> 你知道的越多，你不知道的越多

> 点赞再看，养成习惯

> 本文**GitHub** https://github.com/JavaFamily 已收录，有一线大厂面试点脑图、个人联系方式和技术交流群，欢迎Star和指教

前言

消息队列在互联网技术存储方面使用如此广泛，几乎所有的后端技术面试官都要在**消息队列**的使用和原理方面对小伙伴们进行360°的刁难。

作为一个在互联网公司面一次拿一次Offer的面霸，打败了无数竞争对手，每次都只能看到无数落寞的身影失望的离开，略感愧疚（**请允许我使用一下夸张的修辞手法**）。

于是在一个寂寞难耐的夜晚，我痛定思痛，决定开始写**《吊打面试官》**系列，希望能帮助各位读者以后面试势如破竹，对面试官进行360°的反击，吊打问你的面试官，让一同面试的同僚瞠目结舌，疯狂收割大厂Offer！

捞一下

消息队列系列前面两章分别讲了**消息队列**的基础知识，还有比较常见的问题和常见分布式事务解决方案，那么在实际开发过程中,我们使用频率比较高的消息队列中间件有哪些呢?

帅丙我工作以来接触的消息队列中间件有**RocketMQ**、**Kafka**、**自研**，是的因为我主要接触的都是电商公司，相对而言业务体量还有场景来说都是他们比较适合，再加上杭州阿里系公司偏多，身边同事或者公司老大基本都是阿里出来创业的，那在使用技术栈的时候**阿里系的开源框架**也就成了首选。

就算是自研的中间件多多少少也是借鉴RocketMQ、Kafka的优点自研的，那我后面两章就分别简单的介绍下两者，他们分别在业务场景和大数据领域各自发光发热。

那到底是道德的沦丧，还是人性的泯灭，让我们跟着敖丙走进RocketMQ的内心世界。

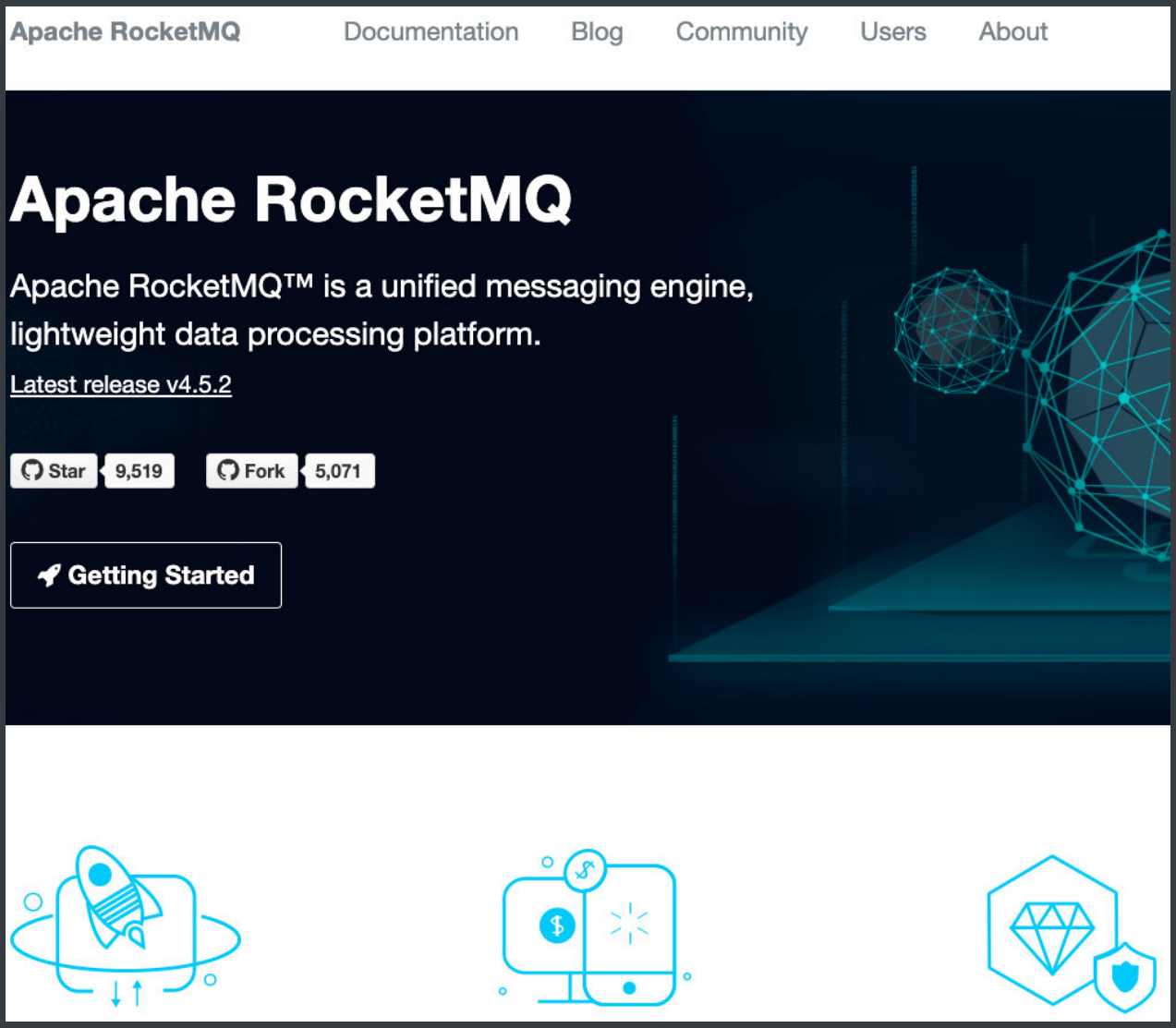
正文

RocketMQ简介

> RocketMQ是一个纯Java、分布式、队列模型的开源消息中间件，前身是MetaQ，是阿里参考Kafka特点研发的一个队列模型的消息中间件，后开源给apache基金会成为了apache的顶级开源项目，具有高性能、高可靠、高实时、分布式特点。

我们再看下阿里给他取的名字哈：**Rocket** 火箭 阿里这是希望他上天呀，不过我觉得这个名字确实挺酷的。

我们先看看他最新的官网



回顾一下他的心路历程

2007年：淘宝实施了“五彩石”项目，“五彩石”用于将交易系统从单机变成分布式，也是在这个过程中产生了阿里巴巴第一代消息引擎——Notify。

2010年：阿里巴巴B2B部门基于ActiveMQ的5.1版本也开发了自己的一款消息引擎，称为Napoli，这款消息引擎在B2B里面广泛地被使用，不仅仅是在交易领域，在很多的后台异步解耦等方面也得到了广泛的应用。

2011年：业界出现了现在被很多大数据领域所推崇的Kafka消息引擎，阿里巴巴在研究了Kafka的整体机制和架构设计之后，基于Kafka的设计使用Java进行了完全重写并推出了**MetaQ 1.0**版本，主要是用于解决顺序消息和海量堆积的问题。

2012年：阿里巴巴开源其自研的第三代分布式消息中间件——**RocketMQ**。

经过几年的技术打磨，阿里称基于RocketMQ技术，目前双十一当天消息容量可达到万亿级。

2016年11月：阿里将RocketMQ捐献给**Apache**软件基金会，正式成为孵化项目。

阿里称会将其打造成顶级项目。**这是阿里迈出的一大步**，因为加入到开源软件基金会需要经过评审方的考核与观察。

坦率而言，业界还对国人的代码开源参与度仍保持着刻板印象；而Apache基金会中的342个项目中，暂时还只有Kylin、CarbonData、Eagle 、Dubbo和 RocketMQ 共计五个中国技术人主导的项目。

2017年2月20日：RocketMQ正式发布4.0版本，专家称新版本适用于电商领域，金融领域，大数据领域，兼有物联网领域的编程模型。

以上就是RocketMQ的整体发展历史，其实在阿里巴巴内部围绕着RocketMQ内核打造了三款产品，分别是**MetaQ**、**Notify**和**Aliware MQ**。

这三者分别采用了不同的模型，**MetaQ**主要使用了拉模型，解决了顺序消息和海量堆积问题；**Notify**主要使用了推模型，解决了事务消息；而云产品**Aliware MQ**则是提供了商业化的版本。

经历多次双11洗礼的英雄

在备战2016年双十一时，**RocketMq**团队重点做了**两件事情**，优化慢请求与统一存储引擎。

- **优化慢请求**：这里主要是解决在海量高并发场景下降低慢请求对整个集群带来的抖动，**毛刺问题**。这是一个极具挑战的技术活，团队同学经过长达1个多月的跟进调优，从双十一的复盘情况来看，99.996%的延迟落在了10ms以内，**而99.6%的延迟在1ms以内**。优化主要集中在**RocketMQ**存储层算法优化、JVM与操作系统调优。更多的细节大家可以参考《万亿级数据洪峰下的分布式消息引擎》。
- **统一存储引擎**：主要解决的消息引擎的高可用，成本问题。在多代消息引擎共存的前提下，我们对Notify的存储模块进行了全面移植与替换。

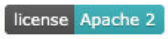
RocketMQ天生为金融互联网领域而生，追求高可靠、高可用、高并发、低延迟，是一个阿里巴巴由内而外成功孕育的典范，除了阿里集团上千个应用外，根据我们不完全统计，国内至少有上百家单位、科研教育机构在使用。

RocketMQ在阿里集团也被广泛应用在订单，交易，充值，流计算，消息推送，日志流式处理，**binglog**分发等场景。

他所拥有的功能

我们直接去**GitHub**上看**Apache**对他的描述可能会好点

Apache RocketMQ



Apache RocketMQ is a distributed messaging and streaming platform with low latency, high performance and reliability, trillion-level capacity and flexible scalability.

It offers a variety of features:

- Pub/Sub messaging model
- Financial grade transactional message
- A variety of cross language clients, such as Java, C/C++, Python, Go
- Pluggable transport protocols, such as TCP, SSL, AIO
- Inbuilt message tracing capability, also support opentracing
- Versatile big-data and streaming ecosytem integration
- Message retroactivity by time or offset
- Reliable FIFO and strict ordered messaging in the same queue
- Efficient pull&push consumption model
- Million-level message accumulation capacity in a single queue
- Multiple messaging protocols like JMS and OpenMessaging
- Flexible distributed scale-out deployment architecture
- Lightning-fast batch message exchange system
- Various message filter mechanics such as SQL and Tag
- Docker images for isolated testing and cloud isolated clusters
- Feature-rich administrative dashboard for configuration, metrics and monitoring
- Authentication and authorisation

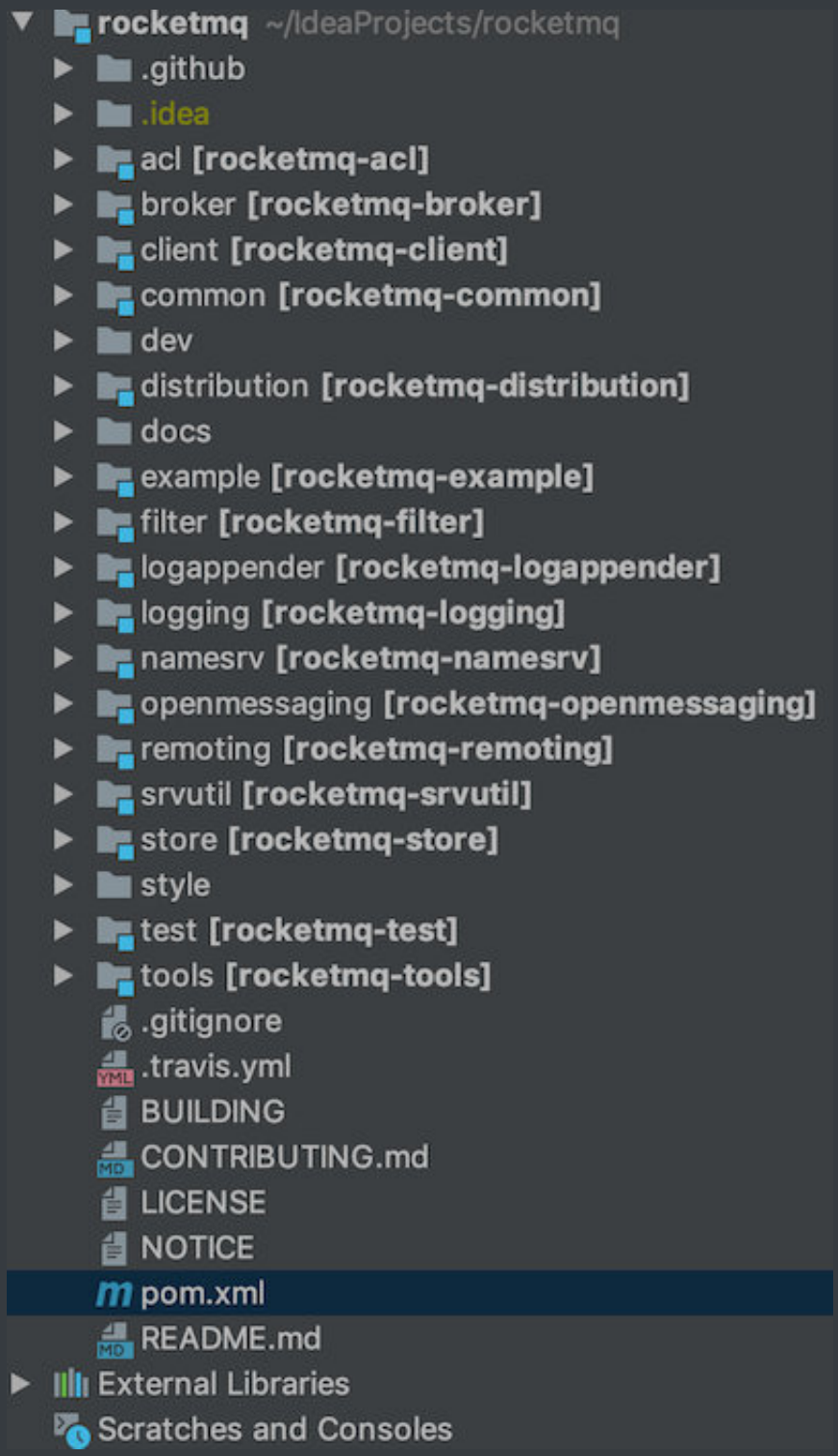
是的功能完整到爆炸基本上开发完全够用，什么？看不懂专业词汇的英文？

帅丙是暖男来的嘛，中文功能如下 ↓

- 发布/订阅消息传递模型
- 财务级交易消息
- 各种跨语言客户端，例如Java，C / C ++，Python，Go
- 可插拔的传输协议，例如TCP，SSL，AIO
- 内置的消息跟踪功能，还支持开放式跟踪
- 多功能的大数据和流生态系统集成
- 按时间或偏移量追溯消息
- 可靠的FIFO和严格的有序消息传递在同一队列中
- 高效的推拉消费模型
- 单个队列中的百万级消息累积容量
- 多种消息传递协议，例如JMS和OpenMessaging
- 灵活的分布式横向扩展部署架构
- 快如闪电的批量消息交换系统
- 各种消息过滤器机制，例如SQL和Tag
- 用于隔离测试和云隔离群集的Docker映像
- 功能丰富的管理仪表板，用于配置，指标和监视
- 认证与授权

他的项目结构组成是怎么样子的?

> GitHub地址：https://github.com/apache/rocketmq



他的核心模块：

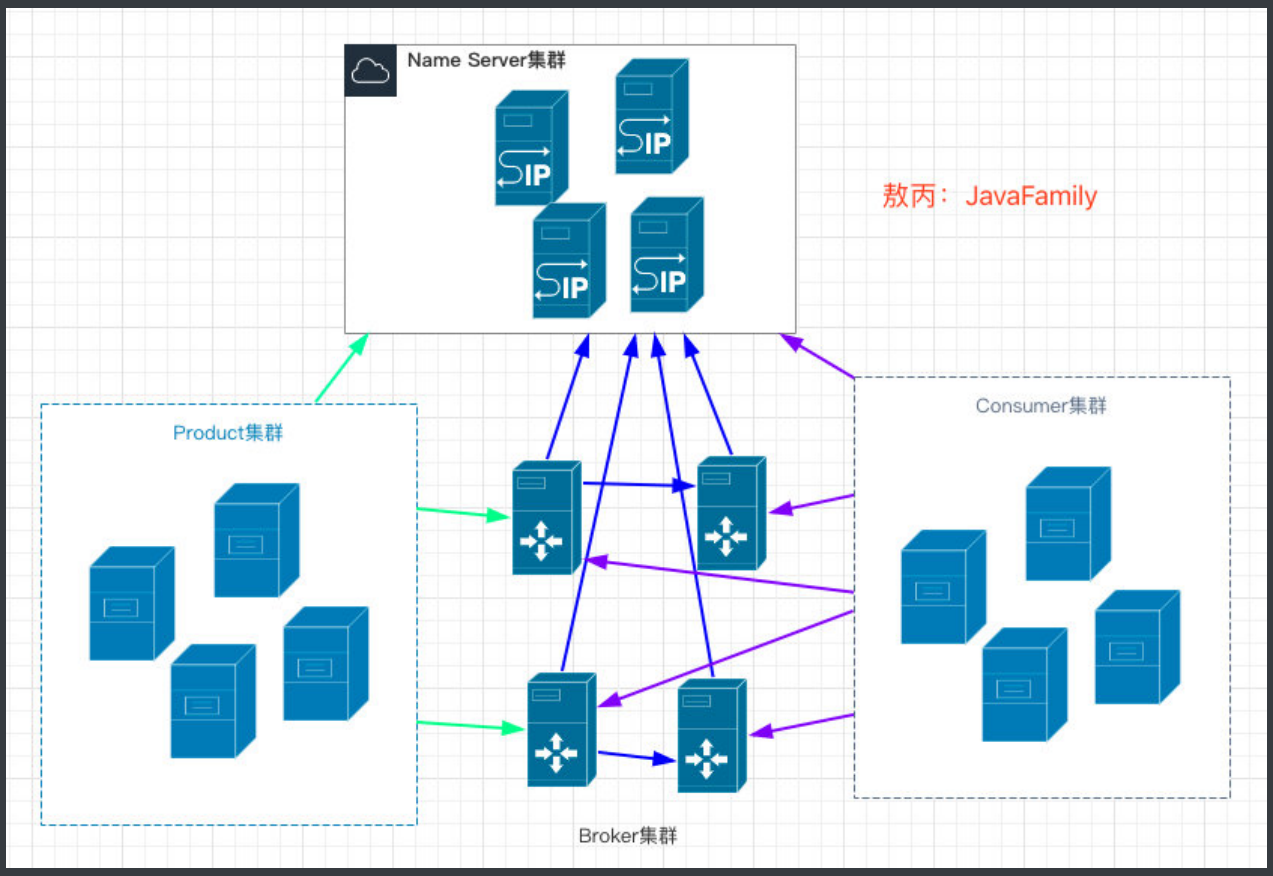
- rocketmq-broker：接受生产者发来的消息并存储（通过调用rocketmq-store），消费者从这里取得消息
- rocketmq-client：提供发送、接受消息的客户端API。
- rocketmq-namesrv：NameServer，类似于Zookeeper，这里保存着消息的TopicName，队列等运

行时的元信息。

- rocketmq-common：通用的一些类，方法，数据结构等。
- rocketmq-remoting：基于Netty4的client/server + fastjson序列化 + 自定义二进制协议。
- rocketmq-store：消息、索引存储等。
- rocketmq-filtersrv：消息过滤器Server，需要注意的是，要实现这种过滤，需要上传代码到MQ！（一般而言，我们利用Tag足以满足大部分的过滤需求，如果更灵活更复杂的过滤需求，可以考虑filtersrv组件）。
- rocketmq-tools：命令行工具。

他的架构组成，或者理解为为什么他这么快？这么强？这么厉害？

他主要有四大核心组成部分：**NameServer**、**Broker**、**Producer**以及**Consumer**四部分。



Tip：我们可以看到**RocketMQ**啥都是**集群**部署的，这是他**吞吐量大**，**高可用**的原因之一，集群的模式也很花哨，可以支持多master 模式、多master多slave异步复制模式、多 master多slave同步双写模式。

而且这个模式好像Kafka啊！（我这里是废话，本身就是阿里基于Kafka的很多特性研发的）。

分别介绍下各个集群组成部分吧

NameServer：

> 主要负责对于源数据的管理，包括了对于**Topic**和路由信息的管理。

NameServer是一个功能齐全的服务器，其角色类似Dubbo中的Zookeeper，但NameServer与Zookeeper相比更轻量。主要是因为每个NameServer节点互相之间是独立的，没有任何信息交互。

NameServer压力不会太大，平时主要开销是在维持心跳和提供Topic-Broker的关系数据。

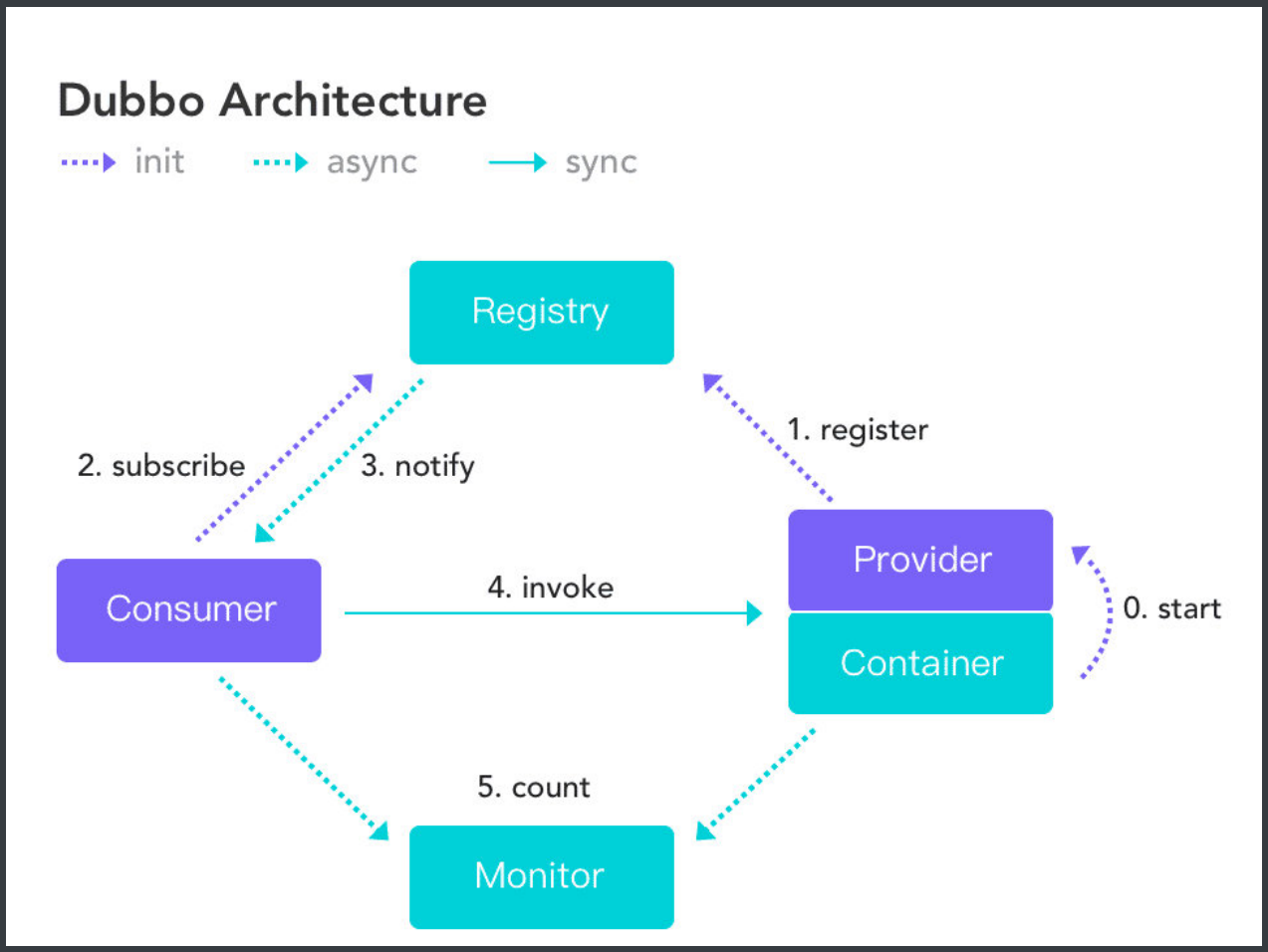
但有一点需要注意，Broker向NameServer发心跳时， 会带上当前自己所负责的所有**Topic**信息，如果**Topic**个数太多（万级别），会导致一次心跳中，就Topic的数据就几十M，网络情况差的话， 网络传输失败，心跳失败，导致NameServer误认为Broker心跳失败。

NameServer 被设计成几乎无状态的，可以横向扩展，节点之间相互之间无通信，通过部署多台机器来标记自己是一个伪集群。

每个 Broker 在启动的时候会到 NameServer 注册，Producer 在发送消息前会根据 Topic 到 **NameServer** 获取到 Broker 的路由信息，Consumer 也会定时获取 Topic 的路由信息。

所以从功能上看NameServer应该是和 ZooKeeper 差不多，据说 RocketMQ 的早期版本确实是使用的 ZooKeeper ，后来改为了自己实现的 NameServer 。

我们看一下**Dubbo**中注册中心的角色，是不是真的一毛一样，师出同门相似点真的很多：



Producer

> 消息生产者，负责产生消息，一般由业务系统负责产生消息。

- **Producer**由用户进行分布式部署，消息由**Producer**通过多种负载均衡模式发送到**Broker**集群，发送低延时，支持快速失败。
- **RocketMQ** 提供了三种方式发送消息：同步、异步和单向
  - **同步发送**：同步发送指消息发送方发出数据后会在收到接收方发回响应之后才发下一个数据包。一般用于重要通知消息，例如重要通知邮件、营销短信。
  - **异步发送**：异步发送指发送方发出数据后，不等接收方发回响应，接着发送下个数据包，一般用于可能链路耗时较长而对响应时间敏感的业务场景，例如用户视频上传后通知启动转码服务。
  - **单向发送**：单向发送是指只负责发送消息而不等待服务器回应且没有回调函数触发，适用于某些耗时非常短但对可靠性要求并不高的场景，例如日志收集。

Broker

> 消息中转角色，负责**存储消息**，转发消息。

- **Broker**是具体提供业务的服务器，单个Broker节点与所有的NameServer节点保持长连接及心跳，并会定时将**Topic**信息注册到NameServer，顺带一提底层的通信和连接都是**基于Netty实现**的。
- **Broker**负责消息存储，以Topic为纬度支持轻量级的队列，单机可以支撑上万队列规模，支持消息推拉模型。
- 官网上有数据显示：具有**上亿级消息堆积能力**，同时可**严格保证消息的有序性**。

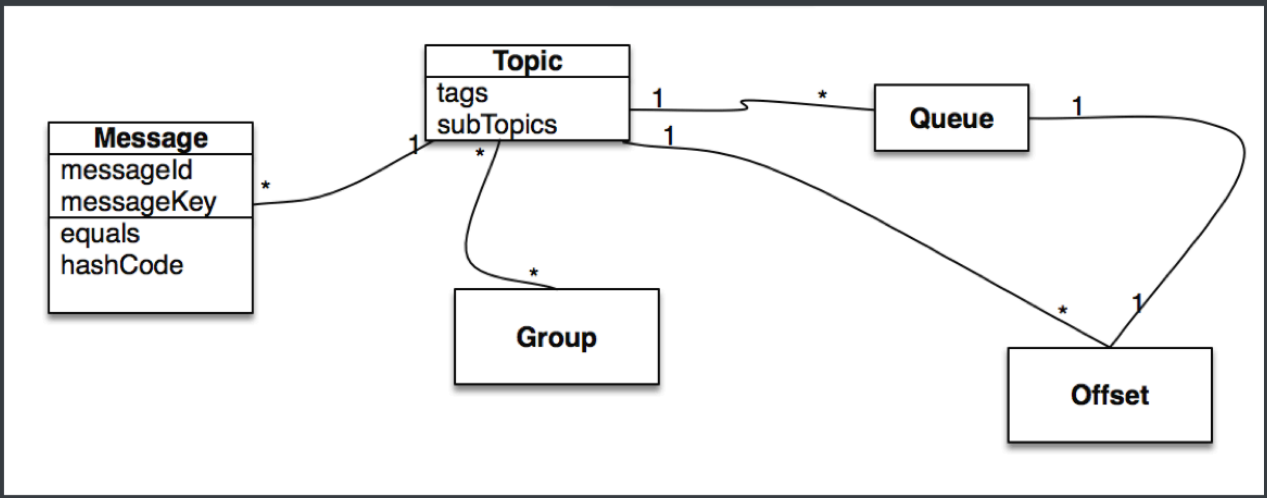
Consumer

> 消息消费者，负责消费消息，一般是后台系统负责异步消费。

- **Consumer**也由用户部署，支持PUSH和PULL两种消费模式，支持**集群消费**和**广播消息**，提供**实时的消息订阅机制**。
  - **Pull**：拉取型消费者（Pull Consumer）主动从消息服务器拉取信息，只要批量拉取到消息，用户应用就会启动消费过程，所以 Pull 称为主动消费型。
- **Push**：推送型消费者（Push Consumer）封装了消息的拉取、消费进度和其他的内部维护工作，将消息到达时执行的回调接口留给用户应用程序来实现。所以 Push 称为被动消费类型，但从实现上看还是从消息服务器中拉取消息，不同于 Pull 的是 Push 首先要注册消费监听器，当监听器处触发后才开始消费消息。

Tip： **GItHub** https://github.com/JavaFamily 有一线大厂面经和面试考察点脑图，也有个人联系方式。

消息领域模型



Message

Message（消息）就是要传输的信息。

一条消息必须有一个主题（Topic），主题可以看做是你的信件要邮寄的地址。

一条消息也可以拥有一个可选的标签（Tag）和额处的键值对，它们可以用于设置一个业务 Key 并在 Broker 上查找此消息以便在开发期间查找问题。

Topic

Topic（主题）可以看做消息的规类，它是消息的第一级类型。比如一个电商系统可以分为：交易消息、物流消息等，一条消息必须有一个 Topic 。

Topic 与生产者和消费者的关系非常松散，一个 Topic 可以有0个、1个、多个生产者向其发送消息，一个生产者也可以同时向不同的 Topic 发送消息。

一个 Topic 也可以被 0个、1个、多个消费者订阅。

Tag

Tag（标签）可以看作子主题，它是消息的第二级类型，用于为用户提供额外的灵活性。使用标签，同一业务模块不同目的的消息就可以用相同 Topic 而不同的 **Tag** 来标识。比如交易消息又可以分为：交易创建消息、交易完成消息等，一条消息可以没有 **Tag** 。

标签有助于保持您的代码干净和连贯，并且还可以为 **RocketMQ** 提供的查询系统提供帮助。

Group

分组，一个组可以订阅多个Topic。

分为ProducerGroup，ConsumerGroup，代表某一类的生产者和消费者，一般来说同一个服务可以作为Group，同一个Group一般来说发送和消费的消息都是一样的

Queue

在**Kafka**中叫Partition，每个Queue内部是有序的，在**RocketMQ**中分为读和写两种队列，一般来说读写队列数量一致，如果不一致就会出现很多问题。

Message Queue

Message Queue（消息队列），主题被划分为一个或多个子主题，即消息队列。

一个 Topic 下可以设置多个消息队列，发送消息时执行该消息的 Topic ，RocketMQ 会轮询该 Topic 下的所有队列将消息发出去。

消息的物理管理单位。一个Topic下可以有多个Queue，Queue的引入使得消息的存储可以分布式集群化，具有了水平扩展能力。

Offset

在**RocketMQ** 中，所有消息队列都是持久化，长度无限的数据结构，所谓长度无限是指队列中的每个存储单元都是定长，访问其中的存储单元使用Offset 来访问，Offset 为 java long 类型，64 位，理论上在100年内不会溢出，所以认为是长度无限。

也可以认为 Message Queue 是一个长度无限的数组，**Offset** 就是下标。

消息消费模式

消息消费模式有两种：**Clustering**（集群消费）和**Broadcasting**（广播消费）。

默认情况下就是集群消费，该模式下一个消费者集群共同消费一个主题的多个队列，一个队列只会被一个消费者消费，如果某个消费者挂掉，分组内其它消费者会接替挂掉的消费者继续消费。

而广播消费消息会发给消费者组中的每一个消费者进行消费。

Message Order

Message Order（消息顺序）有两种：**Orderly**（顺序消费）和**Concurrently**（并行消费）。

顺序消费表示消息消费的顺序同生产者为每个消息队列发送的顺序一致，所以如果正在处理全局顺序是强制性的场景，需要确保使用的主题只有一个消息队列。

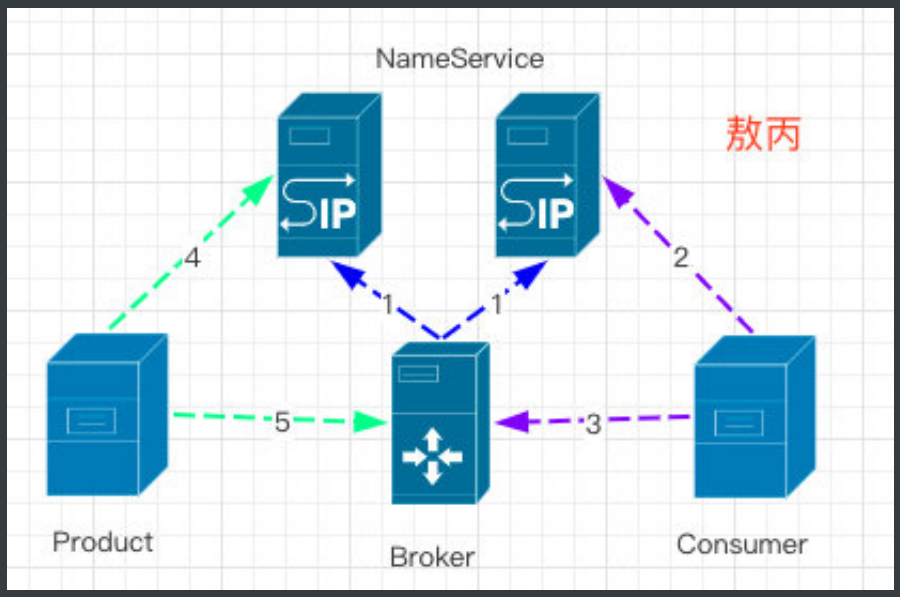
并行消费不再保证消息顺序，消费的最大并行数量受每个消费者客户端指定的线程池限制。

一次完整的通信流程是怎样的?

Producer 与 NameServer集群中的其中一个节点（随机选择）建立长连接，定期从 NameServer 获取 **Topic** 路由信息，并向提供 Topic 服务的 **Broker Master** 建立长连接，且定时向 **Broker** 发送心跳。

Producer 只能将消息发送到 Broker master，但是 **Consumer** 则不一样，它同时和提供 Topic 服务的 Master 和 Slave建立长连接，既可以从 Broker Master 订阅消息，也可以从 Broker Slave 订阅消息。

具体如下图：



我上面说过他跟**Dubbo**像不是我瞎说的，就连他的注册过程都很像**Dubbo**的服务暴露过程。

是不是觉得很简单，但是你同时也产生了好奇心，每一步是怎么初始化启动的呢?

帅丙呀就知道大家都是**求知欲极强的人才**，这不我都准备好了，我们一步步分析一下。

主要是人才群里的仔要求我写出来。。。(文末有进群方式)

NameService启动流程

在org.apache.rocketmq.namesrv目录下的**NamesrvStartup**这个启动类基本上描述了他的启动过程我们可以看一下代码：

- 第一步是初始化配置
- 创建**NamesrvController**实例，并开启两个定时任务：
  - 每隔10s扫描一次**Broker**，移除处于不激活的**Broker**；
  - 每隔10s打印一次KV配置。

```
this.scheduledExecutorService.scheduleAtFixedRate(new Runnable() {

    @Override
    public void run() {
        NamesrvController.this.routeInfoManager.scanNotActiveBroker();
    }
}, 5, 10, TimeUnit.SECONDS);

this.scheduledExecutorService.scheduleAtFixedRate(new Runnable() {

    @Override
    public void run() {
        NamesrvController.this.kvConfigManager.printAllPeriodically();
    }
}, 1, 10, TimeUnit.MINUTES);
```

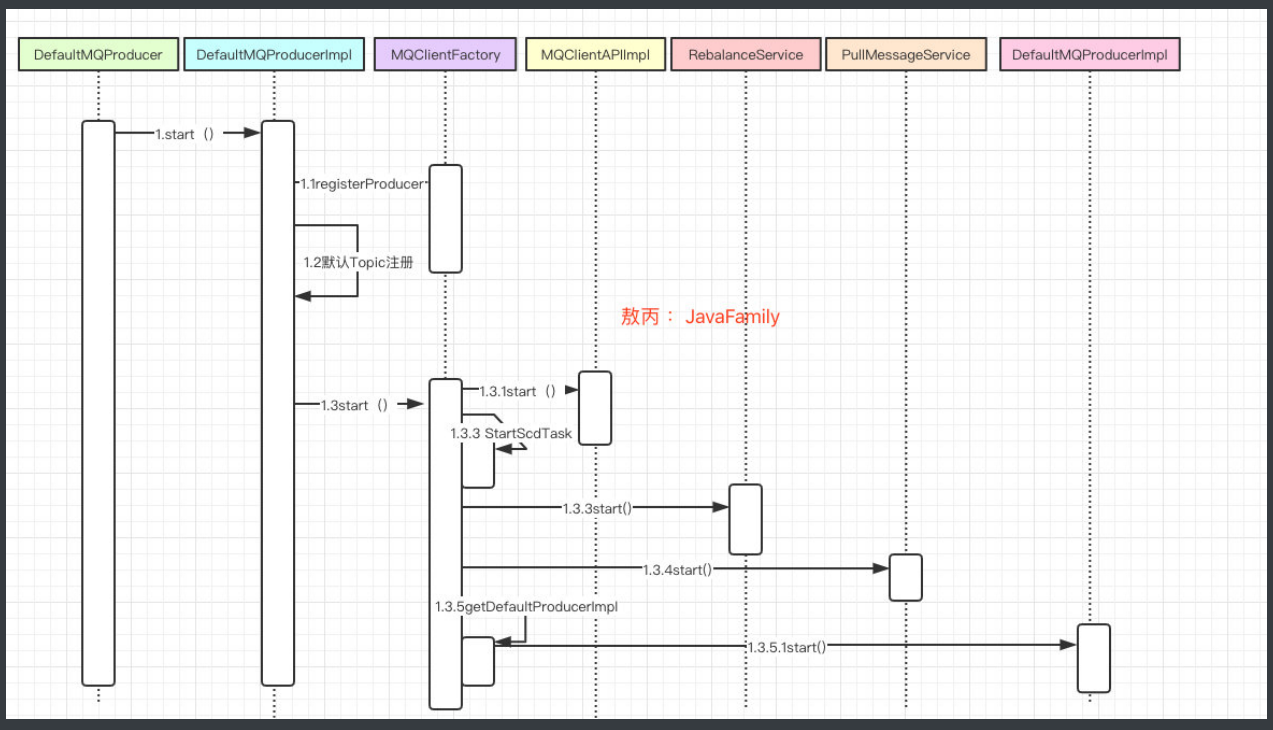
- 第三步注册钩子函数，启动服务器并监听Broker。

NameService还有很多东西的哈我这里就介绍他的启动流程，大家还可以去看看代码，还是很有意思的，比如路由注册会发送心跳包，还有**心跳包的处理流程**，**路由删除**，**路由发现**等等。

Tip：本来我想贴很多源码的，后面跟**歪歪（Java3y）**讨论了很久做出了不贴的决定，大家理解过程为主！我主要是做只是扫盲还有一些痛点分析嘛，深究还是得大家花时间，我要啥都介绍篇幅就不够了。

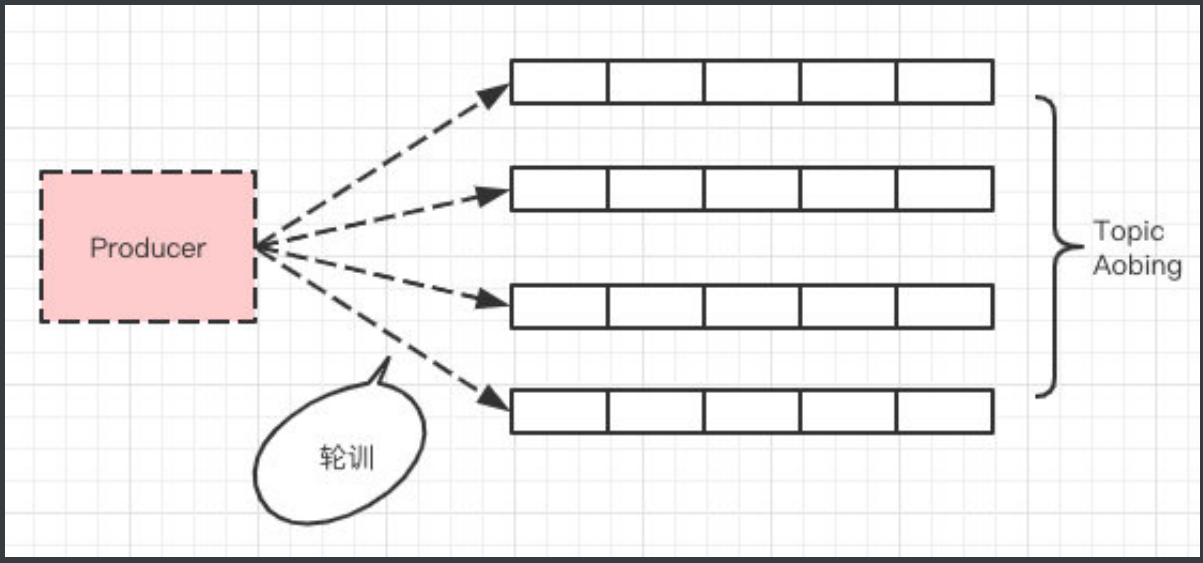
Producer

链路很长涉及的细节也多，我就发一下链路图。



Producer是消息发送方，那他怎么发送的呢?

通过轮训，**Producer**轮训某个**Topic**下面的所有队列实现发送方的负载均衡



Broker

Broker在RocketMQ中是进行处理Producer发送消息请求，Consumer消费消息的请求，并且进行消息的持久化，以及HA策略和服务端过滤，就是集群中很重的工作都是交给了**Broker**进行处理。

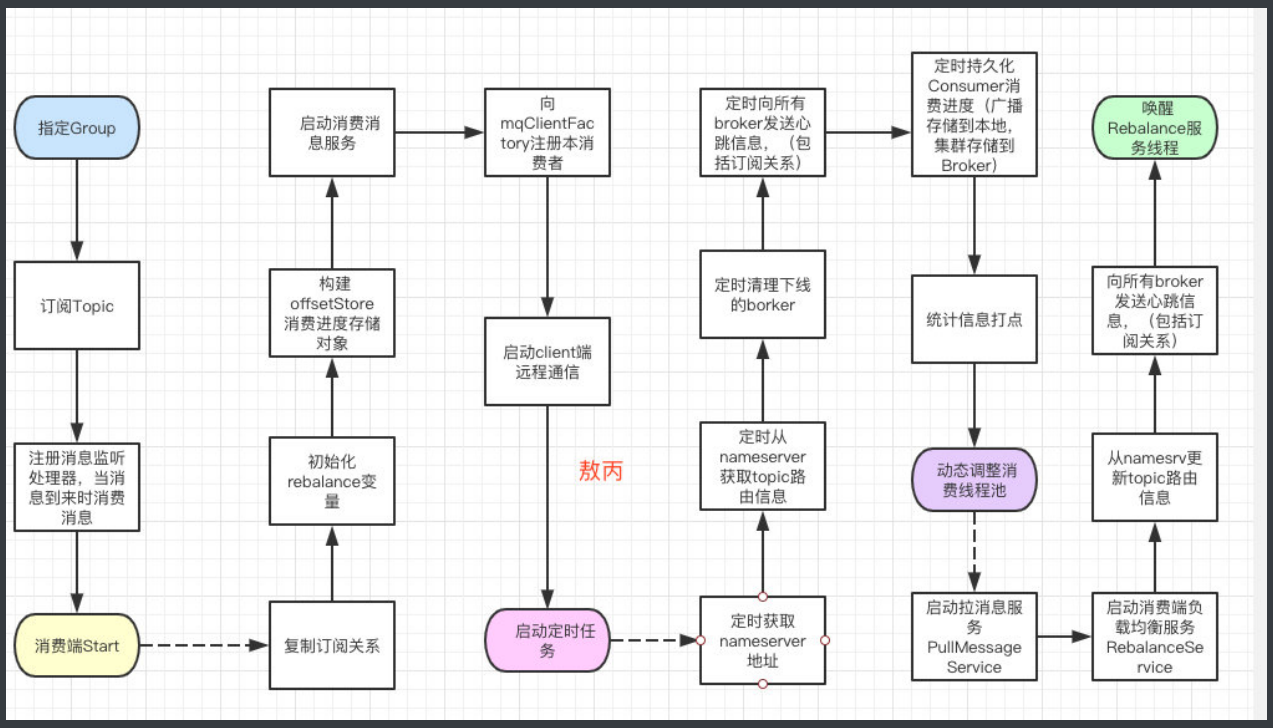
Broker模块是通过BrokerStartup进行启动的，会实例化BrokerController，并且调用其初始化方法

```
public static BrokerController start(BrokerController controller) {
        try {

            controller.start();

            String tip = "The broker[" + controller.getBrokerConfig().getBrokerName() + ", "
                + controller.getBrokerAddr() + "] boot success. serializeType=" +
RemotingCommand.getSerializeTypeConfigInThisServer();

            if (null != controller.getBrokerConfig().getNamesrvAddr()) {
                tip += " and name server is " + controller.getBrokerConfig().getNamesrvAddr();
            }

            log.info(tip);
            System.out.printf("%s%n", tip);
            return controller;
        } catch (Throwable e) {
            e.printStackTrace();
            System.exit(-1);
        }

        return null;
    }
```

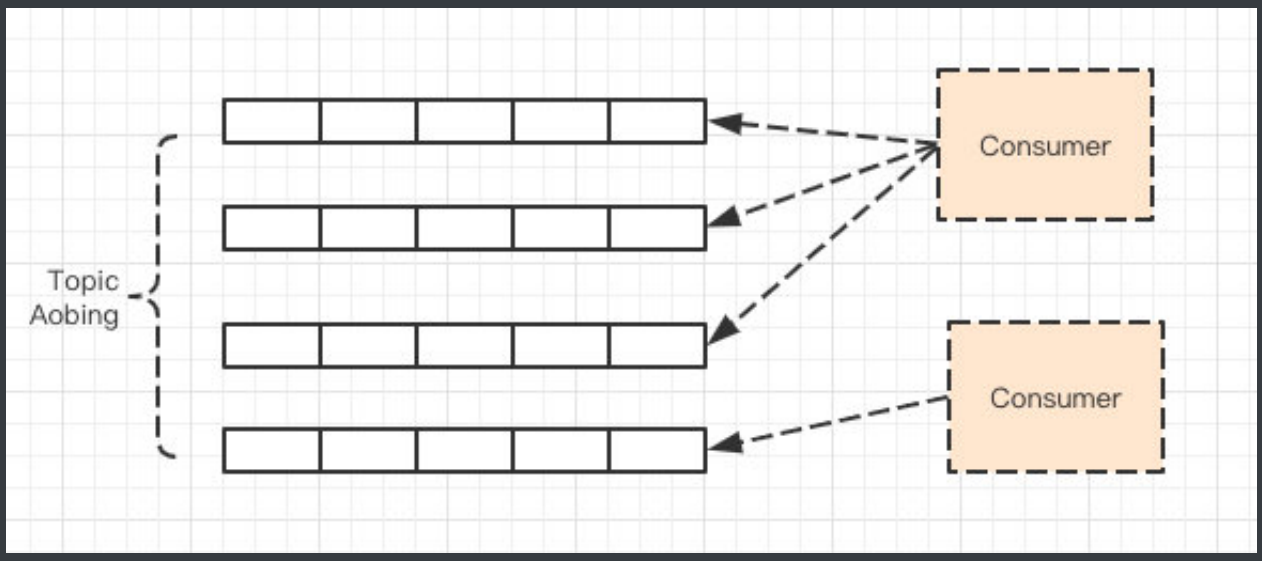
大家去看**Broker**的源码的话会发现，他的**初始化流程很冗长**，会根据配置创建很多线程池主要用来**发送消息**、**拉取消息**、**查询消息**、**客户端管理**和**消费者管理**，也有很多**定时任务**，同时也注册了很多**请求处理器**，用来发送拉取消息查询消息的。

Consumer

不说了直接怼图吧！要死了，下次我还是做扫盲，写点爽文吧555



Consumer是消息接受，那他怎么接收消息的呢?



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

面试常见问题分析

他的优缺点是啥

RocketMQ优点：

- 单机吞吐量：十万级
- 可用性：非常高，分布式架构
- 消息可靠性：经过参数优化配置，消息可以做到0丢失

- 功能支持：MQ功能较为完善，还是分布式的，扩展性好
- 支持10亿级别的消息堆积，不会因为堆积导致性能下降
- 源码是java，我们可以自己阅读源码，定制自己公司的MQ，可以掌控
- 天生为金融互联网领域而生，对于可靠性要求很高的场景，尤其是电商里面的订单扣款，以及业务削峰，在大量交易涌入时，后端可能无法及时处理的情况
- **RoketMQ**在稳定性上可能更值得信赖，这些业务场景在阿里双11已经经历了多次考验，如果你的业务有上述并发场景，建议可以选择**RocketMQ**

RocketMQ缺点：

- 支持的客户端语言不多，目前是java及c++，其中c++不成熟
- 社区活跃度不是特别活跃那种
- 没有在 mq 核心中去实现**JMS**等接口，有些系统要迁移需要修改大量代码

消息去重

去重原则：使用业务端逻辑保持幂等性

幂等性：就是用户对于同一操作发起的一次请求或者多次请求的结果是一致的，不会因为多次点击而产生了副作用，数据库的结果都是唯一的，不可变的。

只要保持幂等性，不管来多少条重复消息，最后处理的结果都一样，需要业务端来实现。

去重策略：保证每条消息都有唯一编号(**比如唯一流水号)**，且保证消息处理成功与去重表的日志同时出现。

建立一个消息表，拿到这个消息做数据库的insert操作。给这个消息做一个唯一主键（primary key）或者唯一约束，那么就算出现重复消费的情况，就会导致主键冲突，那么就不再处理这条消息。

消息重复

消息领域有一个对消息投递的QoS定义，分为：

- 最多一次（At most once）
- 至少一次（At least once）
- 仅一次（ Exactly once）

> QoS：Quality of Service，服务质量

几乎所有的MQ产品都声称自己做到了**At least once**。

既然是至少一次，那避免不了消息重复，尤其是在分布式网络环境下。

比如：网络原因闪断，ACK返回失败等等故障，确认信息没有传送到消息队列，导致消息队列不知道自己已经消费过该消息了，再次将该消息分发给其他的消费者。

不同的消息队列发送的确认信息形式不同，例如**RabbitMQ**是发送一个ACK确认消息，**RocketMQ**是返回一个CONSUME_SUCCESS成功标志，**Kafka**实际上有个offset的概念。

RocketMQ没有内置消息去重的解决方案，最新版本是否支持还需确认。

消息的可用性

当我们选择好了集群模式之后，那么我们需要关心的就是怎么去存储和复制这个数据，**RocketMQ**对消息的刷盘提供了同步和异步的策略来满足我们的，当我们选择同步刷盘之后，如果刷盘超时会给返回FLUSH_DISK_TIMEOUT，如果是异步刷盘不会返回刷盘相关信息，选择同步刷盘可以尽最大程度满足我们的消息不会丢失。

除了存储有选择之后，我们的主从同步提供了同步和异步两种模式来进行复制，当然选择同步可以提升可用性，但是消息的发送RT时间会下降10%左右。

RocketMQ采用的是混合型的存储结构，即为**Broker**单个实例下所有的队列共用一个日志数据文件（即为CommitLog）来存储。

而**Kafka**采用的是独立型的存储结构，每个队列一个文件。

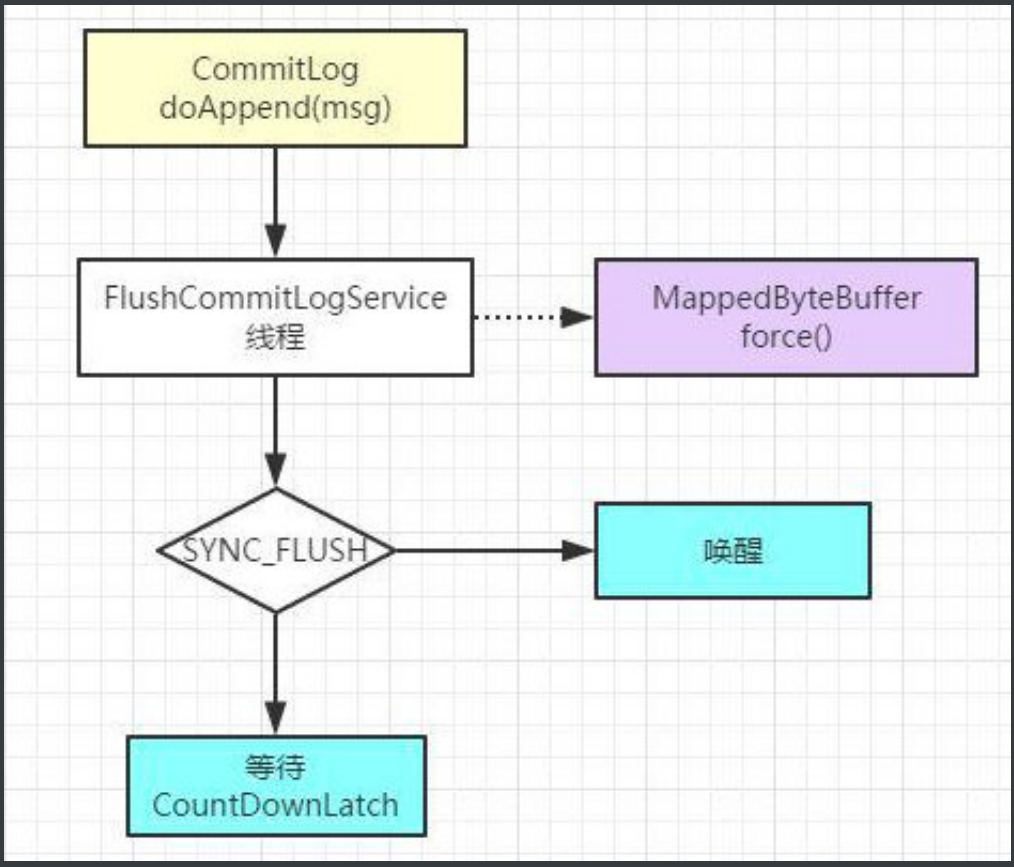
这里帅丙认为，**RocketMQ**采用混合型存储结构的缺点在于，会存在较多的随机读操作，因此读的效率偏低。同时消费消息需要依赖**ConsumeQueue**，构建该逻辑消费队列需要一定开销。

RocketMQ 刷盘实现

Broker 在消息的存取时直接操作的是内存（内存映射文件），这可以提供系统的吞吐量，但是无法避免机器掉电时数据丢失，所以需要持久化到磁盘中。

刷盘的最终实现都是使用**NIO**中的 MappedByteBuffer.force() 将映射区的数据写入到磁盘，如果是同步刷盘的话，在**Broker**把消息写到**CommitLog**映射区后，就会等待写入完成。

异步而言，只是唤醒对应的线程，不保证执行的时机，流程如图所示。



顺序消息：

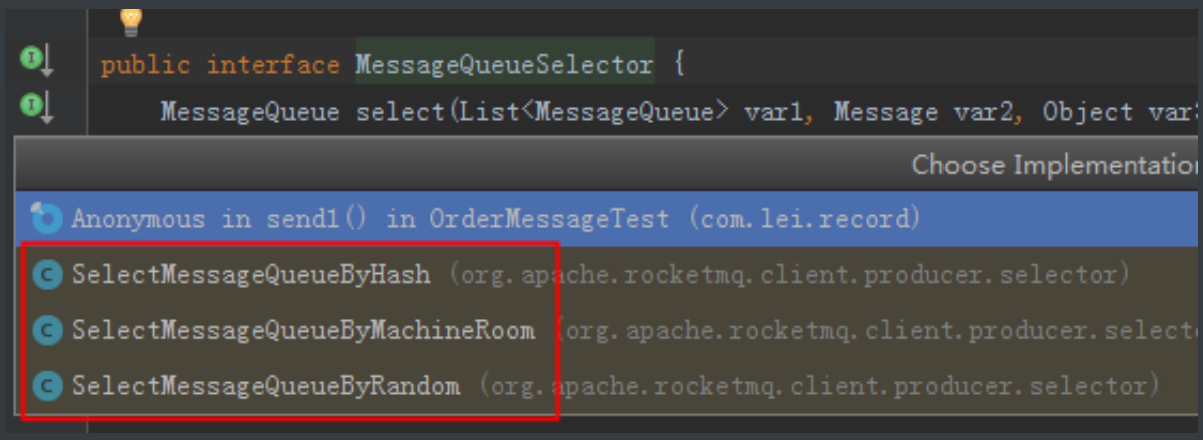
我简单的说一下我们使用的**RocketMQ**里面的一个简单实现吧。

Tip：为啥用**RocketMQ**举例呢，这玩意是阿里开源的，我问了下身边的朋友很多公司都有使用，所以读者大概率是这个的话我就用这个举例吧，具体的细节我后面会在**RocketMQ**和**Kafka**各自章节说到。

生产者消费者一般需要保证顺序消息的话，可能就是一个业务场景下的，比如订单的创建、支付、发货、收货。

那这些东西是不是一个订单号呢？一个订单的肯定是一个订单号的说，那简单了呀。

一个topic下有多个队列，为了保证发送有序，**RocketMQ**提供了**MessageQueueSelector**队列选择机制，他有三种实现：



我们可使用**Hash取模法**，让同一个订单发送到同一个队列中，再使用同步发送，只有同个订单的创建消息发送成功，再发送支付消息。这样，我们保证了发送有序。

RocketMQ的topic内的队列机制,可以保证存储满足**FIFO**（First Input First Output 简单说就是指先进先出）,剩下的只需要消费者顺序消费即可。

RocketMQ仅保证顺序发送，顺序消费由消费者业务保证!!!

这里很好理解，一个订单你发送的时候放到一个队列里面去，你同一个的订单号Hash一下是不是还是一样的结果，那肯定是一个消费者消费，那顺序是不是就保证了?

真正的顺序消费不同的中间件都有自己的不同实现我这里就举个例子，大家思路理解下。

分布式事务:

Half Message(半消息)

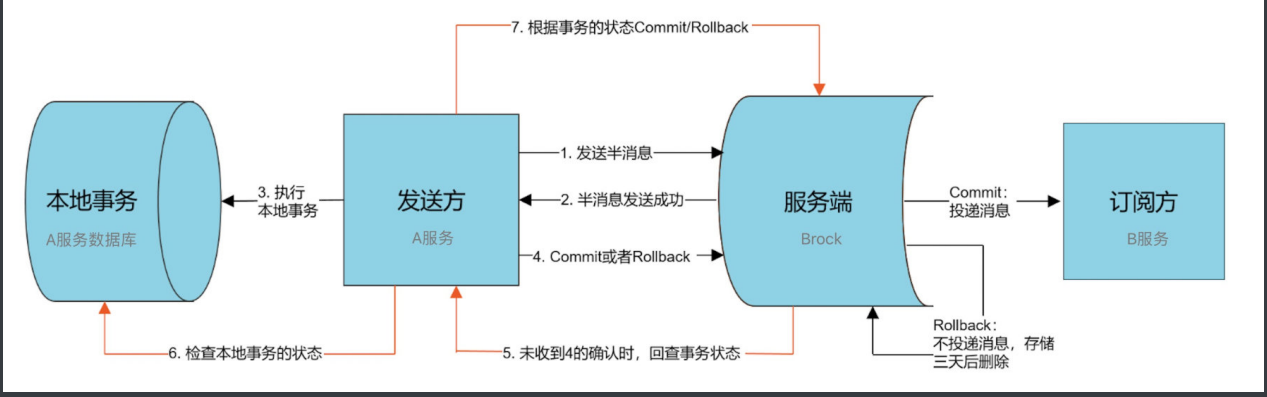
是指暂不能被Consumer消费的消息。Producer 已经把消息成功发送到了 Broker 端，但此消息被标记为 `暂不能投递` 状态，处于该种状态下的消息称为半消息。需要 Producer

对消息的 `二次确认` 后，Consumer才能去消费它。

消息回查

由于网络闪段，生产者应用重启等原因。导致 **Producer** 端一直没有对 **Half Message(半消息)** 进行 **二次确认**。这是**Brock**服务器会定时扫描 `长期处于半消息的消息` ，会

主动询问 **Producer**端 该消息的最终状态(**Commit或者Rollback**),该消息即为 **消息回查**。



1. A服务先发送个Half Message给Brock端，消息中携带 B服务 即将要+100元的信息。
2. 当A服务知道Half Message发送成功后，那么开始第3步执行本地事务。
3. 执行本地事务(会有三种情况1、执行成功。2、执行失败。3、网络等原因导致没有响应)
4. 如果本地事务成功，那么Product像Brock服务器发送Commit,这样B服务就可以消费该message。
5. 如果本地事务失败，那么Product像Brock服务器发送Rollback,那么就会直接删除上面这条半消息。
6. 如果因为网络等原因迟迟没有返回失败还是成功，那么会执行RocketMQ的回调接口,来进行事务的回查。

消息过滤

- **Broker**端消息过滤　　在**Broker**中，按照**Consumer**的要求做过滤，优点是减少了对于**Consumer**无用消息的网络传输。缺点是增加了Broker的负担，实现相对复杂。
- **Consumer**端消息过滤 这种过滤方式可由应用完全自定义实现，但是缺点是很多无用的消息要传输到**Consumer**端。

Broker的Buffer问题

Broker的**Buffer**通常指的是Broker中一个队列的内存Buffer大小，这类**Buffer**通常大小有限。

另外，RocketMQ没有内存**Buffer**概念，RocketMQ的队列都是持久化磁盘，数据定期清除。

RocketMQ同其他MQ有非常显著的区别，RocketMQ的内存**Buffer**抽象成一个无限长度的队列，不管有多少数据进来都能装得下，这个无限是有前提的，Broker会定期删除过期的数据。

例如Broker只保存3天的消息，那么这个**Buffer**虽然长度无限，但是3天前的数据会被从队尾删除。

回溯消费

回溯消费是指Consumer已经消费成功的消息，由于业务上的需求需要重新消费，要支持此功能，Broker在向Consumer投递成功消息后，消息仍然需要保留。并且重新消费一般是按照时间维度。

例如由于Consumer系统故障，恢复后需要重新消费1小时前的数据，那么Broker要提供一种机制，可以按照时间维度来回退消费进度。

RocketMQ支持按照时间回溯消费，时间维度精确到毫秒，可以向前回溯，也可以向后回溯。

消息堆积

消息中间件的主要功能是异步解耦，还有个重要功能是挡住前端的数据洪峰，保证后端系统的稳定性，这就要求消息中间件具有一定的消息堆积能力，消息堆积分以下两种情况：

- 消息堆积在内存**Buffer**，一旦超过内存**Buffer**，可以根据一定的丢弃策略来丢弃消息，如CORBA Notification规范中描述。适合能容忍丢弃消息的业务，这种情况消息的堆积能力主要在于内存**Buffer**大小，而且消息堆积后，性能下降不会太大，因为内存中数据多少对于对外提供的访问能力影响有限。
- 消息堆积到持久化存储系统中，例如DB，KV存储，文件记录形式。 当消息不能在内存Cache命中时，要不可避免的访问磁盘，会产生大量读IO，读IO的吞吐量直接决定了消息堆积后的访问能力。
- 评估消息堆积能力主要有以下四点：
  - 消息能堆积多少条，多少字节？即消息的堆积容量。
  - 消息堆积后，发消息的吞吐量大小，是否会受堆积影响?
  - 消息堆积后，正常消费的Consumer是否会受影响?
  - 消息堆积后，访问堆积在磁盘的消息时，吞吐量有多大?

定时消息

定时消息是指消息发到**Broker**后，不能立刻被**Consumer**消费，要到特定的时间点或者等待特定的时间后才能被消费。

如果要支持任意的时间精度，在**Broker**层面，必须要做消息排序，如果再涉及到持久化，那么消息排序要不可避免的产生巨大性能开销。

RocketMQ支持定时消息，但是不支持任意时间精度，支持特定的level，例如定时5s，10s，1m等。

总结

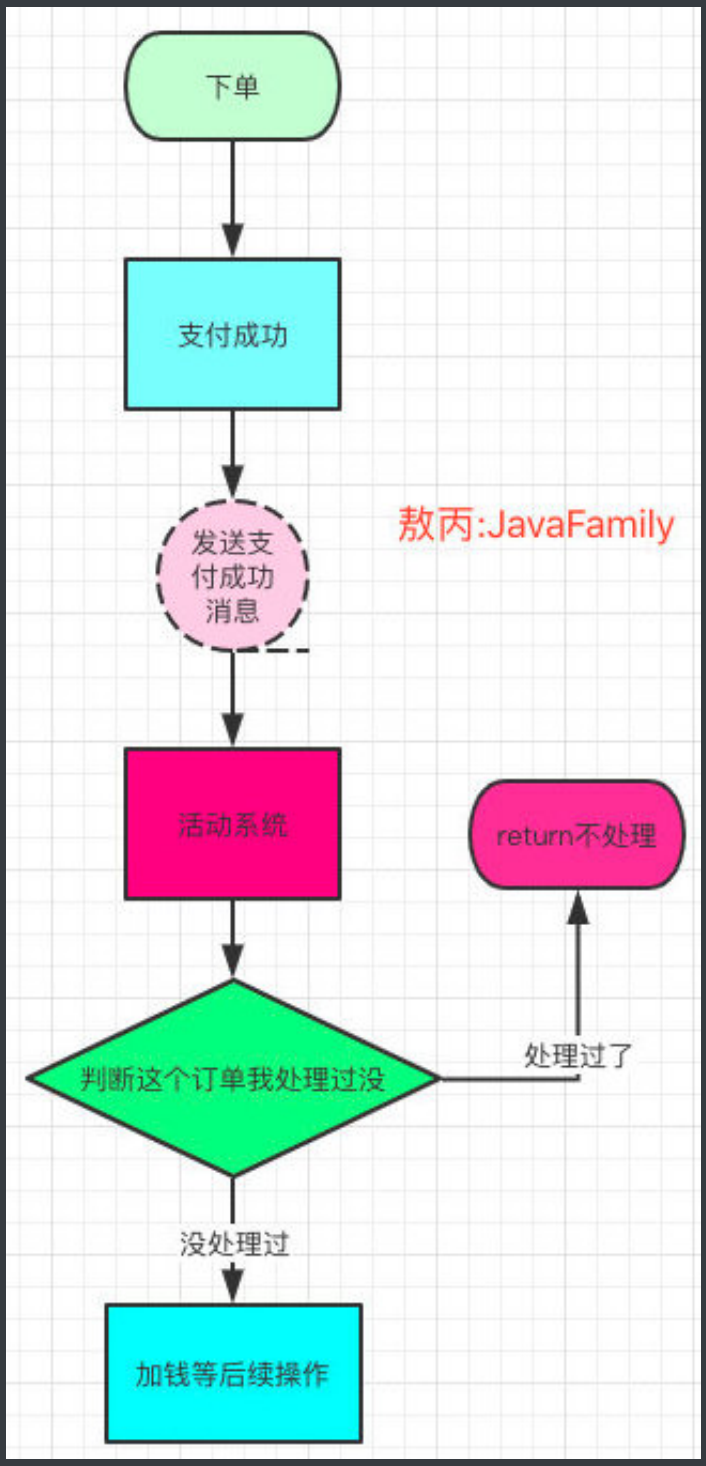
写这种单纯介绍中间件的枯燥乏味，大家看起来估计也累，目前已经破一万个字了，以后我这种类型的少写，大家老是让我写点深度的，我说真的很多东西我源码一贴，看都没人看。

Kafka我就不发博客了，大家可以去**GItHub**上第一时间阅读，后面会出怎么搭建项目在服务器的教程，还有一些大牛个人经历和个人书单的东西，今年应该先这么写，主要是真心太忙了，望理解。

絮叨

我也不过多描述了，反正嘛网络上重拳出击嘛，现实中唯唯诺诺，让他说理由也说不出来，不回我。

他说的是下面这个场景多线程的情况，就是第一个线程还没走完，第二个现在进来，也判断没处理过那不就两个都继续加了么？



订单号+业务场景，组成一个唯一主键，你插入数据库只能成功第一个，后续的都会报错的，报违反唯一主键的错误。

还有就是有人疑惑为啥不直接就不判断就等他插入的时候报错，丢掉后续的就好了？

你要知道**报错有很多种**，你哪里知道不是数据库挂了的错？或者别的运行时异常？

不过你如果可以做到抛特定的异常也可以，反正我们要**减少数据库的报错**，如果并发大，像我现在负责的系统都是10W+QPS，那日志会打满疯狂报警的。（就是正常情况我们都经常报警）

解决问题的思路有很多，喷我可以，讲清楚问题，讲清楚你的理由。

很多大家都只是单方面的知识摄入，就这样还要喷我，还有一上来就问我为啥今天没发文章，我欠你的？我工作日上班，周六周日都怼上去了，时间有限啊，哥哥。

大家都有自己的事情，写文章也耗时耗脑，难免出错，还望理解。

日常求赞

好了各位，以上就是这篇文章的全部内容了，能看到这里的人呀，都是**人才**。

我后面会每周都更新几篇《吊打面试官》系列和互联网常用技术栈相关的文章，非常感谢**人才**们能看到这里，如果这个文章写得还不错，觉得「敖丙」我**有点东西**的话 **求点赞**👍 **求关注**❤️ **求分享**👥 对暖男我来说真的 **非常有用**！！！

创作不易，各位的支持和认可，就是我创作的最大动力，我们下篇文章见！

敖丙 | 文 【原创】【转载请联系本人】 如果本篇博客有任何错误，请批评指教，不胜感激！

> 《吊打面试官》系列每周持续更新，可以关注我的公众号「三太子敖丙」第一时间阅读和催更（公众号比博客早一到两篇哟），本文**GitHub**https://github.com/JavaFamily 已收录，有一线大厂面试点思维导图，欢迎Star和完善，里面也有我个人联系方式有什么问题也可以直接找我，也有技术交流群，我们一起有点东西。