他原因，TCP 连接并未断开，故 nameserver 在 120s 后才感知其宕机，从路由信息表中将该 broker 移除，那按照路由剔除机制，客户端应该在 150 秒的时间内感知其变化，那为什么没感知呢？

继续查看客户端路由信息，查看客户端感知路由信息发生变化的时间点，如下图所示：

```
14:53:46 INFO NettyClientWorkerThread_1 - NETTY CLIENT PIPELINE: CLOSE
14:53:46 INFO NettyClientWorkerThread_1 - closeChannel: the channel  :0911] was removed from channel table
14:53:46 INFO NettyClientWorkerThread_1 - NETTY CLIENT PIPELINE: CLOSE
14:53:46 INFO NettyClientWorkerThread_1 - eventCloseChannel: the channel[null] has been removed from the channel table before
14:53:46 INFO NettyClientSelector_1 - closeChannel: close the connection to remote  result: true
14:53:46 INFO MQClientFactoryScheduledThread - new name server is chosen. OLD:  :9876, NEW:  :9876, 1
14:53:46 INFO MQClientFactoryScheduledThread - createChannel: begin to connect remote  0:9876] asynchronously
14:53:46 INFO NettyClientWorkerThread_2 - NETTY CLIENT PIPELINE: CONNECT  UNKNOWN =>  :9876
14:53:46 INFO MQClientFactoryScheduledThread - createChannel: connect remote host[192.  :9876] success, AbstractBootstr
14:53:46 INFO MQClientFactoryScheduledThread - the topic  sh_topic] route info changed, old[TopicRouteData [orderTopic
14:53:46 INFO MQClientFactoryScheduledThread - updateTopicPublishInfo prev is not null, TopicPublishInfo [orderTopic=false, m
14:53:46 INFO MQClientFactoryScheduledThread - updateTopicPublishInfo prev is not null, TopicPublishInfo [orderTopic=false, m
14:53:46 INFO MQClientFactoryScheduledThread - topicRouteTable.put. Topic =  pic, TopicRouteData[TopicRouteData
```

从客户端日志来看，客户端在 14:53:46 才感知其变化，这又是为什么呢？

原来客户端在更新路由信息时报超时异常，其截图如下所示：

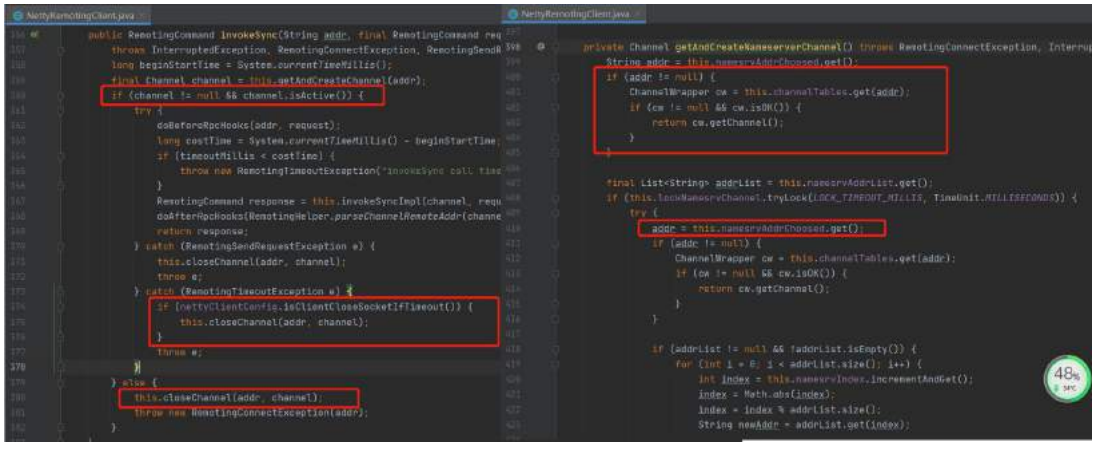
```
2020-12-09 14:44:19 WARN MQClientFactoryScheduledThread - updateTopicRouteInfoFromNameServer Exception
org.apache.rocketmq.remoting.exception.RemotingTimeoutException: wait response on the channel                9876> timeout, 30(
    at org.apache.rocketmq.remoting.netty.NettyRemotingAbstract.invokeSyncImpl(NettyRemotingAbstract.java:369)
    at org.apache.rocketmq.remoting.netty.NettyRemotingClient.invokeSync(NettyRemotingClient.java:340)
    at org.apache.rocketmq.client.impl.MQClientAPIImpl.getTopicRouteInfoFromNameServer(MQClientAPIImpl.java:1214)
    at org.apache.rocketmq.client.impl.factory.MQClientInstance.updateTopicRouteInfoFromNameServer(MQClientInstance.java:603)
    at org.apache.rocketmq.client.impl.factory.MQClientInstance.updateTopicRouteInfoFromNameServer(MQClientInstance.java:491)
    at org.apache.rocketmq.client.impl.factory.MQClientInstance.updateTopicRouteInfoFromNameServer(MQClientInstance.java:360)
    at org.apache.rocketmq.client.impl.factory.MQClientInstance$3.run(MQClientInstance.java:277)
    at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:511)
    at java.util.concurrent.FutureTask.runAndReset(FutureTask.java:308)
    at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.access$301(ScheduledThreadPoolExecutor.java:180)
    at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.run(ScheduledThreadPoolExecutor.java:294)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
```

从发生故障到故障恢复期间，客户端一直尝试从已发生故障的 NameServer 去更新路由信息，但一直返回超时，这样就导致了客户端一直无法获取最新的路由信息，故一直无法感知已宕机的 Broker。

从日志分析来看，到目前来说就比较明朗了，客户端之所有没有在 120s 之内感知其路由信息的变化，是因为客户端一直尝试从已宕机的 nameserver 去更新路由信息，但由于一直无法请求成功，故客户端的缓存路由信息一直无法得到更新，造成了上面的现象

那问题来了，按照我们对 RocketMQ 的认识，NameServer 宕机，客户端会自动去从 nameserver 列表中选择下一个 nameserver，那为什么这里并无发生 nameserver 切换，而是等到 14:53 才切换呢?

接下来我们将目光投向 NameServer 的切换代码，其代码片段如下图所示:



上图中的几个关键分析如下:

- 客户端能通过缓存中的连接发送 RPC 请求的前提条件是 channel 的 isActive 方法返回 true，即底层 TCP 连接处于激活状态。

- 在客户端向服务端发起 RPC 请求时，如出现非超时类异常，会执行 closeChannel 方法，该方法会关闭连接并从连接缓存表中移除，这个非常关键，因为在切换 Name Server 时如果缓存中存在连接并连接处于激活状态，就不会切换 nameserver。
- 如果发送 RPC 超时，rocketmq 会根据 clientCloseSocketIfTimeout 参数来决定是否关闭连接，但遗憾的是该参数默认为 false，并且并未提供修改的入口。

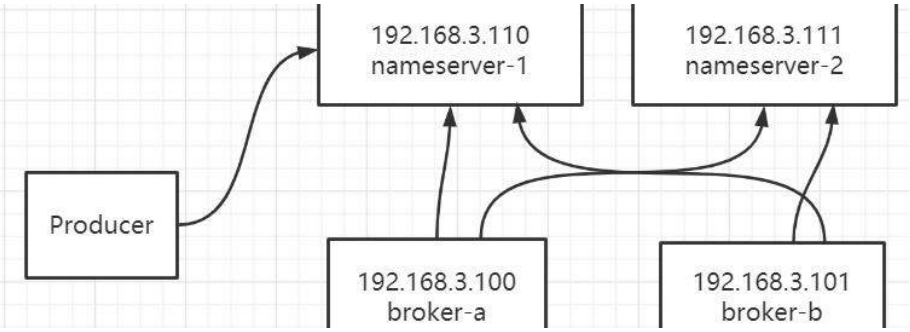
那问题分析到这里，就非常明确了，由于机器内存故障触发重启并且重启前需要自检等因素，造成 nameserver,broker 无法再处理请求但底层 TCP 连接并未断开，导致发生超时错误，但客户端并不会关闭与故障机器 nameserver 的 TCP 连接，导致无法切换，等到机器重新启动后，TCP 连接断开，故障机器重启完成后感知路由信息变化，故障恢复。

经过上面的问题分析，其故障原因如下： 192.168.3.100 机器在内存故障后重启，整个重启耗时 10 分钟，并且在重启过程中 TCP 连接未断开，造成 192.168.3.101 nameserver 在故障发送时 2 分钟左右才感知路由变化，但部分客户端时连接 192.168.3.100 的 nameserver，客户端尝试从该 nameserver 查询路由信息，但一直返回超时，并没有关闭连接，导致客户端并不会切换到 3.101 的 nameserver，直到客户端与 nameserver 的 TCP 连接断开后，切换到另外一个 3.101 的 nameserver，故障在指定时间内得以恢复。

根本原因：其实是 nameserver 的假死导致路由信息无法更新。

三、最佳实践

经过上面的故障，个人觉得 nameserver 不应该与 broker 部署在一起，如果 nameserver 与 broker 并不部署在一起，上面的问题能得到有效避免，其部署架构如下图所示：



这样的部署架构如果面对上面的故障，Broker 假死的情况，能有效避免吗？答案是可以的。

如果 192.168.3.100 的 broker 假死，那么 3.110,3.111 的 nameserver 都能在 2 分钟内感知 broker-a 宕机，然后客户端能成功从 nameserver 处获得最新的路由信息，如果 nameserver 假死，出现超时错误，只要 broker 不宕机，则通过缓存，还是能正常工作的，但如果 nanmeserver,broker 一起假死，则上述架构还是无法规避上面的问题。

故本次的最佳实践主要包含如下两条：

1. nameserver 与 broker 一定要分开部署，进行隔离。
2. nameserver 与客户端的连接，应该在超时后，关闭连接，触发 nameserver 漂移，需要修改源码。

1.12 RocketMQ 一行代码造成大量消息丢失

一、问题现象

首先接到项目反馈使用 RocketMQ 会出现如下错误：

错误信息关键点：MQBrokerException：CODE:2 DESC:[TIMEOUT_CLEAN_QUEUE]broker busy,start flow control for a while,period in queue：205ms，size of queue:880。

由于项目组并没有对消息发送失败做任何补偿，导致丢失消息丢失，故需要对这个问题进行深层次的探讨，并加以解决。

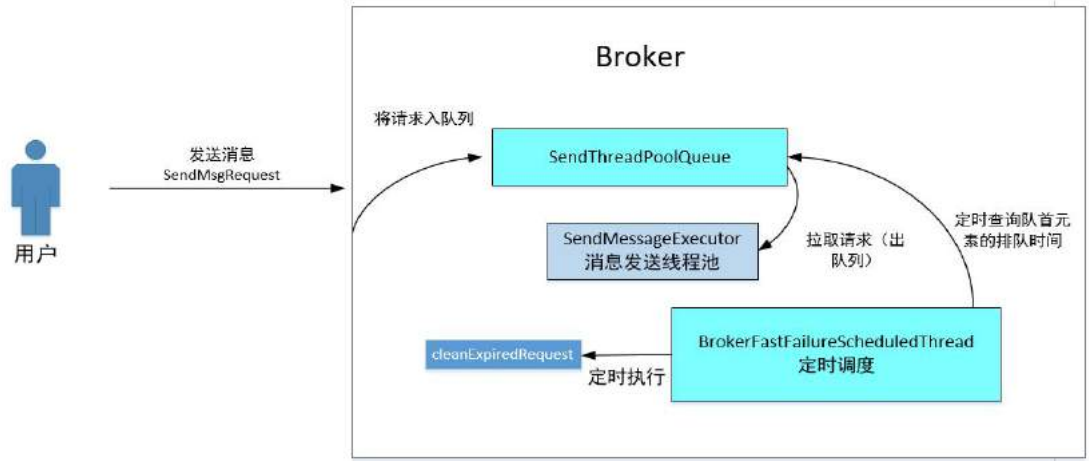
二、问题分析

首先我们根据关键字：TIMEOUT_CLEAN_QUEUE 去 RocketMQ 中查询，去探究在什么时候会抛出如上错误。根据全文搜索如下图所示：

```
void cleanExpiredRequestInQueue(final BlockingQueue<Runnable> blockingQueue, final long maxWaitTimeMillsInQueue) {
    while (true) {
        try {
            if (!blockingQueue.isEmpty()) {
                final Runnable runnable = blockingQueue.peek();
                if (null == runnable) {
                    break;
                }
                final RequestTask rt = castRunnable(runnable);
                if (rt == null || rt.isStopRun()) {
                    break;
                }

                final long behind = System.currentTimeMillis() - rt.getCreateTimestamp();
                if (behind >= maxWaitTimeMillsInQueue) {
                    if (blockingQueue.remove(runnable)) {
                        rt.setStopRun(true);
                        rt.returnResponse(RemotingSysResponseCode.SYSTEM_BUSY, String.format("[TIMEOUT_CLEAN_QUEUE]broker busy, start flow control for a whil
                    }
                } else {
```

该方法是在 BrokerFastFailure 中定义的，通过名称即可以看成其设计目的：Broker 端快速失败机制。

Broker 端快速失败其原理图如下：



消息发送者向 Broker 发送消息写入请求，Broker 端在接收到请求后会首先放入一个队列中(SendThreadPoolQueue)，默认容量为 10000。

Broker 会专门使用一个线程池(SendMessageExecutor)去从队列中获取任务并执行消息写入请求，为了保证消息的顺序处理，该线程池默认线程个数为 1。

如果 Broker 端收到内存抖动等因素造成单条写入数据发生抖动，如果单个 Broker 端积压的请求太对还得不到及时处理，会极大的造成客户端消息发送的延长时间，设想一下，如果由于 Broker 压力增大，写入一条消息需要 500ms 甚至超过 1s，并且队列中积压了

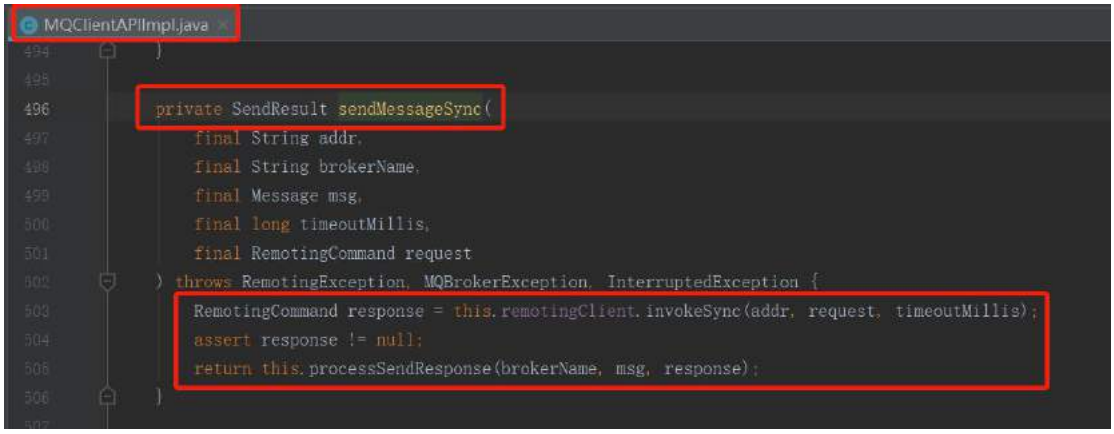
5000 条消息，消息发送端的默认超时时间为 3s，如果按照这样的速度，这些请求在轮到 Broker 执行写入请求时，客户端已经将这个请求超时了，这样不仅会造成大量的无效处理，还会导致客户端发送超时。

故 RocketMQ 为了解决该问题，引入 Broker 端快速失败机制，即开启一个定时调度线程，每隔 10 毫秒去检查队列中的第一个排队节点，如果该节点的排队时间已经超过了 200 ms，就会取消该队列中所有已超过 200ms 的请求，立即向客户端返回失败，这样客户端能尽快进行重试，因为 Broker 都是集群部署，下次重试可以发送到其他 Broker 上，这样能最大程度保证消息发送在默认 3s 的时间内经过重试机制，能有效避免某一台 Broker 由于瞬时压力大而造成的消息发送不可用，从而实现消息发送的高可用。

从 Broker 端快速失败机制引入的初衷来看，快速失败后会发起重试，除非同一深刻集群内所有的 Broker 都繁忙，不然消息会发送成功，用户是不会感知这个错误的，那为什么用户感知了呢？难道 TIMEOUT_CLEAN_QUEUE 错误，Broker 不重试？

为了解开这个谜团，接下来会采用源码分析的手段去探究真相。接下来将以消息同步发送为例揭示其消息发送处理流程中的核心关键点。

MQ Client 消息发送端首先会利用网络通道将请求发送到 Broker，然后接收到请求结果后并调用 processSendResponse 方法对响应结果进行解析，如下图所示：



在这里，RemotingCommand 的 code 为 RemotingSysResponseCode.SYSTEM_BUSY。

我们从 proccessSendResponse 方法中可以得知，如果 code 为 SYSTEM_BUSY，该方法会抛出 MQBrokerException，响应 code 为 SYSTEM_BUSY，其错误描述为开头部分的错误信息。

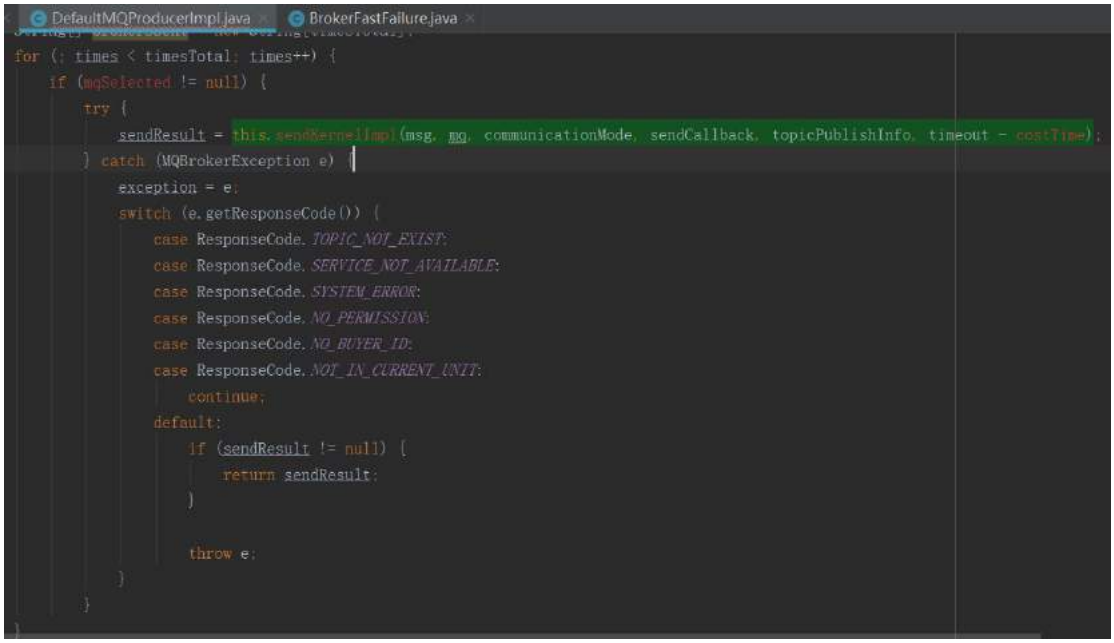
那我们沿着该方法的调用量，可以找到其直接调用方为：DefaultMQProducerImpl 的 sendKernelImpl，我们重点考虑如果底层方法抛出 MQBrokerException 该方法会如何处理。

其关键代码如下图所示：

```
byte[] prevBody = msg.getBody();
try {
    SendResult sendResult = null;
    switch (communicationMode) {
        case SYNC:
            sendResult = this.mQClientFactory.getMQClientAPIImpl().sendMessage(
} catch (RemotingException e) {
    if (this.hasSendMessageHook()) {
        context.setException(e);
        this.executeSendMessageHookAfter(context);
    }
    throw e;
} catch (MQBrokerException e) {
    if (this.hasSendMessageHook()) {
        context.setException(e);
        this.executeSendMessageHookAfter(context);
    }
    throw e;
} catch (InterruptedException e) {
```

可以看出在 sendKernelImpl 方法中首先会捕捉异常，先执行注册的钩子函数，即就算执行失败，对应的消息发送后置钩子函数也会执行，然后再原封不动的将该异常向上抛出。

sendKernelImpl 方法被 DefaultMQProducerImpl 的 sendDefaultImpl 方法调用，下面是其核心实现截图：



从这里可以看出 RocketMQ 消息发送高可用设计一个非常关键的点，重试机制，其实现是在 for 循环中 使用 try catch 将 sendKernelImpl 方法包裹，就可以保证该方法抛出异常后能继续重试。从上文可知，如果 SYSTEM_BUSY 会抛出 MQBrokerException，但发现只有上述几个错误码才会重试，因为如果不是上述错误码，会继续向外抛出异常，此时 for 循环会被中断，即不会重试。

这里非常令人意外的是连 SYSTEM_ERROR 都会重试，却没有包含 SYSTEM_BUSY，显然违背了快速失败的设计初衷，故笔者断定，这是 RocketMQ 的一个 BUG，将 SYSTEM_BUSY 遗漏了，后面与 RocketMQ 核心成员进行过沟通，也印证了这点，后续会提一个 PR，在上面增加一行代码，将 SYSTEM_BUSY 加上即可。

问题分析到这里，该问题应该就非常明了。

三、解决方案

如果大家在网上搜索 TIMEOUT_CLEAN_QUEUE 的解决方法，大家不约而同提出的解决方案是增加 waitTimeMillsInSendQueue 的值，该值默认为 200ms，例如将其设置为 1000s 等等，以前我是反对的，因为我的认知里 Broker 会重试，但现在发现 Broker 不会重试，故提高该值能有效的缓解。

但这是并不是好的解决方案，我会在近期向官方提交一个 PR，将这个问题修复，建议大家在公司尽量对自己使用的版本进行修改，重新打一个包即可，因为这已经违背了 Broker 端快速失败的设计初衷。

但在消息发送的业务方，尽量自己实现消息的重试机制，即不依懒 RocketMQ 本身提供的重试机制，因为受制与网络等因素，消息发送不可能百分之百成功，建议大家在消息发送时捕获一下异常，如果发送失败，可以将消息存入数据库，再结合定时任务对消息进行重试，尽最大程度保证消息不丢失。

、

1.13 RocketMQ DLedger 多副本即主从切换实战

实际操作如何实现从原先的主从同步平滑升级到主从切换。

本文首先先介绍与 DLedger 多副本即 RocketMQ 主从切换相关的核心配置属性，然后尝试搭建一个 DLedger 集群，从原先的 RocketMQ 集群平滑升级到 DLedger 集群的示例，并简单测试一下主从切换功能。

一、RocketMQ DLedger 多副本即主从切换核心配置参数详解

其主要的配置参数如下所示：

- enableDLegerCommitLog

是否启用 DLedger，即是否启用 RocketMQ 主从切换，默认值为 false。如果需要开启主从切换，则该值需要设置为 true 。

- dLegerGroup

节点所属的 raft 组，建议与 brokerName 保持一致，例如 broker-a。

- dLegerPeers

集群节点信息，示例配置如下：n0-127.0.0.1:40911;n1-127.0.0.1:40912;n2-127.0.0.1:40913，多个节点用英文冒号隔开，单个条目遵循 legerSlefId-ip:端口，这里的端口用作 dledger 内部通信。

- dLegerSelfId

当前节点 id。取自 legerPeers 中条目的开头，即上述示例中的 n0，并且特别需要强调，只能第一个字符为英文，其他字符需要配置成数字。

- storePathRootDir

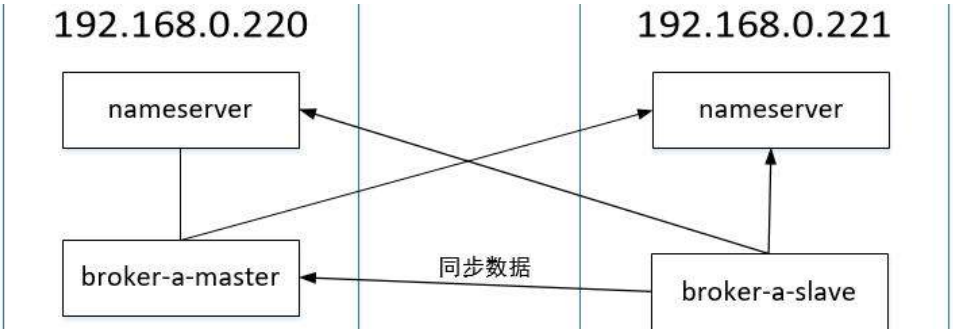
DLedger 日志文件的存储根目录，为了能够支持平滑升级，该值与 storePathCommitLog 设置为不同的目录。

二、搭建主从同步环境

首先先搭建一个传统意义上的主从同步架构，往集群中灌一定量的数据，然后升级到 DLedger 集群。

在 Linux 服务器上大家一个 rocketmq 主从同步集群我想不是一件很难的事情，故本文就不会详细介绍按照过程，只贴出相关配置。

实验环境的部署结构采取 一主一次，其部署图如下：



下面我就重点贴一下 broker 的配置文件。

220 上的 broker 配置文件如下：

```
brokerClusterName = DefaultCluster
brokerName = broker-a
brokerId = 0
deleteWhen = 04
fileReservedTime = 48
brokerRole = ASYNC_MASTER
flushDiskType = ASYNC_FLUSH
brokerIP1=192.168.0.220
brokerIP2=192.168.0.220
namesrvAddr=192.168.0.221:9876;192.168.0.220:9876
storePathRootDir=/opt/application/rocketmq-all-4.5.2-bin-release/store
storePathCommitLog=/opt/application/rocketmq-all-4.5.2-bin-release/store/commitlog
autoCreateTopicEnable=false
autoCreateSubscriptionGroup=false
```

221 上 broker 的配置文件如下：

```
brokerClusterName = DefaultCluster
brokerName = broker-a
brokerId = 1
deleteWhen = 04
fileReservedTime = 48
brokerRole = SLAVE
flushDiskType = ASYNC_FLUSH
brokerIP1=192.168.0.221
brokerIP2=192.168.0.221
namesrvAddr=192.168.0.221:9876;192.168.0.220:9876
storePathRootDir=/opt/application/rocketmq-all-4.5.2-bin-release/store
storePathCommitLog=/opt/application/rocketmq-all-4.5.2-bin-release/store/commitlog
autoCreateTopicEnable=false
autoCreateSubscriptionGroup=false
```

相关的启动命令如下：

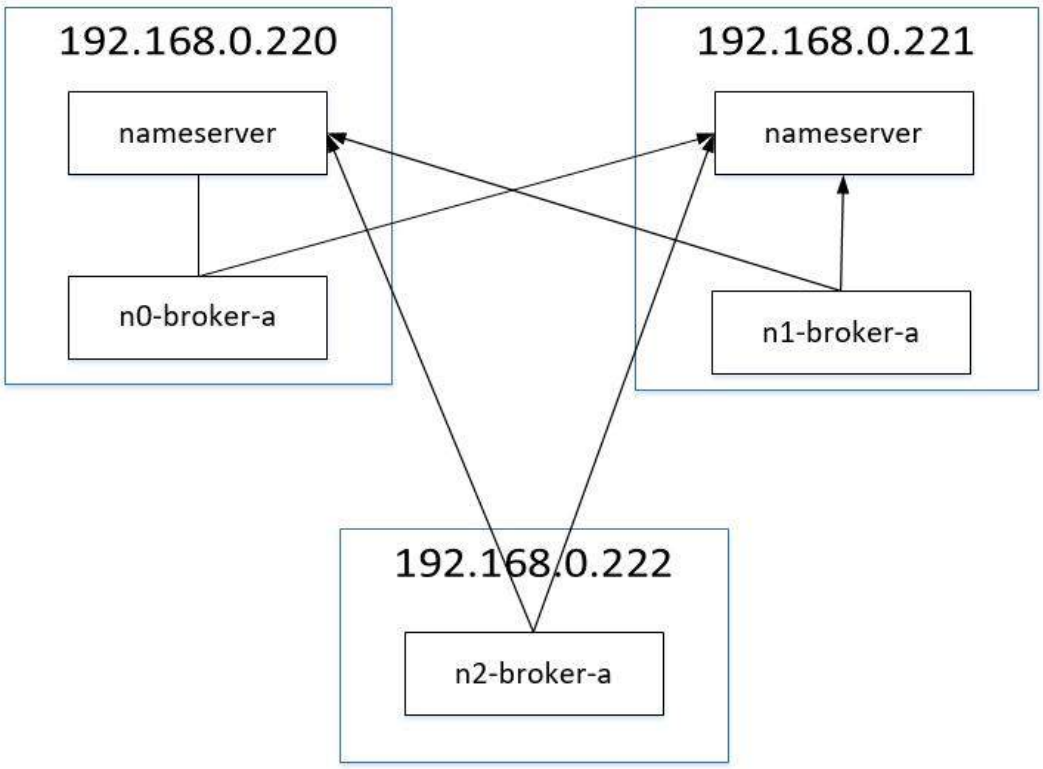
```
nohup bin/mqnamesrv  /dev/null  2>&1 &
nohup bin/mqbroker -c conf/broker.conf  /dev/null  2>&1 &
```

安装后的集群信息如图所示：

三、主从同步集群升级到 DLedger

1. 部署架构

DLedger 集群至少需要 3 台机器，故搭建 DLedger 还需要再引入一台机器，其部署结构图如下：



从主从同步集群升级到 DLedger 集群，用户最关心的还是升级后的集群是否能够兼容原先的数据，即原先存储在消息能否能被消息消费者消费端，甚至于能否查询到。

为了方便后续验证，首先我使用下述程序向 mq 集群中添加了一篇方便查询的消息（设置消息的 key）。

```
    public class Producer {
        public static void main(String[] args) throws MQClientException, InterruptedExcept
ion {
            DefaultMQProducer producer = new DefaultMQProducer("producer_dw_test");
            producer.setNamesrvAddr("192.168.0.220:9876;192.168.0.221:9876");
            producer.start();
            for(int i =600000; i < 600100; i ++) {
                try {
                    Message msg = new Message("topic_dw_test_by_order_01",null , "m"
 + i,("Hello RocketMQ" + i ).getBytes(RemotingHelper.DEFAULT_CHARSET));
                    SendResult sendResult = producer.send(msg);
                   //System.out.printf("%s%n", sendResult);
                } catch (Exception e) {
                    e.printStackTrace();
```

```
                    Thread.sleep(1000);
                }
            }
            producer.shutdown();
            System.out.println("end");
        }
    }
```

消息的查询结果示例如下：

2. 升级步骤

Step1：将 192.168.0.220 的 rocketmq 拷贝到 192.168.0.222，可以使用如下命令进行操作。在 192.168.0.220 上敲如下命令：

```
scp -r rocketmq-all-4.5.2-bin-release/ root@192.168.0.222:/opt/application/rocketmq-all-4.5.2-bin-release
```

温馨提示：示例中由于版本是一样，实际过程中，版本需要升级，故需先下载最新的版本，然后将老集群中的 store 目录完整的拷贝到新集群的 store 目录。

Step2：依次在三台服务器的 broker.conf 配置文件中添加与 dledger 相关的配置属性，修改后的 broker 配置属性如下：

192.168.0.220 broker 配置文件如下：

```
brokerClusterName = DefaultCluster
brokerId = 0
deleteWhen = 04
```

```
fileReservedTime = 48
brokerRole = ASYNC_MASTER
flushDiskType = ASYNC_FLUSH
brokerIP1=192.168.0.220
brokerIP2=192.168.0.220
namesrvAddr=192.168.0.221:9876;192.168.0.220:9876
storePathRootDir=/opt/application/rocketmq-all-4.5.2-bin-release/store
storePathCommitLog=/opt/application/rocketmq-all-4.5.2-bin-release/store/commitlog
autoCreateTopicEnable=false
autoCreateSubscriptionGroup=false
# 与 dledger 相关的属性
enableDLegerCommitLog=true
storePathRootDir=/opt/application/rocketmq-all-4.5.2-bin-release/store/dledger_store
dLegerGroup=broker-a
dLegerPeers=n0-192.168.0.220:40911;n1-192.168.0.221:40911;n2-192.168.0.222:40911
dLegerSelfId=n0
```

192.168.0.221 broker 配置文件如下：

```
brokerClusterName = DefaultCluster
brokerName = broker-a
brokerId = 1
deleteWhen = 04
fileReservedTime = 48
brokerRole = SLAVE
flushDiskType = ASYNC_FLUSH
brokerIP1=192.168.0.221
brokerIP2=192.168.0.221
namesrvAddr=192.168.0.221:9876;192.168.0.220:9876
storePathRootDir=/opt/application/rocketmq-all-4.5.2-bin-release/store
storePathCommitLog=/opt/application/rocketmq-all-4.5.2-bin-release/store/commitlog
autoCreateTopicEnable=false
autoCreateSubscriptionGroup=false
# 与 dledger 相关的配置属性
enableDLegerCommitLog=true
storePathRootDir=/opt/application/rocketmq-all-4.5.2-bin-release/store/dledger_store
dLegerGroup=broker-a
dLegerPeers=n0-192.168.0.220:40911;n1-192.168.0.221:40911;n2-192.168.0.222:40911
dLegerSelfId=n1
```


192.168.0.222 broker 配置文件如下：

```
brokerClusterName = DefaultCluster
brokerName = broker-a
brokerId = 0
deleteWhen = 04
fileReservedTime = 48
brokerRole = ASYNC_MASTER
flushDiskType = ASYNC_FLUSH
brokerIP1=192.168.0.222
brokerIP2=192.168.0.222
namesrvAddr=192.168.0.221:9876;192.168.0.220:9876
storePathRootDir=/opt/application/rocketmq-all-4.5.2-bin-release/store
storePathCommitLog=/opt/application/rocketmq-all-4.5.2-bin-release/store/commitlog
autoCreateTopicEnable=false
autoCreateSubscriptionGroup=false
# 与 dledger 相关的配置
enableDLegerCommitLog=true
storePathRootDir=/opt/application/rocketmq-all-4.5.2-bin-release/store/dledger_store
dLegerGroup=broker-a
dLegerPeers=n0-192.168.0.220:40911;n1-192.168.0.221:40911;n2-192.168.0.222:40911
dLegerSelfId=n2
```

温馨提示：legerSelfId 分别为 n0、n1、n2。在真实的生产环境中，broker 配置文件中的 storePathRootDir、storePathCommitLog 尽量使用单独的根目录，这样判断其磁盘使用率时才不会相互影响。

Step3：将 store/config 下的 所有文件拷贝到 dledger store 的 congfig 目录下：

```
cd /opt/application/rocketmq-all-4.5.2-bin-release/store/
cp config/* dledger_store/config/
```

温馨提示：该步骤按照各自按照时配置的目录进行复制即可。

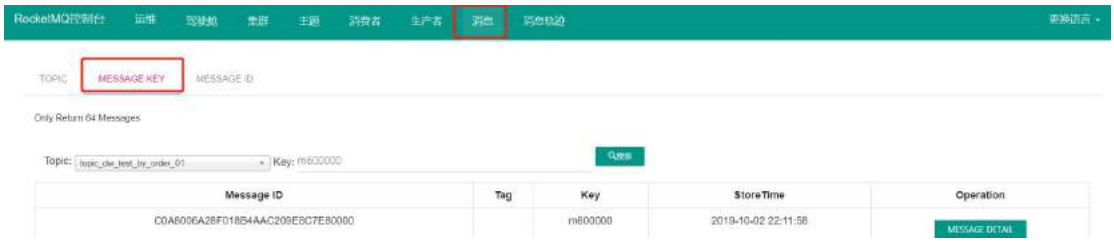
Step4：依次启动三台 broker。

```
nohup bin/mqbroker -c conf/broker.conf  /dev/null  2>&1 &
```

如果启动成功，则在 rocketmq-console 中看到的集群信息如下：

3. 验证消息发送与消息查找

首先我们先验证升级之前的消息是否能查询到，那我们还是查找 key 为 m600000 的消息，查找结果如图所示：



然后我们来测试一下消息发送。测试代码如下：

```
    public class Producer {
        public static void main(String[] args) throws MQClientException, InterruptedExcept
ion {
            DefaultMQProducer producer = new DefaultMQProducer("producer_dw_test");
            producer.setNamesrvAddr("192.168.0.220:9876;192.168.0.221:9876");
            producer.start();
            for(int i =600200; i < 600300; i ++) {
                try {
                    Message msg = new Message("topic_dw_test_by_order_01",null , "m"
 + i,("Hello RocketMQ" + i ).getBytes(RemotingHelper.DEFAULT_CHARSET));
                    SendResult sendResult = producer.send(msg);
                    System.out.printf("%s%n", sendResult);
                } catch (Exception e) {
                    e.printStackTrace();
                    Thread.sleep(1000);
                }
            }
            producer.shutdown();
            System.out.println("end");
        }}
```

执行结果如下：

再去控制台查询一下消息，其结果也表明新的消息也能查询到。

最后我们再来验证一下主节点宕机，消息发送是否会受影响。

在消息发送的过程中，去关闭主节点，其截图如下：

```
    at org.apache.rocketmq.client.producer.DefaultMQProducer.send(DefaultMQProducer.java:783)
    at persistent.prestige.mq.Producer.main(Producer.java:30)
Caused by: org.apache.rocketmq.remoting.exception.RemotingConnectException: connect to <192.168.0.232:10909> failed
    at org.apache.rocketmq.remoting.netty.NettyRemotingClient.invokeSync(NettyRemotingClient.java:392)
    at org.apache.rocketmq.client.impl.MQClientAPIImpl.sendMessageSync(MQClientAPIImpl.java:368)
    at org.apache.rocketmq.client.impl.MQClientAPIImpl.sendMessage(MQClientAPIImpl.java:340)
    at org.apache.rocketmq.client.impl.MQClientAPIImpl.sendMessage(MQClientAPIImpl.java:294)
    at org.apache.rocketmq.client.impl.producer.DefaultMQProducerImpl.sendKernelImpl(DefaultMQProducerImpl.java:608)
    at org.apache.rocketmq.client.impl.producer.DefaultMQProducerImpl.sendDefaultImpl(DefaultMQProducerImpl.java:552)
    ... 4 more
09:50:18.007 [NettyClientSelector_1] INFO  RocketmqRemoting - closeChannel: close the connection to remote address[] result: true
SendResult [sendStatus=SEND_OK, msgId=C0A8006A265018B4AAC20C6923AB0196, offsetMsgId=C0A800DD00002A9F0000000004001E684, messageQueue=MessageQueue
 [topic=topic_dw_test_by_order_01, brokerName=broker-a, queueId=0], queueOffset=37539]
SendResult [sendStatus=SEND_OK, msgId=C0A8006A265018B4AAC20C6927A70197, offsetMsgId=C0A800DD00002A9F0000000004001E77D, messageQueue=MessageQueue
```

```
 [topic=topic_dw_test_by_order_01, brokerName=broker-a, queueId=7], queueOffset=37572]
SendResult [sendStatus=SEND_OK, msgId=C0A8006A265018B4AAC20C693A1603E5, offsetMsgId=C0A800DD00002A9F0000000004004255B, messageQueue=MessageQueue
 [topic=topic_dw_test_by_order_01, brokerName=broker-a, queueId=8], queueOffset=37573]
SendResult [sendStatus=SEND_OK, msgId=C0A8006A265018B4AAC20C693A1903E6, offsetMsgId=C0A800DD00002A9F0000000004004265A, messageQueue=MessageQueue
 [topic=topic_dw_test_by_order_01, brokerName=broker-a, queueId=9], queueOffset=37573]
SendResult [sendStatus=SEND_OK, msgId=C0A8006A265018B4AAC20C693A1B03E7, offsetMsgId=C0A800DD00002A9F0000000004004274D, messageQueue=MessageQueue
 [topic=topic_dw_test_by_order_01, brokerName=broker-a, queueId=10], queueOffset=37575]
09:50:22.771 [NettyClientSelector_1] INFO  RocketmqRemoting - closeChannel: close the connection to remote address[192.168.0.220:10911] result: true
09:50:22.771 [NettyClientSelector_1] INFO  RocketmqRemoting - closeChannel: close the connection to remote address[192.168.0.222:10911] result: true
09:50:22.771 [NettyClientSelector_1] INFO  RocketmqRemoting - closeChannel: close the connection to remote address[192.168.0.221:10911] result: true
09:50:22.772 [NettyClientSelector_1] INFO  RocketmqRemoting - closeChannel: close the connection to remote address[192.168.0.221:9876] result: true
end
09:50:22.872 [NettyClientSelector_1] INFO  RocketmqRemoting - closeChannel: close the connection to remote address[192.168.0.221:10909] result: true

Process finished with exit code 0
```

再来看一下集群的状态：

等待该复制组重新完成主服务器选举后，即可继续处理消息发送。

温馨提示：由于本示例是一主一从，故在选举期间，消息不可用，但在真实的生产环境上，其部署架构是多主主从，即一个复制组在 leader 选举期间，其他复制组可以接替该复制组完成消息的发送，实现消息服务的高可用。

与 DLedger 相关的日志，默认存储在 broker_default.log 文件中。

1.14 RocketMQ msgId 与 offsetMsgId 释疑

消息发送、消息消费、RocketMQ queryMsgById 命令以及 rocketmq-console 等使用场景中究竟是用的哪一个 ID。

一、抛出问题

1. 从消息发送看消息 ID

```
package org.apache.rocketmq.example.quickstart;
import org.apache.rocketmq.client.producer.DefaultMQProducer;
import org.apache.rocketmq.client.producer.SendResult;
import org.apache.rocketmq.common.message.Message;
import org.apache.rocketmq.remoting.common.RemotingHelper;
public class Producer {
    public static void main(String[] args) {
        try {
            DefaultMQProducer producer = new DefaultMQProducer("please_rename_unique_group_name");
            producer.setNamesrvAddr("127.0.0.1:9876");
            producer.start();
            Message msg = new Message("TestTopic" /* Topic */,null /* Tag */, ("Hello RocketMQ test1" ).getBytes(RemotingHelper.DEFAULT_CHARSET) /* Message body */);
            SendResult sendResult = producer.send(msg);
            System.out.printf("%s%n", sendResult);
            producer.shutdown();
        } catch (Throwable e) {
            e.printStackTrace();
        }

    }
}
```

执行效果如图所示：

```
SendResult [sendStatus=SEND_OK, msgId=0000000000000000000000000000000000001000018B4AAC2199E56FE0000,
offsetMsgId=C0A8006500002A9F0000000000000803, messageQueue=MessageQueue [topic=TestTopic, brokerName=broker-a,
queueId=0], queueOffset=0]
```

即消息发送会返回 msgId 与 offsetMsgId。

2. 从消息消费看消息 ID

```
package org.apache.rocketmq.example.quickstart;
import java.util.List;
import org.apache.rocketmq.client.consumer.DefaultMQPushConsumer;
import org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlyContext;
import org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlyStatus;
import org.apache.rocketmq.client.consumer.listener.MessageListenerConcurrently;
import org.apache.rocketmq.client.exception.MQClientException;
import org.apache.rocketmq.common.consumer.ConsumeFromWhere;
import org.apache.rocketmq.common.message.MessageExt;
public class Consumer {
    public static void main(String[] args) throws InterruptedException, MQClientExcept
ion {
        DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("please
_rename_unique_group_name_1");
        consumer.setNamesrvAddr("127.0.0.1:9876");
        consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_F
IRST_OFFSET);
        consumer.subscribe("TestTopic", "*");
        consumer.registerMessageListener(new MessageListenerConcurrently() {
            public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt>
msgs,
                    ConsumeConcurrentlyContext context) {
                    System.out.println("MessageExt msg.getMsgId():" +  msgs.get(0).getM
sgId());
                    System.out.println("--------------------分割线----------------
-");
                    System.out.printf("%s Receive New Messages: %s %n", Thread.curre
ntThread().getName(), msgs);
                    return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
```

```
                }
            });
            consumer.start();
            System.out.printf("Consumer Started.%n");
        }
    }
```

```
Consumer Started.
MessageExt msg.getMsgId():0000000000000000000000000000000001000018B4AAC2199E56FE0000
--------------------分割线----------------
ConsumeMessageThread_1 Receive New Messages: [MessageExt [queueId=0, storeSize=219, queueOffset=0, sysFlag=0,
bornTimestamp=1583421807359, bornHost=/192.168.0.101:53323, storeTimestamp=1583421807367,
storeHost=/192.168.0.101:10911, msgId=C0A8006500002A9F0000000000000803, commitLogOffset=2051,
bodyCRC=1963542861, reconsumeTimes=0, preparedTransactionOffset=0, toString()=Message{topic='TestTopic',
flag=0, properties={MIN_OFFSET=0, MAX_OFFSET=1, CONSUME_START_TIME=1583421977602,
UNIQ_KEY=0000000000000000000000000000000001000018B4AAC2199E56FE0000, CLUSTER=DefaultCluster, WAIT=true},
body=[72, 101, 108, 108, 111, 32, 82, 111, 99, 107, 101, 116, 77, 81, 32, 116, 101, 115, 116, 49],
transactionId='null'}]]
```

不知道大家是否有注意到，调用 msgs.get(0).getMsgId()返回的 msgId 与直接输出 msgs 中的 msgId 不一样，那这又是为什么呢?

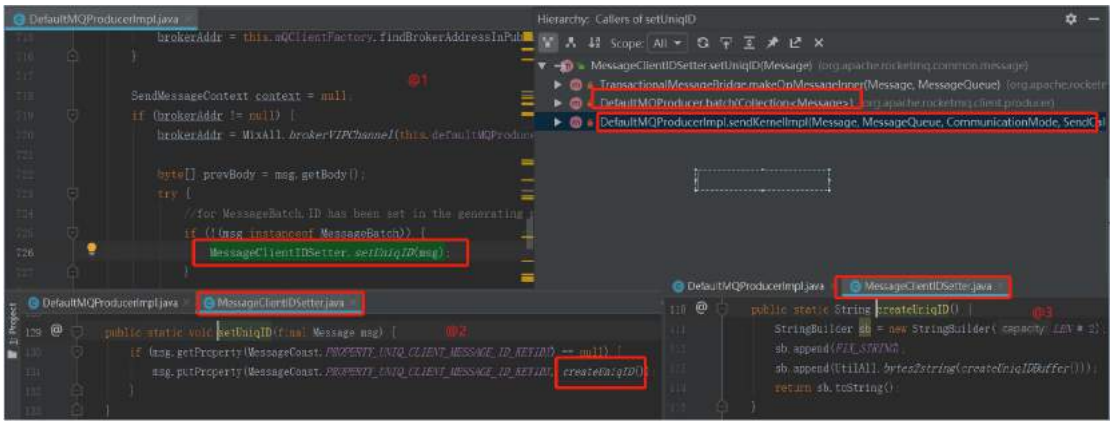
二、消息 ID 释疑

从消息发送的结果可以得知，RocketMQ 发送的返回结果会返回 msgId 与 offsetMsgId，那这两个 msgId 分别是代表什么呢?

msgId：该 ID 是消息发送者在消息发送时会首先在客户端生成，全局唯一，在 RocketMQ 中该 ID 还有另外的一个叫法：uniqId，无不体现其全局唯一性。

offsetMsgId：消息偏移 ID，该 ID 记录了消息所在集群的物理地址，主要包含所存储 Broker 服务器的地址(IP 与端口号)以及所在 commitlog 文件的物理偏移量。

1. msgId 即全局唯一 ID 构建规则



从这张图可以看出，msgId 确实是客户端生成的，接下来我们详细分析一下其生成算法。

```
MessageClientIDSetter#createUniqID
public static String createUniqID() {
    StringBuilder sb = new StringBuilder(LEN * 2);
    sb.append(FIX_STRING);      // @1
    sb.append(UtilAll.bytes2string(createUniqIDBuffer()));   // @2
    return sb.toString();
}
```

一个 uniqID 的构建主要分成两个部分：FIX_STRING 与唯一 ID 生成算法，顾名思义，FIX_STRING 就是一个客户端固定一个前缀，那接下来先看一下固定字符串的生成规则。

FIX_STRING

MessageClientIDSetter 静态代码块：

```
static {
    byte[] ip;
    try {
        ip = UtilAll.getIP();
    } catch (Exception e) {
        ip = createFakeIP();
    }
    LEN = ip.length + 2 + 4 + 4 + 2;
    ByteBuffer tempBuffer = ByteBuffer.allocate(ip.length + 2 + 4);
```

```
        tempBuffer.position(0);
        tempBuffer.put(ip);
        tempBuffer.position(ip.length);
        tempBuffer.putInt(UtilAll.getPid());
        tempBuffer.position(ip.length + 2);
        tempBuffer.putInt(MessageClientIDSetter.class.getClassLoader().hashCode());
        FIX_STRING = UtilAll.bytes2string(tempBuffer.array());
        setStartTime(System.currentTimeMillis());
        COUNTER = new AtomicInteger(0);
    }
```

从这里可以看出 FIX_STRING 的主要由：客户端的 IP、进程 ID、加载 MessageClientIDSetter 的类加载器的 hashcode。

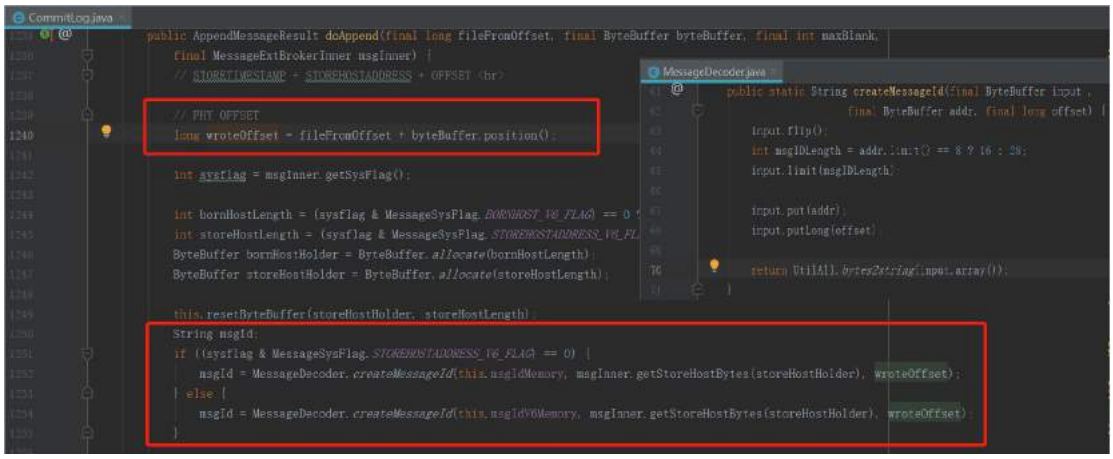
唯一性算法

msgId 的唯一性算法由 MessageClientIDSetter 的 createUniqIDBuffer 方法实现。

```
    private static byte[] createUniqIDBuffer() {
        ByteBuffer buffer = ByteBuffer.allocate(4 + 2);
        long current = System.currentTimeMillis();
        if (current >= nextStartTime) {
            setStartTime(current);
        }
        buffer.position(0);
        buffer.putInt((int) (System.currentTimeMillis() - startTime));
        buffer.putShort((short) COUNTER.getAndIncrement());
        return buffer.array();
    }
```

可以得出 msgId 的后半段主要由：当前时间与系统启动时间的差值，以及自增序号。

2. offsetMsgId 构建规则



在消息 Broker 服务端将消息追加到内存后会返回其物理偏移量，即在 commitlog 文件中的文件，然后会再次生成一个 id，代码中虽然也叫 msgId，其实这里就是我们常说的 offsetMsgId，即记录了消息的物理偏移量，故我们重点来看一下其具体生成规则：

```
MessageDecoder#createMessageId
public static String createMessageId(final ByteBuffer input ,
                final ByteBuffer addr, final long offset) {
 input.flip();
     int msgIDLength = addr.limit() == 8 ? 16 : 28;
     input.limit(msgIDLength);
     input.put(addr);
     input.putLong(offset);
     return UtilAll.bytes2string(input.array());
}
```

首先结合该方法的调用上下文，先解释一下该方法三个入参的含义：

- ByteBuffer input

  用来存放 offsetMsgId 的字节缓存区(NIO 相关的基础知识)

- ByteBuffer addr

  当前 Broker 服务器的 IP 地址与端口号，即通过解析 offsetMsgId 从而得到消息服务器的地址信息。

- long offset

  消息的物理偏移量。

即构成 offsetMsgId 的组成部分：Broker 服务器的 IP 与端口号、消息的物理偏移量。

即在 RocketMQ 中，只需要提供 offsetMsgId，可用不必知道该消息所属的 topic 信息即可查询该条消息的内容。

3. 消息发送与消息消费返回的消息 ID 信息

消息发送时会在 SendSesult 中 返回 msgId、offsetMsgId，在了解了这个两个 ID 的含义时则问题不大，接下来重点介绍一下消息消费时返回的 msgId 到底是哪一个。

在消息消费时，我们更加希望因为 msgId(即客户端生成的全局唯一性 ID)，因为该全局性 ID 非常方便实现消费端的幂等。

在本文的 1.2 节我们也提到一个现象，为什么在 msgs.get(0).getMsgId()返回的 id 与直接用 System.out.println(msg)返回的 msgId 不一样呢?

```
import org.apache.rocketmq.common.message.MessageExt;
public class Consumer {
    public static void main(String[] args) throws InterruptedException, MQClientException {
        DefaultMQPushConsumer consumer = new DefaultMQPushConsumer( consumerGroup: "please_rename_unique_group_name_1");
        consumer.setNamesrvAddr("127.0.0.1:9876");
        consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_OFFSET);
        consumer.subscribe( topic: "TestTopic",  subExpression: "*");
        consumer.registerMessageListener(new MessageListenerConcurrently() {
            public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs,
                ConsumeConcurrentlyContext context) {
                System.out.println("MessageExt msg.getMsgId():" + msgs.get(0).getMsgId());
                System.out.println("----------------分割线----------------");
                System.out.printf("%s Receive New Messages: %s %n", Thread.currentThread().getName() msgs);
                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
            }
        });
        consumer.start();
        System.out.printf("Consumer Started.%n");
    }
}
```

在客户端返回的 msg 信息，其最终返回的对象是 MessageClientExt，继承自 MessageExt。

那我们接下来分别看一下其 getMsgId()方法与 toString 方法即可。

```
@Override
public String getMsgId() {
    String uniqID = MessageClientIDSetter.getUniqID(this);
    if (uniqID == null) {
        return this.getOffsetMsgId();
    } else {
        return uniqID;
    }
}
```

原来在调用 MessageClientExt 中的 getMsgId 方法时，如果消息的属性中存在其唯一 ID，则返回消息的全局唯一 ID，否则返回消息的 offsetMsgId。

而 MessageClientExt 方法并没有重写 MessageExt 的 toString 方法，其实现如图所示：

```
@Override
public String toString() {
    return "MessageExt [queueId=" + queueId + ", storeSize=" + storeSize + ", queueOffset=" + queueOffset
        + ", sysFlag=" + sysFlag + ", bornTimestamp=" + bornTimestamp + ", bornHost=" + bornHost
        + ", storeTimestamp=" + storeTimestamp + ", storeHost=" + storeHost + ", msgId=" + msgId
        + ", commitLogOffset=" + commitLogOffset + ", bodyCRC=" + bodyCRC + ", reconsumeTimes="
        + reconsumeTimes + ", preparedTransactionOffset=" + preparedTransactionOffset
        + ", toString()=" + super.toString() + "]";
}
```

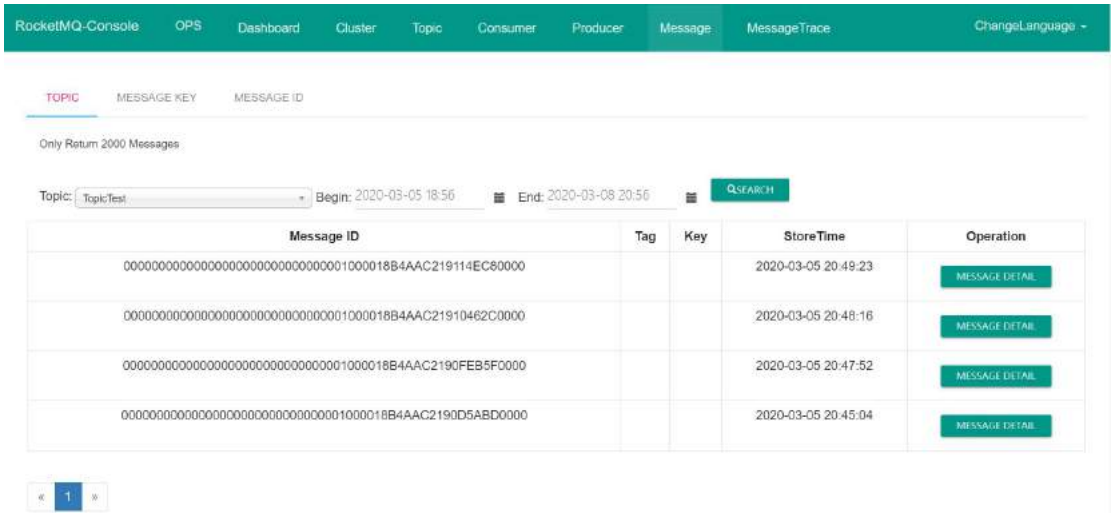
故返回的是 MessageExt 中的 msgId，该 msgId 存放的是 offsetMsgId，所以才造成了困扰。

温馨提示：如果消息消费失败需要重试，RocketMQ 的做法是将消息重新发送到 Broker 服务器，此时全局 msgId 是不会发送变化的，但该消息的 offsetMsgId 会发送变化，因为其存储在服务器中的位置发生了变化。

三、实践经验

在回答了消息发送与消息消费关于 msgId 与 offsetMsgId 的困扰后，再来介绍一下如果根据 msgId 去查询消息。

想必大家对 rocketmq-console ，那在消息查找界面，展示的消息列表中返回的 msgId 又是哪一个呢?



这里的 Message ID 返回的是消息的全局唯一 id。

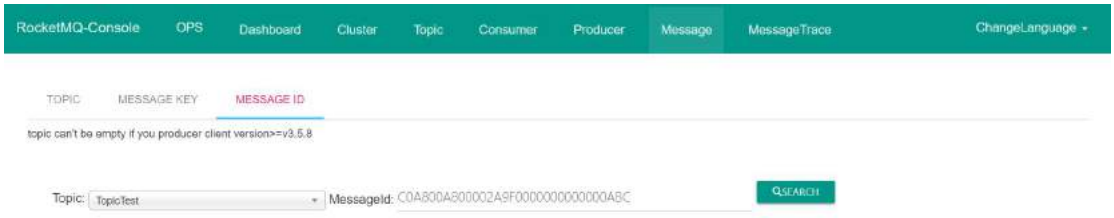
其实 RokcetMQ 也提供了 queryMsgById 命令来查看消息的内容，不过这里的 msgId 是 offsetMsgId，我们首先将全局唯一 ID 传入命令，其执行效果如下:

```
[root@localhost bin]# sh ./mqadmin queryMsgById -n 127.0.0.1:9876  -i 0000000000000000000000000001000018B4AAC2190FEB5F0000
RocketMQLog:WARN No appenders could be found for logger (io.netty.util.internal.PlatformDependent0).
RocketMQLog:WARN Please initialize the logger system properly.
org.apache.rocketmq.tools.command.SubCommandException: QueryMsgByIdSubCommand command failed
        at org.apache.rocketmq.tools.command.message.QueryMsgByIdSubCommand.execute(QueryMsgByIdSubCommand.java:253)
        at org.apache.rocketmq.tools.command.MQAdminStartup.main0(MQAdminStartup.java:139)
        at org.apache.rocketmq.tools.command.MQAdminStartup.main(MQAdminStartup.java:90)
Caused by: org.apache.rocketmq.remoting.exception.RemotingConnectException: connect to 0:0:0:0:0:0:0:1:6324 failed
        at org.apache.rocketmq.remoting.netty.NettyRemotingClient.invokeSync(NettyRemotingClient.java:390)
        at org.apache.rocketmq.client.impl.MQClientAPIImpl.viewMessage(MQClientAPIImpl.java:814)
        at org.apache.rocketmq.client.impl.MQAdminImpl.viewMessage(MQAdminImpl.java:264)
        at org.apache.rocketmq.tools.admin.DefaultMQAdminExtImpl.viewMessage(DefaultMQAdminExtImpl.java:990)
        at org.apache.rocketmq.tools.admin.DefaultMQAdminExt.viewMessage(DefaultMQAdminExt.java:127)
        at org.apache.rocketmq.tools.command.message.QueryMsgByIdSubCommand.queryById(QueryMsgByIdSubCommand.java:47)
        at org.apache.rocketmq.tools.command.message.QueryMsgByIdSubCommand.execute(QueryMsgByIdSubCommand.java:247)
        ... 2 more
```

发现报错，那我们将 offsetMsgId 传入其执行效果如图所示:

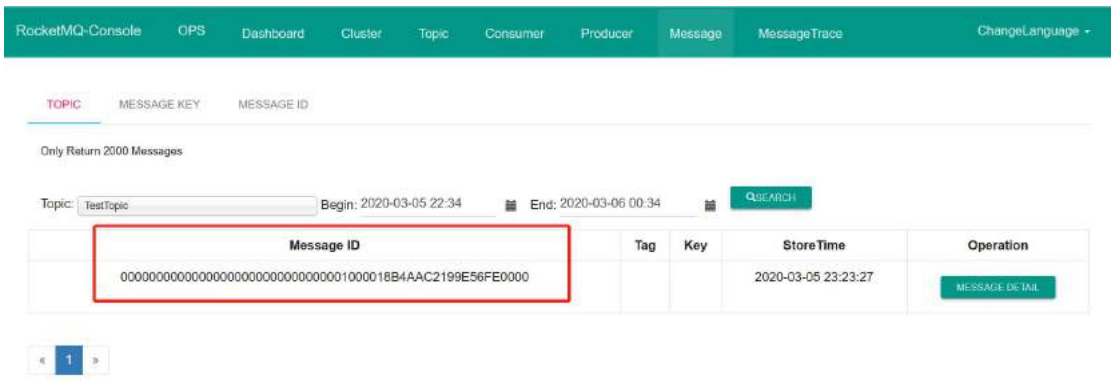
```
[root@localhost bin]# sh ./mqadmin queryMsgById -n 127.0.0.1:9876  -i C0A800A800002A9F0000000000000ABC
RocketMQLog:WARN No appenders could be found for logger (io.netty.util.internal.PlatformDependent0).
RocketMQLog:WARN Please initialize the logger system properly.
OffsetID:           C0A800A800002A9F0000000000000ABC
Topic:              TopicTest
Tags:               [null]
Keys:               [null]
Queue ID:           1
Queue Offset:       2
CommitLog Offset:   2748
Reconsume Times:    0
Born Timestamp:     2020-03-05 07:47:53,697
Store Timestamp:    2020-03-05 07:47:52,909
Born Host:          192.168.0.101:58058
Store Host:         192.168.0.168:10911
System Flag:        0
Properties:         {UNIQ_KEY=0000000000000000000000000000000001000018B4AAC2190FEB5F0000, CLUSTER=DefaultCluster, WAIT=true}
Message Body Path:  /tmp/rocketmq/msgbodys/0000000000000000000000000000000001000018B4AAC2190FEB5F0000
```

但在 rocketmq-console 的根据消息 ID 去查找消息，无论传入哪个 msgId，下图该功能都能返回正确的结果：



这是因为 rocketmq-console 做了兼容，先将传入的 msgId 用 queryMsgById 该命令去查，如果报错，则当成 uniqID(全局 ID)去查，首先全局 id 会存储在消息的属性中，并会创建 Hash 索引，即可用通过 indexFile 快速定位到该条消息。

rocketmq-console 消息查找消息列表中的 id 又是什么呢?



1.15 RocketMQ ACL 使用指南

一、什么是 ACL?

ACL 是 access control list 的简称，俗称访问控制列表。访问控制，基本上会涉及到用户、资源、权限、角色等概念，那在 RocketMQ 中上述会对应哪些对象呢?

- 用户

用户是访问控制的基础要素，也不难理解，RocketMQ ACL 必然也会引入用户的概念，即支持用户名、密码。

- 资源

资源，需要保护的对象，在 RocketMQ 中，消息发送涉及的 Topic、消息消费涉及的消费组，应该进行保护，故可以抽象成资源。

- 权限

针对资源，能进行的操作，

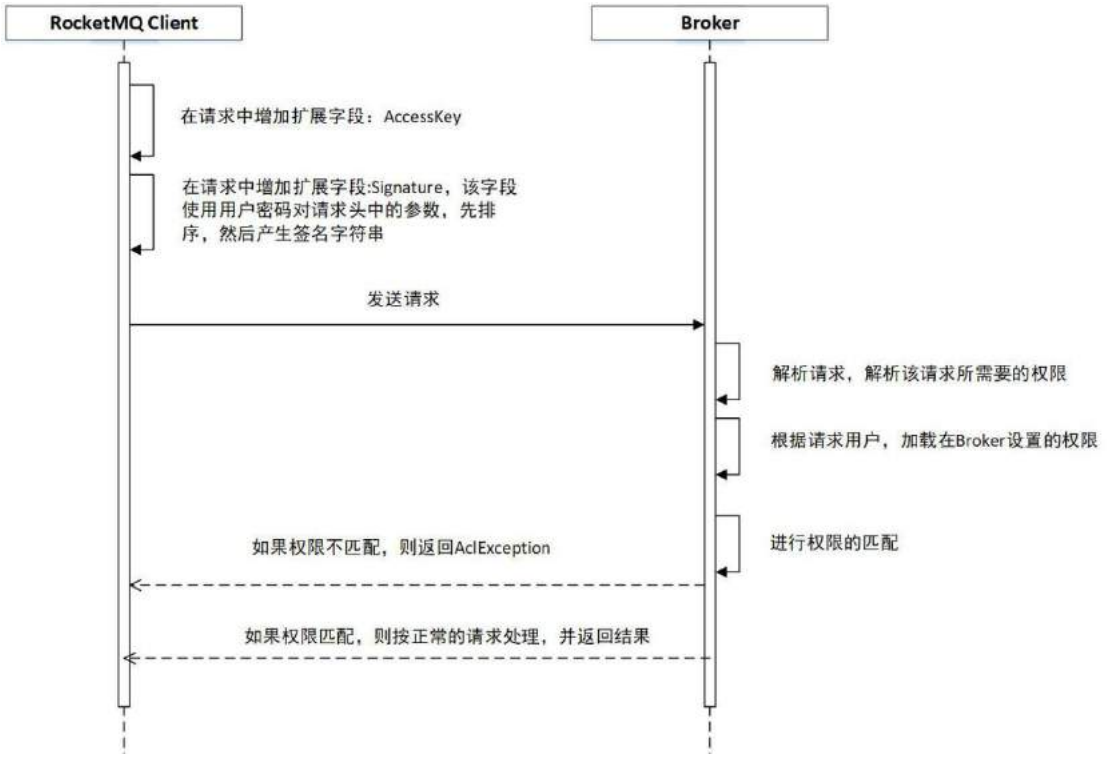
- 角色

RocketMQ 中，只定义两种角色：是否是管理员。

另外，RocketMQ 还支持按照客户端 IP 进行白名单设置。

二、ACL 基本流程图

在讲解如何使用 ACL 之前，我们先简单看一下 RocketMQ ACL 的请求流程:



对于上述具体的实现，将在后续文章中重点讲解，本文的目的只是希望给读者一个大概的了解。

三、如何配置 ACL

1. acl 配置文件

acl 默认的配置文件名：plain_acl.yml,需要放在${ROCKETMQ_HOME}/store/config 目录下。下面对其配置项一一介绍。

globalWhiteRemoteAddresses

全局白名单，其类型为数组，即支持多个配置。其支持的配置格式如下：

- 空

表示不设置白名单，该条规则默认返回 false。

- *

  表示全部匹配，该条规则直接返回 true，将会阻断其他规则的判断，请慎重使用。

- 192.168.0.{100,101}

  多地址配置模式，ip 地址的最后一组，使用{}，大括号中多个 ip 地址，用英文逗号(,)隔开。

- 192.168.1.100,192.168.2.100

  直接使用,分隔，配置多个 ip 地址。

- 192.168.*.*或 192.168.100-200.10-20

  每个 IP 段使用*或-表示范围。

- accounts

  配置用户信息，该类型为数组类型。拥有 accessKey、secretKey、whiteRemoteAddress、admin、defaultTopicPerm、defaultGroupPerm、topicPerms、groupPerms 子元素。

- accessKey

  登录用户名，长度必须大于 6 个字符。

- secretKey

  登录密码。长度必须大于 6 个字符。

- whiteRemoteAddress

  用户级别的 IP 地址白名单。其类型为一个字符串，其配置规则与 globalWhiteRemoteAddresses，但只能配置一条规则。

- admin

  boolean 类型，设置是否是 admin。如下权限只有 admin=true 时才有权限执行。

- UPDATE_AND_CREATE_TOPIC

  更新或创建主题。

- UPDATE_BROKER_CONFIG
  更新 Broker 配置。

- DELETE_TOPIC_IN_BROKER
  删除主题。

- UPDATE_AND_CREATE_SUBSCRIPTIONGROUP
  更新或创建订阅组信息。

- DELETE_SUBSCRIPTIONGROUP
  删除订阅组信息。

- defaultTopicPerm
  默认 topic 权限。该值默认为 DENY(拒绝)。

- defaultGroupPerm
  默认消费组权限，该值默认为 DENY(拒绝)，建议值为 SUB。

- topicPerms
  设置 topic 的权限。其类型为数组，其可选择值在下节介绍。

- groupPerms
  设置消费组的权限。其类型为数组，其可选择值在下节介绍。可以为每一消费组配置不一样的权限。

2. RocketMQ ACL 权限可选值

- DENY
  拒绝。

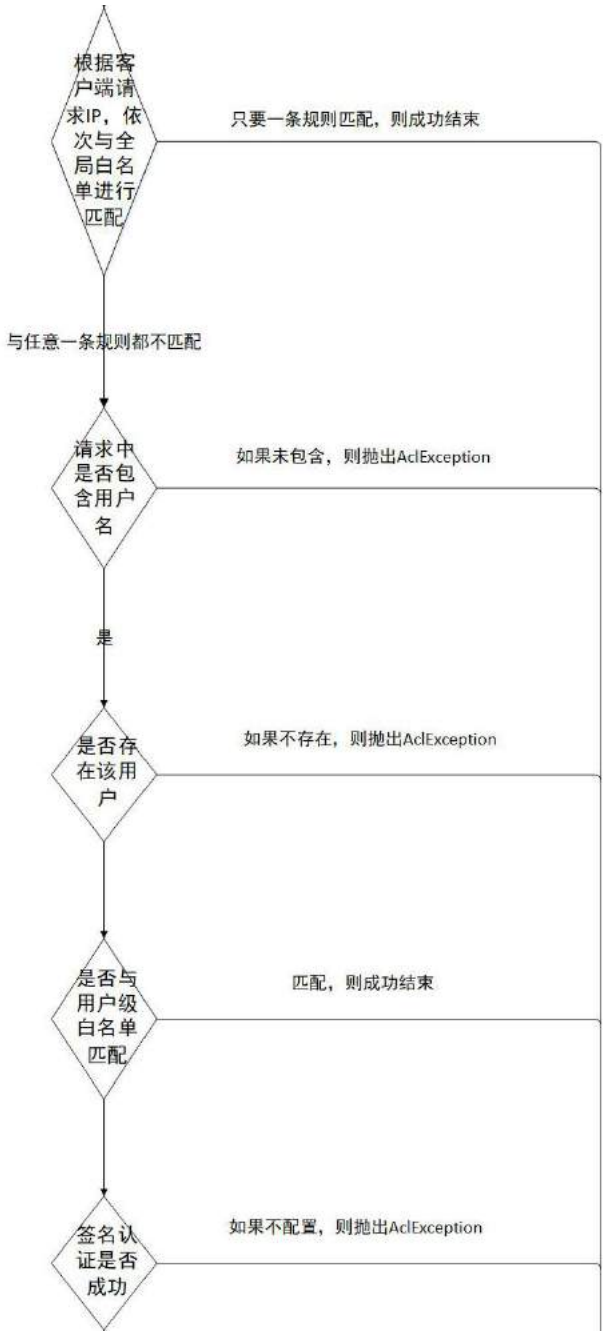
- PUB
  拥有发送权限。

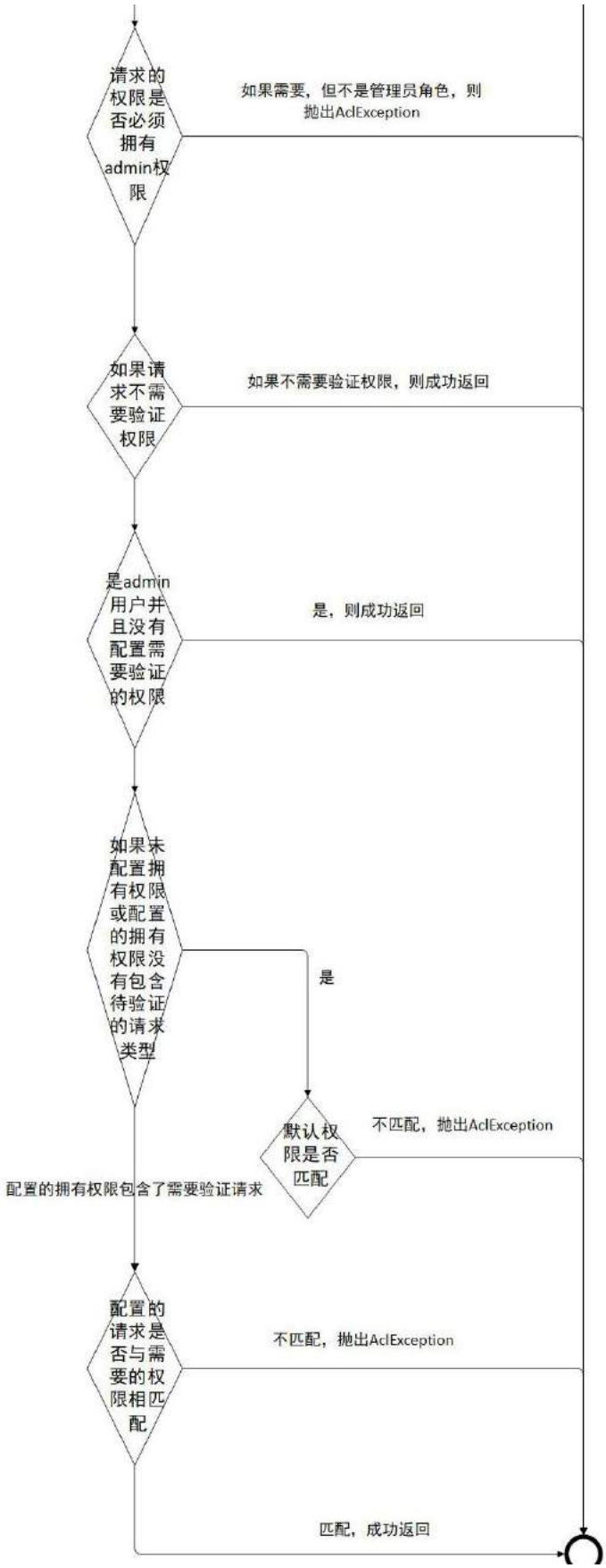
- SUB

  拥有订阅权限。

3. 权限验证流程

上面定义了全局白名单、用户级别的白名单，用户级别的权限，为了更好的配置 ACL 权限规则，下面给出权限匹配逻辑。





四、使用示例

1. Broker 端安装

首先，需要在 broker.conf 文件中，增加参数 aclEnable=true。并拷贝 distribution/conf/plain_acl.yml 文件到${ROCKETMQ_HOME}/conf 目录。

broker.conf 的配置文件如下：

```
brokerClusterName = DefaultCluster
brokerName = broker-b
brokerId = 0
deleteWhen = 04
fileReservedTime = 48
brokerRole = ASYNC_MASTER
flushDiskType = ASYNC_FLUSH
listenPort=10915
storePathRootDir=E:/SH2019/tmp/rocketmq_home/rocketmq4.5MB/store
storePathCommitLog=E:/SH2019/tmp/rocketmq_home/rocketmq4.5MB/store/commitlog
namesrvAddr=127.0.0.1:9876
autoCreateTopicEnable=false
aclEnable=true
```

plain_acl.yml 文件内容如下：

```
globalWhiteRemoteAddresses:

accounts:
- accessKey: RocketMQ
  secretKey: 12345678
  whiteRemoteAddress:
  admin: false
  defaultTopicPerm: DENY
  defaultGroupPerm: SUB
  topicPerms:
  - TopicTest=PUB
  groupPerms:
```

```
    # the group should convert to retry topic
    - please_rename_unique_group_name_4=DENY

  - accessKey: admin
    secretKey: 12345678
    whiteRemoteAddress:
    # if it is admin, it could access all resources
    admin: true
```

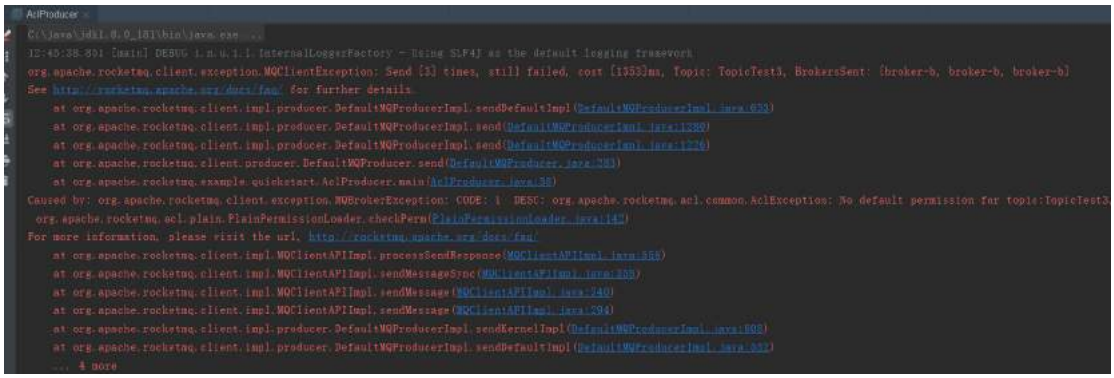
从上面的配置可知，用户 RocketMQ 只能发送 TopicTest 的消息，其他 topic 无权限发送；拒绝 please_rename_unique_group_name_4 消费组的消息消费，其他消费组默认可消费。

2. 消息发送端示例

```
public class AclProducer {
    public static void main(String[] args) throws MQClientException, InterruptedExcept
ion {
        DefaultMQProducer producer = new DefaultMQProducer("please_rename_uni
que_group_name", getAclRPCHook());
        producer.setNamesrvAddr("127.0.0.1:9876");
        producer.start();
        for (int i = 0; i < 1; i++) {
            try {
                Message msg = new Message("TopicTest3" ,"TagA" , ("Hello Rocket
MQ " + i).getBytes(RemotingHelper.DEFAULT_CHARSET));
                SendResult sendResult = producer.send(msg);
                System.out.printf("%s%n", sendResult);
            } catch (Exception e) {
                e.printStackTrace();
                Thread.sleep(1000);
            }
        }
        producer.shutdown();
    }

    static RPCHook getAclRPCHook() {
```

```
        return new AclClientRPCHook(new SessionCredentials("rocketmq","12345678"));
        }
    }
```

运行效果如图所示：



3. 消息消费端示例

```
    public class AclConsumer {

        public static void main(String[] args) throws InterruptedException, MQClientExcept
ion {
            DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("please
_rename_unique_group_name_4", getAclRPCHook(),new AllocateMessageQueueAveragel
y());
            consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_F
IRST_OFFSET);
            consumer.subscribe("TopicTest", "*");
            consumer.setNamesrvAddr("127.0.0.1:9876");
            consumer.registerMessageListener(new MessageListenerConcurrently() {
                @Override
                public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt>
msgs,
                    ConsumeConcurrentlyContext context) {
                    System.out.printf("%s Receive New Messages: %s %n", Thread.curre
ntThread().getName(), msgs);
                    return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
                }
```

```
            });
            consumer.start();
            System.out.printf("Consumer Started.%n");
        }

        static RPCHook getAclRPCHook() {
            return new AclClientRPCHook(new SessionCredentials("rocketmq","12345678
"));
        }
    }
```

发现并不没有消息被消费，符合预期。

关于 RocketMQ ACL 的使用就介绍到这里了，下一篇将介绍 RocketMQ ACL 实现原理。

1.16 RocketMQ 消息轨迹-设计篇

RocketMQ 消息轨迹主要包含两篇文章：设计篇与源码分析篇，本节将详细介绍 RocketMQ 消息轨迹-设计相关。

RocketMQ 消息轨迹，主要跟踪消息发送、消息消费的轨迹，即详细记录消息各个处理环节的日志，从设计上至少需要解决如下三个核心问题：

- 消费轨迹数据格式
- 记录消息轨迹(消息日志)
- 消息轨迹数据存储在哪?

一、消息轨迹数据格式

RocketMQ4.5 版本消息轨迹主要记录如下信息：

- traceType

  跟踪类型，可选值：Pub(消息发送)、SubBefore(消息拉取到客户端，执行业务定义的消费逻辑之前)、SubAfter(消费后)。

- timeStamp

  当前时间戳。

- regionId

  broker 所在的区域 ID，取自 BrokerConfig#regionId。

- groupName

  组名称，traceType 为 Pub 时为生产者组的名称；如果 traceType 为 subBefore 或 subAfter 时为消费组名称。

- requestId

  traceType 为 subBefore、subAfter 时使用，消费端的请求 Id。

- topic
  消息主题。

- msgId
  消息唯一 ID。

- tags
  消息 tag。

- keys
  消息索引 key，根据该 key 可快速检索消息。

- storeHost
  跟踪类型为 PUB 时为存储该消息的 Broker 服务器 IP；跟踪类型为 subBefore、subAfter 时为消费者 IP。

- bodyLength
  消息体的长度。

- costTime
  耗时。

- msgType
  消息的类型，可选值：Normal_Msg(普通消息),Trans_Msg_Half(预提交消息),Trans_msg_Commit(提交消息),Delay_Msg(延迟消息)。

- offsetMsgId
  消息偏移量 ID,该 ID 中包含了 broker 的 ip 以及偏移量。

- success
  是发送成功。

- contextCode

消费状态码，可选值：SUCCESS,TIME_OUT,EXCEPTION,RETURNNULL,FAILED。

二、记录消息轨迹

消息中间件的两大核心主题：消息发送、消息消费，其核心载体就是消息，消息轨迹（消息的流转）主要是记录消息是何时发送到哪台 Broker，发送耗时多少时间，在什么是被哪个消费者消费。记录消息的轨迹主要是集中在消息发送前后、消息消费前后，可以通过 RokcetMQ 的 Hook 机制。通过如下两个接口来定义钩子函数。

通过实行上述两个接口，可以实现在消息发送、消息消费前后记录消息轨迹，为了不明显增加消息发送与消息消费的时延，记录消息轨迹最好使用异步发送模式。

三、如何存储消息轨迹数据

消息轨迹需要存储什么消息以及在什么时候记录消息轨迹的问题都以及解决，那接下来就得思考将消息轨迹存储在哪里？存储在数据库中或其他媒介中，都会加重消息中间件，使其依赖外部组件，最佳的选择还是存储在 Broker 服务器中，将消息轨迹数据也当成一条消息存储到 Broker 服务器。

既然把消息轨迹当成消息存储在 Broker 服务器，那存储消息轨迹的 Topic 如何确定呢？RocketMQ 提供了两种方法来定义消息轨迹的 Topic。

- 系统默认 Topic

如果 Broker 的 traceTopicEnable 配置设置为 true，表示在该 Broker 上创建 topic 名为：RMQ_SYS_TRACE_TOPIC，队列个数为 1，默认该值为 false，表示该 Broker 不承载系统自定义用于存储消息轨迹的 topic。

- 自定义 Topic

在创建消息生产者或消息消费者时，可以通过参数自定义用于记录消息轨迹的 Topic 名称，不过要注意的是，rokcetmq 控制台(rocketmq-console)中只支持配置一个消息轨迹 Topic，故自定义 Topic，在目前这个阶段或许还不是一个最佳实践，建议使用系统默认的 Topic 即可。

通常为了避免消息轨迹的数据与正常的业务数据混合在一起，官方建议，在 Broker 集群中，新增加一台机器，只在这台机器上开启消息轨迹跟踪，这样该集群内的消息轨迹数据只会发送到这一台 Broker 服务器上，并不会增加集群内原先业务 Broker 的负载压力。

RocketMQ 消息轨迹的设计细节就介绍到这里了，下一篇将从源码的角度对其实现细节进行详细的剖析；如果觉得本文对您有帮助的话，期待您的点赞，谢谢。

1.17 消息发送常见问题与解决方案

本文将结合自己使用 RocketMQ 的经验，对消息发送常见的问题进行分享，基本会遵循出现问题，分析问题、解决问题。

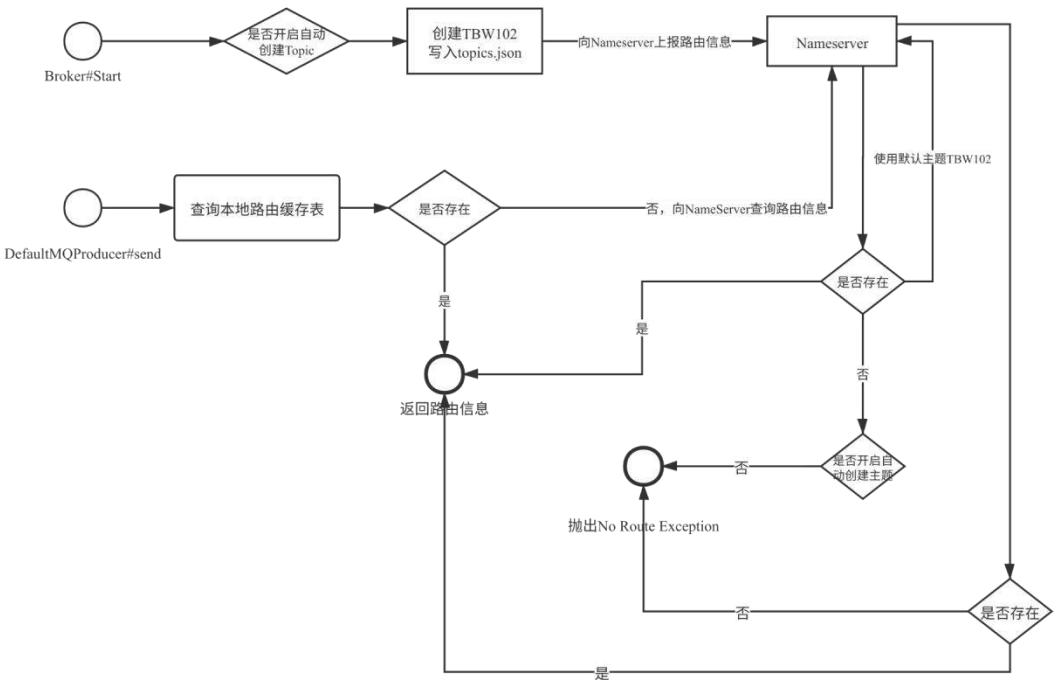
一、No route info of this topic

无法找到路由信息，其完整的错误堆栈信息如下：

```
Exception in thread "main" org.apache.rocketmq.client.exception.MQClientException: No route info of this topic: dw_test05
See http://rocketmq.apache.org/docs/faq/ for further details.
    at org.apache.rocketmq.client.impl.producer.DefaultMQProducerImpl.sendDefaultImpl(DefaultMQProducerImpl.java:684)
    at org.apache.rocketmq.client.impl.producer.DefaultMQProducerImpl.send(DefaultMQProducerImpl.java:1342)
    at org.apache.rocketmq.client.impl.producer.DefaultMQProducerImpl.send(DefaultMQProducerImpl.java:1288)
    at org.apache.rocketmq.client.producer.DefaultMQProducer.send(DefaultMQProducer.java:324)
    at com.example.demo.NoRouterTest.main(NoRouterTest.java:23)
```

而且很多读者朋友会说 Broker 端开启了自动创建主题也会出现上述问题。

RocketMQ 的路由寻找流程如下图所示：



上面的核心关键点如下：

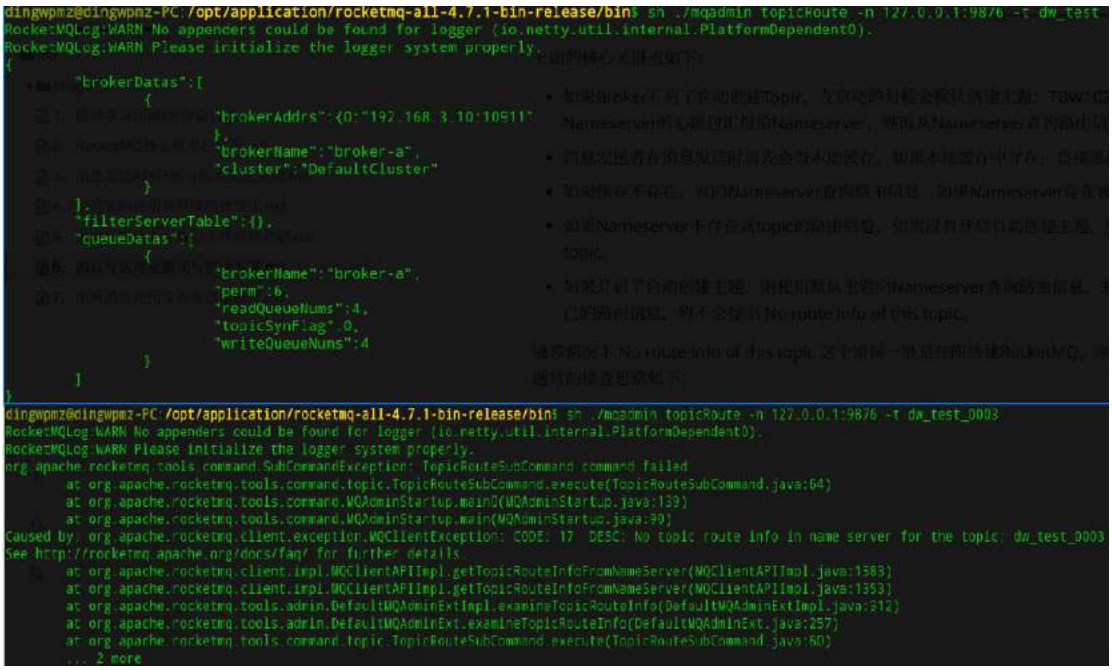
- 如果 Broker 开启了自动创建 Topic，在启动的时候会默认创建主题：TBW102，并会随着 Broker 发送到 Nameserver 的心跳包汇报给 Nameserver，继而从 Nameserver 查询路由信息时能返回路由信息。
- 消息发送者在消息发送时首先会查本地缓存，如果本地缓存中存在，直接返回路由信息。
- 如果缓存不存在，则向 Nameserver 查询路由信息，如果 Nameserver 存在该路由信息，就直接返回。
- 如果 Nameserver 不存在该 topic 的路由信息，如果没有开启自动创建主题，则抛出 No route info of this topic。
- 如果开启了自动创建主题，则使用默认主题向 Nameserver 查询路由信息，并使用默认 Topic 的路由信息为自己的路由信息，将不会抛出 No route info of this topic。

通常情况下 No route info of this topic 这个错误一般是在刚搭建 RocketMQ，刚入门 RocketMQ 遇到的比较多，通常的排查思路如下：

可以通过 rocketmq-console 查询路由信息是否存在，或使用如下命令查询路由信息：

```
cd ${ROCKETMQ_HOME}/bin
sh ./mqadmin topicRoute -n 127.0.0.1:9876 -t dw_test_0003
```

其输出结果如下所示：



- 如果通过命令无法查询到路由信息，则查看 Broker 是否开启了自动创建 topic，参数为：autoCreateTopicEnable,该参数默认为 true。但在生产环境不建议开启。
- 如果开启了自动创建路由信息，但还是抛出这个错误，这个时候请检查客户端(Producer)连接的 Nameserver 地址是否与 Broker 中配置的 nameserver 地址是否一致。

经过上面的步骤，基本就能解决该错误。

二、消息发送超时

消息发送超时，通常客户端的日志如下：

```
org.apache.rocketmq.remoting.exception.RemotingTimeoutException: wait response on the channel <                    > timeout, 3000(ms)
    at org.apache.rocketmq.remoting.netty.NettyRemotingAbstract.invokeSyncImpl(NettyRemotingAbstract.java:369)
    at org.apache.rocketmq.remoting.netty.NettyRemotingClient.invokeSync(NettyRemotingClient.java:340)
    at org.apache.rocketmq.client.impl.MQClientAPIImpl.sendMessageSync(MQClientAPIImpl.java:349)
    at org.apache.rocketmq.client.impl.MQClientAPIImpl.sendMessage(MQClientAPIImpl.java:333)
    at org.apache.rocketmq.client.impl.MQClientAPIImpl.sendMessage(MQClientAPIImpl.java:296)
    at org.apache.rocketmq.client.impl.producer.DefaultMQProducerImpl.sendKernelImpl(DefaultMQProducerImpl.java:693)
    at org.apache.rocketmq.client.impl.producer.DefaultMQProducerImpl.sendSelectImpl(DefaultMQProducerImpl.java:910)
    at org.apache.rocketmq.client.impl.producer.DefaultMQProducerImpl.send(DefaultMQProducerImpl.java:887)
    at org.apache.rocketmq.client.impl.producer.DefaultMQProducerImpl.send(DefaultMQProducerImpl.java:882)
    at org.apache.rocketmq.client.producer.DefaultMQProducer.send(DefaultMQProducer.java:373)
```

客户端报消息发送超时，通常第一怀疑的对象是 RocketMQ 服务器，是不是 Broker 性能出现了抖动，无法抗住当前的量。

那我们如何来排查 RocketMQ 当前是否有性能瓶颈呢?

首先我们执行如下命令查看 RocketMQ 消息写入的耗时分布情况：

```
cd /${USER.HOME}/logs/rocketmqlogs/
grep -n 'PAGECACHERT' store.log | more
```

输出结果如下所示：

```
[PAGECACHERT] TotalPut 226384, PutMessageDistributeTime [<=0ms]:224797 [0~10ms]:1587 [10~50ms]:0 [50~100ms]:0 [100~200ms]:0 [200~500ms]:0 [500ms~1s]:0 [1~2
[PAGECACHERT] TotalPut 222767, PutMessageDistributeTime [<=0ms]:221031 [0~10ms]:1736 [10~50ms]:0 [50~100ms]:0 [100~200ms]:0 [200~500ms]:0 [500ms~1s]:0 [1~2
[PAGECACHERT] TotalPut 229725, PutMessageDistributeTime [<=0ms]:228087 [0~10ms]:1635 [10~50ms]:3 [50~100ms]:0 [100~200ms]:0 [200~500ms]:0 [500ms~1s]:0 [1~2
[PAGECACHERT] TotalPut 228769, PutMessageDistributeTime [<=0ms]:227052 [0~10ms]:1717 [10~50ms]:0 [50~100ms]:0 [100~200ms]:0 [200~500ms]:0 [500ms~1s]:0 [1~2
```

RocketMQ 会每一分钟打印前一分钟内消息发送的耗时情况分布，我们从这里就能窥探 RocketMQ 消息写入是否存在明细的性能瓶颈，其区间如下：

- [<=0ms] 小于 0ms，即微妙级别的。
- [0~10ms] 小于 10ms 的个数。
- [10~50ms] 大于 10ms 小。
- 于 50ms 的个数。

其他区间显示，绝大多数会落在微妙级别完成，按照笔者的经验如果 100-200ms 及以上的区间超过 20 个后，说明 Broker 确实存在一定的瓶颈，如果只是少数几个，说明这个是内存或 pagecache 的抖动，问题不大。

通常情况下超时通常与 Broker 端的处理能力关系不大，还有另外一个佐证，在 RocketMQ broker 中还存在快速失败机制，即当 Broker 收到客户端的请求后会将消息先放入队列，然后顺序执行，如果一条消息队列中等待超过 200ms 就会启动快速失败，向客户端返回[TIMEOUT_CLEAN_QUEUE]broker busy，这个在本文的第 3 部分会详细介绍。

在 RocketMQ 客户端遇到网络超时，通常可以考虑一些应用本身的垃圾回收，是否由于 GC 的停顿时间导致的消息发送超时，这个我在测试环境进行压力测试时遇到过，但生产环境暂时没有遇到过，大家稍微留意一下。

在 RocketMQ 中通常遇到网络超时，通常与网络的抖动有关系，但由于我对网络不是特别擅长，故暂时无法找到直接证据，但能找到一些间接证据，例如在一个应用中同时连接了 kafka、RocketMQ 集群，发现在出现超时的同一时间发现连接到 RocketMQ 集群内所有 Broker，连接到 kafka 集群都出现了超时。

但出现网络超时，我们总得解决，那有什么解决方案吗?

我们对消息中间件的最低期望就是高并发低延迟，从上面的消息发送耗时分布情况也可以看出 RocketMQ 确实符合我们的期望，绝大部分请求都是在微妙级别内，故我给出的方案时，减少消息发送的超时时间，增加重试次数，并增加快速失败的最大等待时长。具体措施如下:

增加 Broker 端快速失败的时长，建议为 1000，在 broker 的配置文件中增加如下配置:

```
maxWaitTimeMillsInQueue=1000
```

主要原因是在当前的 RocketMQ 版本中，快速失败导致的错误为 SYSTEM_BUSY，并不会触发重试，适当增大该值，尽可能避免触发该机制，详情可以参考本文第 3 部分内容，会重点介绍 system_busy、broker_busy。

如果 RocketMQ 的客户端版本为 4.3.0 以下版本(不含 4.3.0)

将超时时间设置消息发送的超时时间为 500ms，并将重试次数设置为 6 次(这个可以适当进行调整，尽量大于 3)，其背后的哲学是尽快超时，并进行重试，因为发现局域网内的网络抖动是瞬时的，下次重试的是就能恢复，并且 RocketMQ 有故障规避机制，重试的时候会尽量选择不同的 Broker，相关的代码如下：

```
DefaultMQProducer producer = new DefaultMQProducer("dw_test_producer_group");
producer.setNamesrvAddr("127.0.0.1:9876");
producer.setRetryTimesWhenSendFailed(5);//  同步发送模式：重试次数
producer.setRetryTimesWhenSendAsyncFailed(5);// 异步发送模式：重试次数
producer.start();
producer.send(msg,500);//消息发送超时时间
```

如果 RocketMQ 的客户端版本为 4.3.0 及以上版本

如果客户端版本为 4.3.0 及其以上版本，由于其设置的消息发送超时时间为所有重试的总的超时时间，故不能直接通过设置 RocketMQ 的发送 API 的超时时间，而是需要对其 API 进行包装，重试需要在外层收到进行，例如示例代码如下：

```
public static SendResult send(DefaultMQProducer producer, Message msg, int
                              retryCount) {
    Throwable e = null;
    for(int i =0; i < retryCount; i ++ ) {
        try {
            return producer.send(msg,500); //设置超时时间，为 500ms，内部有重试机制
        } catch (Throwable e2) {
            e = e2;
        }
    }
```

```
        throw new RuntimeException("消息发送异常",e);
    }
```

三、System busy、Broker busy

在使用 RocketMQ 中，如果 RocketMQ 集群达到 1W/tps 的压力负载水平，System busy、Broker busy 就会是大家经常会遇到的问题。例如如下图所示的异常栈。

```
nt.exception.MQBrokerException: CODE: 2  DESC: [TIMEOUT_CLEAN_QUEUE]broker busy, start flow control for a while, period in queue: 200ms, size of queue: 92
ease visit the url, http://rocketmq.apache.org/docs/faq/
client.impl.MQClientAPIImpl.processSendResponse(MQClientAPIImpl.java:551)
client.impl.MQClientAPIImpl.sendMessageSync(MQClientAPIImpl.java:351)
client.impl.MQClientAPIImpl.sendMessage(MQClientAPIImpl.java:333)
client.impl.MQClientAPIImpl.sendMessage(MQClientAPIImpl.java:296)
client.impl.producer.DefaultMQProducerImpl.sendKernelImpl(DefaultMQProducerImpl.java:693)
client.impl.producer.DefaultMQProducerImpl.sendSelectImpl(DefaultMQProducerImpl.java:910)
client.impl.producer.DefaultMQProducerImpl.send(DefaultMQProducerImpl.java:887)
client.impl.producer.DefaultMQProducerImpl.send(DefaultMQProducerImpl.java:882)
client.producer.DefaultMQProducer.send(DefaultMQProducer.java:373)
```

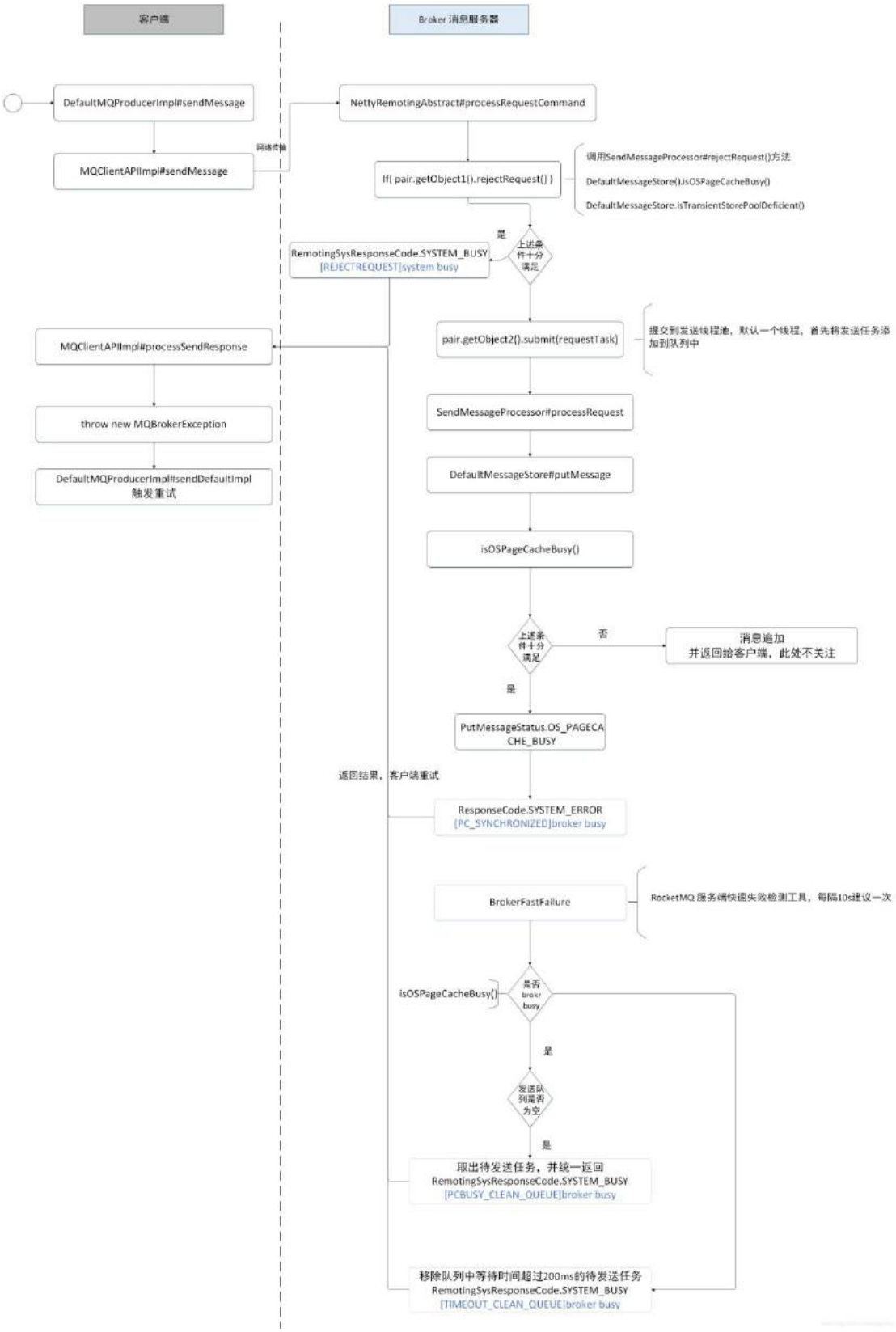
```
org.apache.rocketmq.client.exception.MQBrokerException: CODE: 2  DESC: [REJECTREQUEST]system busy, start flow control for a while
For more information, please visit the url, http://rocketmq.apache.org/docs/faq/
 at org.apache.rocketmq.client.impl.MQClientAPIImpl.processSendResponse(MQClientAPIImpl.java:551)
 at org.apache.rocketmq.client.impl.MQClientAPIImpl.sendMessageSync(MQClientAPIImpl.java:351)
 at org.apache.rocketmq.client.impl.MQClientAPIImpl.sendMessage(MQClientAPIImpl.java:333)
 at org.apache.rocketmq.client.impl.MQClientAPIImpl.sendMessage(MQClientAPIImpl.java:296)
 at org.apache.rocketmq.client.impl.producer.DefaultMQProducerImpl.sendKernelImpl(DefaultMQProducerImpl.java:693)
 at org.apache.rocketmq.client.impl.producer.DefaultMQProducerImpl.sendSelectImpl(DefaultMQProducerImpl.java:910)
 at org.apache.rocketmq.client.impl.producer.DefaultMQProducerImpl.send(DefaultMQProducerImpl.java:887)
 at org.apache.rocketmq.client.impl.producer.DefaultMQProducerImpl.send(DefaultMQProducerImpl.java:882)
 at org.apache.rocketmq.client.producer.DefaultMQProducer.send(DefaultMQProducer.java:373)
```

纵观 RocketMQ 与 system busy、broker busy 相关的错误关键字，总共包含如下 5 个：

- [REJECTREQUEST]system busy
- too many requests and system thread pool busy
- [PC_SYNCHRONIZED]broker busy
- [PCBUSY_CLEAN_QUEUE]broker busy
- [TIMEOUT_CLEAN_QUEUE]broker busy

1. 原理分析

我们先用一张图来阐述一下在消息发送的全生命周期中分别在什么时候会抛出上述错误。



根据上述 5 类错误日志，其触发的原有可以归纳为如下 3 种。

pagecache 压力较大

其中如下三类错误属于此种情况：

- [REJECTREQUEST]system busy
- [PC_SYNCHRONIZED]broker busy
- [PCBUSY_CLEAN_QUEUE]broker busy

判断 pagecache 是否忙的依据就是在写入消息时，在向内存追加消息时加锁的时间，默认的判断标准是加锁时间超过 1s，就认为是 pagecache 压力大，向客户端抛出相关的错误日志。

发送线程池挤压的拒绝策略

在 RocketMQ 中处理消息发送的是一个只有一个线程的线程池，内部会维护一个有界队列，默认长度为 1W，如果当前队列中挤压的数量超过 1w，执行线程池的拒绝策略，从而抛出[too many requests and system thread pool busy]错误。

Broker 端快速失败

默认情况下 Broker 端开启了快速失败机制，就是在 Broker 端还未发生 pagecache 繁忙(加锁超过 1s)的情况，但存在一些请求在消息发送队列中等待 200ms 的情况，RocketMQ 会不再继续排队，直接向客户端返回 system busy，但由于 rocketmq 客户端目前对该错误没有进行重试处理，所以在解决这类问题的时候需要额外处理。

2. PageCache 繁忙解决方案

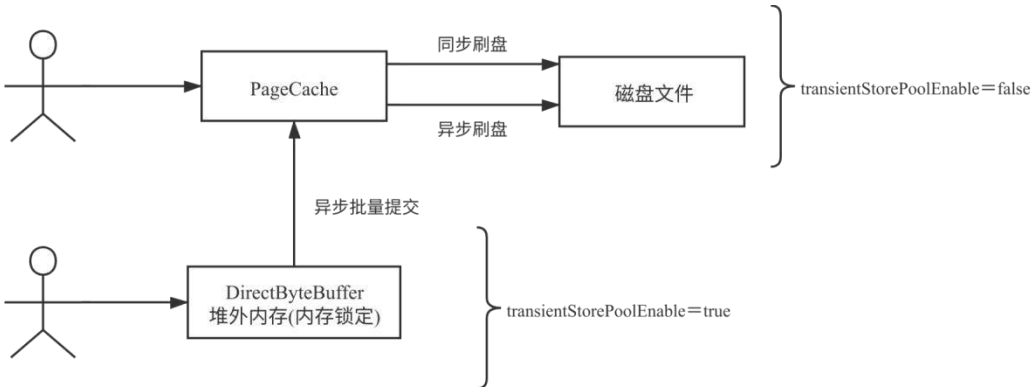
一旦消息服务器出现大量 pagecache 繁忙(在向内存追加数据加锁超过 1s)的情况，这个是比较严重的问题，需要人为进行干预解决，解决的问题思路如下：

transientStorePoolEnable

开启 transientStorePoolEnable 机制，即在 broker 中配置文件中增加如下配置：

transientStorePoolEnable=true

transientStorePoolEnable 的原理如下图所示：



引入 transientStorePoolEnable 能缓解 pagecache 的压力背后关键如下：

- 消息先写入到堆外内存中，该内存由于启用了内存锁定机制，故消息的写入是接近直接操作内存，性能能得到保证。
- 消息进入到堆外内存后，后台会启动一个线程，一批一批将消息提交到 pagecache，即写消息时对 pagecache 的写操作由单条写入变成了批量写入，降低了对 pagecache 的压力。

引入 transientStorePoolEnable 会增加数据丢失的可能性，如果 Broker JVM 进程异常退出，提交到 PageCache 中的消息是不会丢失的，但存在堆外内存(DirectByteBuffer)中但还未提交到 PageCache 中的这部分消息，将会丢失。但通常情况下，RocketMQ 进程退出的可能性不大，通常情况下，如果启用了 transientStorePoolEnable，消息发送端需要有重新推送机制(补偿思想)。

扩容

如果在开启了 transientStorePoolEnable 后，还会出现 pagecache 级别的繁忙，那需要集群进行扩容，或者对集群中的 topic 进行拆分，即将一部分 topic 迁移到其他集群中，降低集群的负载。关于 RocketMQ 优雅停机、扩容方案等，将在本专栏的[运维实战部分]做专题介绍。

温馨提示：在 RocketMQ 出现 pagecache 繁忙造成的 broker busy，RocketMQ Client 会有重试机制。

3. TIMEOUT_CLEAN_QUEUE 解决方案

由于如果出现 TIMEOUT_CLEAN_QUEUE 的错误，客户端暂时不会对其进行重试，故现阶段的建议是适当增加快速失败的判断标准，即在 broker 的配置文件中增加如下配置：

```
# 该值默认为 200，表示 200ms
waitTimeMillsInSendQueue=1000
```

温馨提示，关于 Broker busy，笔者发表过两篇文章，大家也可以结合着看：

- RocketMQ 消息发送 system busy、broker busy 原因分析与解决方案
- 再谈 RocketMQ broker busy

四、小结

本文主要对实际中常遇到的关于消息发送方面经常遇到的问题进行剖析，从而提出解决方案。

扫一扫加入作者公众号
中间件兴趣圈

扫一扫关注
RocketMQ 官微

扫一扫关注【阿里巴巴云原生】公众号
获取第一手技术干货

阿里云开发者“藏经阁”
海量免费电子书下载