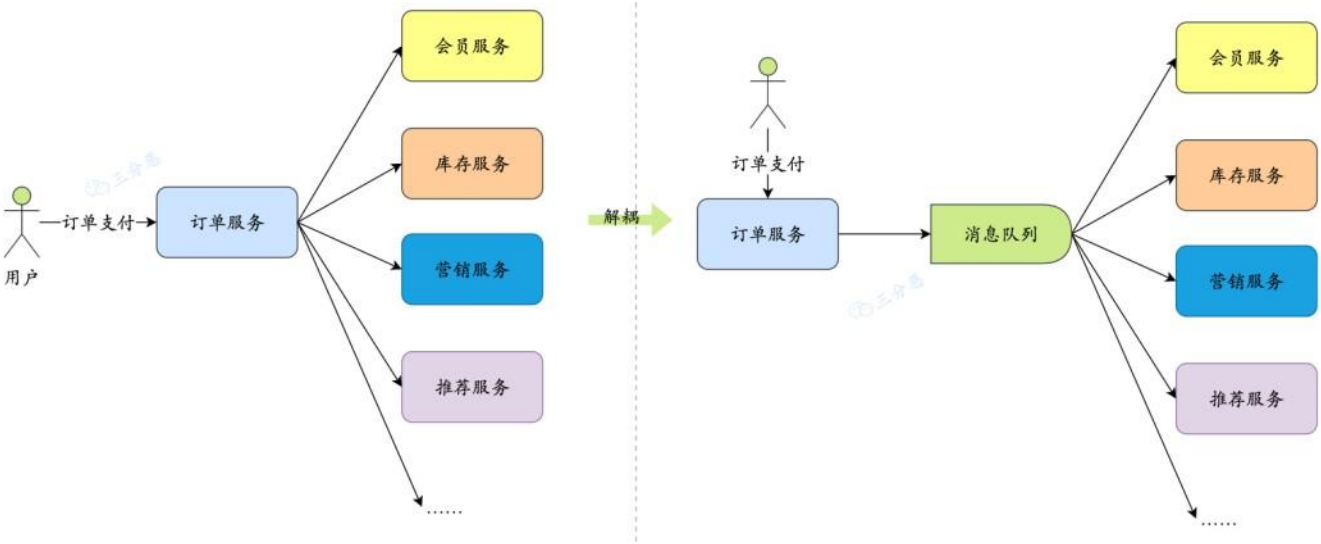
> 图文详解 RocketMQ 面试高频题，这次吊打面试官，我觉得稳了（手动 dog）。整理：楼仔，作者：三分恶，戳原文链接。

# 基础

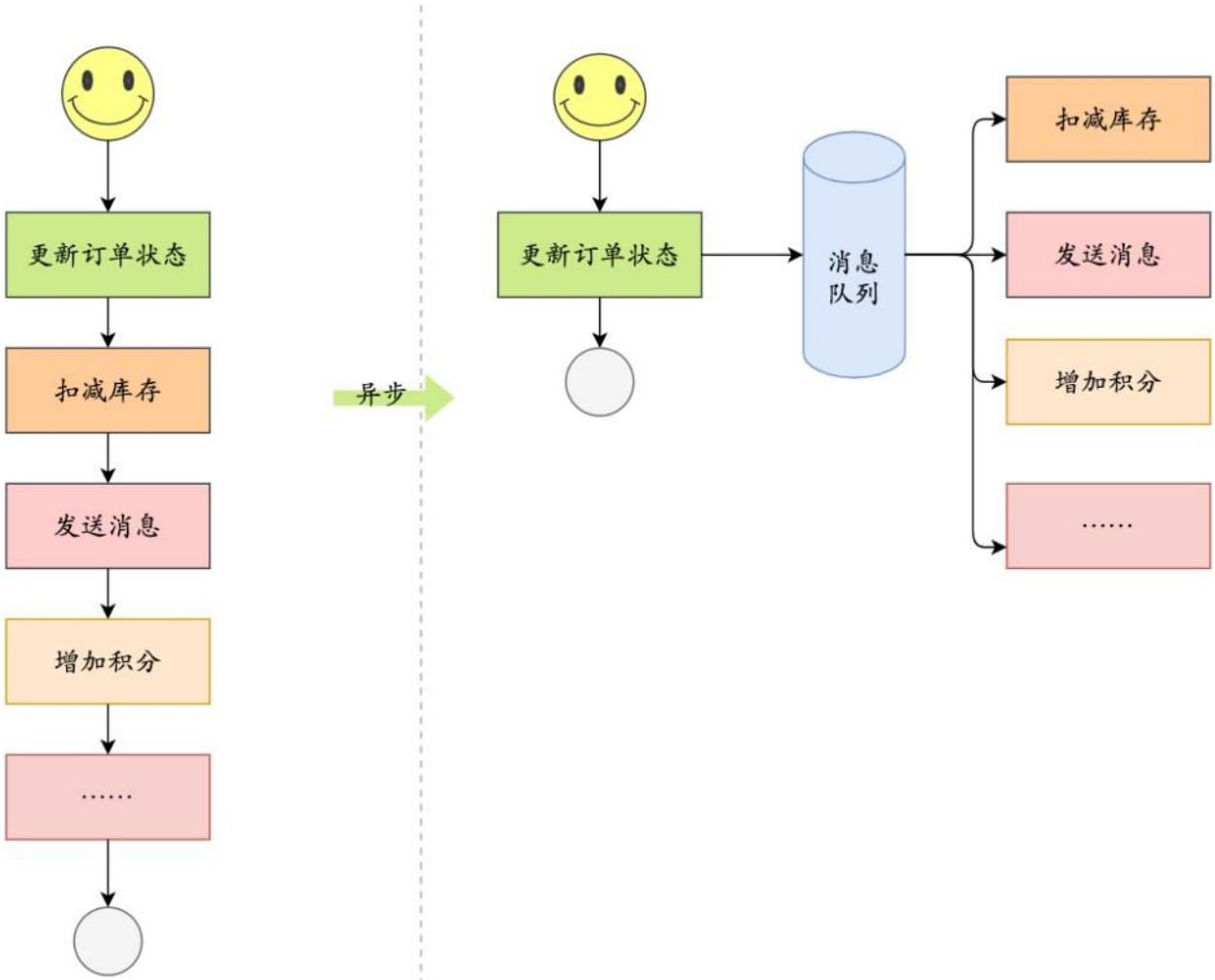
## 1.为什么要使用消息队列呢？

消息队列主要有三大用途，我们拿一个电商系统的下单举例：

- **解耦**：引入消息队列之前，下单完成之后，需要订单服务去调用库存服务减库存，调用营销服务加营销数据……引入消息队列之后，可以把订单完成的消息丢进队列里，下游服务自己去调用就行了，这样就完成了订单服务和其它服务的解耦合。

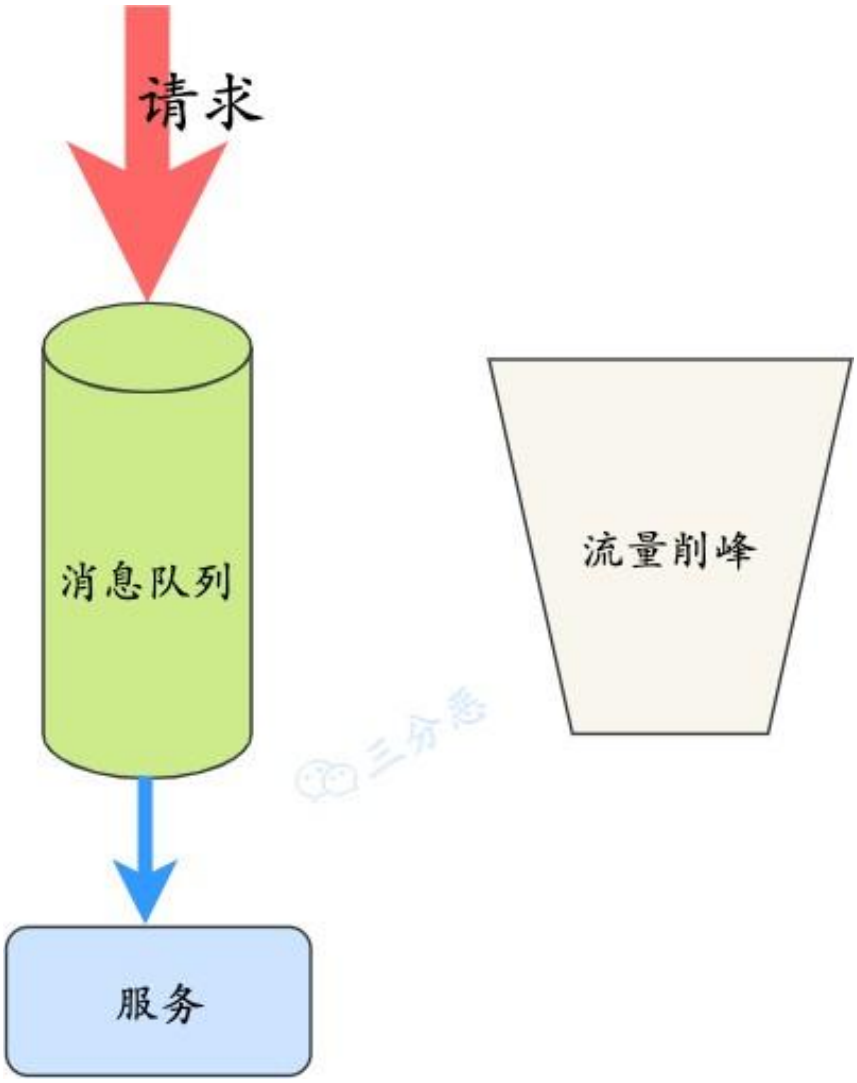



- **异步**：订单支付之后，我们要扣减库存、增加积分、发送消息等等，这样一来这个链路就长了，链路一长，响应时间就变长了。引入消息队列，除了更新订单状态，其它的都可以异步去做，这样一来就来，就能降低响应时间。




- **削峰**：消息队列合一用来削峰，例如秒杀系统，平时流量很低，但是要做秒杀活动，秒杀的时候流量疯狂怼进来，我们的服务器，Redis，MySQL各自的承受能力都不一样，直接全部流量照单全收肯定有问题啊，严重点可能直接打挂了。

我们可以把请求扔到队列里面，只放出我们服务能处理的流量，这样就能抗住短时间的大流量了。




解耦、异步、削峰，是消息队列最主要的三大作用。

## 2.为什么要选择RocketMQ?

市场上几大消息队列对比如下：

| | RabbitMQ | ActiveMQ | RocketMQ | Kafka |
|---|---|---|---|---|
| 公司/社区 | Rabbit | Apache | 阿里 | Apache |
| 语言 | Erlang | Java | Java | Scala&Java |
| | | OpenWire、 | | 自定义协议，社 |

| | | | | |
|---|---|---|---|---|
| 协议支持 | AMQP | STOMP、REST、XMPP、AMQP | 自定义 | 区封装了http协议支持 |
| 客户端支持语言 | 官方支持Erlang、Java、Ruby等，社区查出多种API，几乎支持所有语言 | Java、C、C++、Python、PHP、Perl、.net 等 | Java<br>C++（不成熟） | 官方支持Java，社区产出多种API，如PHP，Python等 |
| 单机吞吐量 | 万级<br>③ | 万级<br>④ | 十万级<br>① | 十万级<br>② |
| 消息延迟 | 微秒级 | 毫秒级 | 毫秒级 | 毫秒以内 |
| 可用性 | 高，基于主从架构实现可用性 | 高，基于主从架构实现可用性 | 非常高，分布式架构 | 非常高，分布式架构，一个数据多副本 |
| 消息可靠性 | - | 有较低的概率丢失数据 | 经过参数优化配置，可以做到零丢失 | 经过参数配置，消息可以做到零丢失 |
| 功能支持 | 基于erlang开发，所以并发性能极强，性能极好，延时低 | MQ领域的功能极其完备 | MQ功能较为完备，分布式扩展性好 | 功能较为简单，主要支持加单MQ功能 |
| 优势 | erlang语言开发，性能极好、延时很低，吞吐量万级、MQ功能完备，管理界面非常好，社区活跃；互联网公司使用较多 | 非常成熟，功能强大，在业内大量公司和项目中都有应用 | 接口简单易用，阿里出品有保障，吞吐量大，分布式扩展方便、社区比较活跃，支持大规模的Topic、支持复杂的业务场景，可以基于源码进行定制开发 | 超高吞吐量，ms级的时延，极高的可用性和可靠性，分布式扩展方便 |
| 劣势 | 吞吐量较低，erlang语音开发不容易进行定制开发，集群动态扩展麻烦 | 偶尔有较低概率丢失消息，社区活跃度不高 | 接口不是按照标准JMS规范走的，有的系统迁移要修改大量的代码，技术有被抛弃的风险 | 有可能进行消息的重复消费 |

【三分恶】原创

| | | | | |
|---|---|---|---|---|
| 应用 | 都有使用 | 主要用于解耦和异步，较少用在大规模吞吐的场景中 | 用于大规模吞吐、复杂业务中 | 在大数据的实时计算和日志采集中被大规模使用，是业界的标准 |

**总结一下：**

选择中间件的可以从这些维度来考虑：可靠性，性能，功能，可运维行，可拓展性，社区活跃度。目前常用的几个中间件，ActiveMQ作为“老古董”，市面上用的已经不多，其它几种：

- RabbitMQ：
- 优点：轻量，迅捷，容易部署和使用，拥有灵活的路由配置缺
- 点：性能和吞吐量不太理想，不易进行二次开发 RocketMQ：
- 优点：性能好，高吞吐量，稳定可靠，有活跃的中文社区
- 缺点：兼容性上不是太好
- Kafka：
- 优点：拥有强大的性能及吞吐量，兼容性很好
- 缺点：由于“攒一波再处理”导致延迟比较高

我们的系统是面向用户的C端系统，具有一定的并发量，对性能也有比较高的要求，所以选择了低延迟、吞吐量比较高，可用性比较好的RocketMQ。

## 3.RocketMQ有什么优缺点？

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，方便结合公司自己的业务二次开发
- 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：

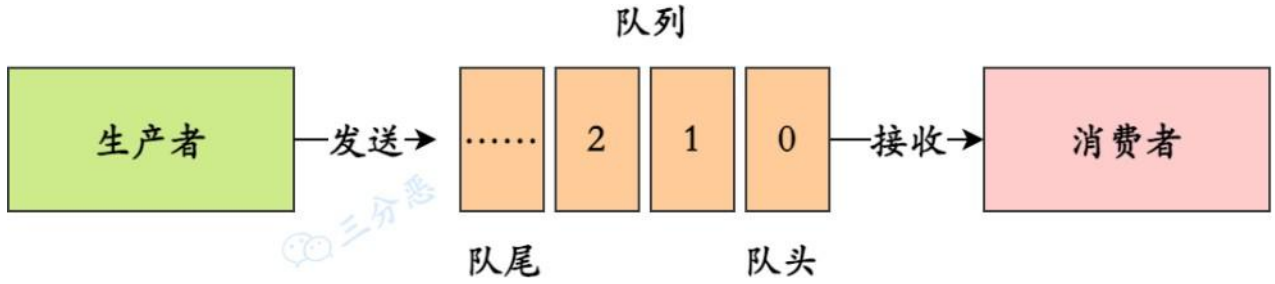
- 支持的客户端语言不多，目前是Java及c++，其中c++不成熟
- 没有在 MQ核心中去实现JMS等接口，有些系统要迁移需要修改大量代码

# 4.消息队列有哪些消息模型？

消息队列有两种模型：**队列模型**和**发布/订阅模型**。

- **队列模型**

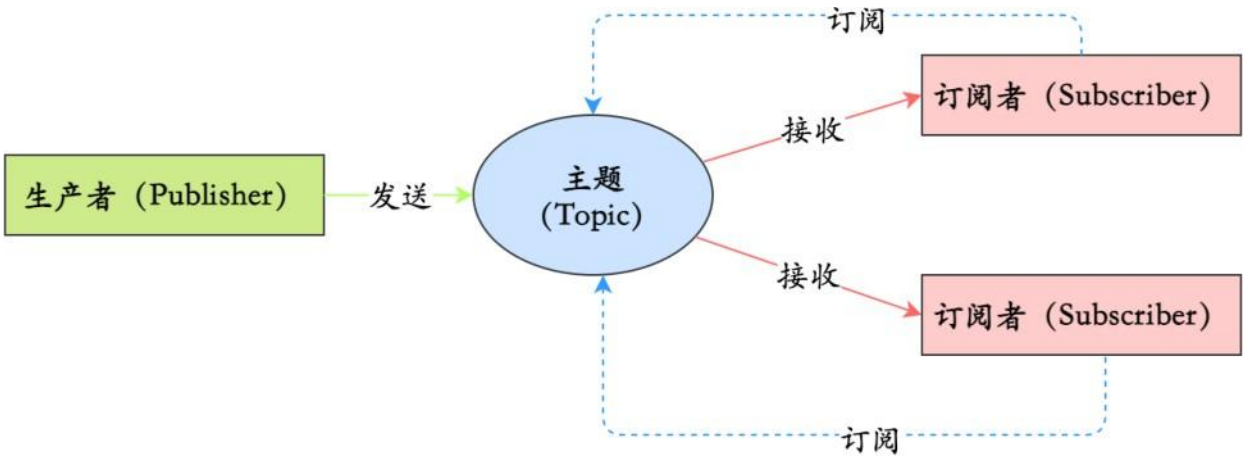
这是最初的一种消息队列模型，对应着消息队列“发-存-收”的模型。生产者往某个队列里面发送消息，一个队列可以存储多个生产者的消息，一个队列也可以有多个消费者，但是消费者之间是竞争关系，也就是说每条消息只能被一个消费者消费。




- **发布/订阅模型**

如果需要将一份消息数据分发给多个消费者，并且每个消费者都要求收到全量的消息。很显然，队列模型无法满足这个需求。解决的方式就是发布/订阅模型。

在发布 - 订阅模型中，消息的发送方称为发布者（Publisher），消息的接收方称为订阅者（Subscriber），服务端存放消息的容器称为主题（Topic）。发布者将消息发送到主题中，订阅者在接收消息之前需要先“订阅主题”。“订阅”在这里既是一个动作，同时还可以认为是主题在消费时的一个逻辑副本，每份订阅中，订阅者都可以接收到主题的所有消息。

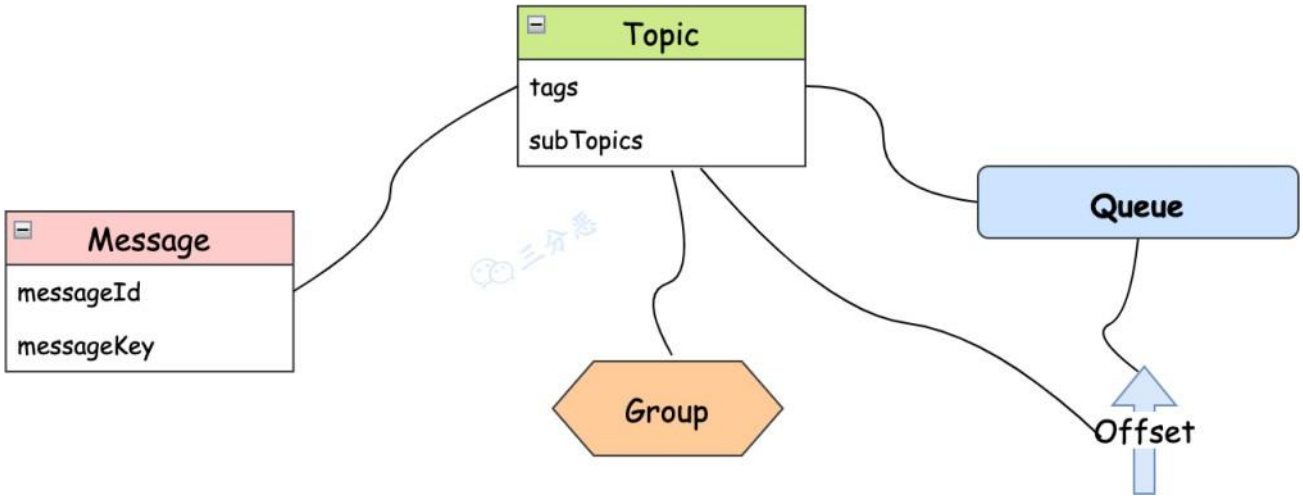



它和 “队列模式” 的异同：生产者就是发布者，队列就是主题，消费者就是订阅者，无本质区别。唯一的不同点在于：一份消息数据是否可以被多次消费。

## 5.那RocketMQ的消息模型呢？

RocketMQ使用的消息模型是标准的发布-订阅模型，在RocketMQ的术语表中，生产者、消费者和主题，与发布-订阅模型中的概念是完全一样的。

RocketMQ本身的消息是由下面几部分组成：




- 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）可以看做消息的归类，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 。

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

- Group

RocketMQ中，订阅者的概念是通过消费组（Consumer Group）来体现的。每个消费组都消费主题中一份完整的消息，不同消费组之间消费进度彼此不受影响，也就是说，一条消息被Consumer Group1消费过，也会再给 Consumer Group2消费。

消费组中包含多个消费者，同一个组内的消费者是竞争消费的关系，每个消费者负责消费组内的一部分消息。默认情况，如果一条消息被消费者Consumer1消费了，那同组的其他消费者就不会再收到这条消息。

- Message Queue

Message Queue（消息队列），一个 Topic 下可以设置多个消息队列，Topic 包括多个 Message Queue ，如果一个 Consumer 需要获取 Topic下所有的消息，就要遍历所有的 Message Queue。

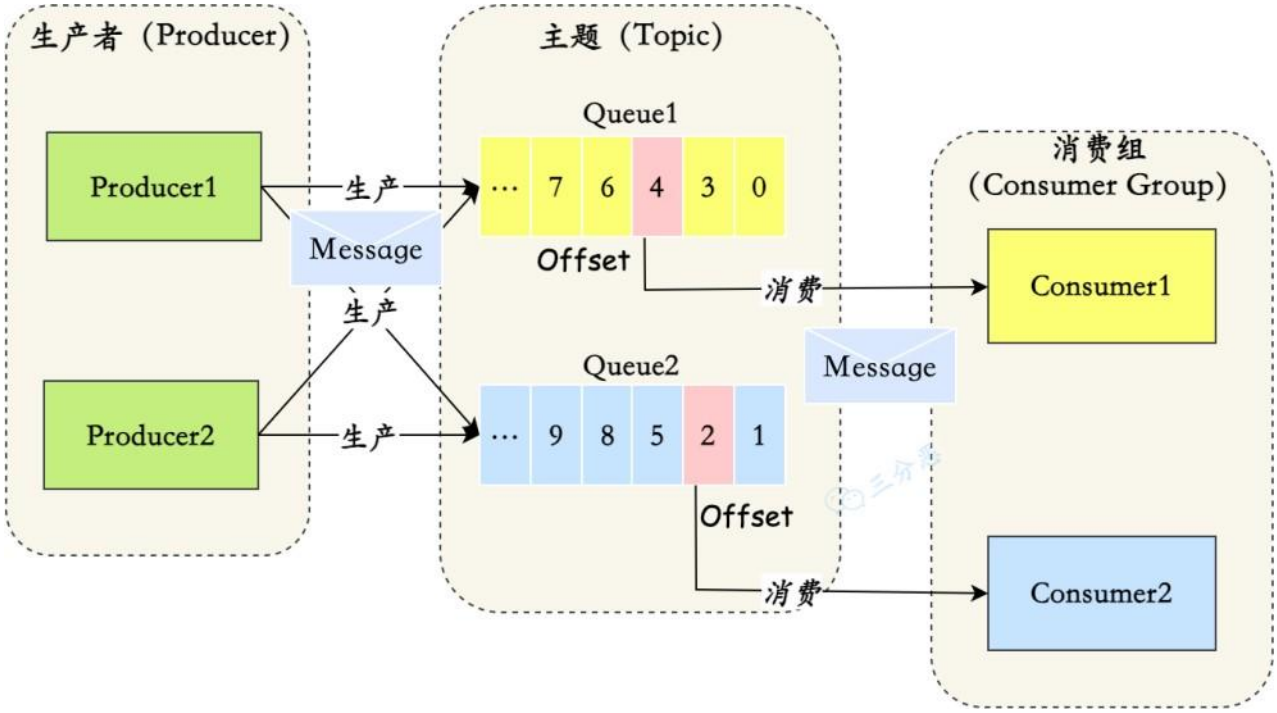
RocketMQ还有一些其它的Queue——例如ConsumerQueue。

- Offset

在Topic的消费过程中，由于消息需要被不同的组进行多次消费，所以消费完的消息并不会立即被删除，这就需要RocketMQ为每个消费组在每个队列上维护一个消费位置（Consumer Offset），这个位置之前的消息都被消费过，之后的消息都没有被消费过，每成功消费一条消息，消费位置就加一。

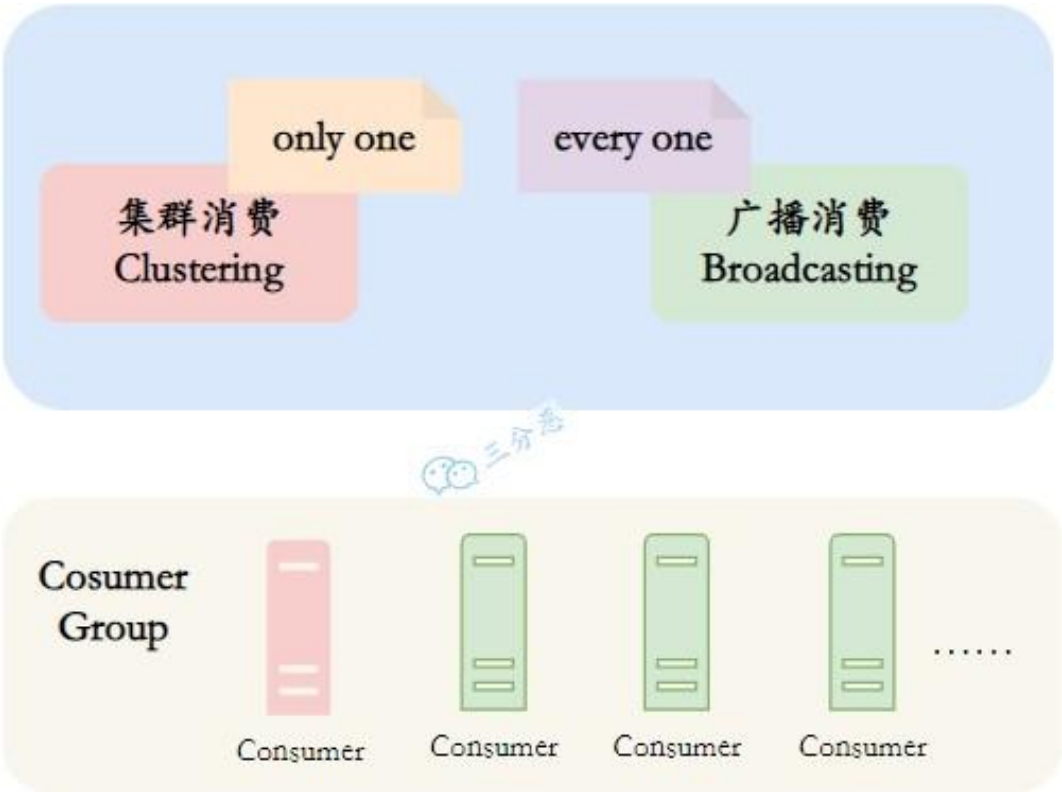
也可以这么说，`Queue` 是一个长度无限的数组，Offset 就是下标。

RocketMQ的消息模型中，这些就是比较关键的概念了。画张图总结一下：




## 6.消息的消费模式了解吗？

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

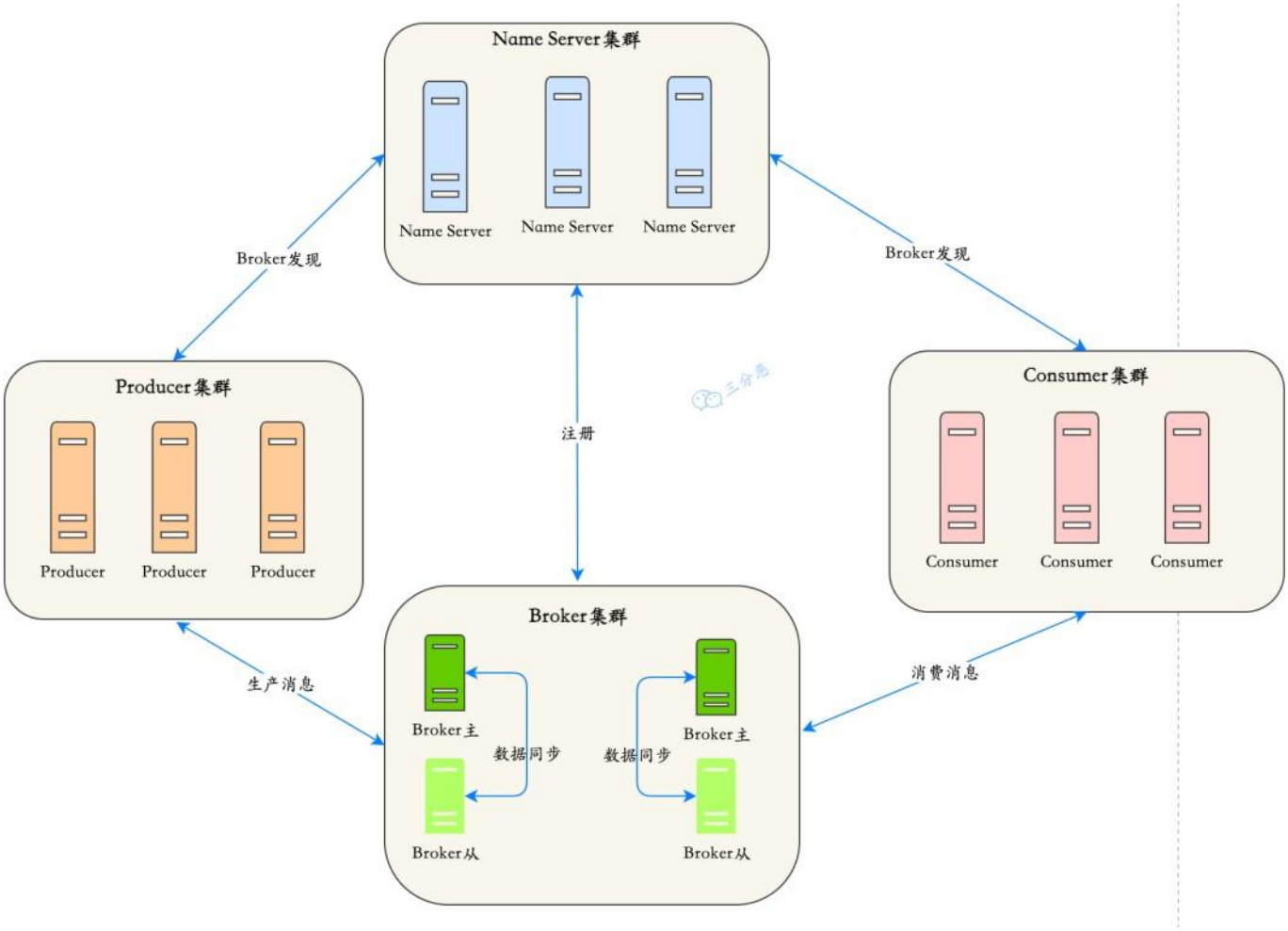



默认情况下就是集群消费，这种模式下一个消费者组共同消费一个主题的多个队列，一个队列只会被一个消费者消费，如果某个消费者挂掉，分组内其它消费者会接替挂掉的消费者继续消费。

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。

## 7.RoctetMQ基本架构了解吗？

先看图，RocketMQ的基本架构：

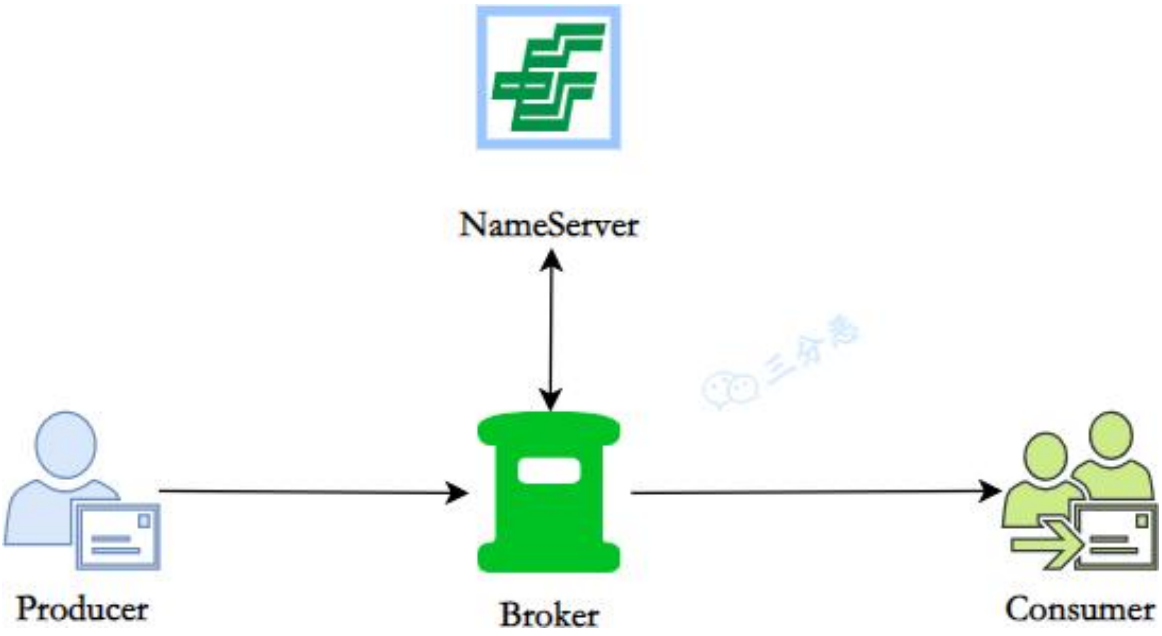



RocketMQ 一共有四个部分组成：NameServer，Broker，Producer 生产者，Consumer 消费者，它们对应了：发现、发、存、收，为了保证高可用，一般每一部分都是集群部署的。

## 8.那能介绍一下这四部分吗？

类比一下我们生活的邮政系统——

邮政系统要正常运行，离不开下面这四个角色， 一是发信者，二 是收信者， 三是负责暂存传输的邮局， 四是负责协调各个地方邮局的管理机构。对应到 RocketMQ 中，这四个角色就是 Producer、 Consumer、 Broker 、 NameServer。




NameServer

NameServer 是一个无状态的服务器，角色类似于 Kafka使用的 Zookeeper，但比 Zookeeper 更轻量。特点：

- 每个 NameServer 结点之间是相互独立，彼此没有任何信息交互。
- Nameserver 被设计成几乎是无状态的，通过部署多个结点来标识自己是一个伪集群，Producer 在发送消息前从 NameServer 中获取 Topic 的路由信息也就是发往哪个 Broker，Consumer 也会定时从 NameServer获取 Topic 的路由信息，Broker 在启动时会向 NameServer 注册，并定时进行心跳连接，且定时同步维护的 Topic 到 NameServer。

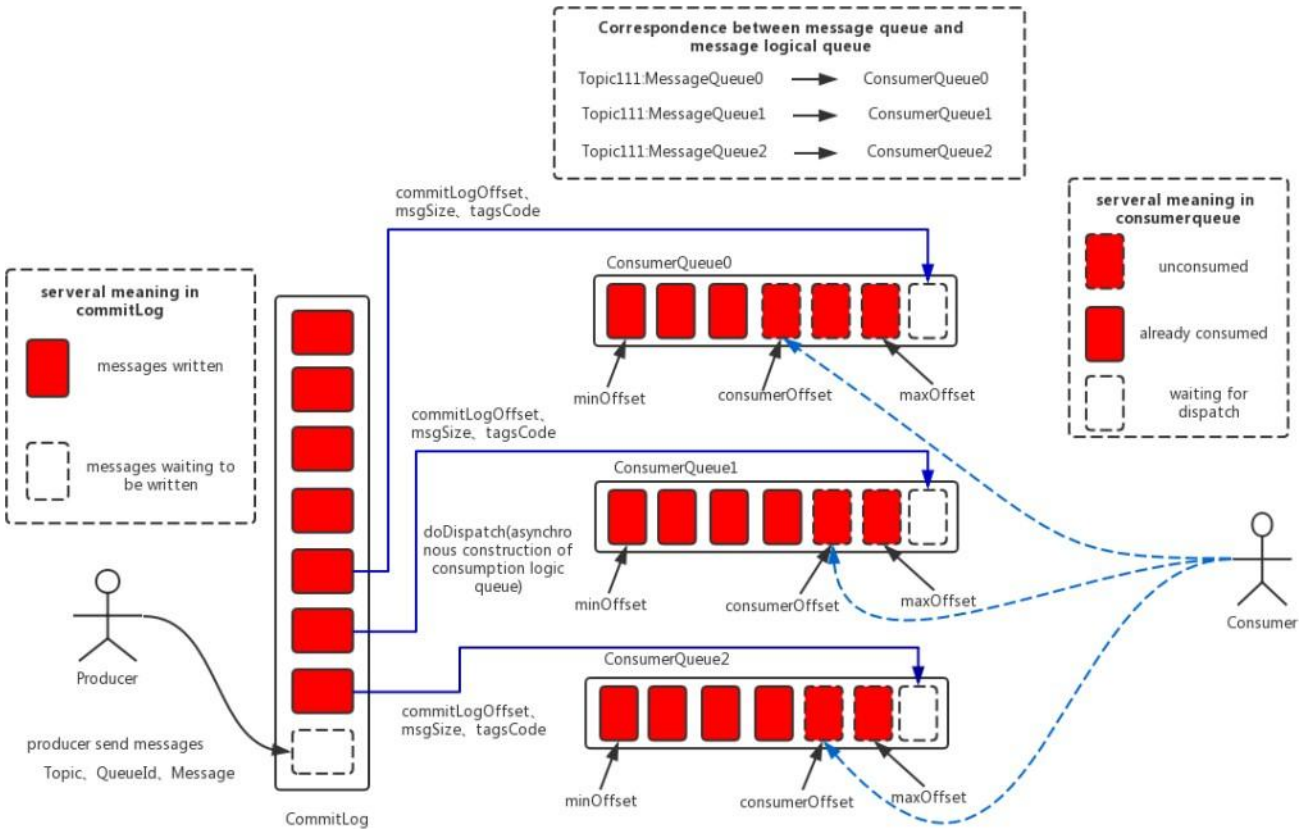
功能主要有两个：

- 1、和Broker 结点保持长连接。
- 2、维护 Topic 的路由信息。

Broker

消息存储和中转角色，负责存储和转发消息。

- Broker 内部维护着一个个 Consumer Queue，用来存储消息的索引，真正存储消息的地方是 CommitLog（日志文件）。




- 单个 Broker 与所有的 Nameserver 保持着长连接和心跳，并会定时将 Topic 信息同步到 NameServer，和 NameServer 的通信底层是通过 Netty 实现的。

Producer

消息生产者，业务端负责发送消息，由用户自行实现和分布式部署。

- 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。

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**广播消费**，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，但其实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。

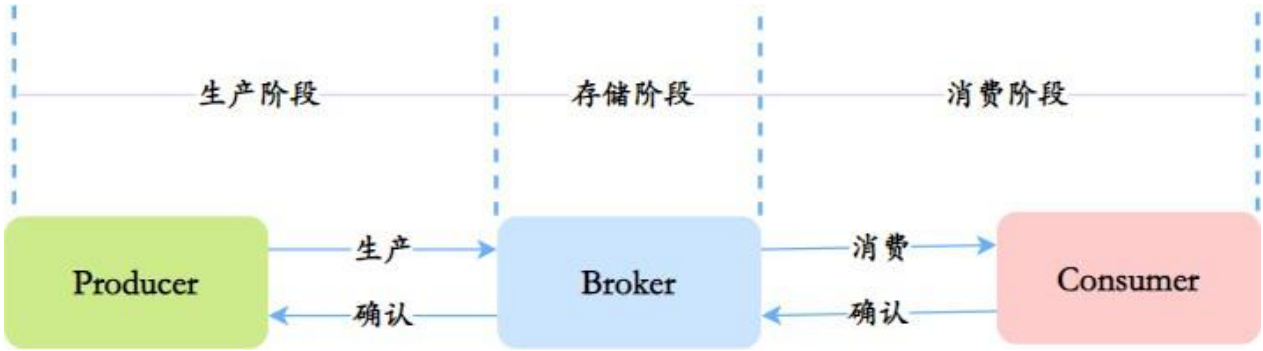
---

# 进阶

## 9.如何保证消息的可用性/可靠性/不丢失呢？

消息可能在哪些阶段丢失呢？可能会在这三个阶段发生丢失：生产阶段、存储阶段、消费阶段。

所以要从这三个阶段考虑：



**生产**

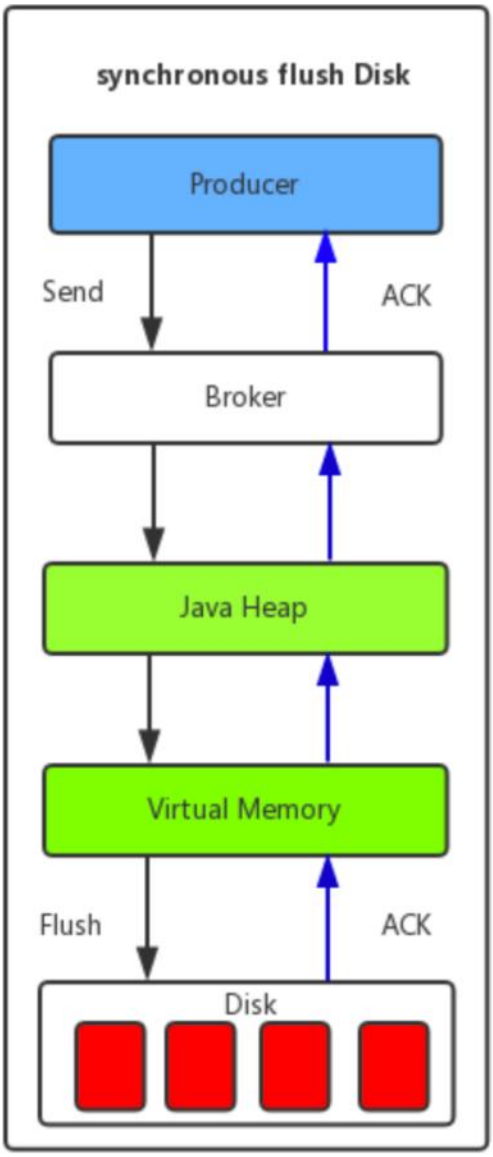
在生产阶段，主要**通过请求确认机制，来保证消息的可靠传递**。

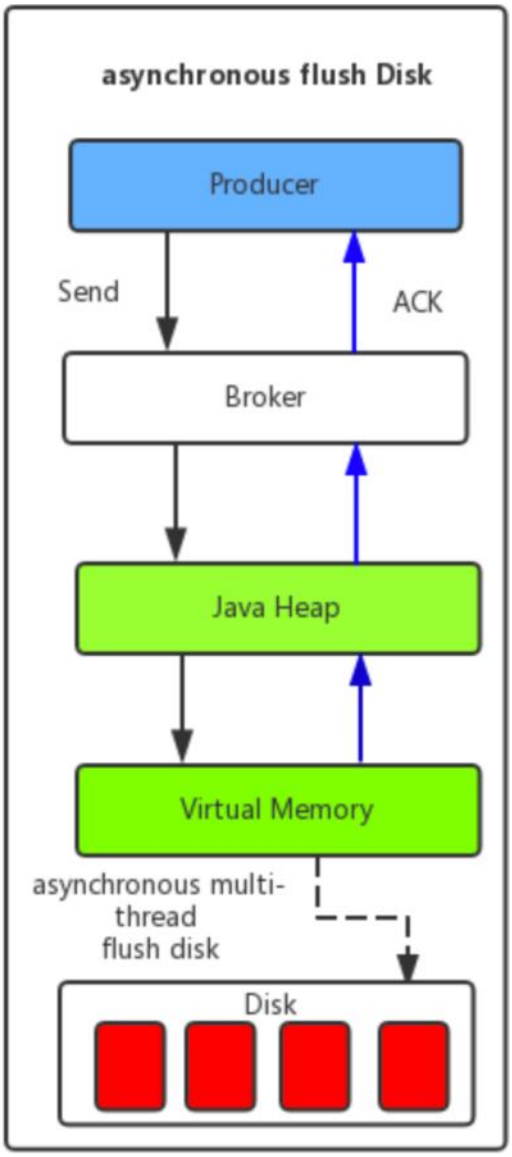
- 1、同步发送的时候，要注意处理响应结果和异常。如果返回响应OK，表示消息成功发送到了Broker，如果响应失败，或者发生其它异常，都应该重试。
- 2、异步发送的时候，应该在回调方法里检查，如果发送失败或者异常，都应该进行重试。
- 3、如果发生超时的情况，也可以通过查询日志的API，来检查是否在Broker存储成功。

**存储**

存储阶段，可以通过**配置可靠性优先的** Broker **参数来避免因为宕机丢消息**，简单说就是可靠性优先的场景都应该使用同步。

- 1、消息只要持久化到CommitLog（日志文件）中，即使Broker宕机，未消费的消息也能重新恢复再消费。
- 2、Broker的刷盘机制：同步刷盘和异步刷盘，不管哪种刷盘都可以保证消息一定存储在pagecache中（内存中），但是同步刷盘更可靠，它是Producer发送消息后等数据持久化到磁盘之后再返回响应给Producer。



- 3、Broker通过主从模式来保证高可用，Broker支持Master和Slave同步复制、Master和Slave异步复制模式，生产者的消息都是发送给Master，但是消费既可以从Master消费，也可以从Slave消费。同步复制模式可以保证即使Master宕机，消息肯定在Slave中有备份，保证了消息不会丢失。

**消费**

从Consumer角度分析，如何保证消息被成功消费？

- Consumer保证消息成功消费的关键在于确认的时机，不要在收到消息后就立即发送消费确认，而是应该在执行完所有消费业务逻辑之后，再发送消费确认。因为消息队列维护了消费的位置，逻辑执行失败了，没有确认，再去队列拉取消息，就还是之前的一条。

# 10. 如何处理消息重复的问题呢？

对分布式消息队列来说，同时做到确保一定投递和不重复投递是很难的，就是所谓的“有且仅有一次”。RocketMQ择了确保一定投递，保证消息不丢失，但有可能造成消息重复。

处理消息重复问题，主要有业务端自己保证，主要的方式有两种：**业务幂等**和**消息去重**。

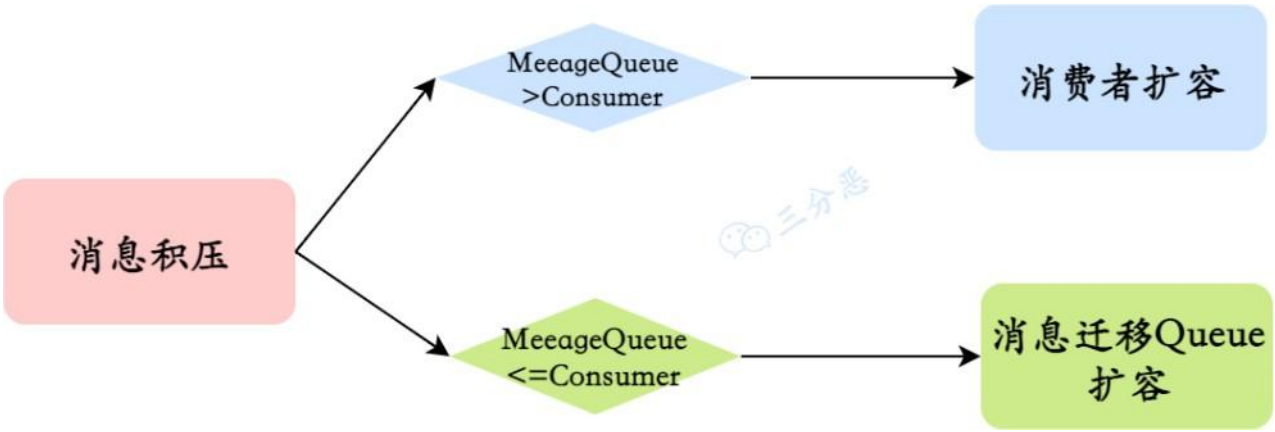

**业务幂等**：第一种是保证消费逻辑的幂等性，也就是多次调用和一次调用的效果是一样的。这样一来，不管消息消费多少次，对业务都没有影响。

**消息去重**：第二种是业务端，对重复的消息就不再消费了。这种方法，需要保证每条消息都有一个惟一的编号，通常是业务相关的，比如订单号，消费的记录需要落库，而且需要保证和消息确认这一步的原子性。

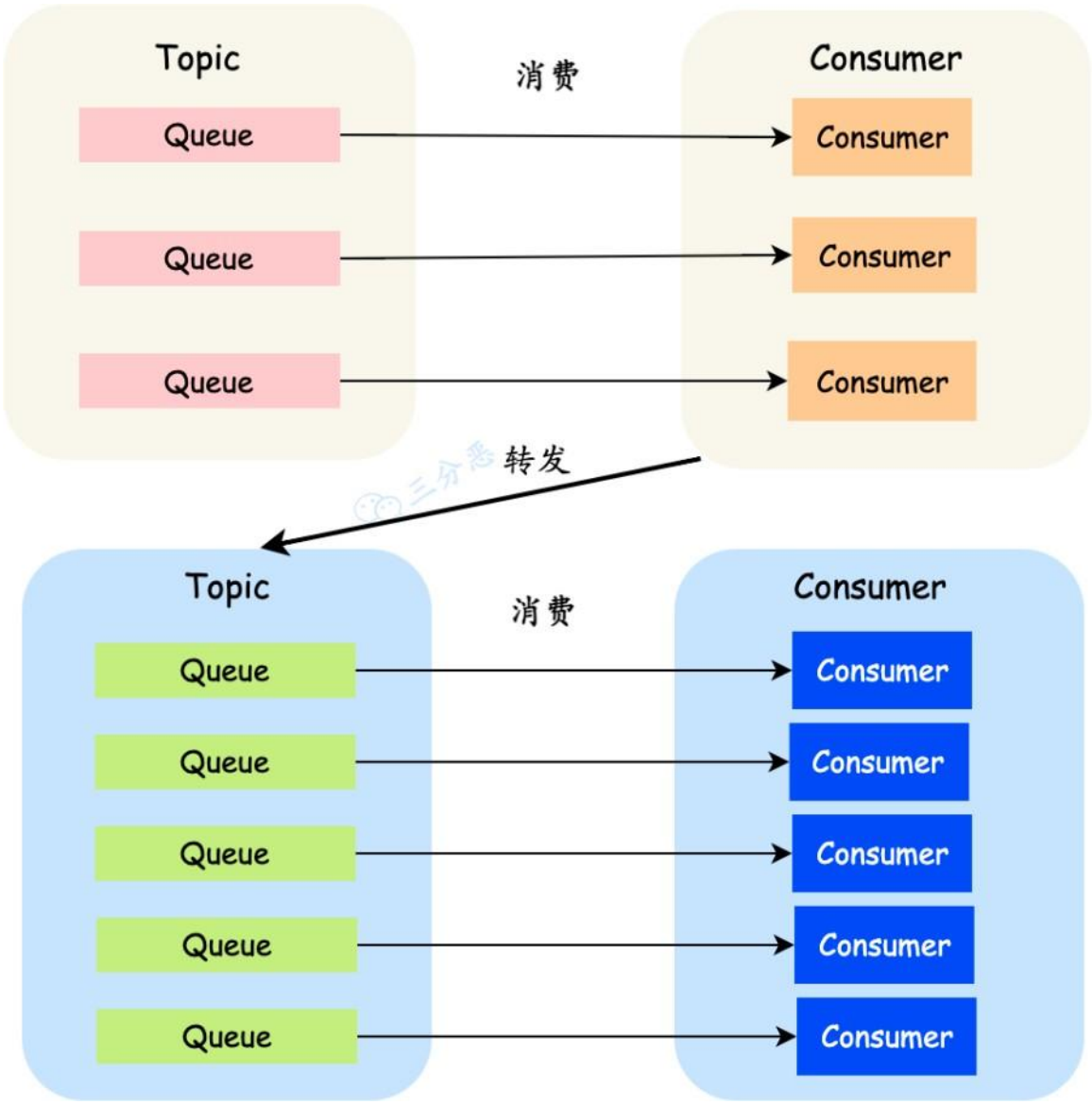
具体做法是可以建立一个消费记录表，拿到这个消息做数据库的insert操作。给这个消息做一个唯一主键（primary key）或者唯一约束，那么就算出现重复消费的情况，就会导致主键冲突，那么就不再处理这条消息。

# 11. 怎么处理消息积压？

发生了消息积压，这时候就得想办法赶紧把积压的消息消费完，就得考虑提高消费能力，一般有两种办法：

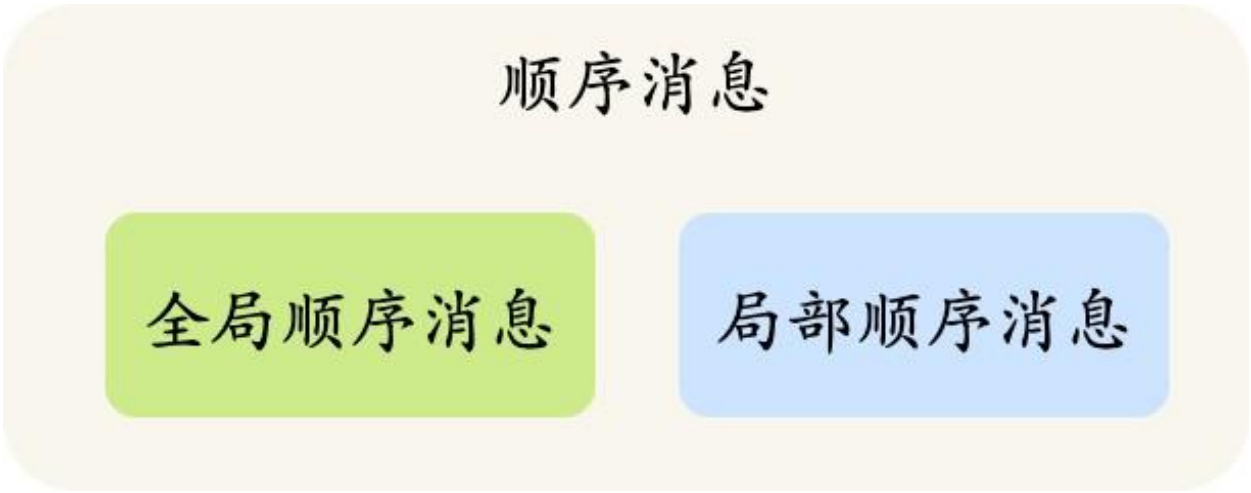



- **消费者扩容**：如果当前Topic的Message Queue的数量大于消费者数量，就可以对消费者进行扩容，增加消费者，来提高消费能力，尽快把积压的消息消费玩。
- **消息迁移Queue扩容**：如果当前Topic的Message Queue的数量小于或者等于消费者数量，这种情况，再扩容消费者就没什么用，就得考虑扩容Message Queue。可以新建一个临时的Topic，临时的Topic多设置一些Message Queue，然后先用一些消费者把消费的数据丢到临时的Topic，因为不用业务处理，只是转发一下消息，还是很快的。接下来用扩容的消费者去消费新的Topic里的数据，消费完了之后，恢复原状。




## 12. 顺序消息如何实现？

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

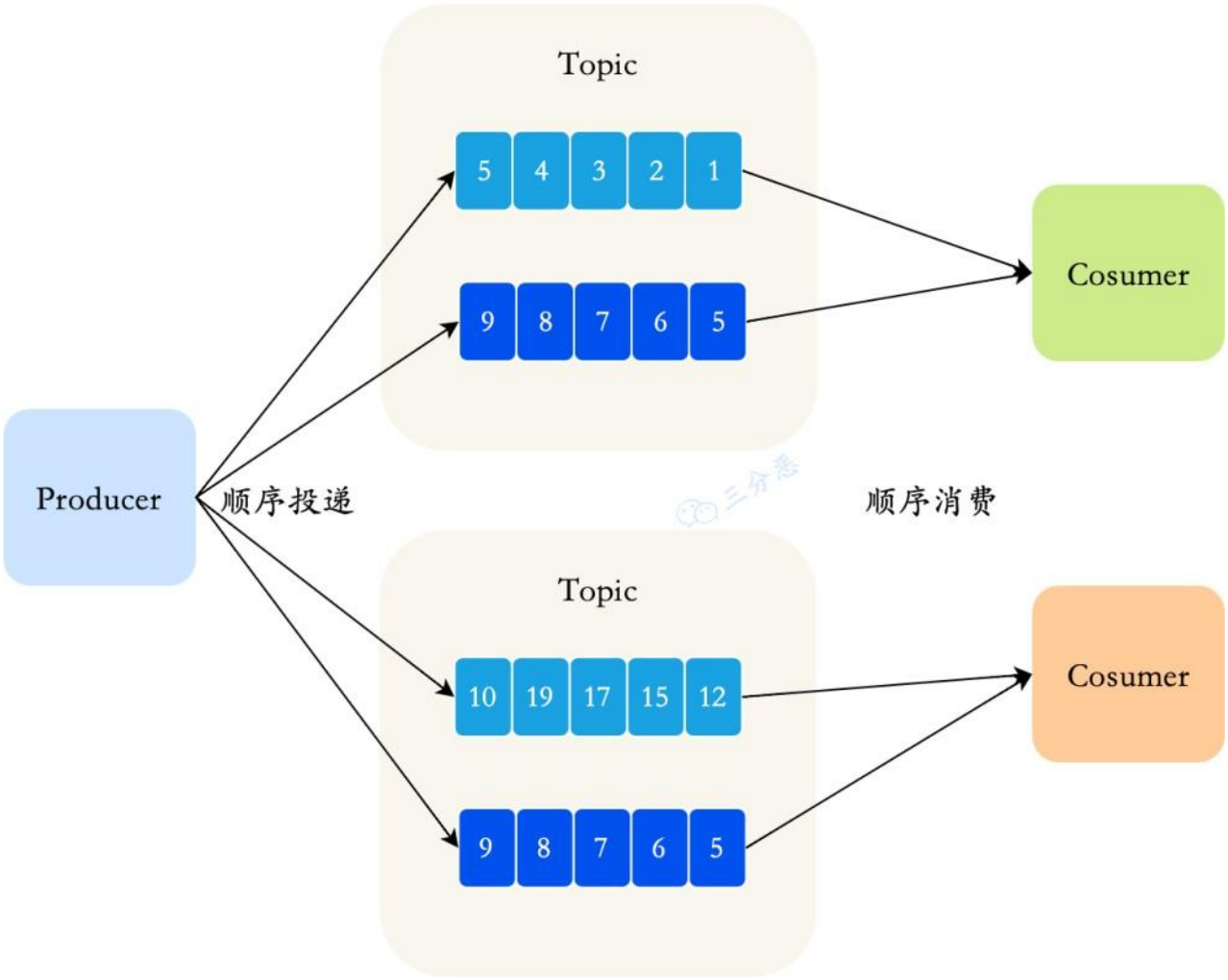



顺序消息分为全局顺序消息和部分顺序消息，全局顺序消息指某个 Topic 下的所有消息都要保证顺序；

部分顺序消息只要保证每一组消息被顺序消费即可，比如订单消息，只要保证同一个订单 ID 个消息能按顺序消费即可。

**部分顺序消息**

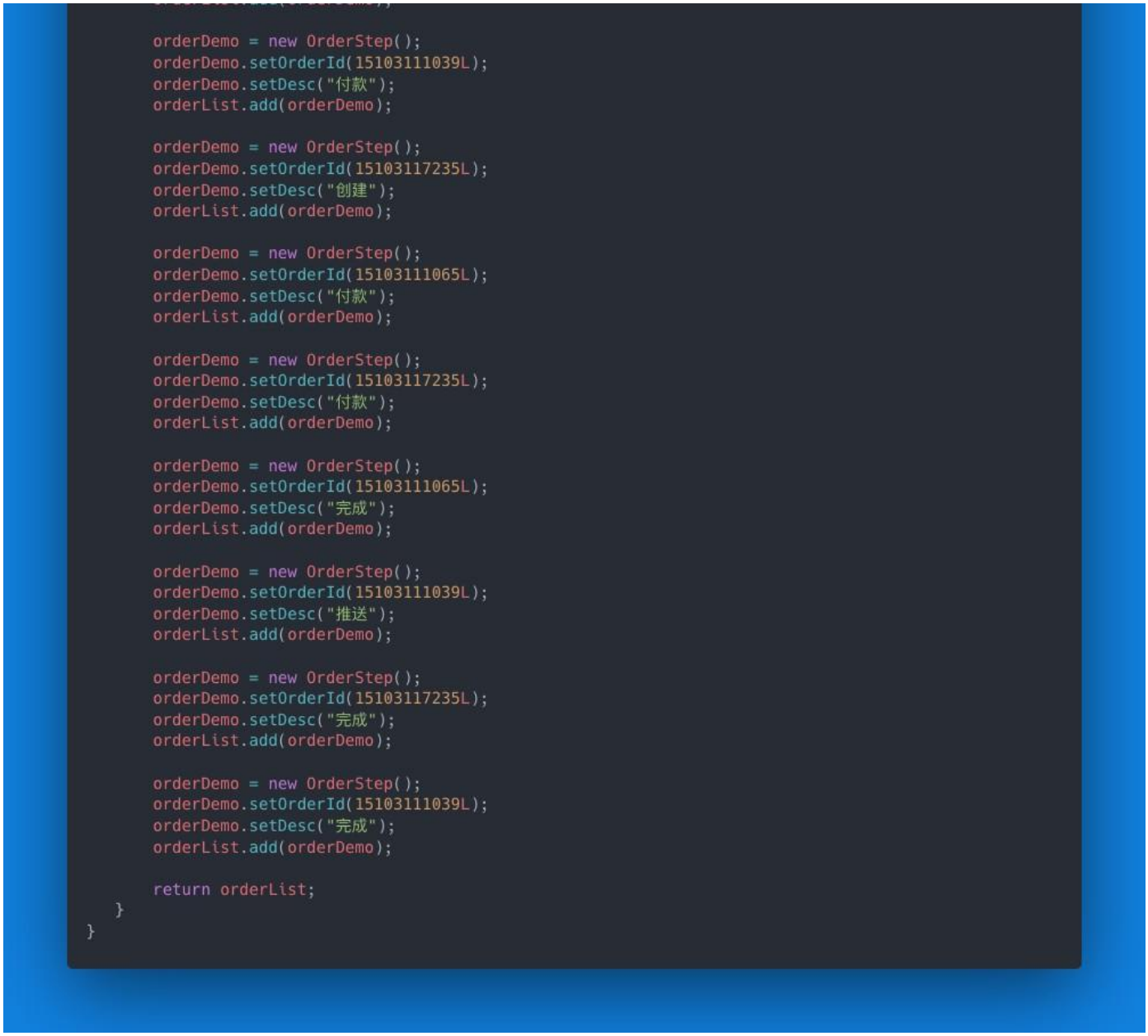
部分顺序消息相对比较好实现，生产端需要做到把同 ID 的消息发送到同一个 Message Queue ；在消费过程中，要做到从同一个Message Queue读取的消息顺序处理——消费端不能并发处理顺序消息，这样才能达到部分有 序。




发送端使用 MessageQueueSelector 类来控制 把消息发往哪个 Message Queue 。

```
package org.apache.rocketmq.example.order2;

import org.apache.rocketmq.client.producer.DefaultMQProducer;
import org.apache.rocketmq.client.producer.MessageQueueSelector;
import org.apache.rocketmq.client.producer.SendResult;
import org.apache.rocketmq.common.message.Message;
import org.apache.rocketmq.common.message.MessageQueue;

import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

/**
* Producer, 发送顺序消息
*/
public class Producer {

    public static void main(String[] args) throws Exception {
        DefaultMQProducer producer = new DefaultMQProducer("please_rename_unique_group_name");

        producer.setNamesrvAddr("127.0.0.1:9876");

        producer.start();
```

```
        String[] tags = new String[]{"TagA", "TagC", "TagD"};

        // 订单列表
        List<OrderStep> orderList = new Producer().buildOrders();

        Date date = new Date();
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        String dateStr = sdf.format(date);
        for (int i = 0; i < 10; i++) {
            // 加个时间前缀
            String body = dateStr + " Hello RocketMQ " + orderList.get(i);
            Message msg = new Message("TopicTest", tags[i % tags.length], "KEY" + i,
body.getBytes());
            SendResult sendResult = producer.send(msg, new MessageQueueSelector() {
                @Override
                public MessageQueue select(List<MessageQueue> mqs, Message msg, Object arg) {
                    Long id = (Long) arg;  //根据订单id选择发送queue
                    long index = id % mqs.size();
                    return mqs.get((int) index);
                }
            }, orderList.get(i).getOrderId());//订单id

            System.out.println(String.format("SendResult status:%s, queueId:%d, body:%s",
                sendResult.getSendStatus(),
                sendResult.getMessageQueue().getQueueId(),
                body));
        }

        producer.shutdown();
    }

    /**
     * 订单的步骤
     */
    private static class OrderStep {
        private long orderId;
        private String desc;

        public long getOrderId() {
            return orderId;
        }

        public void setOrderId(long orderId) {
            this.orderId = orderId;
        }

        public String getDesc() {
            return desc;
        }

        public void setDesc(String desc) {
            this.desc = desc;
        }

        @Override
        public String toString() {
            return "OrderStep{" +
                "orderId=" + orderId +
                ", desc='" + desc + '\'' +
                '}';
        }
    }

    /**
     * 生成模拟订单数据
     */
    private List<OrderStep> buildOrders() {
        List<OrderStep> orderList = new ArrayList<OrderStep>();

        OrderStep orderDemo = new OrderStep();
        orderDemo.setOrderId(15103111039L);
        orderDemo.setDesc("创建");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(15103111065L);
        orderDemo.setDesc("创建");
        orderList.add(orderDemo);
```



```
        orderDemo = new OrderStep();
        orderDemo.setOrderId(15103111039L);
        orderDemo.setDesc("付款");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(15103117235L);
        orderDemo.setDesc("创建");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(15103111065L);
        orderDemo.setDesc("付款");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(15103117235L);
        orderDemo.setDesc("付款");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(15103111065L);
        orderDemo.setDesc("完成");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(15103111039L);
        orderDemo.setDesc("推送");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(15103117235L);
        orderDemo.setDesc("完成");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(15103111039L);
        orderDemo.setDesc("完成");
        orderList.add(orderDemo);

        return orderList;
    }
}
```

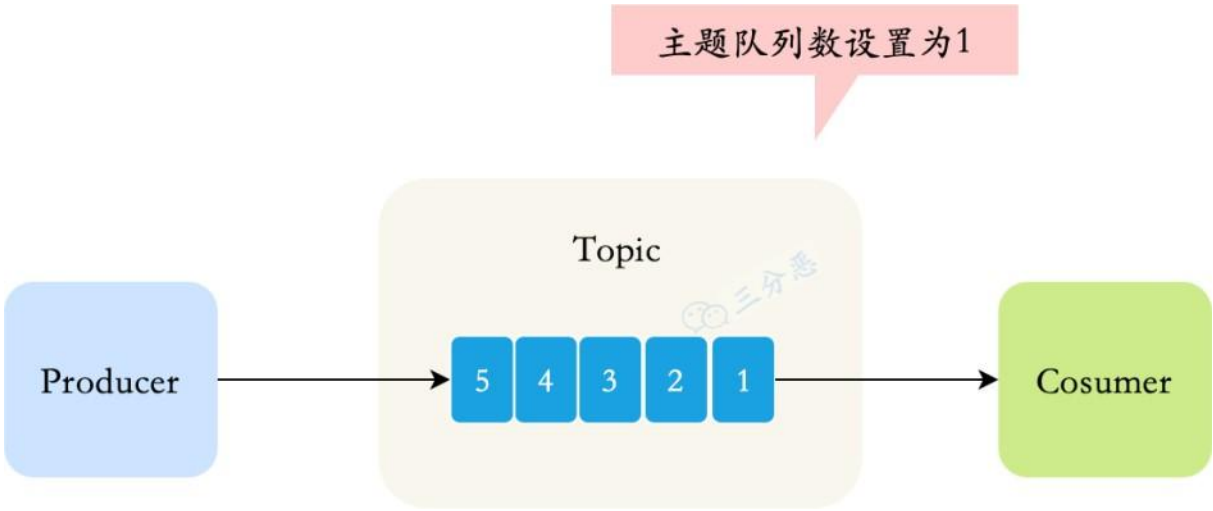
消费端通过使用 MessageListenerOrderly 来解决单 Message Queue 的消息被并发处理的问题。

```java
package org.apache.rocketmq.example.order2;

import org.apache.rocketmq.client.consumer.DefaultMQPushConsumer;
import org.apache.rocketmq.client.consumer.listener.ConsumeOrderlyContext;
import org.apache.rocketmq.client.consumer.listener.ConsumeOrderlyStatus;
import org.apache.rocketmq.client.consumer.listener.MessageListenerOrderly;
import org.apache.rocketmq.common.consumer.ConsumeFromWhere;
import org.apache.rocketmq.common.message.MessageExt;

import java.util.List;
import java.util.Random;
import java.util.concurrent.TimeUnit;

/**
* 顺序消息消费，带事务方式（应用可控制Offset什么时候提交）
*/
public class ConsumerInOrder {

    public static void main(String[] args) throws Exception {
        DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("please_rename_unique_group_name_3");
        consumer.setNamesrvAddr("127.0.0.1:9876");
        /**
         * 设置Consumer第一次启动是从队列头部开始消费还是队列尾部开始消费<br>
         * 如果非第一次启动，那么按照上次消费的位置继续消费
         */
        consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_OFFSET);

        consumer.subscribe("TopicTest", "TagA || TagC || TagD");

        consumer.registerMessageListener(new MessageListenerOrderly() {

            Random random = new Random();

            @Override
            public ConsumeOrderlyStatus consumeMessage(List<MessageExt> msgs, ConsumeOrderlyContext
context) {
                context.setAutoCommit(true);
                for (MessageExt msg : msgs) {
                    // 可以看到每个queue有唯一的consume线程来消费，订单对每个queue(分区)有序
                    System.out.println("consumeThread=" + Thread.currentThread().getName() + "queueId="
+ msg.getQueueId() + ", content:" + new String(msg.getBody()));
                }

                try {
                    //模拟业务逻辑处理中...
                    TimeUnit.SECONDS.sleep(random.nextInt(10));
                } catch (Exception e) {
                    e.printStackTrace();
                }
                return ConsumeOrderlyStatus.SUCCESS;
            }
        });

        consumer.start();

        System.out.println("Consumer Started.");
    }
}
```

**全局顺序消息**

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

要保证全局顺序消息， 需要先把 Topic 的读写队列数设置为 一，然后Producer Consumer 的并发设置，也要是一。简单来说，为了保证整个 Topic全局消息有序，只能消除所有的并发处理，各部分都设置成单线程处理，这时候就完全牺牲RocketMQ的高并发、高吞吐的特性了。



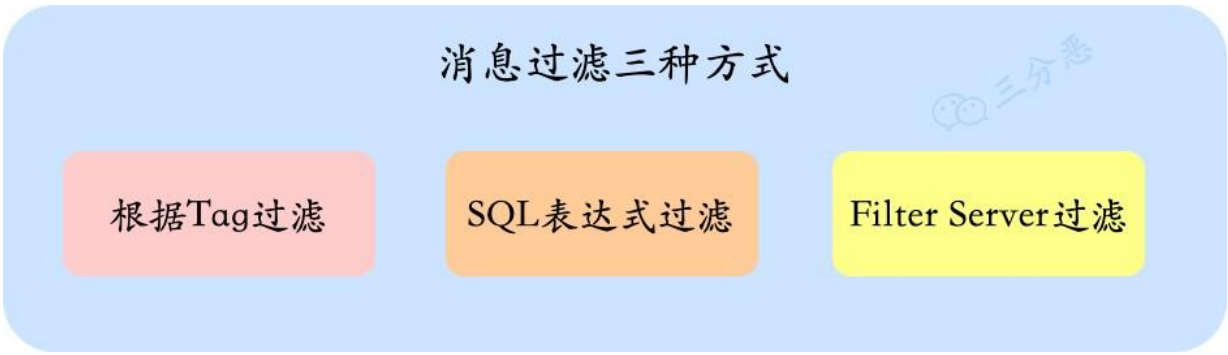

## 13. 如何实现消息过滤？

有两种方案：

- 一种是在 Broker 端按照 Consumer 的去重逻辑进行过滤，这样做的好处是避免了无用的消息传输到Consumer 端，缺点是加重了Broker 的负担，实现起来相对复杂。
- 另一种是在 Consumer 端过滤，比如按照消息设置的 tag 去重，这样的好处是实现起来简单，缺点是有大量无用的消息到达了Consumer 端只能丢弃不处理。

一般采用Cosumer端过滤，如果希望提高吞吐量，可以采用Broker过滤。对

消息的过滤有三种方式：




- 根据Tag过滤：这是最常见的一种，用起来高效简单

```
DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("CID_EXAMPLE");
consumer.subscribe("TOPIC", "TAGA || TAGB || TAGC");
```

- SQL 表达式过滤：SQL表达式过滤更加灵活

```
DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("please_rename_unique_group_
name_4");
// 只有订阅的消息有这 属性a, a >=0 and a <= 3
consumer.subscribe("TopicTest", MessageSelector.bySql("a between 0 and 3");
consumer.registerMessageListener(new MessageListenerConcurrently() {
   @Override
   public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs, ConsumeConcur
rentlyContext context) {
        return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
   }
});
consumer.start();
```

- Filter Server 方式：最灵活，也是最复杂的一种方式，允许用户自定义函数进行过滤

## 14. 延时消息了解吗？

电商的订单超时自动取消，就是一个典型的利用延时消息的例子，用户提交了一个订单，就可以发送一个延时消息，1h后去检查这个订单的状态，如果还是未付款就取消订单释放库存。

RocketMQ是支持延时消息的，只需要在生产消息的时候设置消息的延时级别：

```
// 实例化一个生产者来产生延时消息
DefaultMQProducer producer = new DefaultMQProducer("ExampleProducerGroup");
// 启动生产者
producer.start();
int totalMessagesToSend = 100;
for (int i = 0; i < totalMessagesToSend; i++) {

Message message = new Message("TestTopic", ("Hello scheduled message " + i).getBytes())
;
    // 设置延时等级3,这个消息将在10s之后发送(现在只支持固定的时间,详看delayTimeLevel)
    message.setDelayTimeLevel(3);
    // 发送消息
    producer.send(message);
}
```

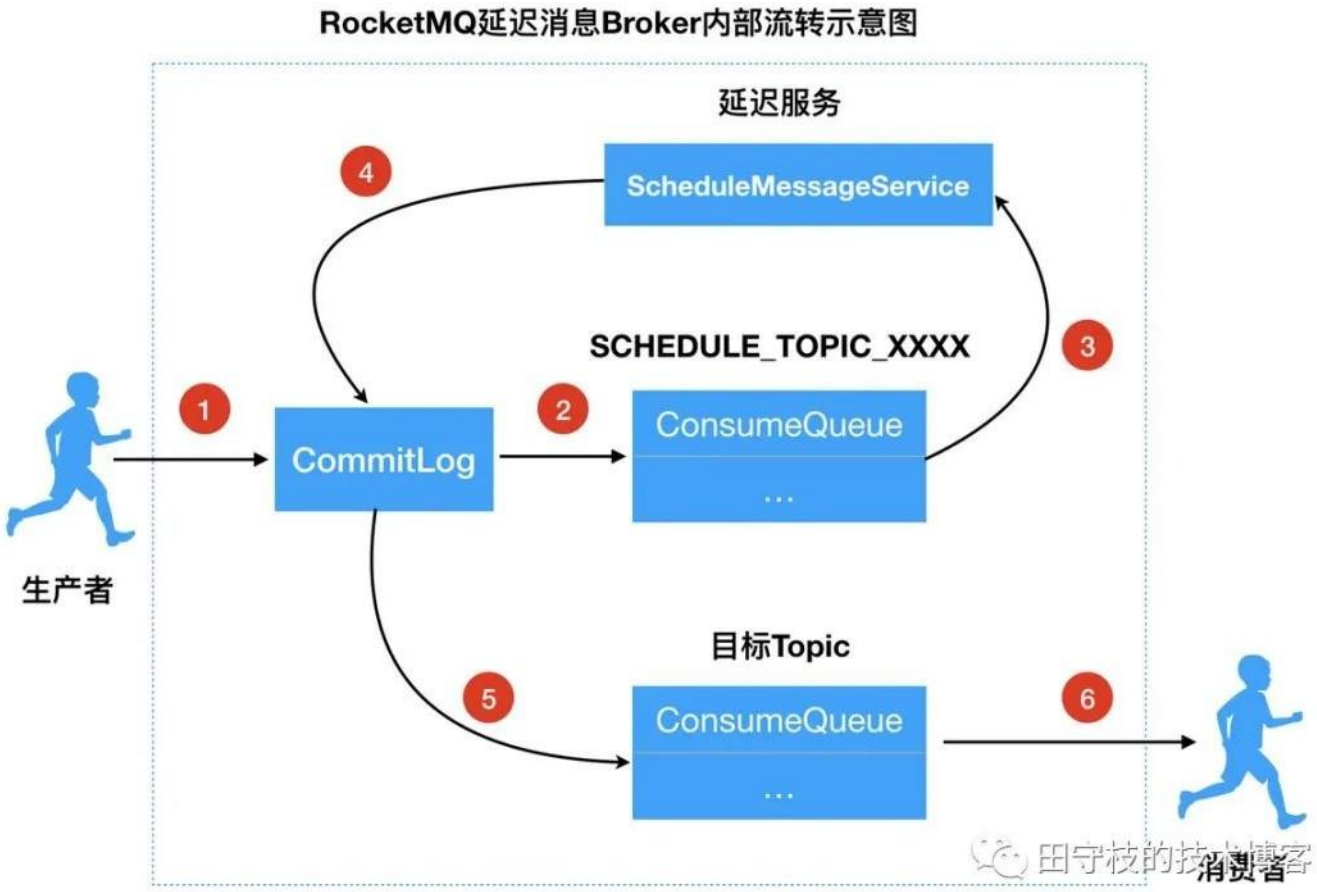
但是目前RocketMQ支持的延时级别是有限的：

```
private String messageDelayLevel = "1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m 30
m 1h 2h";
```

RocketMQ怎么实现延时消息的？

简单，八个字：`临时存储` + `定时任务` 。

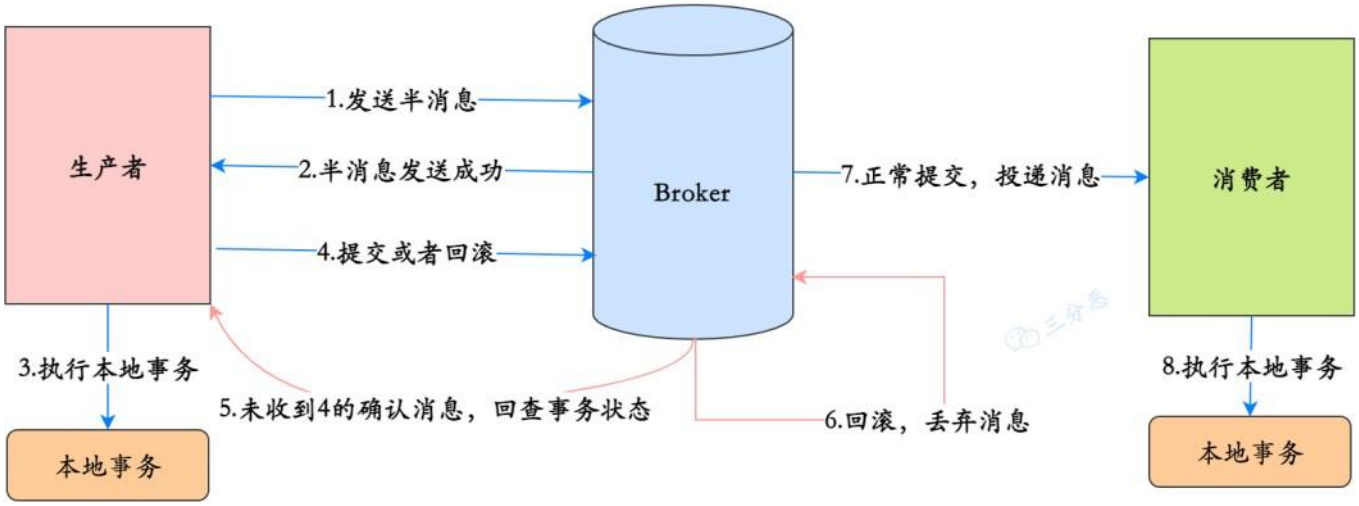
Broker收到延时消息了，会先发送到主题（SCHEDULE_TOPIC_XXXX）的相应时间段的Message Queue中，然后通过一个定时任务轮询这些队列，到期后，把消息投递到目标Topic的队列中，然后消费者就可以正常消费这些消息。




## 15. 怎么实现分布式消息事务的？半消息？

半消息：是指暂时还不能被 Consumer 消费的消息，Producer 成功发送到 Broker 端的消息，但是此消息被标记为“暂不可投递”状态，只有等 Producer 端执行完本地事务后经过二次确认了之后，Consumer 才能消费此条消息。

依赖半消息，可以实现分布式消息事务，其中的关键在于二次确认以及消息回查：




- 1、Producer 向 broker 发送半消息
- 2、Producer 端收到响应，消息发送成功，此时消息是半消息，标记为“不可投递”状态，Consumer 消费不了。
- 3、Producer 端执行本地事务。
- 4、正常情况本地事务执行完成，Producer 向 Broker 发送 Commit/Rollback，如果是 Commit，Broker 端将半消息标记为正常消息，Consumer 可以消费，如果是 Rollback，Broker 丢弃此消息。
- 5、异常情况，Broker 端迟迟等不到二次确认。在一定时间后，会查询所有的半消息，然后到 Producer 端查询半消息的执行情况。
- 6、Producer 端查询本地事务的状态
- 7、根据事务的状态提交 commit/rollback 到 broker 端。（5，6，7 是消息回查）
- 8、消费者段消费到消息之后，执行本地事务，执行本地事务。

## 16.死信队列知道吗？

死信队列用于处理无法被正常消费的消息，即死信消息。

当一条消息初次消费失败，**消息队列**RocketMQ **会自动进行消息重试**；达到最大重试次数后，若消费依然失败，则表明消费者在正常情况下无法正确地消费该消息，此时，消息队列RocketMQ 不会立刻将消息丢弃，而是将其发送到该**消费者对应的特殊队列中**，该特殊队列称为**死信队列**。

**死信消息的特点：**

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

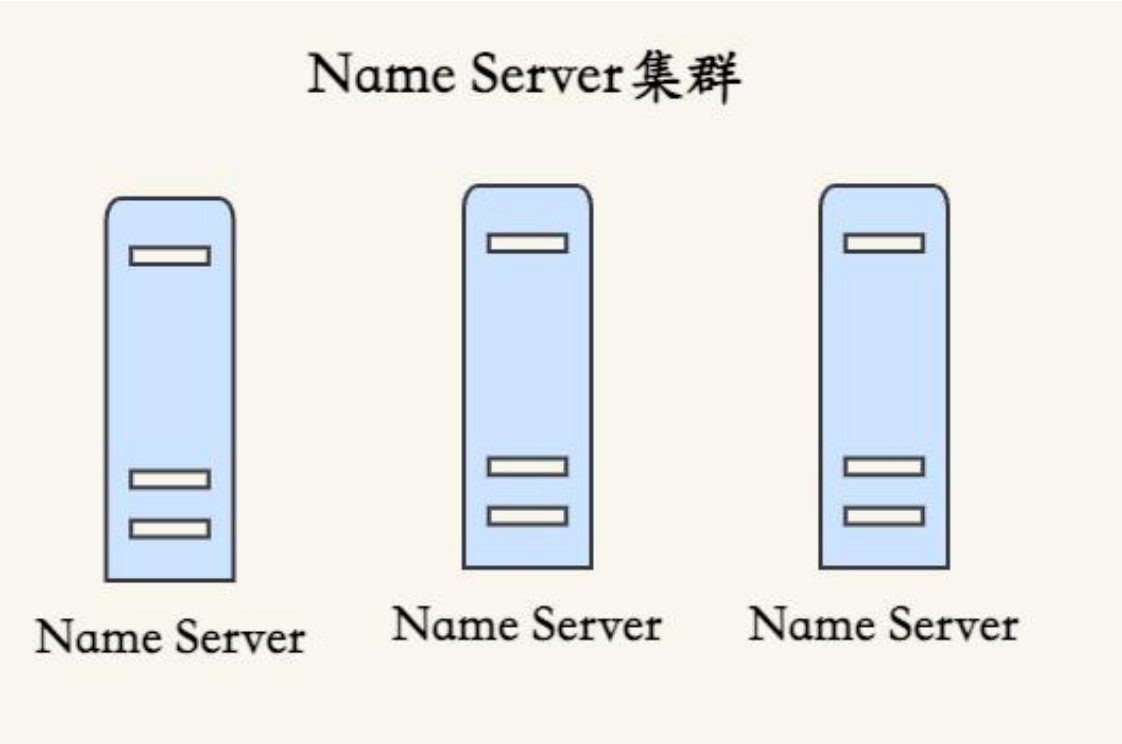
**死信队列的特点：**

- 一个死信队列对应一个 Group ID， 而不是对应单个消费者实例。
- 如果一个 Group ID 未产生死信消息，消息队列RocketMQ 不会为其创建相应的死信队列。一
- 个死信队列包含了对应 Group ID 产生的所有死信消息，不论该消息属于哪个 Topic。

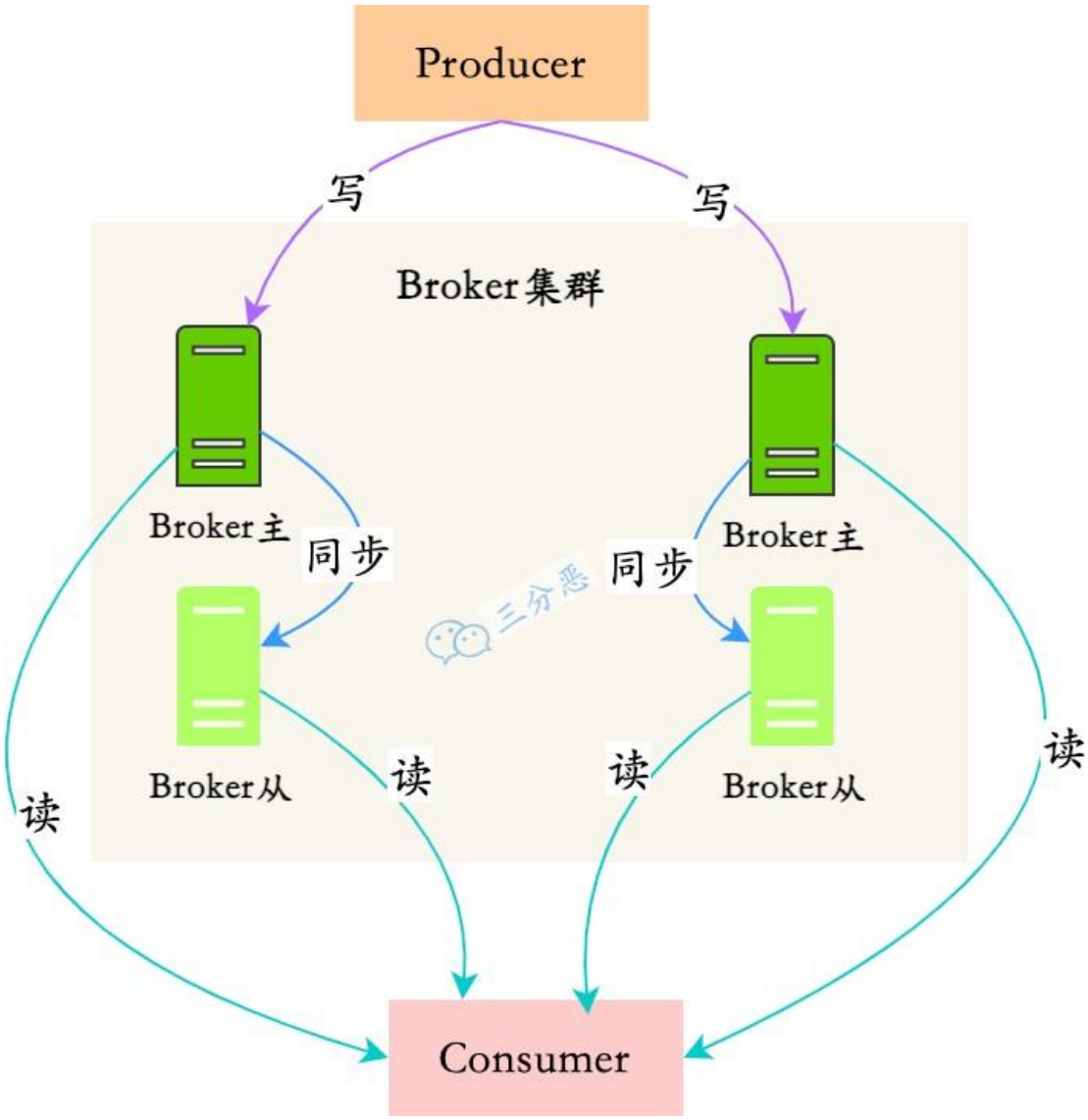
RocketMQ 控制台提供对死信消息的查询、导出和重发的功能。

## 17. 如何保证RocketMQ的高可用？

NameServer因为是无状态，且不相互通信的，所以只要集群部署就可以保证高可用。




RocketMQ的高可用主要是在体现在Broker的读和写的高可用，Broker的高可用是通过 集群 和 主从 实现的。




Broker可以配置两种角色：Master和Slave，Master角色的Broker支持读和写，Slave角色的Broker只支持读， Master会向Slave同步消息。

也就是说Producer只能向Master角色的Broker写入消息，Cosumer可以从Master和Slave角色的Broker读取消息。

Consumer 的配置文件中，并不需要设置是从 Master 读还是从 Slave读，当 Master 不可用或者繁忙的时候，Consumer 的读请求会被自动切换到从 Slave。有了自动切换 Consumer 这种机制，当一个 Master 角色的机器出现故障后，Consumer 仍然可以从 Slave 读取消息，不影响 Consumer 读取消息，这就实现了读的高可用。

如何达到发送端写的高可用性呢？在创建 Topic 的时候，把 Topic 的多个Message Queue 创建在多个 Broker 组上（相同 Broker 名称，不同 brokerId机器组成 Broker 组），这样当 Broker 组的 Master 不可用后，其他组 Master 仍然可用， Producer 仍然可以发送消息 RocketMQ 目前还不支持把Slave自动转成 Master ，如果机器资源不足，需要把 Slave 转成 Master ，则要手动停止 Slave 色的 Broker ，更改配置文件，用新的配置文件启动 Broker。

---

# 原理

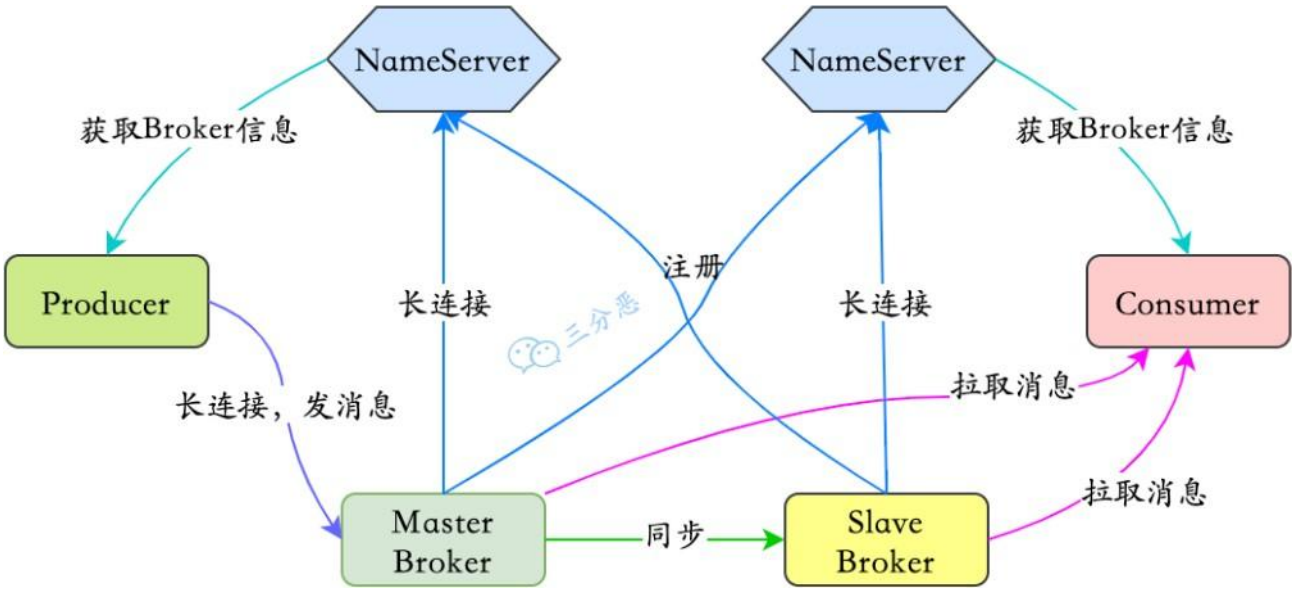
## 18. 说一下RocketMQ的整体工作流程？

简单来说，RocketMQ是一个分布式消息队列，也就是 消息队列 + 分布式系统 。

作为消息队列，它是发-存-收的一个模型，对应的就是Producer、Broker、Cosumer；作为分布式系统，它要有服务端、客户端、注册中心，对应的就是Broker、Producer/Consumer、NameServer

所以我们看一下它主要的工作流程：RocketMQ由NameServer注册中心集群、Producer生产者集群、Consumer消费者集群和若干Broker（RocketMQ进程）组成：

1. Broker在启动的时候去向所有的NameServer注册，并保持长连接，每30s发送一次心跳
2. Producer在发送消息的时候从NameServer获取Broker服务器地址，根据负载均衡算法选择一台服务器来发送消息
3. Conusmer消费消息的时候同样从NameServer获取Broker地址，然后主动拉取消息来消费



## 19. 为什么RocketMQ不使用Zookeeper作为注册中心呢？

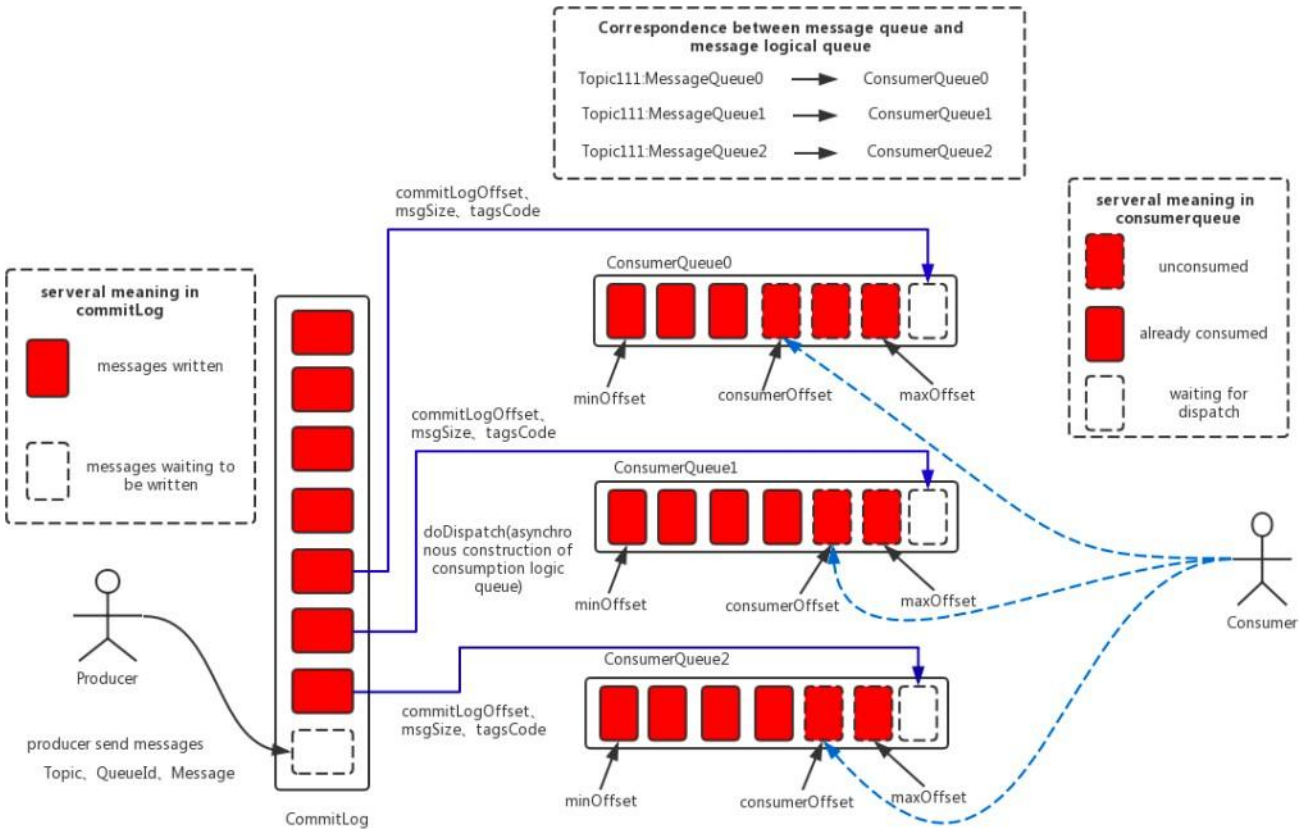
Kafka我们都知道采用Zookeeper作为注册中心——当然也开始逐渐去Zookeeper，RocketMQ不使用Zookeeper
其实主要可能从这几方面来考虑：

1. 基于可用性的考虑，根据CAP理论，同时最多只能满足两个点，而Zookeeper满足的是CP，也就是说 Zookeeper 并不能保证服务的可用性，Zookeeper在进行选举的时候，整个选举的时间太长，期间整个集群都处于不可用的状态，而这对于一个注册中心来说肯定是不能接受的，作为服务发现来说就应该是为可用性而设计。
2. 基于性能的考虑，NameServer本身的实现非常轻量，而且可以通过增加机器的方式水平扩展，增加集群的抗压能力，而Zookeeper的写是不可扩展的，Zookeeper要解决这个问题只能通过划分领域，划分多个 Zookeeper集群来解决，首先操作起来太复杂，其次这样还是又违反了CAP中的A的设计，导致服务之间是不连通的。
3. 持久化的机制来带的问题，ZooKeeper 的 ZAB 协议对每一个写请求，会在每个 ZooKeeper 节点上保持写一个事务日志，同时再加上定期的将内存数据镜像（Snapshot）到磁盘来保证数据的一致性和持久性，而对于一个简单的服务发现的场景来说，这其实没有太大的必要，这个实现方案太重了。而且本身存储的数据应该是高度定制化的。
4. 消息发送应该弱依赖注册中心，而RocketMQ的设计理念也正是基于此，生产者在第一次发送消息的时候从NameServer获取到Broker地址后缓存到本地，如果NameServer整个集群不可用，短时间内对于生产者和消费者并不会产生太大影响。

## 20. Broker是怎么保存数据的呢？

RocketMQ主要的存储文件包括CommitLog文件、ConsumeQueue文件、Indexfile文件。


消息存储的整体的设计：



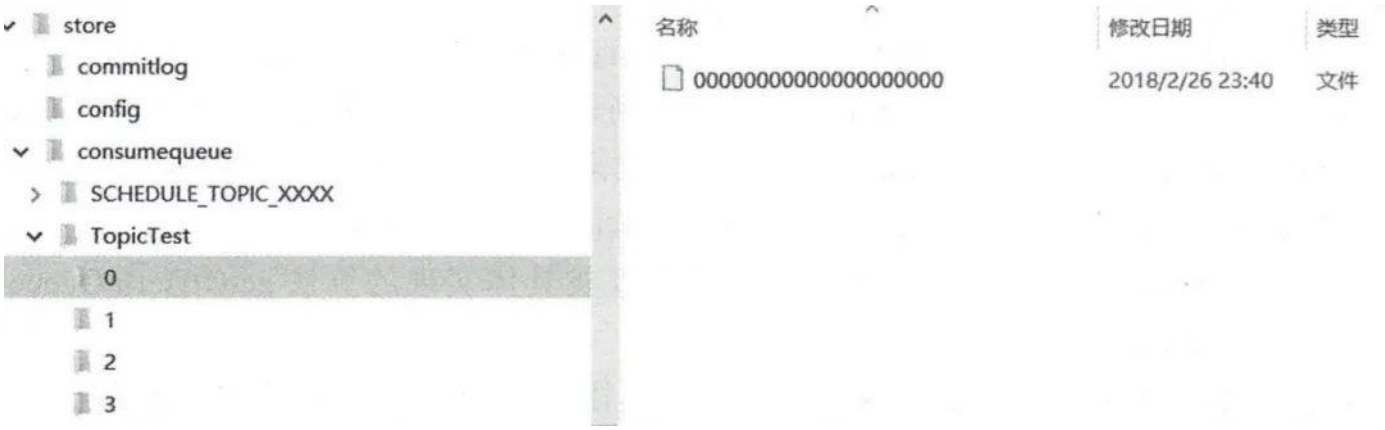

- CommitLog：消息主体以及元数据的存储主体，存储Producer端写入的消息主体内容,消息内容不是定长 的。单个文件大小默认1G, 文件名长度为20位，左边补零，剩余为起始偏移量，比如 00000000000000000000代表了第一个文件，起始偏移量为0，文件大小为1G=1073741824；当第一个文件写满了，第二个文件为00000000001073741824，起始偏移量为1073741824，以此类推。消息主要是顺序写入日志文件，当文件满了，写入下一个文件。

CommitLog文件保存于${Rocket_Home}/store/commitlog目录中，从图中我们可以明显看出来文件名的偏移量，每个文件默认1G，写满后自动生成一个新的文件。

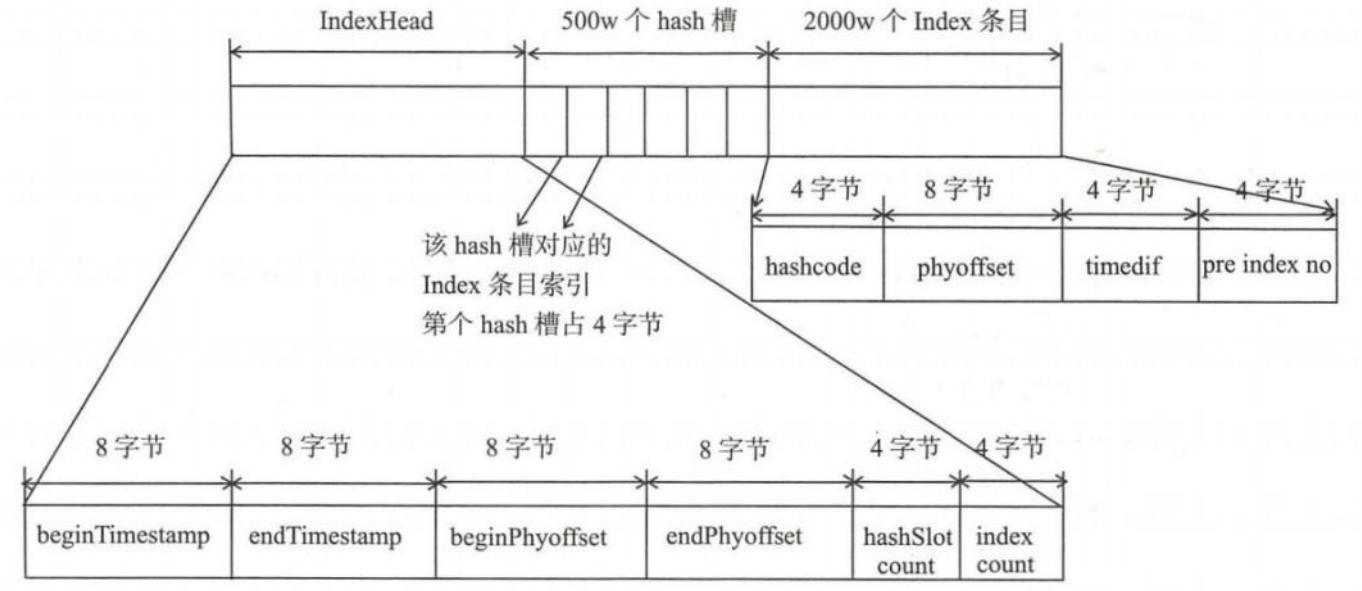

- ConsumeQueue：消息消费队列，引入的目的主要是提高消息消费的性能，由于RocketMQ是基于主题 topic的订阅模式，消息消费是针对主题进行的，如果要遍历commitlog文件中根据topic检索消息是非常低效的。

Consumer即可根据ConsumeQueue来查找待消费的消息。其中，ConsumeQueue（逻辑消费队列）作为消费消息的索引，保存了指定Topic下的队列消息在CommitLog中的起始物理偏移量offset，消息大小size和消息Tag的 HashCode值。

ConsumeQueue文件可以看成是基于Topic的CommitLog索引文件，故ConsumeQueue文件夹的组织方式如下：topic/queue/file三层组织结构，具体存储路径为：$HOME/store/consumequeue/{topic}/{queueId}/{fileName}。同样ConsumeQueue文件采取定长设计，每一个条目共20个字节，分别为8字节的CommitLog物理偏移量、4字节的消息长度、8字节tag hashcode，单个文件由30W个条目组成，可以像数组一样随机访问每一个条目，每个ConsumeQueue文件大小约5.72M；




- IndexFile：IndexFile（索引文件）提供了一种可以通过key或时间区间来查询消息的方法。Index文件的存储位置是：{fileName}，文件名fileName是以创建时的时间戳命名的，固定的单个IndexFile文件大小约为 400M，一个IndexFile可以保存 2000W个索引，IndexFile的底层存储设计为在文件系统中实现HashMap结构，故RocketMQ的索引文件其底层实现为hash索引。




总结一下：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。

RocketMQ的混合型存储结构(多个Topic的消息实体内容都存储于一个CommitLog中)针对Producer和Consumer分别采用了数据和索引部分相分离的存储结构，Producer发送消息至Broker端，然后Broker端使用同步或者异步的方式对消息刷盘持久化，保存至CommitLog中。

只要消息被刷盘持久化至磁盘文件CommitLog中，那么Producer发送的消息就不会丢失。正因为如此， Consumer也就肯定有机会去消费这条消息。当无法拉取到消息后，可以等下一次消息拉取，同时服务端也支持长轮询模式，如果一个消息拉取请求未拉取到消息，Broker允许等待30s的时间，只要这段时间内有新消息到达，将直接返回给消费端。

这里，RocketMQ的具体做法是，使用Broker端的后台服务线程—ReputMessageService不停地分发请求并异步构建ConsumeQueue（逻辑消费队列）和IndexFile（索引文件）数据。


## 21. 说说RocketMQ怎么对文件进行读写的？

RocketMQ对文件的读写巧妙地利用了操作系统的一些高效文件读写方式—— PageCache、 顺序读写 、 零拷贝 。

- PageCache、顺序读取

在RocketMQ中，ConsumeQueue逻辑消费队列存储的数据较少，并且是顺序读取，在page cache机制的预读取作用下，Consume Queue文件的读性能几乎接近读内存，即使在有消息堆积情况下也不会影响性能。而对于 CommitLog消息存储的日志数据文件来说，读取消息内容时候会产生较多的随机访问读取，严重影响性能。如果选择合适的系统IO调度算法，比如设置调度算法为“Deadline”（此时块存储采用SSD的话），随机读的性能也会有所提升。

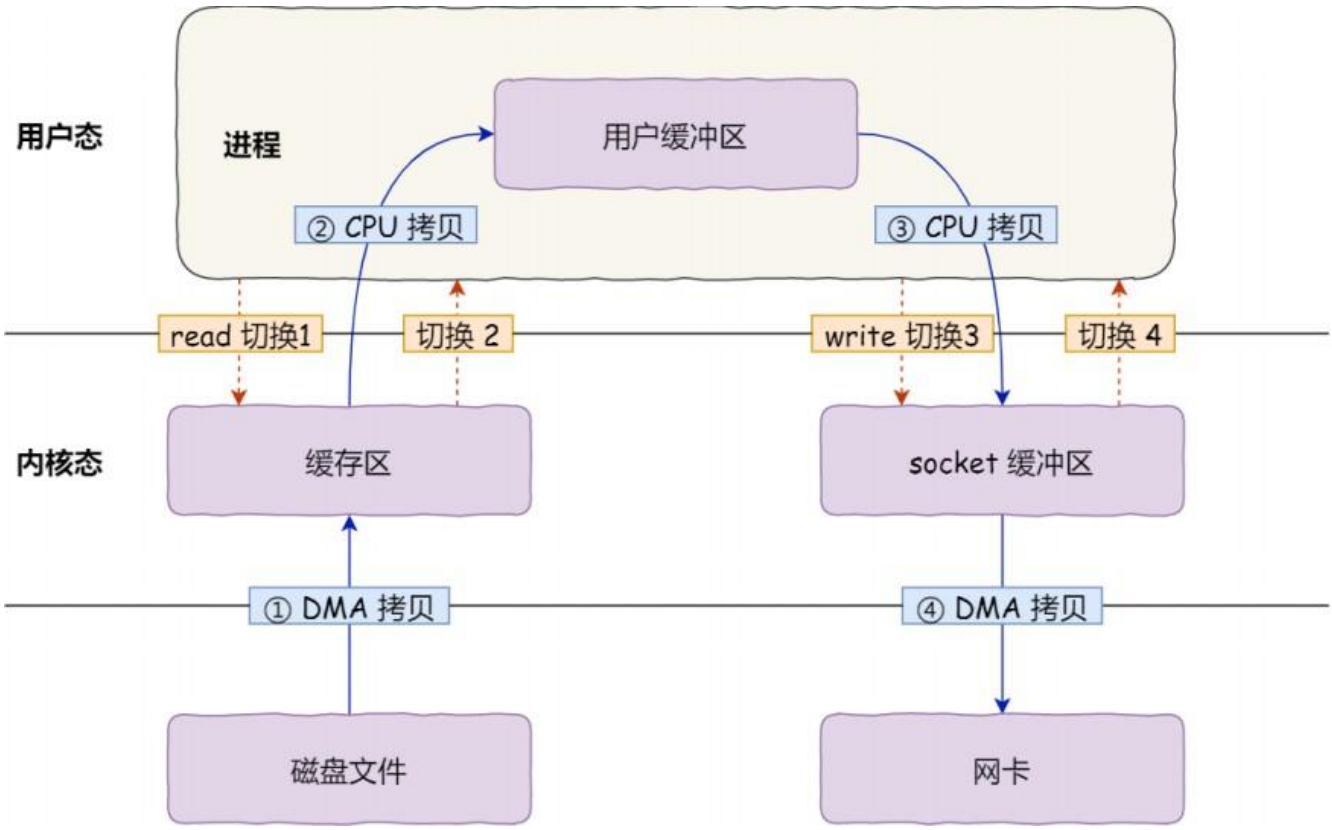
页缓存（PageCache)是OS对文件的缓存，用于加速对文件的读写。一般来说，程序对文件进行顺序读写的速度几乎接近于内存的读写速度，主要原因就是由于OS使用PageCache机制对读写访问操作进行了性能优化，将一部分的内存用作PageCache。对于数据的写入，OS会先写入至Cache内，随后通过异步的方式由pdflush内核线程将 Cache内的数据刷盘至物理磁盘上。对于数据的读取，如果一次读取文件时出现未命中PageCache的情况，OS从物理磁盘上访问读取文件的同时，会顺序对其他相邻块的数据文件进行预读取。

- 零拷贝

另外，RocketMQ主要通过MappedByteBuffer对文件进行读写操作。其中，利用了NIO中的FileChannel模型将磁盘上的物理文件直接映射到用户态的内存地址中（这种Mmap的方式减少了传统IO，将磁盘文件数据在操作系统内核地址空间的缓冲区，和用户应用程序地址空间的缓冲区之间来回进行拷贝的性能开销），将对文件的操作转化为直接对内存地址进行操作，从而极大地提高了文件的读写效率（正因为需要使用内存映射机制，故RocketMQ的文件存储都使用定长结构来存储，方便一次将整个文件映射至内存）。

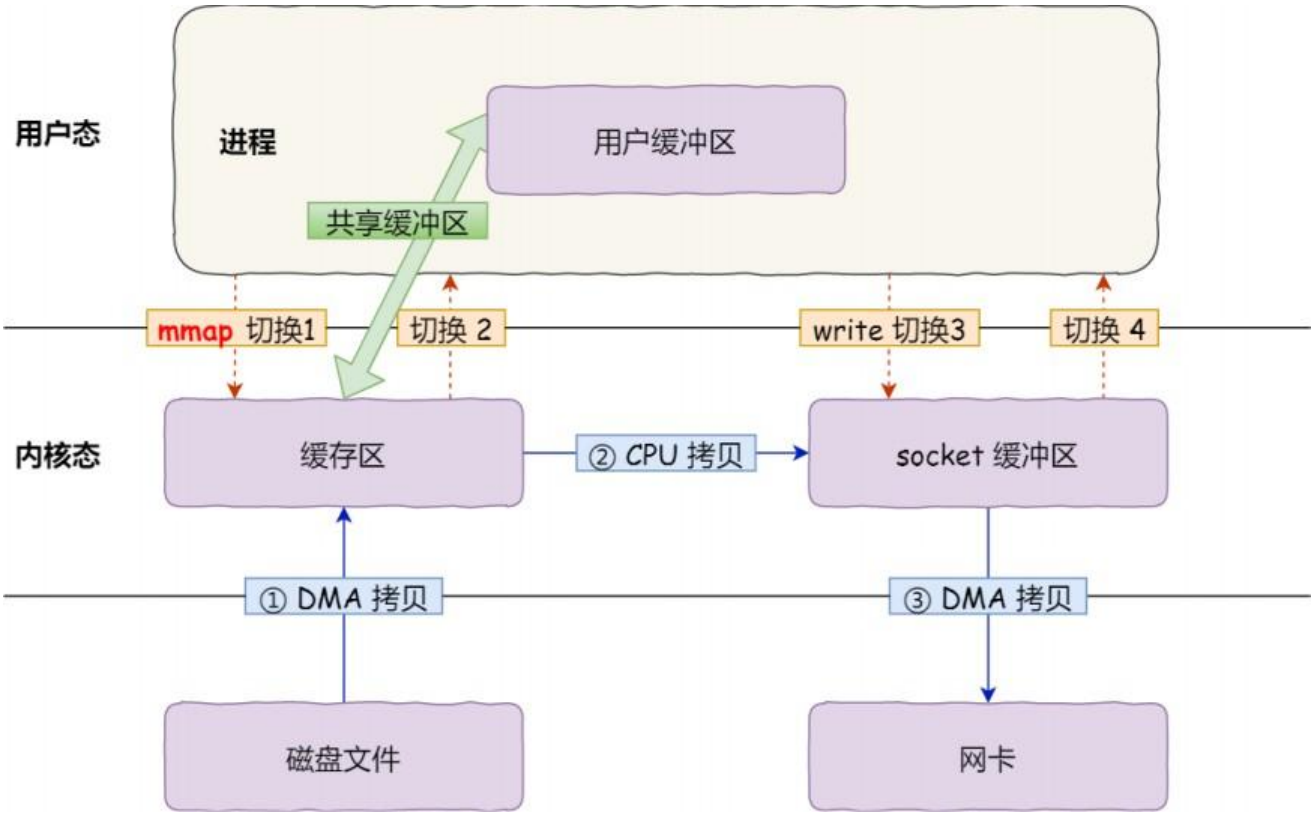
**说说什么是零拷贝?**

在操作系统中，使用传统的方式，数据需要经历几次拷贝，还要经历用户态/内核态切换。




1. 从磁盘复制数据到内核态内存；
2. 从内核态内存复制到用户态内存；
3. 然后从用户态内存复制到网络驱动的内核态内存；
4. 最后是从网络驱动的内核态内存复制到网卡中进行传输。

所以，可以通过零拷贝的方式，**减少用户态与内核态的上下文切换**和**内存拷贝的次数**，用来提升I/O的性能。零拷贝比较常见的实现方式是mmap，这种机制在Java中是通过MappedByteBuffer实现的。




## 22. 消息刷盘怎么实现的呢？

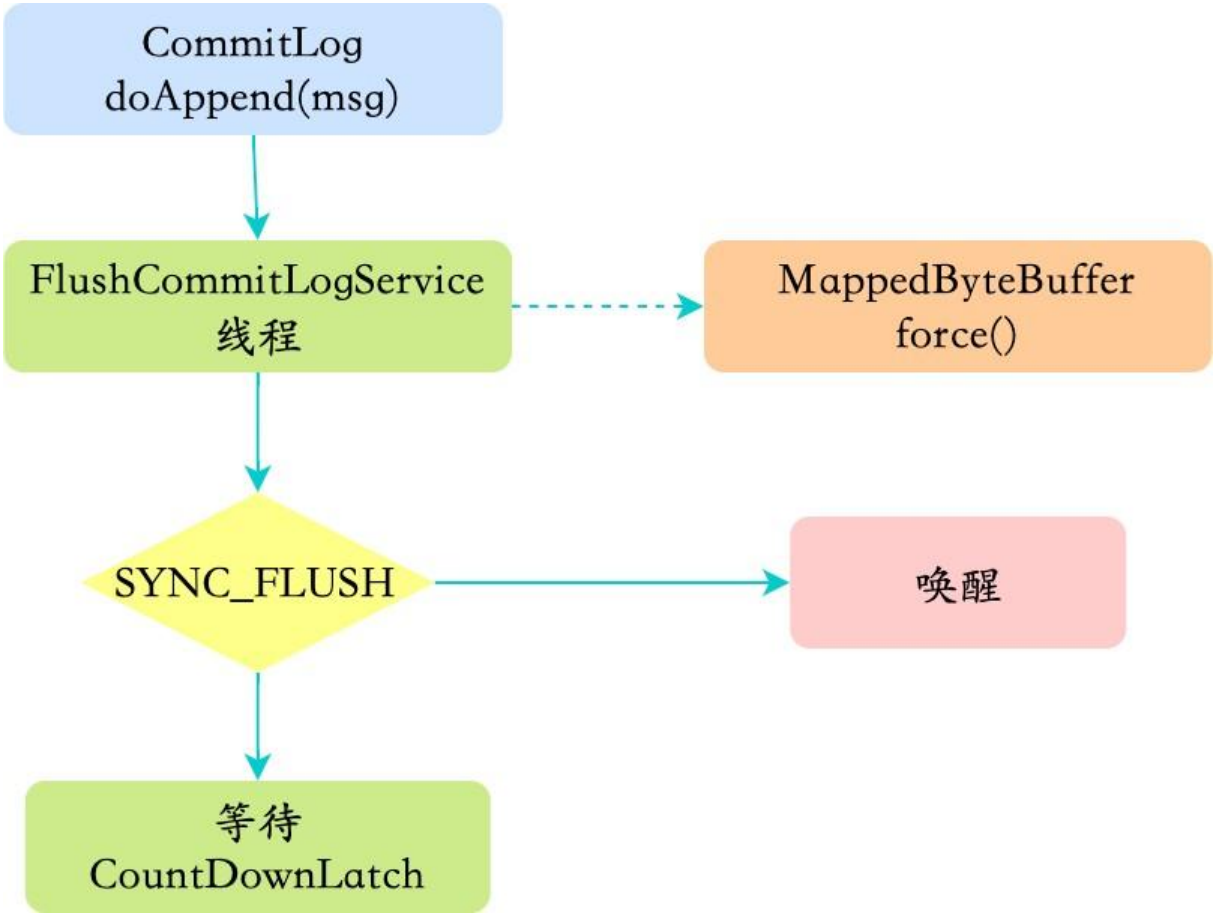
RocketMQ提供了两种刷盘策略：同步刷盘和异步刷盘

- 同步刷盘：在消息达到Broker的内存之后，必须刷到commitLog日志文件中才算成功，然后返回Producer数据已经发送成功。
- 异步刷盘：异步刷盘是指消息达到Broker内存后就返回Producer数据已经发送成功，会唤醒一个线程去将数据持久化到CommitLog日志文件中。

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。




## 22. 能说下 RocketMQ 的负载均衡是如何实现的？

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

Producer的负载均衡

Producer端在发送消息的时候，会先根据Topic找到指定的TopicPublishInfo，在获取了TopicPublishInfo路由信息后，RocketMQ的客户端在默认方式下selectOneMessageQueue()方法会从TopicPublishInfo中的 messageQueueList中选择一个队列（MessageQueue）进行发送消息。具这里有一个sendLatencyFaultEnable开关变量，如果开启，在随机递增取模的基础上，再过滤掉not available的Broker代理。

Producer负载均衡：索引递增随机取模

```java
/**
 * 选择一个MessageQueue
 */
public MessageQueue selectOneMessageQueue() {
    //索引递增
    int index = this.sendWhichQueue.incrementAndGet();
    //利用索引取随机数，取余数
    int pos = Math.abs(index) % this.messageQueueList.size();
    if (pos < 0)
        pos = 0;
    return this.messageQueueList.get(pos);
}
```

所谓的"latencyFaultTolerance"，是指对之前失败的，按一定的时间做退避。例如，如果上次请求的latency超过550Lms，就退避3000Lms；超过1000L，就退避60000L；如果关闭，采用随机递增取模的方式选择一个队列（MessageQueue）来发送消息，latencyFaultTolerance机制是实现消息发送高可用的核心关键所在。

Consumer的负载均衡

在RocketMQ中，Consumer端的两种消费模式（Push/Pull）都是基于拉模式来获取消息的，而在Push模式只是对pull模式的一种封装，其本质实现为消息拉取线程在从服务器拉取到一批消息后，然后提交到消息消费线程池 后，又“马不停蹄”的继续向服务器再次尝试拉取消息。如果未拉取到消息，则延迟一下又继续拉取。在两种基于拉模式的消费方式（Push/Pull）中，均需要Consumer端知道从Broker端的哪一个消息队列中去获取消息。因此，有必要在Consumer端来做负载均衡，即Broker端中多个MessageQueue分配给同一个ConsumerGroup中的哪些 Consumer消费。

1. Consumer端的心跳包发送

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

2. Consumer端实现负载均衡的核心类—RebalanceImpl

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

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

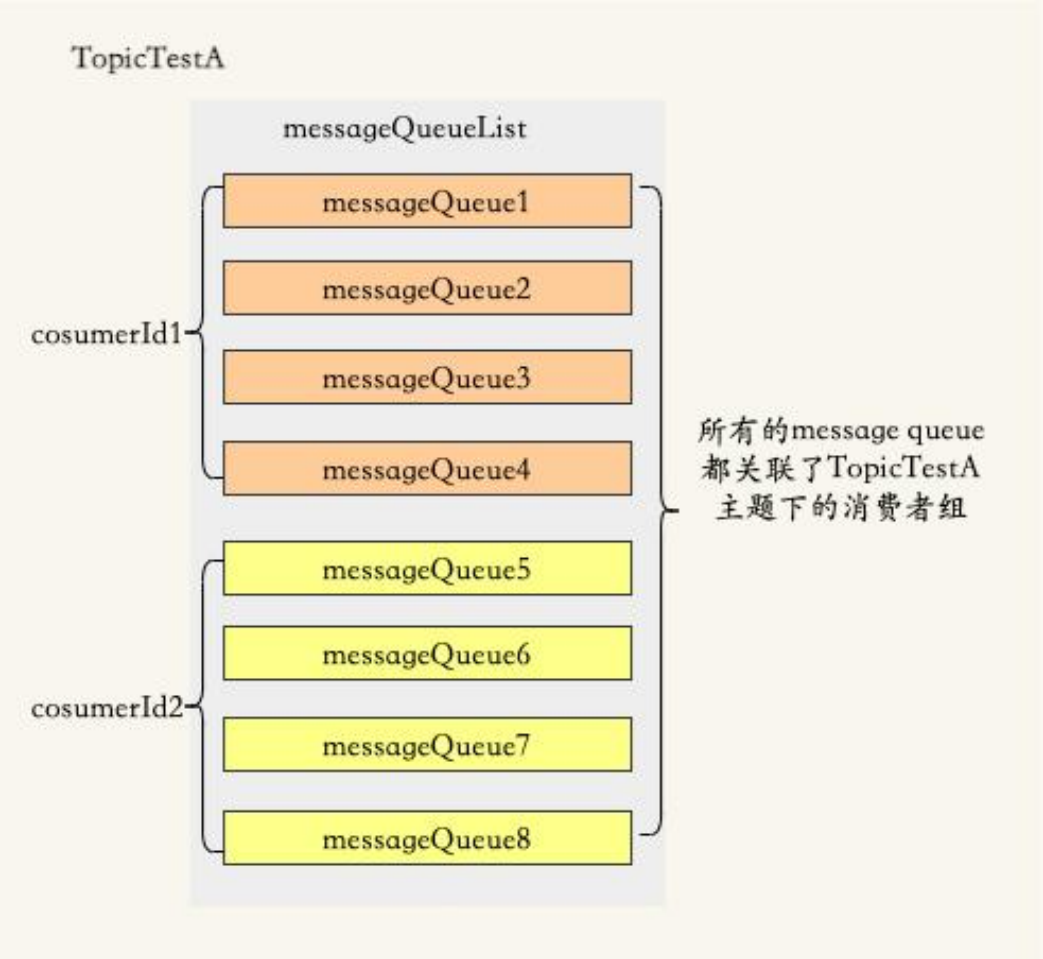
rebalanceByTopic()方法会根据消费者通信类型为“广播模式”还是“集群模式”做不同的逻辑处理。这里主要来看下集群模式下的主要处理流程：

```
            //集群模式负载均衡
            case CLUSTERING: {
                //获取该Topic主题下的消息消费队列集合
                Set<MessageQueue> mqSet = this.topicSubscribeInfoTable.get(topic);
                //找到消费者Id列表
                List<String> cidAll = this.mQClientFactory.findConsumerIdList(topic, consumerGroup);
                if (null == mqSet) {
                    if (!topic.startsWith(MixAll.RETRY_GROUP_TOPIC_PREFIX)) {
                        this.messageQueueChanged(topic, Collections.<MessageQueue>emptySet(),
Collections.<MessageQueue>emptySet());
                        log.warn("doRebalance, {}, but the topic[{}] not exist.", consumerGroup,
topic);
                    }
                }

                if (null == cidAll) {
                    log.warn("doRebalance, {} {}, get consumer id list failed", consumerGroup, topic);
                }

                if (mqSet != null && cidAll != null) {
                    List<MessageQueue> mqAll = new ArrayList<MessageQueue>();
                    mqAll.addAll(mqSet);

                    Collections.sort(mqAll);
                    Collections.sort(cidAll);
                    //消费者消息分配负载均衡策略
                    AllocateMessageQueueStrategy strategy = this.allocateMessageQueueStrategy;
                    //分配队列的结果
                    List<MessageQueue> allocateResult = null;
                    try {
                        //相应的策略类获取负载队列结果
                        allocateResult = strategy.allocate(
                                this.consumerGroup,
                                this.mQClientFactory.getClientId(),
                                mqAll,
                                cidAll);
                    } catch (Throwable e) {
                        log.error("allocate message queue exception. strategy name: {}, ex: {}",
strategy.getName(), e);
                        return false;
                    }
                    //消息队列结果去重
                    Set<MessageQueue> allocateResultSet = new HashSet<MessageQueue>();
                    if (allocateResult != null) {
                        allocateResultSet.addAll(allocateResult);
                    }

                    boolean changed = this.updateProcessQueueTableInRebalance(topic, allocateResultSet,
isOrder);
                    if (changed) {
                        log.info(
                                "client rebalanced result changed. allocateMessageQueueStrategyName={},
group={}, topic={}, clientId={}, mqAllSize={}, cidAllSize={}, rebalanceResultSize={},
rebalanceResultSet={}",
                                strategy.getName(), consumerGroup, topic,
this.mQClientFactory.getClientId(), mqSet.size(), cidAll.size(),
                                allocateResultSet.size(), allocateResultSet);
                        this.messageQueueChanged(topic, mqSet, allocateResultSet);
                    }

                    balanced = allocateResultSet.equals(getWorkingMessageQueue(topic));
                }
                break;
            }
```

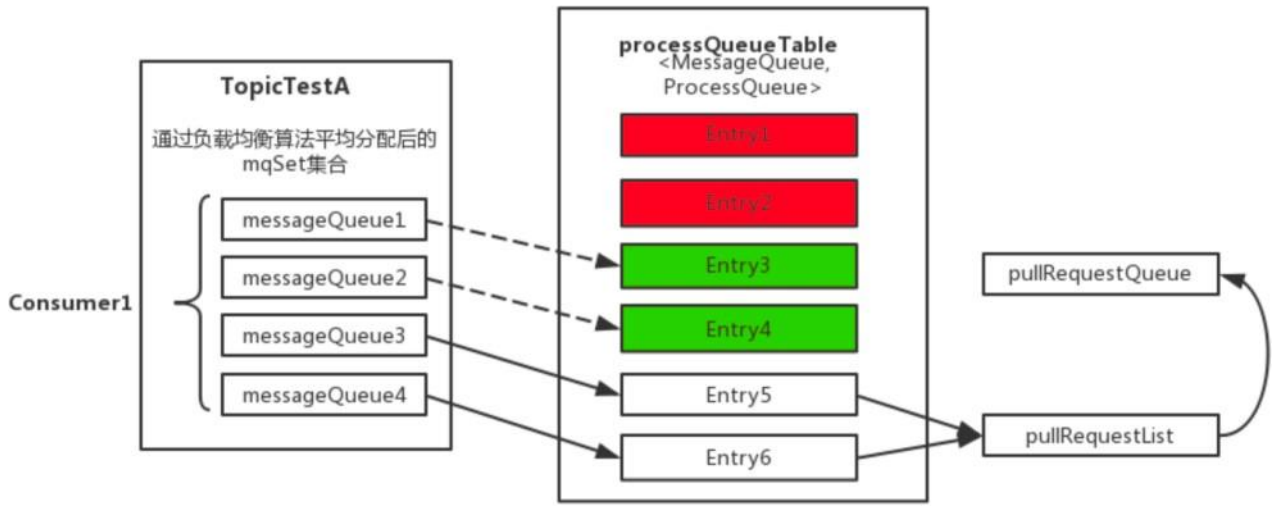
(1) 从rebalanceImpl实例的本地缓存变量—topicSubscribeInfoTable中，获取该Topic主题下的消息消费队列集合（mqSet）；

(2) 根据topic和consumerGroup为参数调用mQClientFactory.findConsumerIdList()方法向Broker端发送通信请求，获取该消费组下消费者Id列表；

(3) 先对Topic下的消息消费队列、消费者Id排序，然后用消息队列分配策略算法（默认为：消息队列的平均分配算法），计算出待拉取的消息队列。这里的平均分配算法，类似于分页的算法，将所有MessageQueue排好序类似于记录，将所有消费端Consumer排好序类似页数，并求出每一页需要包含的平均size和每个页面记录的范围 range，最后遍历整个range而计算出当前Consumer端应该分配到的的MessageQueue。




(4) 然后，调用updateProcessQueueTableInRebalance()方法，具体的做法是，先将分配到的消息队列集合（mqSet）与processQueueTable做一个过滤比对。

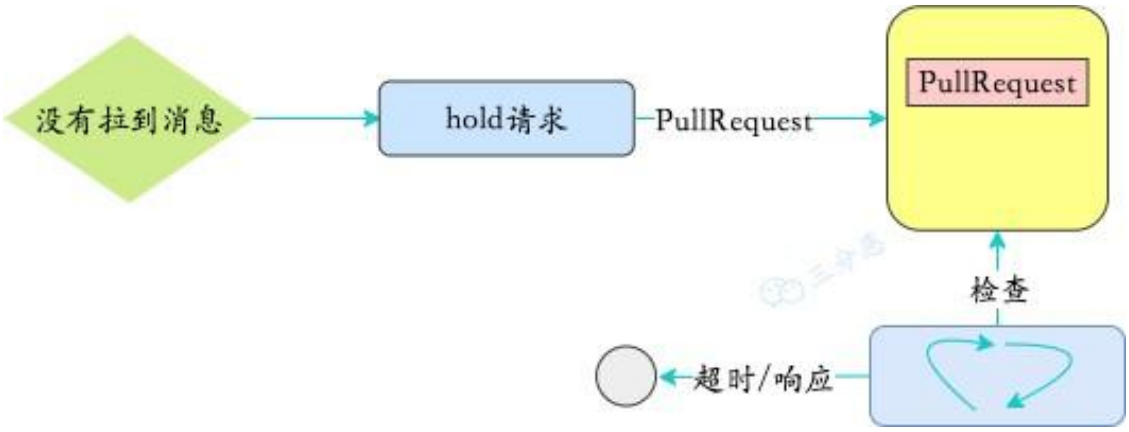



- 上图中processQueueTable标注的红色部分，表示与分配到的消息队列集合mqSet互不包含。将这些队列设置Dropped属性为true，然后查看这些队列是否可以移除出processQueueTable缓存变量，这里具体执行removeUnnecessaryMessageQueue()方法，即每隔1s 查看是否可以获取当前消费处理队列的锁，拿到的话返回true。如果等待1s后，仍然拿不到当前消费处理队列的锁则返回false。如果返回true，则从 processQueueTable缓存变量中移除对应的Entry；
- 上图中processQueueTable的绿色部分，表示与分配到的消息队列集合mqSet的交集。判断该 ProcessQueue是否已经过期了，在Pull模式的不用管，如果是Push模式的，设置Dropped属性为true，并且调用removeUnnecessaryMessageQueue()方法，像上面一样尝试移除Entry；
- 最后，为过滤后的消息队列集合（mqSet）中的每个MessageQueue创建一个ProcessQueue对象并存入RebalanceImpl的processQueueTable队列中（其中调用RebalanceImpl实例的 computePullFromWhere(MessageQueue mq)方法获取该MessageQueue对象的下一个进度消费值 offset，随后填充至接下来要创建的pullRequest对象属性中），并创建拉取请求对象—pullRequest添加到拉取列表—pullRequestList中，最后执行dispatchPullRequest()方法，将Pull消息的请求对象PullRequest依次放入PullMessageService服务线程的阻塞队列—pullRequestQueue中，待该服务线程取出后向Broker端发起 Pull消息的请求。其中，可以重点对比下，RebalancePushImpl和RebalancePullImpl两个实现类的 dispatchPullRequest()方法不同，RebalancePullImpl类里面的该方法为空。

消息消费队列在同一消费组不同消费者之间的负载均衡，其核心设计理念是在一个消息消费队列在同一时间只允许被同一消费组内的一个消费者消费，一个消息消费者能同时消费多个消息队列。

## 23. RocketMQ消息长轮询了解吗？

所谓的长轮询，就是Consumer 拉取消息，如果对应的 Queue 如果没有数据，Broker 不会立即返回，而是把PullReuqest hold起来，等待 queue 有了消息后，或者长轮询阻塞时间到了，再重新处理该 queue 上的所有PullRequest。




- PullMessageProcessor#processRequest

```
//如果没有拉到数据
case ResponseCode.PULL_NOT_FOUND:
// broker 和 consumer 都允许 suspend，默认开启 if
(brokerAllowSuspend && hasSuspendFlag) {
        long pollingTimeMills = suspendTimeoutMillisLong;
        if (!this.brokerController.getBrokerConfig().isLongPollingEnable()) {

pollingTimeMills = this.brokerController.getBrokerConfig().getShortPollingTimeMills();
        }

        String topic = requestHeader.getTopic();
        long offset = requestHeader.getQueueOffset(); int queueId =
        requestHeader.getQueueId();
        //封装一个PullRequest
        PullRequest pullRequest = new PullRequest(request, channel, pollingTimeMills,

this.brokerController.getMessageStore().now(), offset, subscriptionData, messageFilter)
;
        //把PullRequest挂起来

this.brokerController.getPullRequestHoldService().suspendPullRequest(topic, queueId, pu llRequest);
        response = null; break;
}
```

挂起的请求，有一个服务线程会不停地检查，看queue中是否有数据，或者超时。

- PullRequestHoldService#run()

```
@Override
```

```
public void run() {
    log.info("{} service started", this.getServiceName());
    while (!this.isStopped()) {
        try {
            if (this.brokerController.getBrokerConfig().isLongPollingEnable())
                { this.waitForRunning(5 * 1000);
            } else {

this.waitForRunning(this.brokerController.getBrokerConfig().getShortPollingTimeMills())
;
            }

            long beginLockTimestamp = this.systemClock.now();
            //检查hold住的请求
            this.checkHoldRequest();
            long costTime = this.systemClock.now() - beginLockTimestamp;
            if (costTime > 5 * 1000) {
                log.info("[NOTIFYME] check hold request cost {} ms.", costTime);
            }
        } catch (Throwable e) {
            log.warn(this.getServiceName() + " service has exception. ", e);
        }
    }

    log.info("{} service end", this.getServiceName());
}
```