RocketMQ 介绍

消息队列 RocketMQ 是阿里巴巴集团基于高可用分布式集群技术，自主研发的云正式商用的专业消息中间件，既可为分布式应用系统提供异步解耦和削峰填谷的能力，同时也具备互联网应用所需的海量消息堆积、高吞吐、可靠重试等特性，是阿里巴巴双 11 使用的核心产品。

RocketMQ 的设计基于主题的发布与订阅模式，其核心功能包括消息发送、消息存储(Broker)、消息消费，整体设计追求简单与性能第一。

1. NameServer 设计及其简单，RocketMQ 摈弃了业界常用的 Zookeeper 充当消息管理的“注册中心”，而是使用自主研发的 NameServer 来实现各种元数据的管理（Topic 路由信息等）
2. 高效的 I/O 存储，RocketMQ 追求消息发送的高吞吐量，RocketMQ 的消息存储设计成文件组的概念，组内单个文件固定大小，引入了内存映射机制，所有主题的消息存储基于顺序读写，极大提高消息写性能，同时为了兼顾消息消费与消息查找，引入消息消费队列文件与索引文件
3. 容忍存在设计缺陷，适当将某些工作下放给 RocketMQ 的使用者，比如消息只消费一次，这样极大的简化了消息中间件的内核，使得 RocketMQ 的实现发送变得非常简单与高效。

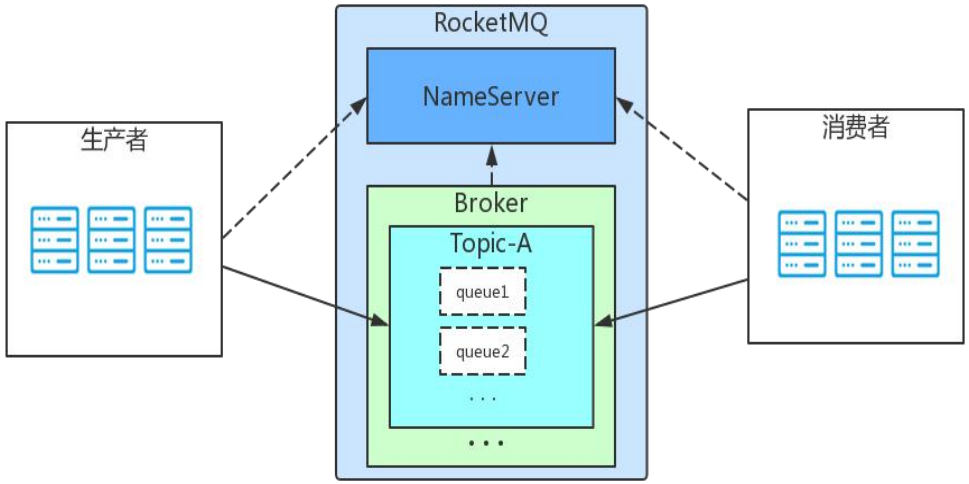
RocketMQ 原先阿里巴巴内部使用，与 2017 年提交到 Apache 基金会成为 Apache 基金会的顶级开源项目，GitHub 代码库链接:https://github.com/apache/rocketmq.git

RocketMQ 的官网 http://rocketmq.apache.org/

MQ:消息中间件是什么？

消息中间件属于分布式系统中一个子系统，关注于数据的发送和接收，利用高效可靠的异步消息传递机制对分布式系统中的其余各个子系统进行集成

核心概念



NameServer

NameServer 是整个 RocketMQ 的“大脑”，它是 RocketMQ 的服务注册中心,所以 RocketMQ 需要先启动 NameServer 再启动 Rocket 中的 Broker。

Broker 在启动时向所有 NameServer 注册（主要是服务器地址等），生产者在发送消息之前先从 NameServer 获取 Broker 服务器地址列表（消费者一样），然后根据负载均衡算法从列表中选择一台服务器进行消息发送。

NameServer 与每台 Broker 服务保持长连接，并间隔 30S 检查 Broker 是否存活，如果检测到 Broker 宕机，则从路由注册表中将其移除。这样就可以实现 RocketMQ 的高可用。具体细节后续的课程会进行讲解。

主题

主题，Topic，消息主题，一级消息类型，生产者向其发送消息。消费者负责从 Topic 接收并消费消息。

生产者

生产者：也称为消息发布者，负责生产并发送消息至 Topic。

消费者

消费者：也称为消息订阅者，负责从 Topic 接收并消费消息。

消息

消息：生产者或消费者进行消息发送或消费的主题，对于 RocketMQ 来说，消息就是字节数组。

核心概念

以下我们将总结下 Rocket 的整体运转。

1. NameServer 先启动
2. Broker 启动时向 NameServer 注册
3. 生产者在发送某个主题的消息之前先从 NamerServer 获取 Broker 服务器地址列表（有可能是集群），然后根据负载均衡算法从列表中选择一台 Broker 进行消息发送。

4. NameServer 与每台 Broker 服务器保持长连接，并间隔 30S 检测 Broker 是否存活，如果检测到 Broker 宕机（使用心跳机制，如果检测超过 120S），则从路由注册表中将其移除。
5. 消费者在订阅某个主题的消息之前从 NamerServer 获取 Broker 服务器地址列表（有可能是集群），但是消费者选择从 Broker 中 订阅消息，订阅规则由 Broker 配置决定。

RocketMQ 的设计理念和目标

设计理念

整体设计思想 80%借鉴 Kafka.

基于主题的发布和订阅，其核心功能，消息发送、消息存储和消息消费。整体设计追求简单与性能。

NameServer 性能对比 Zookeeper 有极大的提升

高效的 IO 存储机制，基于文件顺序读写，内存映射机制

容忍设计缺陷，比如消息只消费一次，Rocket 自身不保证，从而简化 Rocket 的内核使得 Rocket 简单与高效，这个问题交给消费者去实现（幂等）。

设计目标

架构模式：发布订阅模式，主要组件：消息发送者、消息服务器（消息存储）、消息消费、路由发现。

顺序消息：RocketMQ 可以严格保证消息有序

消息过滤：消息消费是，消费者可以对同一主题下的消息按照规则只消费自己感兴趣的消息，可以支持在服务端与消费端的消息过滤机制。

消息存储：一般 MQ 核心就是消息的存储，对存储一般来说两个维度:消息堆积能力和消息存储性能。RocketMQ 追求消息存储的高性能，引入内存映射机制，所有的主题消息顺序存储在同一个文件中。同时为了防止无限堆积，引入消息文件过期机制和文件存储空间报警机制。

消息高可用：

1. Rocket 关机、断电等情况下，Rokcet 可以确保不丢失消息（同步刷盘机制不丢失，异步刷盘会丢失少量）。

2. 另外如果 Rocket 服务器因为 CPU、内存、主板、磁盘等关键设备损坏导致无法开机，这个属于单点故障，该节点上的消息全部丢失，如果开启了异步复制机制，Rocket 可以确保只丢失很少量消息。
3. 如果引入双写机制，这样基本上可以满足消息可靠性要求极高的场景（毕竟两台主服务器同时故障的可能性还是非常小）

消息消费低延迟：RocketMQ 在消息不发生消息堆积时，以长轮询模式实现准实时的消息推送模式。

确保消息必须被消费一次：消息确认机制(ACK)来确保消息至少被消费一次，一般 ACK 机制只能做到消息只被消费一次，有重复消费的可能。

消息回溯：已经消费完的消息，可以根据业务要求重新消费消息。

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，RocketMQ 采用磁盘文件存储，所以堆积能力比较强，同时提供文件过期删除机制。

定时消息：定时消息，定时消息是指消息发送到 Rocket Broker 上之后，不被消费者理解消费，要到等待一定的时间才能进行消费，apache 的版本目前只支持等待指定的时间才能被消费，不支持任意精度的定时消息消费。（一个说法是任意精度的定时消息会带来性能损耗，但是阿里云版本的 RocketMQ 却提供这样的功能，充值收费优先策略？）

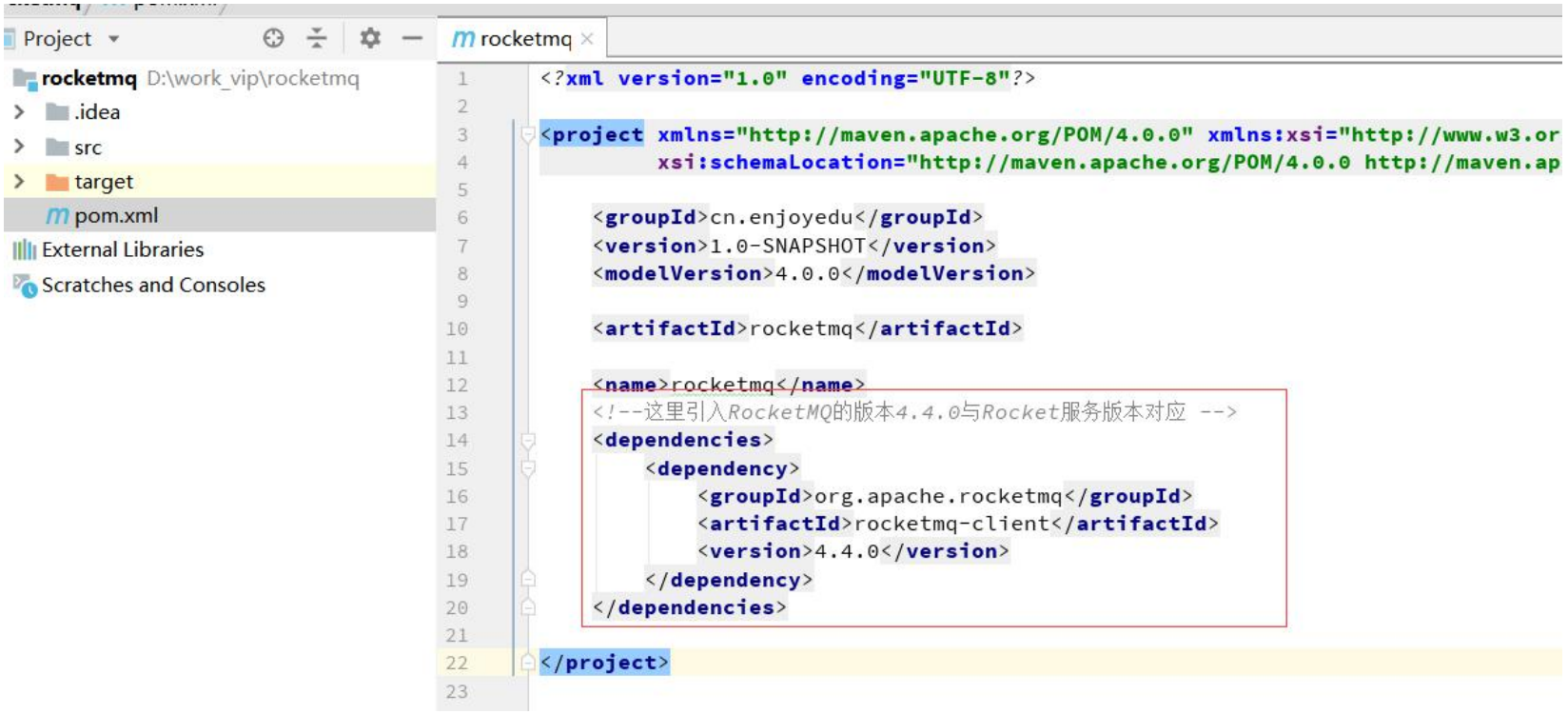
消息重试机制：消息重试是指在消息消费时，如果发送异常，那么消息中间件需要支持消息重新投递，RocketMQ 支持消息重试机制。

RocketMq 中消息的发送

普通消息是指消息队列 RocketMQ 中无特性的消息，区别于有特性的定时/延时消息、顺序消息和事务消息。这些后续会细讲。

RocketMQ 发送普通消息有三种实现方式：单向(OneWay)发送、可靠同步发送、可靠异步发送。

消息生产的客户端依赖如下：



broker.conf

```
# 是否允许 Broker 自动创建 Topic，建议线下开启，线上关闭
autoCreateTopicEnable=true
# 是否允许 Broker 自动创建订阅组，建议线下开启，线上关闭
autoCreateSubscriptionGroup=true
```

单向(OneWay)发送

代码演示

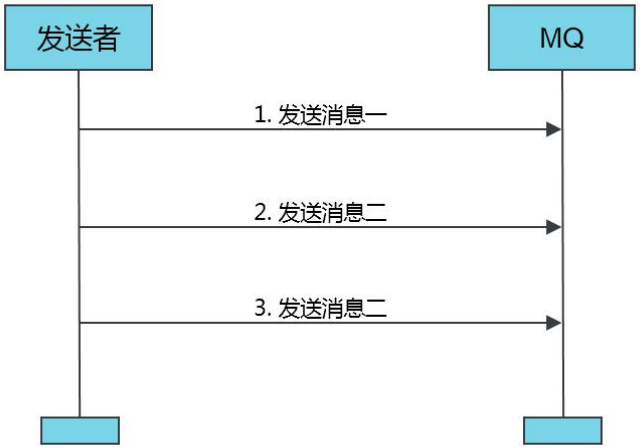
```
*/
public class OnewayProducer {
    public static void main(String[] args) throws Exception{
        //生产者实例化
        DefaultMQProducer producer = new DefaultMQProducer( producerGroup: "oneway");
        //指定rocket服务器地址
        producer.setNamesrvAddr("192.168.56.101:9876");

        //启动实例
        producer.start();
        for (int i = 0; i < 10; i++) {
            //创建一个消息实例，指定topic、tag和消息体
            Message msg = new Message( topic: "TopicTest" /* Topic */,
                    tags: "TagA" /* Tag */,
                    ("Hello RocketMQ " +
                            i).getBytes(RemotingHelper.DEFAULT_CHARSET) /* Message body */
            );
            //发送消息
            producer.sendOneway(msg);
            System.out.printf("%s%n",  new String(msg.getBody()));
        }
        //生产者实例不再使用时关闭.
        producer.shutdown();
    }
}
```

生产者分区

标签：二级分类

单向（Oneway）发送特点为发送方只负责发送消息，不等待服务器回应且没有回调函数触发，即只发送请求不等待应答。此方式发送消息的过程耗时非常短，一般在微秒级别。cn.enjoyedu.normal.OnewayProducer



Producer Group（生产者分组）

生产者组，简单来说就是多个发送同一类消息的生产者称之为一个生产者组。在这里可以不用关心，只要知道有这么一个概念即可。RocketMQ 中的生产者组只能有一个在用的生产者。分组的作用如下（简单的场景不需要了解这个概念）：

1. 标识一类 Producer
2. 可以通过运维工具查询这个发送消息应用下有多个 Producer 实例
3. 发送分布式事务消息时，如果 Producer 中途意外宕机，Broker 会主动回调 Producer Group 内的任意一台机器来确认事务状态。

Producer 实例

Producer 的一个对象实例，不同的 Producer 实例可以运行在不同进程内或者不同机器上。Producer 实例线程安全，可在同一进程内多线程之间共享。

Message Key

Key 一般用于消息在业务层面的唯一标识。对发送的消息设置好 Key，以后可以根据这个 Key 来查找消息。比如消息异常，消息丢失，进行查找会很方便。RocketMQ 会创建专门的索引文件，用来存储 Key 与消息的映射，由于是 Hash 索引，应尽量使 Key 唯一，避免潜在的哈希冲突。

Tag 和 Key 的主要差别是使用场景不同，Tag 用在 Consumer 代码中，用于服务端消息过滤，Key 主要用于通过命令进行查找消息

RocketMQ 并不能保证 message id 唯一，在这种情况下，生产者在 push 消息的时候可以给每条消息设定唯一的 key，消费者可以通过 message key 保证对消息幂等处理。

Tag

消息标签，二级消息类型，用来进一步区分某个 Topic 下的消息分类。

Topic 与 Tag 都是业务上用来归类的标识，区分在于 Topic 是一级分类，而 Tag 可以理解为是二级分类。

以天猫交易平台为例，订单消息和支付消息属于不同业务类型的消息，分别创建 Topic_Order 和 Topic_Pay，其中订单消息根据商品品类以不同的 Tag 再进行细分，如电器类、男装类、女装类、化妆品类，最后他们都被各个不同的系统所接收。

通过合理的使用 Topic 和 Tag，可以让业务结构清晰，更可以提高效率。

您可能会有这样的疑问：到底什么时候该用 Topic，什么时候该用 Tag？

1）消息类型是否一致：如普通消息、事务消息、定时（延时）消息、顺序消息，不同的消息类型使用不同的 Topic，无法通过 Tag 进行区分。

2）业务是否相关联：没有直接关联的消息，如淘宝交易消息，京东物流消息使用不同的 Topic 进行区分；而同样是天猫交易消息，电器类订单、女装类订单、化妆品类订单的消息可以用 Tag 进行区分。

3）消息优先级是否一致：如同样是物流消息，盒马必须小时内送达，天猫超市 24 小时内送达，淘宝物流则相对会慢一些，不同优先级的消息用不同的 Topic 进行区分。

4）消息量级是否相当：有些业务消息虽然量小但是实时性要求高，如果跟某些万亿量级的消息使用同一个 Topic，则有可能会因为过长的等待时间而“饿死”，此时需要将不同量级的消息进行拆分，使用不同的 Topic。

可靠同步发送

代码演示

```
import ...

/**
 * @author 【享学课堂】 King老师
 * 同步发送
 */
public class SyncProducer {
    public static void main(String[] args) throws Exception {
        DefaultMQProducer producer = new DefaultMQProducer( producerGroup: "sync");
        producer.setNamesrvAddr("localhost:9876");
        producer.start();
        for (int i = 0; i < 10; i++) {
            Message msg = new Message( topic: "TopicTest" ,
                    tags: "TagB" ,
                    ("Hello RocketMQ " + i).getBytes(RemotingHelper.DEFAULT_CHARSET)
            );
            SendResult sendResult = producer.send(msg);
            System.out.printf("%s%n%n%n", sendResult.getSendStatus()+":(MsgId):"
                    +sendResult.getMsgId()+":(queueId):"
                    +sendResult.getMessageQueue().getQueueId()
                    +"(value):"+ new String(msg.getBody()));
        }
        producer.shutdown();
    }
```

同步发送是指消息发送方发出数据后，同步等待，直到收到接收方发回响应之后才发下一个请求。

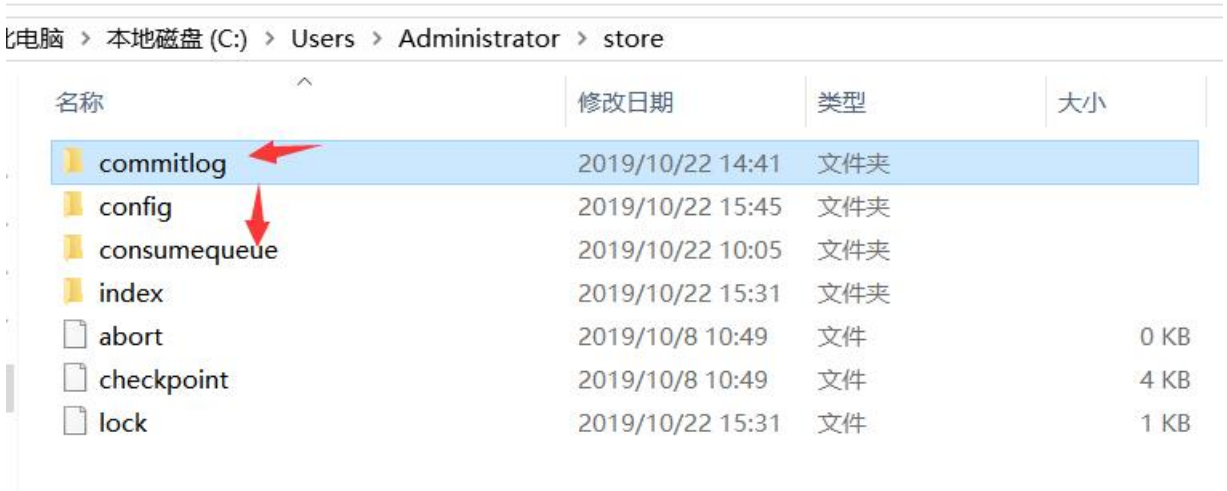
Message ID

消息的全局唯一标识（内部机制的 ID 生成是使用机器 IP 和消息偏移量的组成，所以有可能重复，如果是幂等性还是最好考虑 Key），由消息队列 MQ 系统自动生成，唯一标识某条消息。

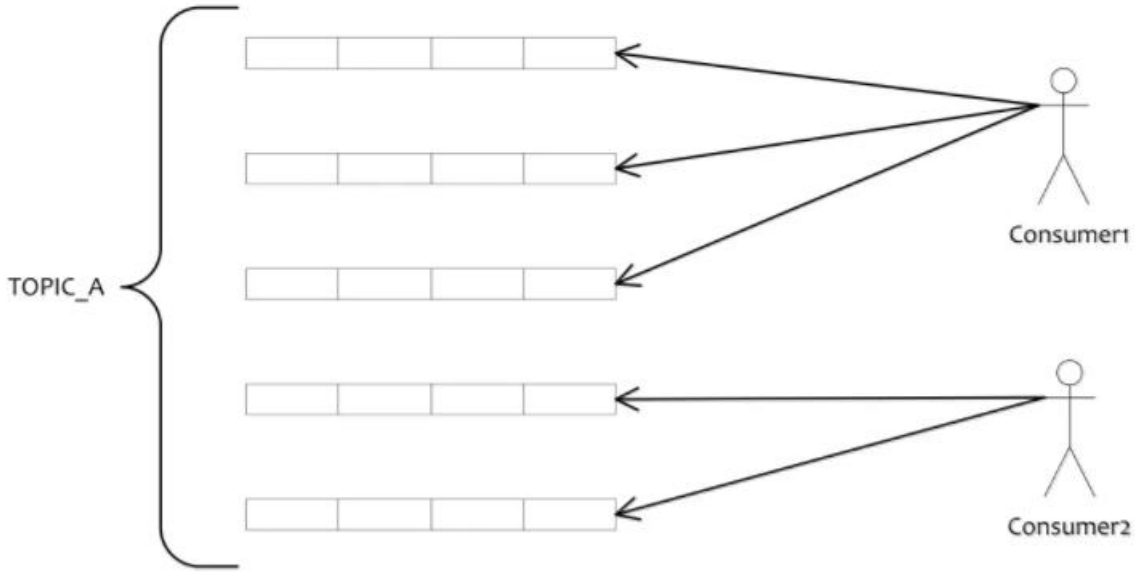
SendStatus

发送的标识。成功，失败等

Queue



RocketMQ 收到消息后，所有主题的消息都存储在 commitlog 文件中，当消息到达 commitlog 后，将会采用异步转发到消息队列，也就是 consumerqueue，Queue 是数据分片的产物，数据分片可以提高消费者的效率。(这个也是 RocketMQ 对比 Kafka 的不同，存储设计时有队列的概念)



broker.conf

```
# 在发送消息时，自动创建服务器不存在的 topic，默认创建的队列数
defaultTopicQueueNums=4
```

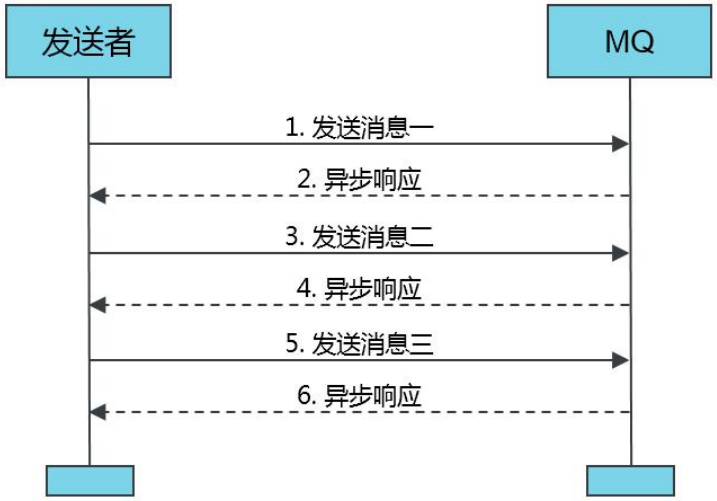
可靠异步发送

代码演示

```java
 * @author 【享学课堂】 King老师
 * 异步发送
 */
public class AsyncProducer {
    public static void main(
            String[] args) throws MQClientException, InterruptedException, UnsupportedEncodingE
        //生产者实例化
        DefaultMQProducer producer = new DefaultMQProducer( producerGroup: "async");
        //指定rocket服务器地址
        producer.setNamesrvAddr("localhost:9876");
        //启动实例
        producer.start();
        //发送异步失败时的重试次数(这里不重试)
        producer.setRetryTimesWhenSendAsyncFailed(0);

        int messageCount = 10;
        final CountDownLatch countDownLatch = new CountDownLatch(messageCount);
        for (int i = 0; i < messageCount; i++) {
            try {
                final int index = i;
                Message msg = new Message( topic: "TopicTest",
                        tags: "TagC",
                        keys: "OrderID"+index,
                        ("Hello world "+index).getBytes(RemotingHelper.DEFAULT_CHARSET));
                //生产者异步发送
                producer.send(msg, new SendCallback() {
                    @Override
                    public void onSuccess(SendResult sendResult) {
                        countDownLatch.countDown();
                        System.out.printf("%-10d OK %s %n", index, new String(msg.getBody()));
                    }
                    @Override
                    public void onException(Throwable e) {
                        countDownLatch.countDown();
                        System.out.printf("%-10d Exception %s %n", index, e);
```

消息发送方在发送了一条消息后，不等接收方发回响应，接着进行第二条消息发送。发送方通过回调接口的方式接收服务器响应，并对响应结果进行处理



RocketMQ 中消息发送的权衡

三种发送方式的对比

发送方式	发送 TPS	发送结果反馈	可靠性	适用场景
同步可靠发送	快	有	不丢失	重要通知邮件、报名短信通知、营销短信系统等
异步可靠发送	快	有	不丢失	用户视频上传后通知启动转码服务，转码完成后通知推送转码结果等
单向发送	最快	无	可能丢失	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

RocketMQ 消息消费

集群消费和广播消费

基本概念

消息队列 RocketMQ 是基于发布/订阅模型的消息系统。消息的订阅方订阅关注的 Topic，以获取并消费消息。由于订阅方应用一般是分布式系统，以集群方式部署有多台机器。因此消息队列 RocketMQ 约定以下概念。

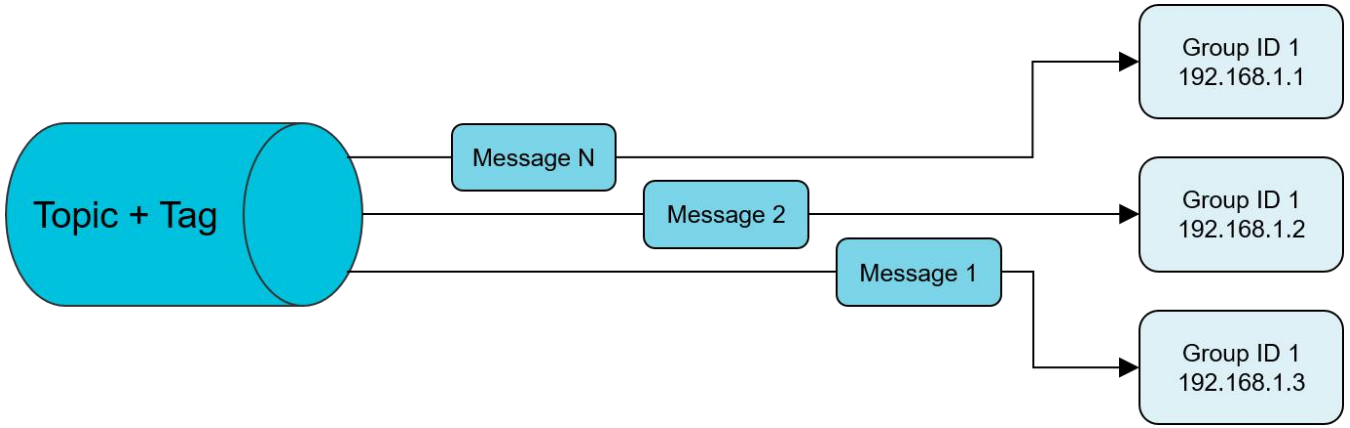
集群：使用相同 Group ID 的订阅者属于同一个集群。同一个集群下的订阅者消费逻辑必须完全一致（包括 Tag 的使用），这些订阅者在逻辑上可以认为是一个消费节点。

集群消费：当使用集群消费模式时，消息队列 RocketMQ 认为任意一条消息只需要被集群内的任意一个消费者处理即可。

广播消费：当使用广播消费模式时，消息队列 RocketMQ 会将每条消息推送给集群内所有注册过的客户端，保证消息至少被每台机器消费一次。

场景对比

集群消费模式：



适用场景&注意事项

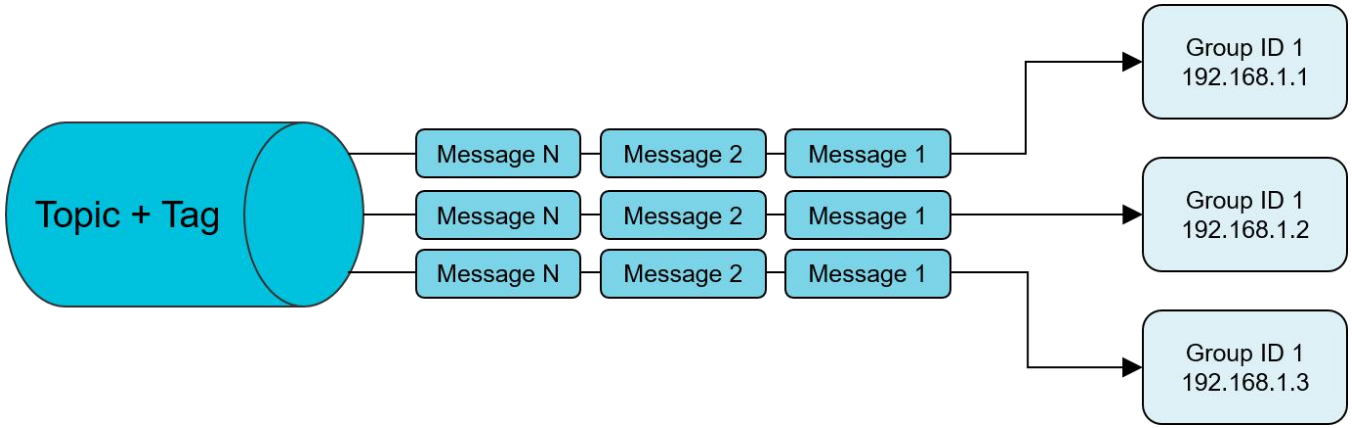
消费端集群化部署，每条消息只需要被处理一次。

由于消费进度在服务端维护，可靠性更高。

集群消费模式下，每一条消息都只会被分发到一台机器上处理。如果需要被集群下的每一台机器都处理，请使用广播模式。

集群消费模式下，不保证每一次失败重投的消息路由到同一台机器上，因此处理消息时不应该做任何确定性假设。

广播消费模式：



适用场景&注意事项

广播消费模式下不支持顺序消息。

广播消费模式下不支持重置消费位点。

每条消息都需要被相同逻辑的多台机器处理。

消费进度在客户端维护，出现重复的概率稍大于集群模式。

广播模式下，消息队列 RocketMQ 保证每条消息至少被每台客户端消费一次，但是并不会对消费失败的消息进行失败重投，因此业务方需要关注消费失败的情况。

广播模式下，客户端每一次重启都会从最新消息消费。客户端在被停止期间发送至服务端的消息将会被自动跳过，请谨慎选择。

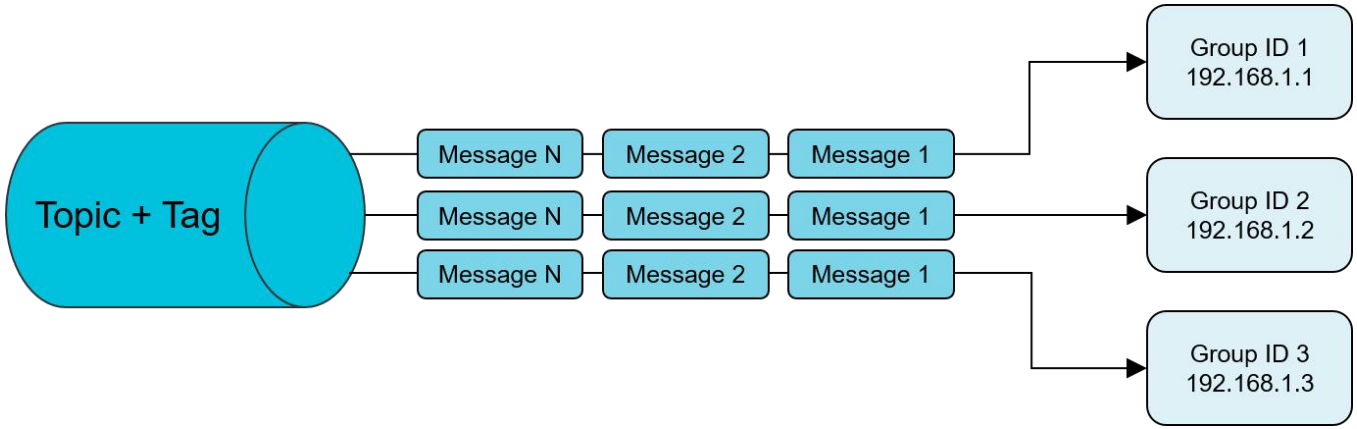
广播模式下，每条消息都会被大量的客户端重复处理，因此推荐尽可能使用集群模式。

目前仅 Java 客户端支持广播模式。

广播模式下服务端不维护消费进度，所以消息队列 RocketMQ 控制台不支持消息堆积查询、消息堆积报警和订阅关系查询功能。

使用集群模式模拟广播

如果业务需要使用广播模式，也可以创建多个 Group ID，用于订阅同一个 Topic。



适用场景&注意事项

每条消息都需要被多台机器处理，每台机器的逻辑可以相同也可以不一样。

消费进度在服务端维护，可靠性高于广播模式。

对于一个 Group ID 来说，可以部署一个消费端实例，也可以部署多个消费端实例。 当部署多个消费端实例时，实例之间又组成了集群模式（共同分担消费消息）。 假设 Group ID 1 部署了三个消费者实例 C1、C2、C3，那么这三个实例将共同分担服务器发送给 Group ID 1 的消息。 同时，实例之间订阅关系必须保持一致。

消费方式

推模式

代码上使用 DefaultMQPushConsumer

这种模型下，系统收到消息后自动调用处理函数来处理消息，自动保存 Offset,并且加入新的消费者后会自动做负载均衡。

底层实现上，推模式还是使用的 pull 来实现的，pull 就是拉取，push 方式是 Server 端接收到消息后，主动把消息推给 Client 端，实时性高。但是使用 Push 方式有很多弊端，首先加大 Server 端的工作量，其次不同的 Client 端处理能力不同，Client 的状态不受 Server 控制，如果 Client 不能及时处理 Server 推送过来的消息，会造成各种潜在问题。

所以 RocketMQ 是通过“长轮询”的方式，同时通过 Client 端和 Server 端的配合，达到既拥有 Pull 的优点，又能达到确保实时性的目的。

拉模式

代码上使用 DefaultMQPullConsumer

使用方式类似，但是更加复杂，除了像推模式一样需要设置各种参数之外，还需要处理额外三件事情：

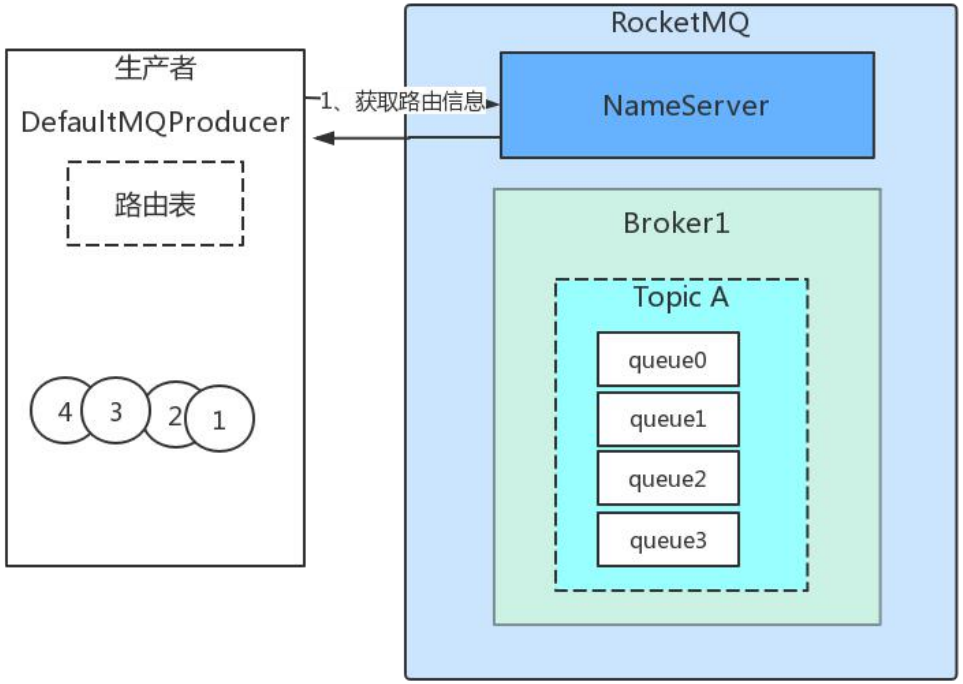
1）获取 MessageQueues 并遍历（一个 Topic 包括多个 MessageQueue），如果是特殊情况，也可以选择指定的 MessageQueue 来读取消息

2）维护 Offsetstore,从一个 MessageQueue 里拉取消息时，要传入 Offset 参数，随着不断的读取消息，Offset 会不断增长。这个时候就需要用户把 Offset 存储起来，根据实际的情况存入内存、写入磁盘或者数据库中。

3）根据不同的消息状态做不同的处理。

深入消息发送

消息生产者流程



生产者的流程主要讲述 DefaultMQProducer 类的具体实现。

消息发送的主要流程：验证消息、查找路由、消息发送（包含异常机制）

验证消息：主要是要求主题名称、消息体不能为空、消息长度不能等于 0，且不能超过消息的最大的长度 4M(生产者对象中配置 maxMessageSize=1024*1024*4)

查找路由：客户端（生产者）会缓存 topic 路由信息（如果是第一次发送消息，本地没有缓存，查询 NameServer 尝试获取），路由信息主要包含了消息队列（queue 相关信息），

消息发送：选择消息队列，发送消息，发送成功则返回。选择消息队列两种方式（一般有两种，这里不做详细讲解，后续做详细讲解）

批量消息发送

```java
import ...

public class SimpleBatchProducer {

    public static void main(String[] args) throws Exception {
        DefaultMQProducer producer = new DefaultMQProducer( producerGroup: "BatchProducer");
        producer.setNamesrvAddr("localhost:9876");
        producer.start();

        //如果一次只发送不超过4M的消息，那么批处理很容易使用
        //同一批的消息应该有：相同的主题，相同的waitstoremsgok，不支持调度
        String topic = "TopicTest";
        List<Message> messages = new ArrayList<>();
        messages.add(new Message(topic,  tags: "Tag",  keys: "OrderID001", "Hello world 0".getBytes()))
        messages.add(new Message(topic,  tags: "Tag",  keys: "OrderID002", "Hello world 1".getBytes()))
        messages.add(new Message(topic,  tags: "Tag",  keys: "OrderID003", "Hello world 2".getBytes()))

        producer.send(messages);
    }
}
```

注意单批次不能超过消息的最大的长度 4M(生产者对象中配置 maxMessageSize=1024*1024*4)

消息重试机制

```
 * @author 【学学课堂】 King老师
 * 异步发送
 */
public class AsyncProducer {
    public static void main(
            String[] args) throws MQClientException, InterruptedException, UnsupportedEncodingExcep
        //生产者实例化
        DefaultMQProducer producer = new DefaultMQProducer( producerGroup: "async");
        //指定rocket服务器地址
        producer.setNamesrvAddr("localhost:9876");
        //启动实例
        producer.start();
        //发送异步失败时的重试次数(这里不重试)
        producer.setRetryTimesWhenSendAsyncFailed(0);

        int messageCount = 10;
        final CountDownLatch countDownLatch = new CountDownLatch(messageCount);
        for (int i = 0; i < messageCount; i++) {
            try {
                final int index = i;
                Message msg = new Message( topic: "TopicTest",
```

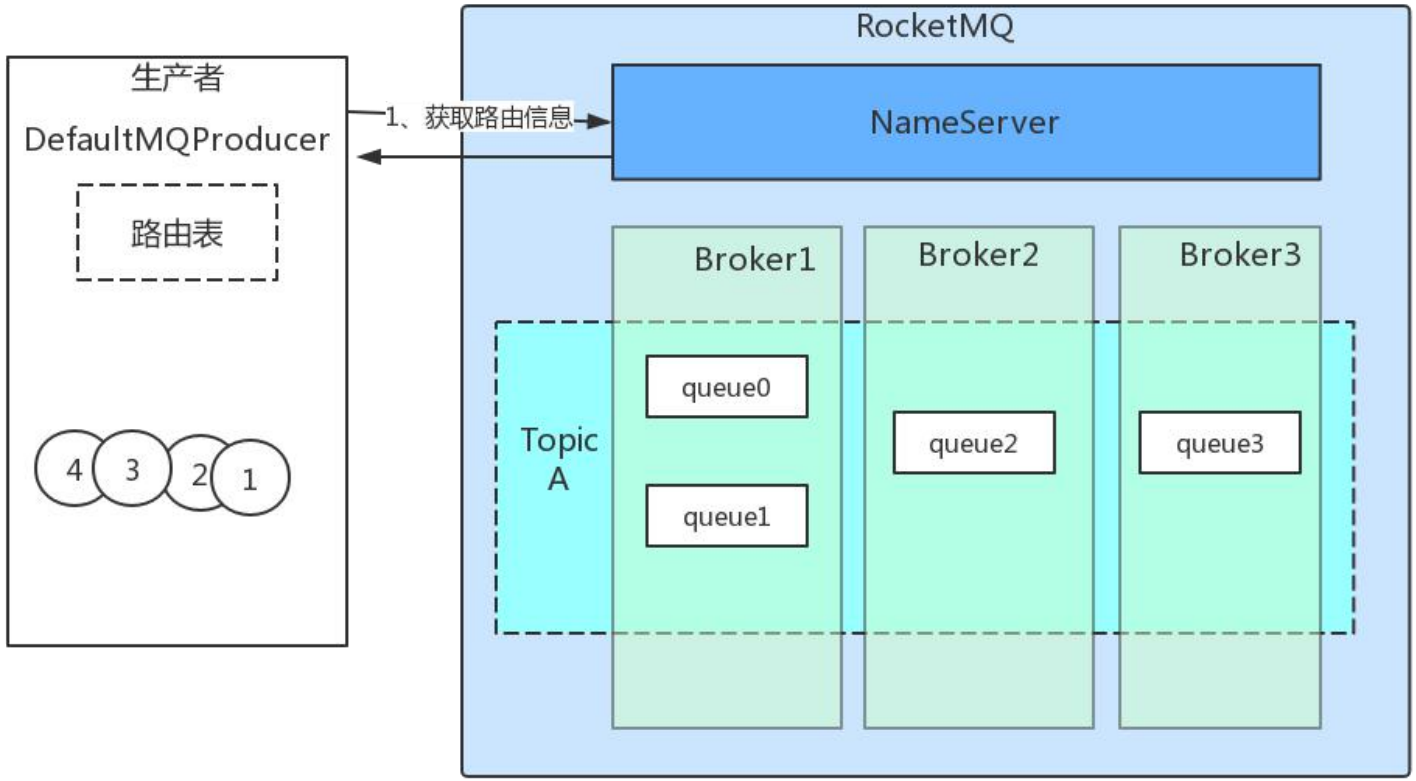
注意重试的原则，一般会采用规避原则（规避原则就是上一次消息发送过程中发现错误，在某一段时间内，消息生产者不会选择该 Broker 上的消息队列，这样可以提高发送消息的成功率）

规避原则

如下图，注意了，这里我们发现，有可能在实际的生产过程中，我们的 RocketMQ 有几台服务器构成的集群（集群后续会细讲）。

其中有可能是一个主题 TopicA 中的 4 个队列分散在 Broker1、Broker2、Broker3 服务器上。

如果这个时候 Broker2 挂了，我们知道，但是生产者不知道（因为生产者客户端每隔 30S 更新一次路由，但是 NamServer 与 Broker 之间的心跳检测间隔是 10S，所以生产者最快也需要 30S 才能感知 Broker2 挂了），所以发送到 queue2 的消息会失败，RocketMQ 发现这次消息发送失败后，就会将 Broker2 排除在消息的选择范围，下次再次发送消息时就不会发送到 Broker2,这样做的目的就是为了提高发送消息的成功率。


RocketMQ
生产者
DefaultMQProducer
路由表
4
3
2
1
1、获取路由信息
NameServer
Broker1
Broker2
Broker3
Topic
A
queue0
queue1
queue2
queue3

深入消息模式

RocketMQ 提供两种模式进行消费

拉模式

代码上使用 DefaultMQPullConsumer

1）获取 MessageQueues 并遍历（一个 Topic 包括多个 MessageQueue），如果是特殊情况，也可以选择指定的 MessageQueue 来读取消息

2）维护 Offsetstore,从一个 MessageQueue 里拉取消息时，要传入 Offset 参数，随着不断的读取消息，Offset 会不断增长。这个时候就需要用户把 Offset 存储起来，根据实际的情况存入内存、写入磁盘或者数据库中。

3）根据不同的消息状态做不同的处理。

拉取消息的请求后，会返回：FOUND（获取到消息），NO_MATCHED_MSG（没有匹配的消息），NO_NEW_MSG（没有新消息），OFFSET_ILLEGAL（非法偏移量）四种状态，其中必要重要的是 FOUND（获取到消息）和 NO_NEW_MSG（没有新消息）。

总结：这种模式下用户需要自己处理 Queue,并且自己保存偏移量，所以这种方式太过灵活，往往我们业务的关注重点不在内部消息的处理上，所以一般情况下我们会使用推模式，

推模式

代码上使用 DefaultMQPushConsumer

Push 方式是 Server 端接收到消息后，主动把消息推给 Client 端，实时性高，但是使用 Push 方式主动推送也存在一些问题：比如加大 Server 端的工作量，其次 Client 端的处理能力各不相同，如果 Client 不能及时处理 Server 推过来的消息，会造成各种潜在的问题。

长轮询

所以 RocketMQ 使用“长轮询”的方式来解决以上问题，核心思想是这样，客户端还是拉取消息，Broker 端 HOLD 住客户端发过来的请求一小段时间，在这个时间内（5s）有新消息达到，就利用现有的连接立刻返回消息给 Consunmer。“长轮询”的主动权还是掌握在 Consumer 手中，Broker 即使有大量消息积压，也不会主动推送给 Consumer。因为长轮询方式的有局限性，是在 HOLD 住 Comsumer 请求的时候需要占用资源，所以它适合在消息队列这种客户端连接数可控的场景中。

流量控制

Push 模式基于拉取，消费者会判断获取但还未处理的消息个数、消息总大小、Offset 的跨度 3 个维度来控制，如果任一值超过设定的大小就隔一段时间再拉取消息，从而达到流量控制的目的。

两种情况会限流，限流的做法是放弃本次拉取消息的动作，并且这个队列的下一次拉取任务将在 50 毫秒后才加入到拉取任务队列。

1:当前的 ProcessQueue（一个主题有多个队列，每一个队列会对应有一个 ProcessQueue 来处理消息）正在处理的消息数量>1000

2:队列中最大最小偏移量差距>2000，这个是为了避免一条消息堵塞，消息进度无法向前推进，可能造成大量消息重复消费。

消息队列负载与重新分布机制

在集群消费模式中，往往会有很多个消费者，对应消费一个主题(topic)，一个主题中有很多个消费者队列(queue),我们要考虑的问题是，集群内多个消费者是如何负载主题下的多个消费者队列，并且如果有新的消费者加入是，消息队列又会如何重新分布。

从源码的角度上看，RocketMQ 消息队列重新分布是由 RebalanceService 线程来实现的，一个 MQClientInstance 持有一个 RebalanceService 实现，并且随着 MQClientInstance 的启动而启动。

备注：（MQClientInstance 是生产者和消费者中最大的一个实例，作为生产者或者消费者引用 RocketMQ 客户端，在一个 JVM 中所有消费者，生产者都持有同一个 MQClientInstance，MQClientInstance 只会启动一次）

RocketMQ 默认提供 5 中分配算法

如果有 8 个消息队列(q1,q2,q3,q4,q5,q6,q7,q8)，有 3 个消费者(c1,c2,c3)

1） 平均分配（AllocateMessageQueueAveragely）

c1:q1,q2,q3

c2:q4,q5,q6

c3:q7,q8,

2） 平均轮询分配（AllocateMessageQueueAveragelyByCircle）

c1:q1,q4,q7

c2:q2,q5,q8

c3:q3,q6

3） 一直性 Hash（AllocateMessageQueueConsistentHash）

不推荐使用，因为消息队列负载均衡信息不容易跟踪

4） 根据配置（AllocateMessageQueueByConfig）

为每一个消费者配置固定的消费队列

5） 根据 Broker 部署机房名（AllocateMessageQueueByMachineRoom）

对每一个消费者负载不同 Broker 上的队列

一般尽量使用“平均分配”“平均轮询分配”，因为分配算法比较直观。无论哪种算法，遵循的原则是一个消费者可以分配多个消息队列，同一个消息队列只会分配一个消费者，所以如果消费者个数大于消息队列数量，则有些消费者无法消费消息。

RebalanceService 每隔 20S 进行一次队列负载

每次进行队列重新负载时会查询出当前所有的消费者，并且对消息队列、消费者列表进行排序。因为在一个 JVM 中只会有一个 pullRequestQueue 对象。具体可见源码中 PullMessageService

消息确认(ACK)

PushConsumer 为了保证消息肯定消费成功，只有使用方明确表示消费成功，RocketMQ 才会认为消息消费成功。中途断电，抛出异常等都不会认为成功——即都会重新投递。

业务实现消费回调的时候，当且仅当此回调函数返回 ConsumeConcurrentlyStatus.CONSUME_SUCCESS，RocketMQ 才会认为这批消息（默认是 1 条）是消费完成的

```java
public class PushConsumerA {

    public static void main(String[] args) throws InterruptedException, MQClientException {
        DefaultMQPushConsumer consumer = new DefaultMQPushConsumer( consumerGroup: "group1");
        consumer.subscribe( topic: "TopicTest",  subExpression: "*");
        consumer.setNamesrvAddr("localhost:9876");
        consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_LAST_OFFSET);//每次从最后一次消费的地址
        //
        consumer.setAllocateMessageQueueStrategy(new AllocateMessageQueueByConfig());
        consumer.registerMessageListener(new MessageListenerConcurrently() {
            @Override
            public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs, ConsumeConcurrentlyContext context
                System.out.printf("queueID:%d:%s:Messages:%s %n",  msgs.get(0).getQueueId(),Thread.currentThread().ge
                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
            }
        });
        consumer.start();
        System.out.printf("ConsumerPartOrder Started.%n");
    }
}
```

如果这时候消息消费失败，例如数据库异常，余额不足扣款失败等一切业务认为消息需要重试的场景，只要返回 ConsumeConcurrentlyStatus.RECONSUME_LATER，RocketMQ 就会认为这批消息消费失败了。

返回 ConsumeConcurrentlyStatus.RECONSUME_LATER,rocketmq 会放到重试队列,这个重试 TOPIC 的名字是%RETRY%+consumergroup 的名字

为了保证消息是肯定被至少消费成功一次，RocketMQ 会把这批消息重发回 Broker（topic 不是原 topic 而是这个消费者的 RETRY topic），在延迟的某个时间点（默认是 10 秒，业务可设置）后，再次投递到这个 ConsumerGroup。而如果一直这样重复消费都持续失败到一定次数（默认 16 次），就会投递到 DLQ 死信队列。应用可以监控死信队列来做人工干预。

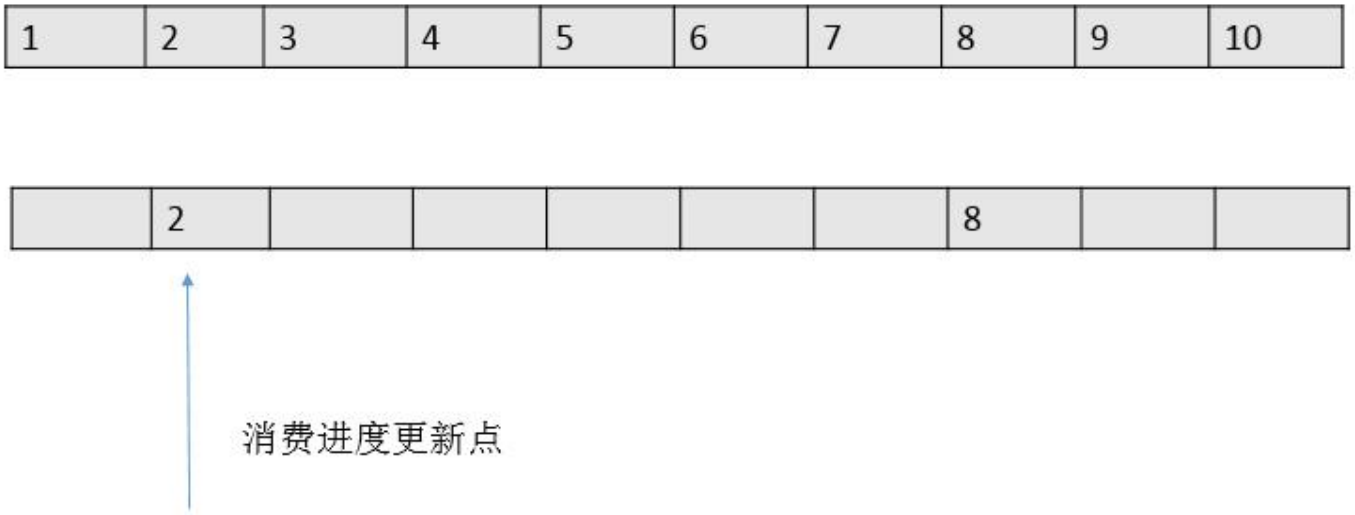
消息 ACK 机制

RocketMQ 是以 consumer group+queue 为单位是管理消费进度的，以一个 consumer offset 标记这个这个消费组在这条 queue 上的消费进度。

如果某已存在的消费组出现了新消费实例的时候，依靠这个组的消费进度，就可以判断第一次是从哪里开始拉取的。

每次消息成功后，本地的消费进度会被更新，然后由定时器定时同步到 broker，以此持久化消费进度。

但是每次记录消费进度的时候，只会把一批消息中最小的 offset 值为消费进度值



这钟方式和传统的一条 message 单独 ack 的方式有本质的区别。性能上提升的同时，会带来一个潜在的重复问题——由于消费进度只是记录了一个下标，就可能出现拉取了 100 条消息如 2101-2200 的消息，后面 99 条都消费结束了，只有 2101 消费一直没有结束的情况。

在这种情况下，RocketMQ 为了保证消息肯定被消费成功，消费进度职能维持在 2101，直到 2101 也消费结束了，本地的消费进度才能标记 2200 消费结束了（注：consumerOffset=2201）。

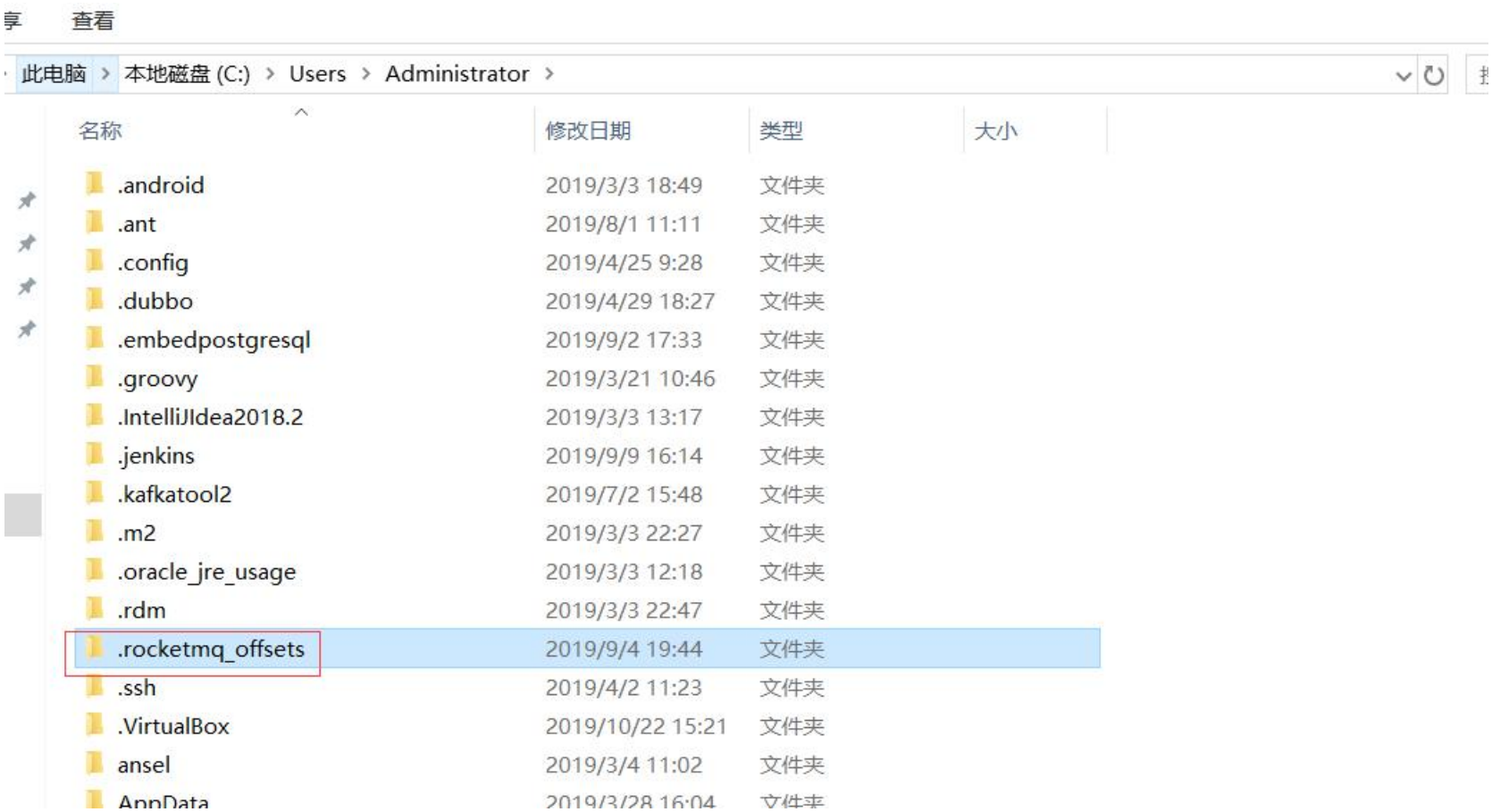
在这种设计下，就有消费大量重复的风险。如 2101 在还没有消费完成的时候消费实例突然退出（机器断电，或者被 kill）。这条 queue 的消费进度还是维持在 2101，当 queue 重新分配给新的实例的时候，新的实例从 broker 上拿到的消费进度还是维持在 2101，这时候就会又从 2101 开始消费，2102-2200 这批消息实际上已经被消费过还是会投递一次。

对于这个场景，RocketMQ 暂时无能为力，所以业务必须要保证消息消费的幂等性，这也是 RocketMQ 官方多次强调的态度。

消息进度存储

广播模式

同一个消费组的所有消费者都需要消费主题下的所有消息，因为消费者的行为都是独立的，互不影响，固消息进度需要独立存储，所以这种模式下消息进度存储在消费者本地。



集群模式

集群模式消息进度存储文件存放在服务器 Broker 上。

推模式总结

RocketMQ 中最难理解的就是推模式，下面用一个故事总结下：

King 老师是一个全能性人才，从市场拉项目做并且完成项目。市场就比如成 RocketMQ，市场上的一个项目就是一个主题，队列就是这个项目中的子任务。拉模式就是 King 老师做所有的事情，可以搞定项目中的所有事情，同时准确的完成项目中各个子任务的提交（比如成一个消费者消费队列中消息的偏移量），但是 King 老师的目标是一年赚它一个，所以我必须换个模式玩，这个模式就是开外包公司。这种开外包公司的模式就是推模式。

1. 外包公司你需要招人来做一个项目，这个项目就是一个主题，所以我会招不同的人做一个项目中不同的事情，这个就是群组消费，一个项目中一个子任务只能由一个人来做，但是一个任务不能被两个人同时做（群组消费者的原则），所以如果外包公司的人数多于子任务数的话，有的人就没事做。
2. 推模式其实还是拉模式封装，因为外包公司的员工是站在等待分配任务，就是起一个监听而已，市场上的项目信息还是需要 King 老师拉下来，为了不把外包的人给累死，King 老师必须要做控制。就是流控。流控两个纬度，这个没处理完的任务量太多，第二个他处理的时候任务跨度太大（最大偏移量和最小）这样才保险。
3. King 老师拉任务肯定是批量（拉消息肯定批量），再分给外包的人，外包公司的人做完一个事情就提交任务，但是对于 King 老师来说只能提交整体的任务进度（因为是拉模式一批批拉的）。所以推模式中你认为的提交偏移量在代码中你认为是一条条提交，其实对于市场（RocketMQ）来说，本质上还是走的拉模式所以还是批量提交的，所以 King 老师每次拉一批量，提交批量中最早的那个偏移量，这样做的好处就是之前所有的消息我都拉到了，缺点是这样就有可能有消息重复（外包公司的人工作过程中离职了），那没办法。

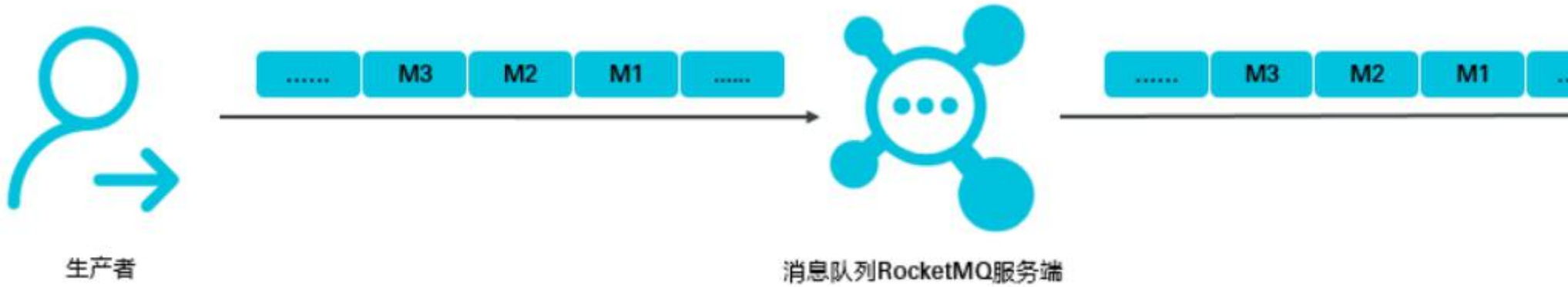
顺序消息

顺序消息（FIFO 消息）是消息队列 RocketMQ 提供的一种严格按照顺序来发布和消费的消息。顺序发布和顺序消费是指对于指定的一个 Topic，生产者按照一定的先后顺序发布消息；消费者按照既定的先后顺序订阅消息，即先发布的消息一定会先被客户端接收到。

顺序消息分为全局顺序消息和分区顺序消息。

全局顺序消息

RocketMQ 在默认情况下不保证顺序，要保证全局顺序，需要把 Topic 的读写队列数设置为 1，然后生产者和消费者的并发设置也是 1。所以这样的话高并发，高吞吐量的功能完全用不上。



适用场景

适用于性能要求不高，所有的消息严格按照 FIFO 原则来发布和消费的场景。

示例

要确保全局顺序消息，需要先把 Topic 的读写队列数设置为 1，然后生产者和消费者的并发设置也是 1。

mqadmin update Topic -t AllOrder -c DefaultCluster -r 1 -w 1 -n 127.0.0.1:9876

在证券处理中，以人民币兑换美元为 Topic，在价格相同的情况下，先出价者优先处理，则可以按照 FIFO 的方式发布和消费全局顺序消息。

部分顺序消息

对于指定的一个 Topic，所有消息根据 Sharding Key 进行区块分区。同一个分区内的消息按照严格的 FIFO 顺序进行发布和消费。Sharding Key 是顺序消息中用来区分不同分区的关键字段，和普通消息的 Key 是完全不同的概念。

延时消息

概念介绍

延时消息：Producer 将消息发送到消息队列 RocketMQ 服务端，但并不期望这条消息立马投递，而是延迟一定时间后才投递到 Consumer 进行消费，该消息即延时消息。

适用场景

消息生产和消费有时间窗口要求：比如在电商交易中超时未支付关闭订单的场景，在订单创建时会发送一条延时消息。这条消息将会在 30 分钟以后投递给消费者，消费者收到此消息后需要判断对应的订单是否已完成支付。 如支付未完成，则关闭订单。如已完成支付则忽略。

使用方式

Apache RocketMQ 目前只支持固定精度的定时消息，因为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。（阿里云 RocketMQ 提供了任意时刻的定时消息功能，Apache 的 RocketMQ 并没有,阿里并没有开源）

发送延时消息时需要设定一个延时时间长度，消息将从当前发送时间点开始延迟固定时间之后才开始投递。

延迟消息是根据延迟队列的 level 来的，延迟队列默认是

msg.setDelayTimeLevel(5)代表延迟一分钟

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

是这 18 个等级（秒（s）、分（m）、小时（h）），level 为 1，表示延迟 1 秒后消费，level 为 5 表示延迟 1 分钟后消费，level 为 18 表示延迟 2 个小时消费。生产消息跟普通的生产消息类似，只需要在消息上设置延迟队列的 level 即可。消费消息跟普通的消费消息一致。

死信队列

概念介绍

死信队列用于处理无法被正常消费的消息。当一条消息初次消费失败，消息队列 MQ 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 Consumer 在正常情况下无法正确地消费该消息。此时，消息队列 MQ 不会立刻将消息丢弃，而是将这条消息发送到该 Consumer 对应的特殊队列中。

消息队列 MQ 将这种正常情况下无法被消费的消息称为死信消息（Dead-Letter Message），将存储死信消息的特殊队列称为死信队列（Dead-Letter Queue）。

适用场景

死信消息的特性

不会再被消费者正常消费。

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

死信队列的特性

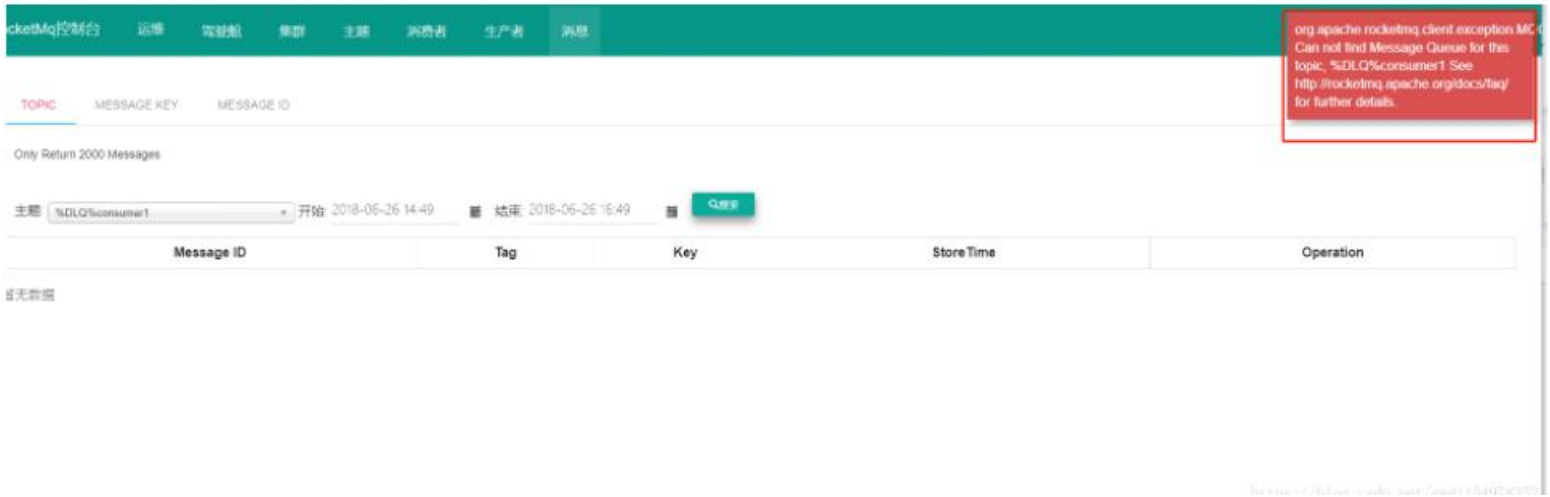
一个死信队列对应一个 Group ID， 而不是对应单个消费者实例。

如果一个 Group ID 未产生死信消息，消息队列 MQ 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。

消息队列 MQ 控制台提供对死信消息的查询的功能。

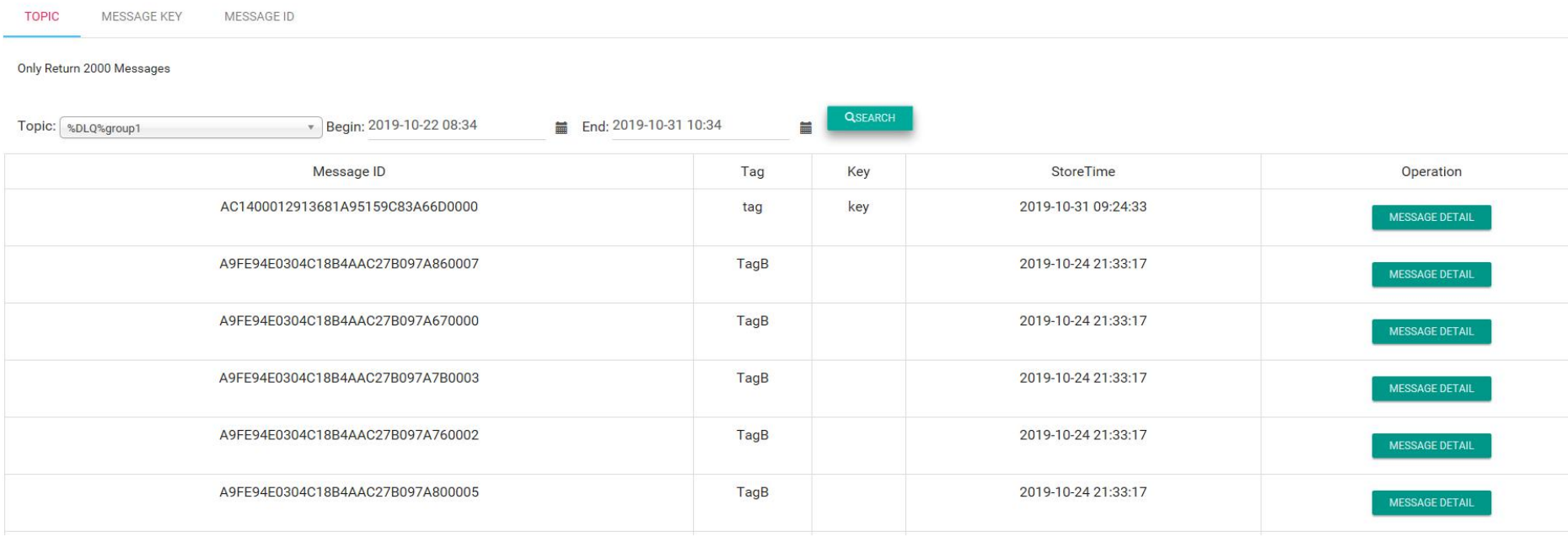
一般控制台直接查看死信消息会报错。



进入 RocketMQ 中服务器对应的 RocketMQ 中的/bin 目录，执行以下脚本

sh mqadmin updateTopic -b 192.168.0.128:10911 -n 192.168.0.128:9876 -t %DLQ%group1 -p 6

```
[root@reg king]# cd rocketmq-all-4.4.0-bin-release
[root@reg rocketmq-all-4.4.0-bin-release]# cd bin/
[root@reg bin]#  sh mqadmin  updateTopic -b 192.168.0.128:10911 -n 192.168.0.128:9876 -t %DLQ%group1 -p 6
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option PermSize=128m; support was removed in 8.0
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=128m; support was removed in 8.0
create topic to 192.168.0.128:10911 success.
TopicConfig [topicName=%DLQ%group1, readQueueNums=8, writeQueueNums=8, perm=RW-, topicFilterType=SINGLE_T
AG, topicSysFlag=0, order=false][root@reg bin]#
```



消费幂等

为了防止消息重复消费导致业务处理异常，消息队列 MQ 的消费者在接收到消息后，有必要根据业务上的唯一 Key 对消息做幂等处理。本文介绍消息幂等的概念、适用场景以及处理方法。

什么是消息幂等

当出现消费者对某条消息重复消费的情况时，重复消费的结果与消费一次的结果是相同的，并且多次消费并未对业务系统产生任何负面影响，那么这整个过程就实现可消息幂等。

例如，在支付场景下，消费者消费扣款消息，对一笔订单执行扣款操作，扣款金额为 100 元。如果因网络不稳定等原因导致扣款消息重复投递，消费者重复消费了该扣款消息，但最终的业务结果是只扣款一次，扣费 100 元，且用户的扣款记录中对应的订单只有一条扣款流水，不会多次扣除费用。那么这次扣款操作是符合要求的，整个消费过程实现了消费幂等。

需要处理的场景

在互联网应用中，尤其在网络不稳定的情况下，消息队列 MQ 的消息有可能会出现重复。如果消息重复会影响您的业务处理，请对消息做幂等处理。

消息重复的场景如下：

1. 发送时消息重复

当一条消息已被成功发送到服务端并完成持久化，此时出现了网络闪断或者客户端宕机，导致服务端对客户端应答失败。 如果此时生产者意识到消息发送失败并尝试再次发送消息，消费者后续会收到两条内容相同并且 Message ID 也相同的消息。

2. 投递时消息重复

消息消费的场景下，消息已投递到消费者并完成业务处理，当客户端给服务端反馈应答的时候网络闪断。为了保证消息至少被消费一次，消息队列 MQ 的服务端将在网络恢复后再次尝试投递之前已被处理过的消息，消费者后续会收到两条内容相同并且 Message ID 也相同的消息。

3. 负载均衡时消息重复（包括但不限于网络抖动、Broker 重启以及消费者应用重启）

当消息队列 MQ 的 Broker 或客户端重启、扩容或缩容时，会触发 Rebalance，此时消费者可能会收到重复消息。

处理方法

因为 Message ID 有可能出现冲突（重复）的情况，所以真正安全的幂等处理，不建议以 Message ID 作为处理依据。最好的方式是以业务唯一标识作为幂等处理的关键依据，而业务的唯一标识可以通过消息 Key 设置。

以支付场景为例，可以将消息的 Key 设置为订单号，作为幂等处理的依据。具体代码示例如下：

```
Message message = new Message();
message.setKey("ORDERID_100");
SendResult sendResult = producer.send(message);
```

消费者收到消息时可以根据消息的 Key，即订单号来实现消息幂等：

```
consumer.subscribe("ons_test", "*", new MessageListener() {
    public Action consume(Message message, ConsumeContext context) {
        String key = message.getKey()
        // 根据业务唯一标识的 Key 做幂等处理
    }
});
```

消息过滤

概念介绍

RocketMQ 分布式消息队列的消息过滤方式有别于其它 MQ 中间件，是可以实现服务端的过滤。

表达式过滤

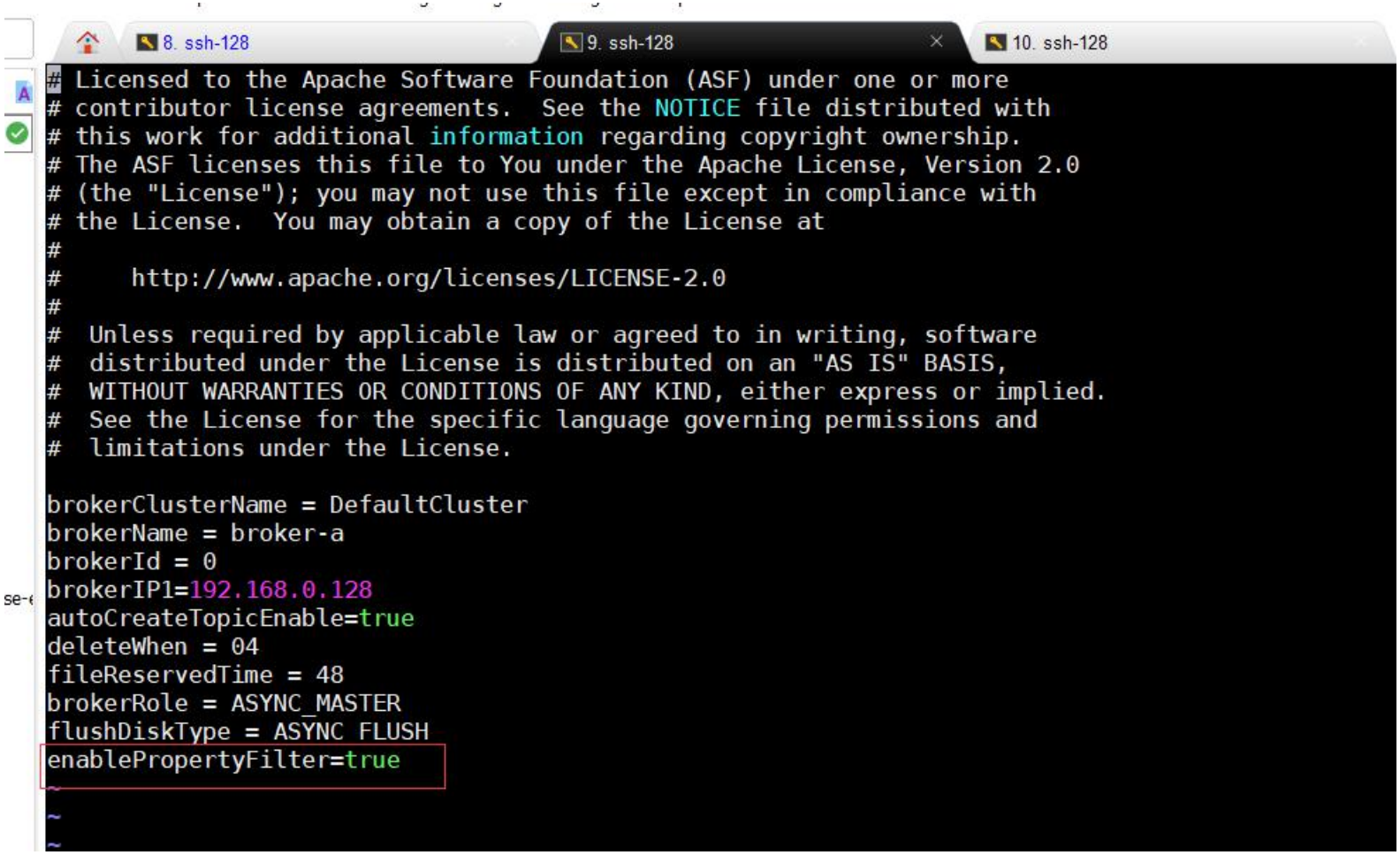
主要支持如下 2 种的过滤方式

(1) Tag 过滤方式：Consumer 端在订阅消息时除了指定 Topic 还可以指定 TAG，如果一个消息有多个 TAG，可以用||分隔。其中，Consumer 端会将这个订阅请求构建成一个 SubscriptionData，发送一个 Pull 消息的请求给 Broker 端。Broker 端从 RocketMQ 的文件存储层—Store 读取数据之前，会用这些数据先构建一个 MessageFilter，然后传给 Store。Store 从 ConsumeQueue 读取到一条记录后，会用它记录的消息 tag hash 值去

做过滤，由于在服务端只是根据 hashcode 进行判断，无法精确对 tag 原始字符串进行过滤，故在消息消费端拉取到消息后，还需要对消息的原始 tag 字符串进行比对，如果不同，则丢弃该消息，不进行消息消费。

(2) SQL92 的过滤方式：这种方式的大致做法和上面的 Tag 过滤方式一样，只是具体过滤过程不太一样，真正的 SQL expression 的构建和执行由 rocketmq-filter 模块负责的。具体使用见 http://rocketmq.apache.org/docs/filter-by-sql92-example/

注意如果开启 SQL 过滤的话，Broker 需要开启参数 enablePropertyFilter=true，然后服务器重启生效。


8. ssh-128
9. ssh-128
10. ssh-128
Licensed to the Apache Software Foundation (ASF) under one or more
contributor license agreements. See the NOTICE file distributed with
this work for additional information regarding copyright ownership.
The ASF licenses this file to You under the Apache License, Version 2.0
(the "License"); you may not use this file except in compliance with
the License. You may obtain a copy of the License at
#
http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
brokerClusterName = DefaultCluster
brokerName = broker-a
brokerId = 0
brokerIP1=192.168.0.128
autoCreateTopicEnable=true
deleteWhen = 04
fileReservedTime = 48
brokerRole = ASYNC_MASTER
flushDiskType = ASYNC_FLUSH
enablePropertyFilter=true

类过滤

新版本（>=4.3.0）已经不支持（代码中 FilterServerConsumer 新版本已经不支持了）

RocketMQ 存储概要设计

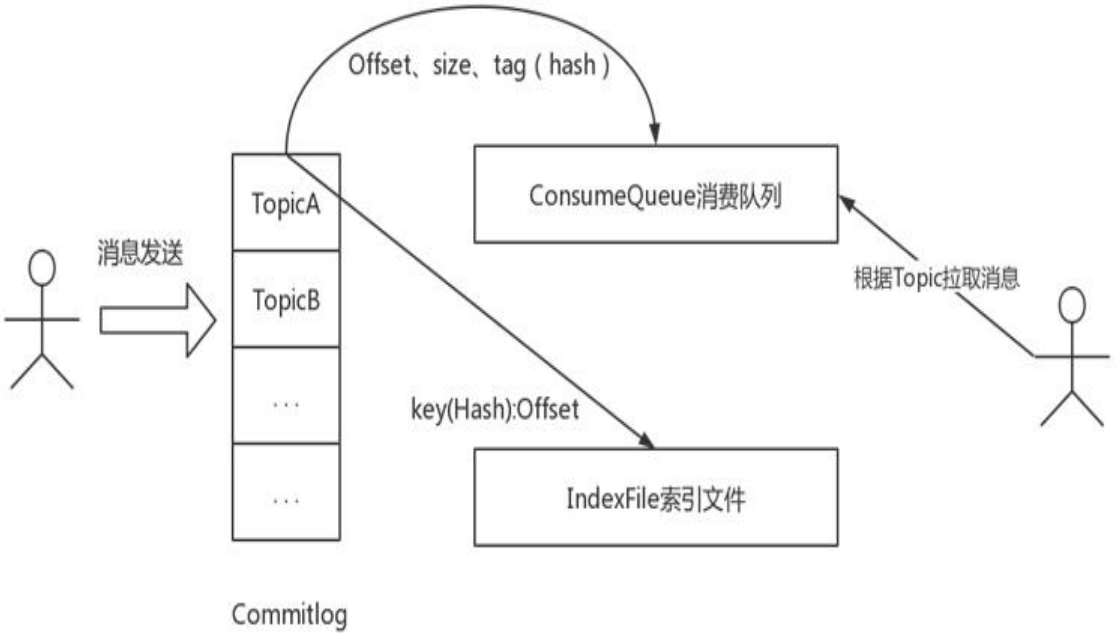
目前的 MQ 中间件从存储模型来 ，分为需要持久化和不需要持久化的两种模型，现在大多数的 是支持持久化存储的，比如 ActiveMQ RabbitMQ Kafka,RocketMQ

ZeroMQ 却不需要支持持久化存储 而业务系统也大多需要 MQ 有持久存储的能力，这样可以大大增加系统的高可用性。

从存储方式和效率来看，文件系统高于 KV 存储， KV 存储又高于关系型数据库，直接操作文件系统肯定是最快的，但如果从可靠性的角度出发直接操作文件系统是最低的，而关系型数据库的可靠性是最高的。

RocketMQ 主要存储的文件包括 Commitlog 文件、 ConsumeQueue 文件、 IndexFile。RocketMQ 将所有主题的消息存储在同一文件，确保消息发送时顺序写文件，尽最大的能力确保消息发送的高性能与高吞吐量。

但由于一般的消息中间件是基于消息主题的订阅机制，这样便给按照消息主题检索消息带来了极大的不便。为了提高消息消费的效率， RocketMQ 引入了 ConsumeQueue 消息队列文件，每个消息主题包含多个消息消费队列，每个消息队列有一个消息文件 IndexFile 索引文件，其主要设计理念就是为了加速消息的检索性能，可以根据消息的属性快速从 Commitlog 文件中检索消息。整体如下：



1 ） CommitLog ： 消息存储文件，所有消息主题的消息都存储在 CommitLog 文件中

2 ） ConsumeQueue ： 消息消费队列，消息到达 CommitLog 文件后，将异步转发到消息消费队列，供消息消费者消费

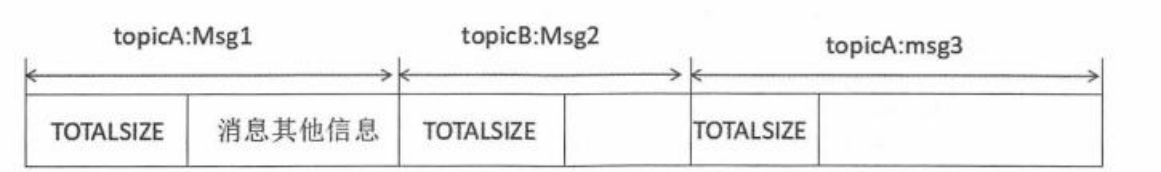
3 ） IndexFile ： 消息索引文件，主要存储消息 Key 与 Offset 的对应关系

消息存储结构

CommitLog

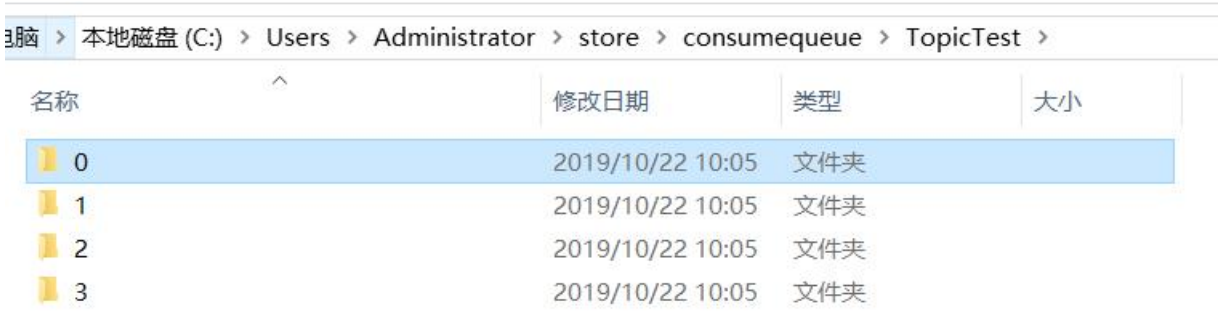
CommitLog 以物理文件的方式存放，每台 Broker 上的 CommitLog 被本机器所有 ConsumeQueue 共享，文件地址：$ {user.home} \store\$ { commitlog} \ $ { fileName}。在 CommitLog 中，一个消息的存储长度是不固定的， RocketMQ 采取一些机制，尽量向 CommitLog 中顺序写 ，但是随机读。commitlog 文件默认大小为 1G ，可通过在 broker 置文件中设置 mapedFileSizeCommitLog 属性来改变默认大小。

Commitlog 文件存储的逻辑视图如下，每条消息的前面 4 个字节存储该条消息的总长度。但是一个消息的存储长度是不固定的。

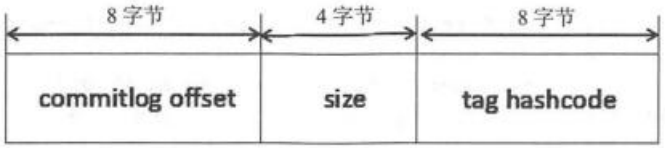


ConsumeQueue

ConsumeQueue 是消息的逻辑队列，类似数据库的索引文件，存储的是指向物理存储的地址。每个 Topic 下的每个 Message Queue 都有一个对应的 ConsumeQueue 文件， 文件地址在$ {$storeRoot} \consumequeue\$ {topicName} \$ { queueId} \$ {fileName}。



ConsumeQueue 中存储的是消息条目，为了加速 ConsumeQueue 消息条目的检索速度与节省磁盘空间，每一个 Consumequeue 条目不会存储消息的全量信息，消息条目如下：



ConsumeQueue 即为 Commitlog 文件的索引文件， 其构建机制是 当消息到达 Commitlog 文件后 由专门的线程产生消息转发任务，从而构建消息消费队列文件（ConsumeQueue ）与下文提到的索引文件。

存储机制这样设计有以下几个好处：

1 ） CommitLog 顺序写 ，可以大大提高写入效率。

（实际上，磁盘有时候会比你想象的快很多，有时候也比你想象的慢很多，关键在如何使用，使用得当，磁盘的速度完全可以匹配上网络的数据传输速度。目前的高性能磁盘，顺序写速度可以达到 600MB/s ，超过了一般网卡的传输速度，这是磁盘比想象的快的地方 但是磁盘随机写的速度只有大概 100KB/s,和顺序写的性能相差 6000 倍！）

2 ）虽然是随机读，但是利用操作系统的 pagecache 机制，可以批量地从磁盘读取，作为 cache 存到内存中，加速后续的读取速度。

3 ）为了保证完全的顺序写，需要 ConsumeQueue 这个中间结构 ，因为 ConsumeQueue 里只存偏移量信息，所以尺寸是有限的，在实际情况中，大部分的 ConsumeQueue 能够被全部读入内存，所以这个中间结构的操作速度很快，可以认为是内存读取的速度。此外为了保证 CommitLog 和 ConsumeQueue 的一致性， CommitLog 里存储了 Consume Queues 、Message Key、 Tag 等所有信息，即使 ConsumeQueue 丢失，也可以通过 commitLog 完全恢复出来。

IndexFile

index 存的是索引文件，这个文件用来加快消息查询的速度。消息消费队列 RocketMQ 专门为消息订阅构建的索引文件 ，提高根据主题与消息检索消息的速度 ，使用 Hash 索引机制，具体是 Hash 槽与 Hash 冲突的链表结构。（这里不做过多解释）

Config

config 文件夹中 存储着 Topic 和 Consumer 等相关信息。主题和消费者群组相关的信息就存在在此。

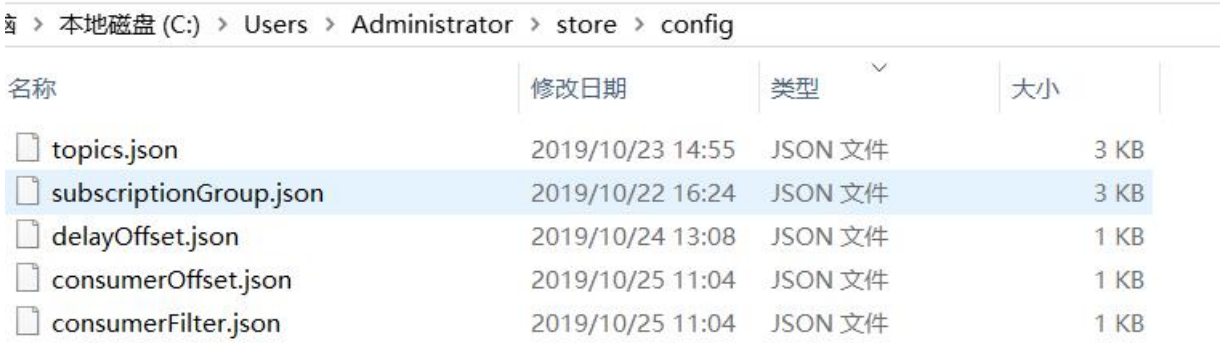
topics.json : topic 配置属性

subscriptionGroup.json :消息消费组配置信息。

delayOffset.json ： 延时消息队列拉取进度。

consumerOffset.json ： 集群消费模式消息消进度。

consumerFilter.json ： 主题消息过滤信息。



其他

abort ： 如果存在 abort 文件说明 Broker 非正常闭，该文件默认启动时创建，正常退出之前删除

checkpoint ： 文件检测点，存储 commitlog 文件最后一次刷盘时间戳、 consumequeue 最后一次刷盘时间、 index 索引文件最后一次刷盘时间戳。

内存映射

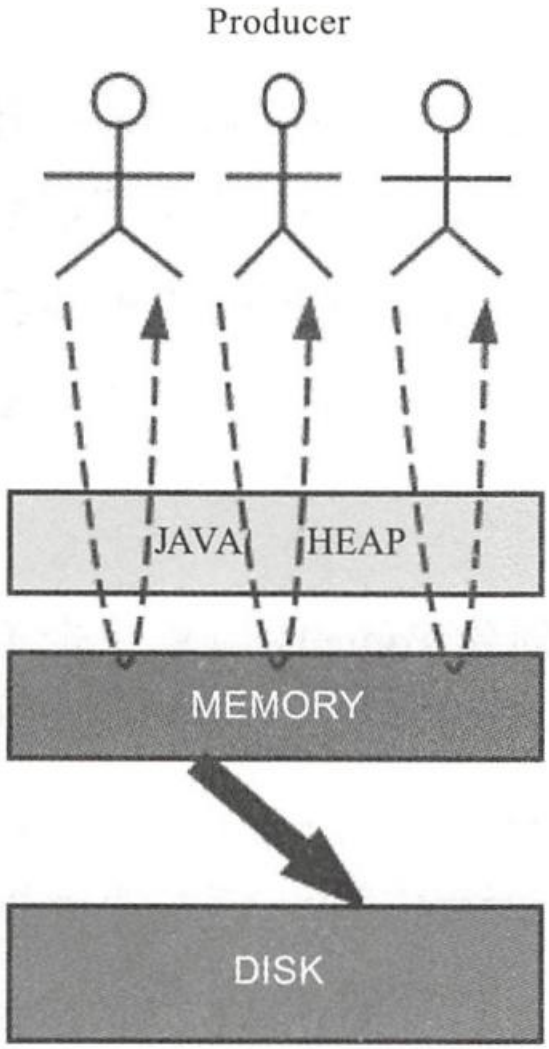
内存映射文件，是由一个文件到一块内存的映射。文件的数据就是这块区域内存中对应的数据，读写文件中的数据，直接对这块区域的地址操作就可以，减少了内存复制的环节。所以说，内存映射文件比起文件 I/O 操作，效率要高，而且文件越大，体现出来的差距越大。

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

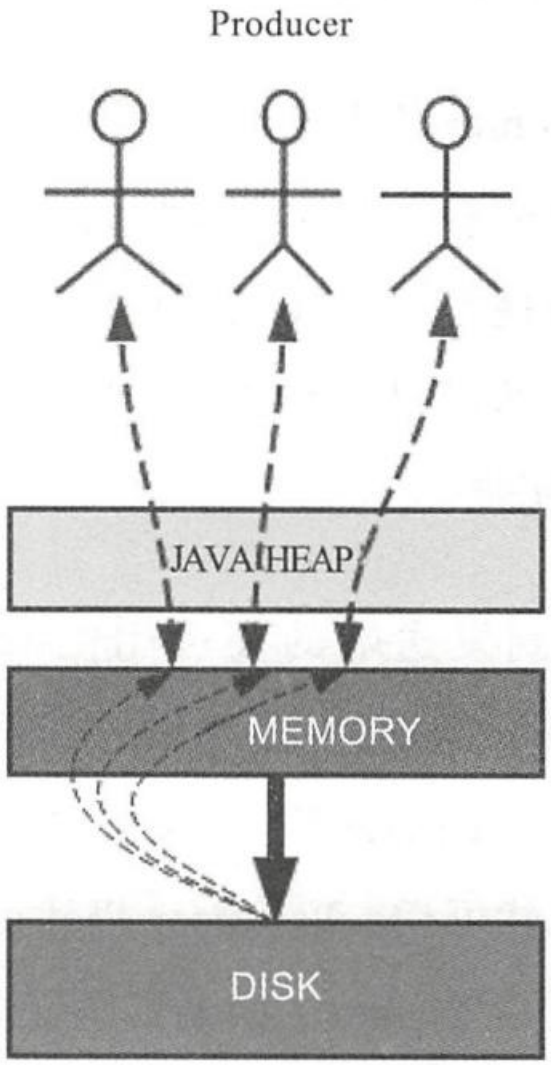
文件刷盘机制

RocketMQ 存储与读写是基于 JDK NIO 的内存映射机制，具体使用 MappedByteBuffer（基于 MappedByteBuffer 操作大文件的方式，其读写性能极高）

RocketMQ 的消息是存储到磁盘上的，这样既能保证断电后恢复，又可以让存储的消息 超出内存的限制 RocketMQ 为了提高性能，会尽可能地保证磁盘的顺序写 消息在通过 Producer 写人 RocketMQ 的时候，有两种写磁盘方式：



同步刷盘



异步刷盘

异步刷盘方式

在返回写成功状态时 ，消息可能只是被写入了内存的 PAGECACHE ，写操作的返回快，吞吐量大 ；当内存里的消息积累到一定程度时，统一触发写磁盘动作，快速写入。

同步刷盘方式

在返回写成功状态时，消息已经被写人磁盘 具体流程是，消息写入内存的 PAGECACHE 后，立刻通知刷盘线程刷盘，然后等待刷盘完成，刷盘线程执行完成后唤醒等待的线程，返回消息写成功的状态

消息存储时首先将消息追加到内存，再根据配值的刷盘 略在不同时间进行刷写磁盘 如果是同步刷盘，消息追加到内存后，将同步调用 MappedByteBuffer force （）方法；如果是异步刷盘，在消息追加到内存后立刻返回给消息发送端 RocketMQ 使用 个单独的线程按照某个设定的频 执行刷盘操作。通过在 broker 配置文件中配置 flushDiskType 来设定刷盘方式，可选值为 ASYNC_FLUSH （异步刷盘）， SYNC_FLUSH 同步刷盘） 默认为异步。

总结

实际应用中要结合业务场景，合理设置刷盘方式，尤其是同步刷盘的方式，由于频繁的触发磁盘写动作， 会明显降低性能。通常情况下，应该把 Rocket 置成异步刷盘方式。

过期文件删除

由于 RocketMQ 操作 CommitLog，ConsumeQueue 文件是基于内存映射机制并在启动的时候会加载 commitlog，ConsumeQueue 目录下的所有文件，为了避免内存与磁盘的浪费，不可能将消息永久存储在消息服务器上，所以需要引入一种机制来删除已过期的文件。

删除过程分别执行清理消息存储文件（ Commitlog ）与消息消费 队列文件（ ConsumeQueue 文件）， 消息消费队列文件与消息存储文件（ Commitlog ）共用一套过期文件机制。

RocketMQ 清除过期文件的方法是 ：如果非当前写文件在一定时间间隔内没有再次被更新，则认为是过期文件，可以被删除， RocketMQ 不会关注这个文件上的消息是否全部被消费。默认每个文件的过期时间为 42 小时（不同版本的默认值不同，这里以 4.4.0 为例） ，通过在 Broker 配置文件中设置 fileReservedTime 来改变过期时间，单位为小时。

触发文件清除操作的是一个定时任务，而且只有定时任务，文件过期删除定时任务的周期由该删除决定，默认每 10s 执行一次。

过期判断

文件删除主要是由这个配置属性：fileReservedTime：文件保留时间。也就是从最后一次更新时间到现在，如果超过了该时间，则认为是过期文件，可以删除。

另外还有其他两个配置参数：

deletePhysicFilesInterval：删除物理文件的时间间隔（默认是 100MS），在一次定时任务触发时，可能会有多个物理文件超过过期时间可被删除，因此删除一个文件后需要间隔 deletePhysicFilesInterval 这个时间再删除另外一个文件，由于删除文件是一个非常耗费 IO 的操作，会引起消息插入消费的延迟（相比于正常情况下），所以不建议直接删除所有过期文件。

destroyMapedFileIntervalForcibly：在删除文件时，如果该文件还被线程引用，此时会阻止此次删除操作，同时将该文件标记不可用并且纪录当前时间戳 destroyMapedFileIntervalForcibly 这个表示文件在第一次删除拒绝后，文件保存的最大时间，在此时间内一直会被拒绝删除，当超过这个时间时，会将引用每次减少 1000，直到引用 小于等于 0 为止，即可删除该文件.

删除条件

1）指定删除文件的时间点， RocketMQ 通过 deleteWhen 设置一天的固定时间执行一次。删除过期文件操作， 默认为凌晨 4 点。

2）磁盘空间是否充足，如果磁盘空间不充足(DiskSpaceCleanForciblyRatio。磁盘空间强制删除文件水位。默认是 85)，会触发过期文件删除操作。

另外还有 RocketMQ 的磁盘配置参数：

1:物理使用率大于 diskSpaceWarningLevelRatio（默认 90%可通过参数设置）,则会阻止新消息的插入。

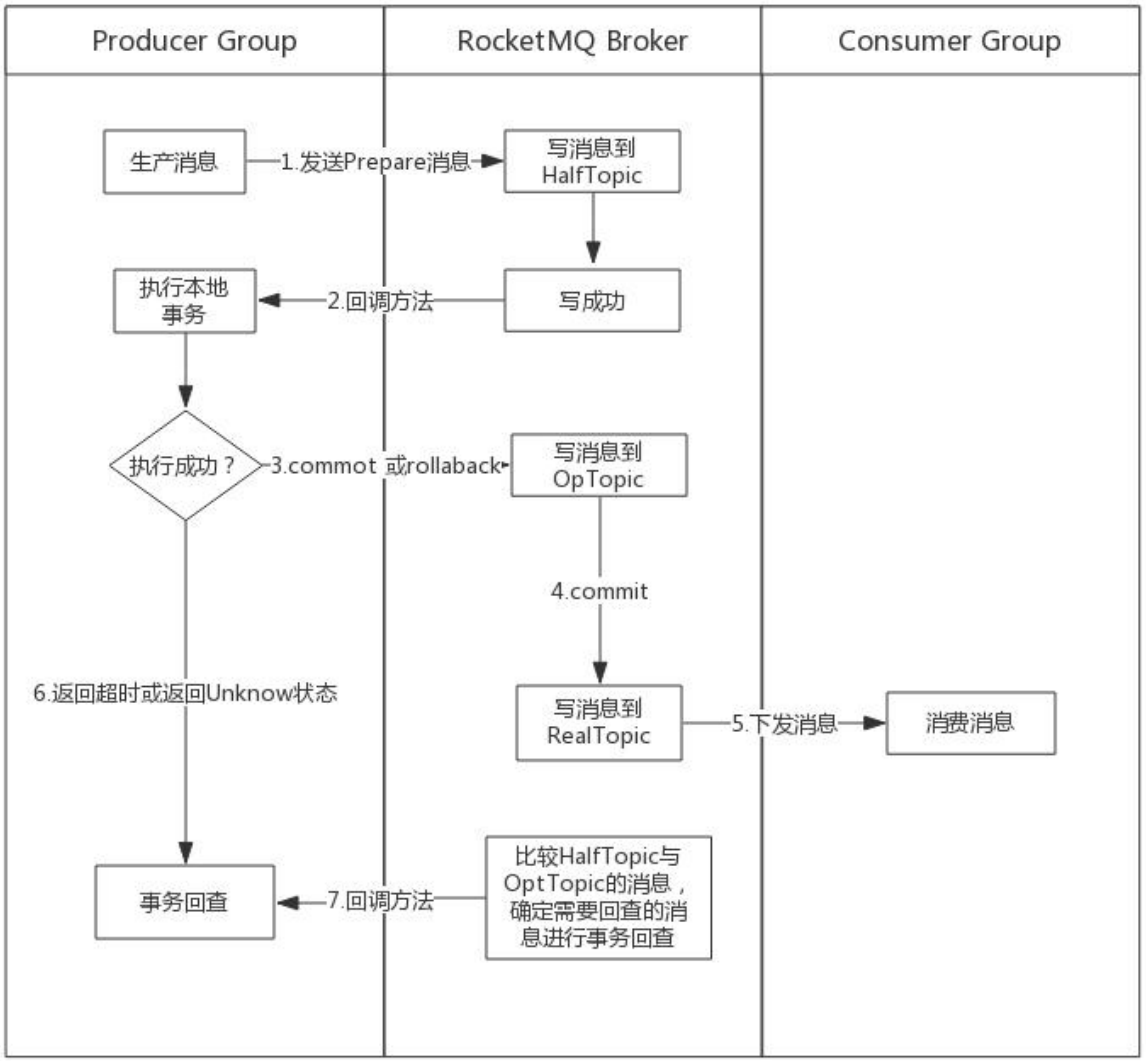
2:物理磁盘使用率小于 diskMaxUsedSpaceRatio(默认 75%) 表示磁盘使用正常。

RocketMQ 中的事务消息

事务消息实现思想

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

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


Producer Group
RocketMQ Broker
Consumer Group
生产消息
1.发送Prepare消息
写消息到HalfTopic
执行本地事务
2.回调方法
写成功
执行成功？
3.commot 或rollaback
写消息到OpTopic
4.commit
6.返回超时或返回Unknow状态
写消息到RealTopic
5.下发消息
消费消息
事务回查
7.回调方法
比较HalfTopic与OptTopic的消息，确定需要回查的消息进行事务回查

1）发送方向 RocketMQ 发送“待确认”（Prepare）消息。

2) RocketMQ 将收到的“待确认”(一般写入一个 HalfTopic 主题<RMQ_SYS_TRANS_HALF_TOPIC>)消息持化成功后， 向发送方回复消息已经发送成功，此时第一阶段消息发送完成。

发送方开始执行本地事件逻辑.

3）发送方根据事件执行结果向 RocketMQ 发送二次确认（ Commit 还是 Rollback）消息 RocketMQ 收到 Commit 则将第一阶段消息标记为可投递(这些消息才会进入生产时发送实际的主题 RealTopic)，订阅方将能够收到该消息；收到 Rollback 状态则删除第一阶段的消息，订阅方接收不到该消息。

4）如果出现异常情况，步骤 3 提交的二次确认最终未到达 RocketMQ,服务器在经过固定时间段后将对“待确认”消息、发起回查请求.

5）发送方收到消息回查请求后(如果发送一阶段消息的 Producer 不能工作，回查请求将被发送到和 Producer 在同一个 Group 里的其他 Producer)，通过检查对应消息的本地事件执行结果返回 Commit Roolback 状态。

两阶段提交

提交半事务是一个阶段，提交全事务和事务回查是另外一个阶段，所以称之为两阶段提交。

事务状态回查机制

RocketMQ 通过 TransactionalMessageCheckService 线程定时去检测 RMQ_SYS_ TRANS_ HALF_TOPIC 主题中的消息，回查消息的事务状态 TransactionalMessageCheckService 的检测频率默认为 1 分钟，可通过在 broker.conf 文件中设置 transactionCheckInterval 来改变默认值，单位为毫秒。

代码实现

cn.enjoyedu.transaction 包中

LocalTransactionState 枚举类，

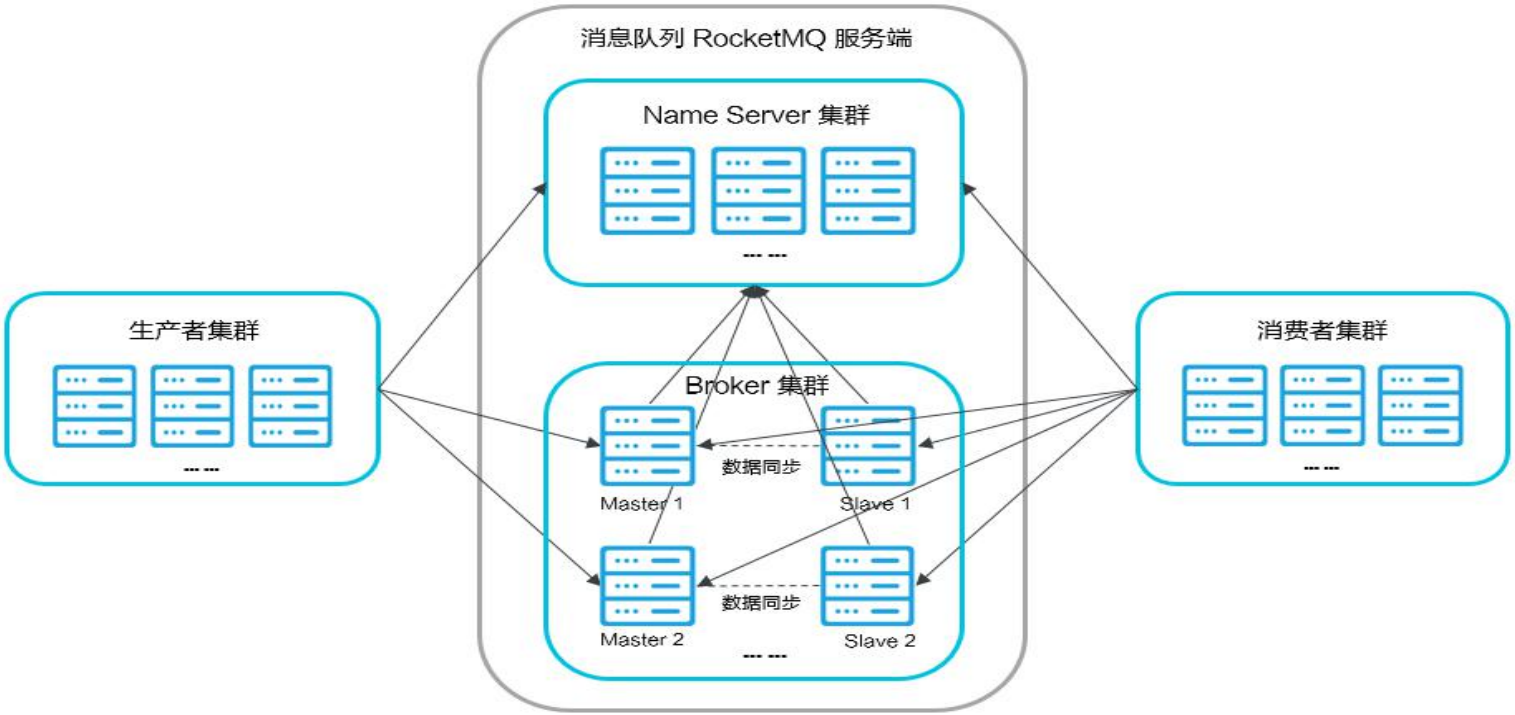
COMMIT_MESSAGE 提交消息，即 broker 确认了这条消息的正确性之后执行提交，标记这条消息可被消费，这样的话 consumer 就可以正常消费这条消息了；

ROLLBACK_MESSAGE 回滚消息，意思是当我们的本地主事务发生异常的时候，回滚本地事务的同时，同样需要一种方法通知到 rocketMq 不要继续发送消息了，当 broker 收到这个命令时候就会标记消息为 rollBack 的状态，consumer 就不能收到了

UNKNOW 消息回查的状态

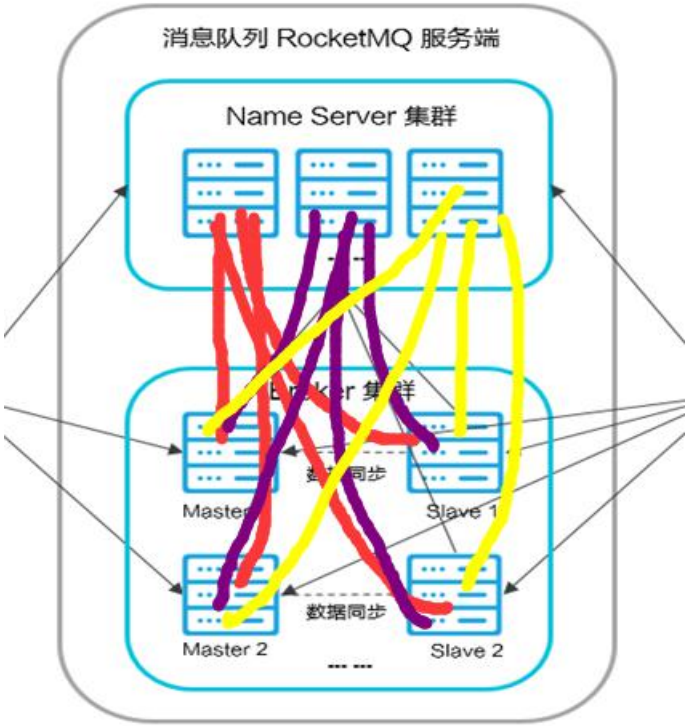
RocketMQ 主从同步(HA)机制

高可用(HA 机制)特性是目前分布式系统中必备的特性之一,对一个中间件来说没有 HA 机制必然是一个重大的缺陷。RocketMQ 的 Broker 分为 Master（主）和 Slave（从） 两个角色，为了保证高可用性，Master 角色的机器接收到消息后 ，要把内容同步到 Slave 机器上，这样一旦 Master 宕机，Slave 机器依然可以提供服务。这个就是 RocketMQ 实现高可用(HA 机制)的原理。



为了提高消息消费的高可用性，避免 Broker 发生单点故障引起存储在 Broker 上的消息无法及时消费， RocketMQ 引入了 Broker 主从机制：即消息消费到达主服务器（Master）后需要消息同步到消息从服务器（Slave），如果主服务器 Broker 宕机后，消息消费者可以从从服务器拉取消息。

同时 RocketMQ 依赖 NameServer, 所以为了确保高可用，同时要确保 NameServer 的高可用，一般通过部署多台 NamesServer 服务器来实现，但彼此之间互不通信，也就是 NameServer 务器之间在某一时刻的数据并不会完全相同，但这对消息发送不会造成任何影响，这也是 RocketMQ NameServer 设计的一个亮点。


消息队列 RocketMQ 服务端
Name Server 集群
Master
Slave 1
数据同步
Master 2
数据同步
Slave 2

RocketMQ 集群部署模式

集群部署模式

1）单 master 模式

也就是只有一个 master 节点，称不上是集群，一旦这个 master 节点宕机，那么整个服务就不可用。

2）多 master 模式

多个 master 节点组成集群，单个 master 节点宕机或者重启对应用没有影响。

优点：所有模式中性能最高

缺点：单个 master 节点宕机期间，未被消费的消息在节点恢复之前不可用，消息的实时性就受到影响。

注意：使用同步刷盘可以保证消息不丢失，同时 Topic 相对应的 queue 应该分布在集群中各个节点，而不是只在某各节点上，否则，该节点宕机会对订阅该 topic 的应用造成影响。

3）多master 多 slave 异步复制模式

在多 master 模式的基础上，每个 master 节点都有至少一个对应的 slave。master 节点可读可写，但是 slave 只能读不能写，类似于 mysql 的主备模式。

优点： 一般情况下都是 master 消费，在 master 宕机或超过负载时，消费者可以从 slave 读取消息，消息的实时性不会受影响，性能几乎和多 master 一样。

缺点：使用异步复制的同步方式有可能会有消息丢失的问题。

4）多 master 多 slave 同步双写模式

同多 master 多 slave 异步复制模式类似，区别在于 master 和 slave 之间的数据同步方式。

优点：同步双写的同步模式能保证数据不丢失。

缺点：发送单个消息响应时间会略长，性能相比异步复制低 10%左右。

同步方式：同步双写和异步复制（指的一组 master 和 slave 之间数据的同步）

刷盘策略：同步刷盘和异步刷盘（指的是节点自身数据是同步还是异步存储进入磁盘）

注意：对数据要求较高的场景，建议的持久化策略是主 broker 和从 broker 采用同步复制方式，主从 broker 都采用异步刷盘方式。通过同步复制方式，保存数据热备份，通过异步刷盘方式，保证 rocketMQ 高吞吐量。

多主模式与数据重复

多主模式，rocketmq 生产者发送一条消息，只会写入到一台 broker 的一个 queue 中，所以有几台 master 跟消息会不会重复没有直接关系，两台 master 就是 8 个 queue,生产者一条消息只会写入其中一个 queue，双主模式下，一台主挂了后 nameserver 能感知到，而生产者从 nameserver 能获取到可用的 broker 信息，就会将消息写入另一台可用的 broker。

安装部署过程

RocketMQ 提供了初始的集群部署模式下的配置文件，如下图：

安装过程见：

主从复制原理

RocketMQ 主从同步（HA）实现过程如下：

1)主服务器启动，并在特定端口上监听从服务器的连接。

2)从服务器主动连接主服务器，主服务器接受客户端的连接，并建立相关 TCP 连接。

3)从服务器主动向服务器发送待拉取消息偏移 ， 主服务器解析请求并返回消息给从服务器。

4)从服务器保存消息并继续发送新的消息同步请求。

核心实现

从服务器在启动的时候主动向主服务器建立 TCP 长连接，然后获取服务器的 commitlog 最大偏移，以此偏移向主服务器主动拉取消息，主服务器根据偏移量，与自身 commitlog 文件的最大偏移进行比较，如果大于从服务器 commitlog 偏移 ，主服务器将向从服务器返回一定数量的消息， 该过程循环进行 ，达到主从服务器数据同步。

读写分离机制

RocketMQ 读写分离与他中间件的实现方式完全不同， RocketMQ 是消费者首先服务器发起拉取消息请求，然后主服务器返回一批消息，然后会根据主服务器负载压力与主从同步情况，向从服务器建议下次消息拉取是从主服务器还是从从服务器拉取。

那消息服务端是根据何种规则来建议哪个消息消费队列该从哪台 Broker 服务器上拉取消息呢？

一般都是从主服务器拉取，如果主阶段拉取的消息已经超出了常驻内存的大小，表示主服务器繁忙，此时从从服务器拉取。

如果主服务器繁忙则建议下 次从从服务器拉取消息，设置 suggestWhichBrokerId 配置文件中 whichBrokerWhenConsumeSlowly 属性，默认为 1。如果一个 Master 拥有多台 Slave 服务器，参与消息拉取负载的从服务器只会是其中一个。

与 Spring 集成

具体代码实现，参见 rocket-with-spring

pom 文件

```
<!--RocketMQ-->
    <dependency>
      <groupId>org.apache.rocketmq</groupId>
      <artifactId>rocketmq-client</artifactId>
      <version>4.4.0</version>
</dependency>
```

生产者

applicationContext.xml

```
<!-- 生产者配置 -->
<bean id="rocketMQProducer" class="cn.enjoyedu.producer.RocketMQProducer"
      init-method="init" destroy-method="destroy">
    <property name="producerGroup" value="ProducerGroup" />
    <property name="namesrvAddr" value="192.168.0.128:9876" />
</bean>
```

消费者

applicationContext.xml 中使用监听器的方式

```xml
<!-- 消费者监听1 -->
<bean id="messageListeners" class="cn.enjoyedu.listener.MessageListenerImpl"></bean>

<!-- 消费者监听2 -->
<bean id="bmessageListeners" class="cn.enjoyedu.listener.BMessageListenerImpl"></bean>
<!-- 消费者配置1 -->
<bean id="rocketmqConsumer" class="org.apache.rocketmq.client.consumer.DefaultMQPushConsumer"
      init-method="start" destroy-method="shutdown">
    <property name="consumerGroup" value="ConsumerGroup" />
    <property name="namesrvAddr" value="192.168.0.128:9876" />
    <property name="messageListener" ref="messageListeners" />
    <property name="subscription">
        <map>
            <entry key="rocket-spring-topic" value="TAG1" />
        </map>
    </property>
</bean>
<!-- 消费者配置2 -->
<bean id="rocketmqConsumer2" class="org.apache.rocketmq.client.consumer.DefaultMQPushConsumer"
      init-method="start" destroy-method="shutdown">
    <property name="consumerGroup" value="ConsumerGroup2" />
    <property name="namesrvAddr" value="192.168.0.128:9876" />
    <property name="messageListener" ref="bmessageListeners" />
    <property name="subscription">
        <map>
            <entry key="rocket-spring-topic-b" value="TAG1" />
        </map>
    </property>
</bean>
```

与 SpringBoot 集成

具体代码实现，参见 rocket-with-springboot

跟 Spring 和原生非常类似

限时订单实战

什么是限时订单

在各种电商网站下订单后会保留一个时间段，时间段内未支付则自动将订单状态设置为已过期，这种订单称之为限时订单。

如何实现限时订单

限时订单的流程

电商平台都会包含以下 5 种状态。

待付款：代表买家下单了但是还没有付款。

待发货：代表买家付款了卖家还没有发货。

已发货：代表卖家已经发货并寄出商品了。

已完成：代表买家已经确认收到货了。

已关闭：代表订单过期了买家也没付款、或者卖家关闭了订单。

限时订单实现的关键

我们可以看到，订单中的很多状态都是可以用户触发的，唯独订单过期了买家也没付款我们需要自动的把订单给关闭，这个操作是没有用户或者是人工干预的，所以限时订单的关键就是如何检查订单状态，如果订单过期了则把该订单设置为关闭状态。

轮询数据库？

轮询数据库在实现限时订单上是可行的，而且实现起来很简单。写个定时器去每隔一段时间扫描数据库，检查到订单过期了，做适当的业务处理。

但是轮询会带来什么问题？

1、轮询大部分时间其实是在做无用功，我们假设一张订单是 45 分钟过期，每 1 分钟我们扫描一次，对这张订单来说，要扫描 45 次以后，才会检查到这张订单过期，这就意味着数据库的资源（连接，IO）被白白浪费了；

2、处理上的不及时，一个待支付的电影票订单我们假设是 12:00:35 过期，但是上次扫描的时间是 12:00:30，那么这个订单实际的过期时间是什么时候？12:01:30，和我本来的过期时间差了 55 秒钟。放在业务上，会带来什么问题？这张电影票，假设是最后一张，有个人 12:00:55 来买票，买得到吗？当然买不到了。那么这张电影票很有可能就浪费了。如果缩短扫描的时间间隔，第一只能改善不能解决，第二，又会对数据库造成更大的压力。

那么我们能否有种机制，不用定时扫描，当订单到期了，自然通知我们的应用去处理这些到期的订单呢？

Java 本身的提供的解决方案

java 其实已经为我们提供了问题的方法。我们想，要处理限时支付的问题，肯定是要有个地方保存这些限时订单的信息的，意味着我们需要一个容器，于是我们在 Java 容器中去寻找。Map? List? Queue?

看看 java 为我们提供的容器，我们是个多线程下的应用，会有多个用户同时下订单，所以所有并发不安全的容器首先被排除，并发安全的容器有哪些？一一排除，很巧，java 在阻塞队列里为我们提供了一种叫延迟队列 delayQueue 的容器，刚好可以为我们解决问题。

DelayQueue： 阻塞队列（先进先出）

1）支持阻塞的插入方法：意思是当队列满时，队列会阻塞插入元素的线程，直到队列不满。

2）支持阻塞的移除方法：意思是在队列为空时，获取元素的线程会等待队列变为非空。

延迟期满时才能从中提取元素（光队列里有元素还不行）。

Delayed 接口使对象成为延迟对象，它使存放在 DelayQueue 类中的对象具有了激活日期。该接口强制实现下列两个方法。

- CompareTo(Delayed o)：Delayed 接口继承了 Comparable 接口，因此有了这个方法。让元素按激活日期排队
- getDelay(TimeUnit unit):这个方法返回到激活日期的剩余时间，时间单位由单位参数指定。

阻塞队列更多详情，参考 VIP 课程《并发编程》

架构师应该多考虑一点！

架构师在设计和实现系统时需要考虑些什么？

功能，这个没什么好说，实现一个应用，连基本的功能都没实现，要这个应用有何用？简直就是“一顿操作猛如虎，一看战绩零比五”

高性能，能不能尽快的为用户提供服务和能为多少用户同时提供服务，性能这个东西是个很综合性的东西，从前端到后端，从架构（缓存机制、异步机制）到 web 容器、数据库本身再到虚拟机到算法、java 代码、sql 语句的编写，全部都对性能有影响。如何提升性能，要建立在充分的性能测试的基础上，然后一个个的去解决性能瓶颈。对我们今天的应用来讲，我们不想去轮询数据库，其实跟性能有非常大的关系。

高可用，应用正确处理业务，服务用户的时间，这个时间当然是越长越好，希望可以 7*24 小时。而且哪怕服务器出现了升级，宕机等等情况下，能够以最短的时间恢复，为用户继续服务，但是实际过程中没有哪个网站可以说做到 100%，不管是 Google，FaceBook，阿里，腾讯，一般来说可以做到 99.99%的可用性，已经是相当厉害了，这个水平大概就是一个服务在一年可以做到只有 50 分钟不可用。这个需要技术、资金、技术人员的水平和责任心，还要运气。

高伸缩，伸缩性是指通过不断向集群中加入服务器的手段来缓解不断上升的用户并发访问压力和不断增长的数据存储需求。就像弹簧一样挂东西一样，用户多，伸一点，用户少，缩一点。衡量架构是否高伸缩性的主要标准就是是否可用多台服务器构建集群，是否容易向集群中添加新的服务器。加入新的服务器后是否可以提供和原来服务器无差别的服务。集群中可容纳的总的服务器数量是否有限制。

高扩展，的主要标准就是在网站增加新的业务产品时，是否可以实现对现有产品透明无影响，不需要任何改动或者很少改动既有业务功能就可以上线新产品。比如购买电影票的应用，用户购买电影票，现在我们要增加一个功能，用户买了票后，随机抽取用户送限量周边。怎么做到不改动用户下订单功能的基础上增加这个功能。熟悉设计模式的同学，应该很眼熟，这是设计模式中的开闭原则（对扩展开放，对修改关闭）在架构层面的一个原则。

普利兹克奖（建筑界的诺贝尔奖之称）2016 年该奖得主：48 岁的智利建筑师亚历杭德罗·阿拉维纳。他设计的房子对扩展开放，对修改关闭。

从系统可用性角度考虑

应用重启带来的问题：

保存在 Queue 中的订单会丢失，这些丢失的订单会在什么时候过期，因为队列里已经没有这个订单了，无法检查了，这些订单就得不到处理了。

已过期的订单不会被处理，在应用的重启阶段，可能会有一部分订单过期，这部分过期未支付的订单同样也得不到处理，会一直放在数据库里，过期未支付订单所对应的资源比如电影票所对应的座位，就不能被释放出来，让别的用户来购买。

解决之道 ：在系统启动时另行处理

从系统伸缩性角度考虑

集群化了会带来什么问题？应用之间会相互抢夺订单，特别是在应用重启的时候，重新启动的那个应用会把不属于自己的订单，也全部加载到自己的队列里去，一是造成内存的浪费，二来会造成订单的重复处理，而且加大了数据库的压力。

解决方案

让应用分区处理

1、 给每台服务器编号，然后在订单表里登记每条订单的服务器编号；2，更简单的，在订单表里登记每台服务器的 IP 地址，修改相应的 sql 语句即可。

几个问题：如果有一台服务器挂了怎么办？运维吃干饭的吗？服务器挂了赶紧启动啊。如果是某台服务器下线或者宕机，起不来怎么搞？这个还是还是稍微有点麻烦，需要人工干预一下，手动把库里的每条订单数据的服务器编号改为目前正常的服务器的编号，不过也就是一条 sql 语句的事，然后想办法让正常的服务器进行处理（重启正常的服务器）。

用 RocketMQ 实现限时订单

引入 RocketMQ 呢,使用延时消息，一举解决我们限时订单的伸缩性和扩展性问题

延时消息

概念介绍

延时消息：Producer 将消息发送到消息队列 RocketMQ 服务端，但并不期望这条消息立马投递，而是延迟一定时间后才投递到 Consumer 进行消费，该消息即延时消息。

适用场景

消息生产和消费有时间窗口要求：比如在电商交易中超时未支付关闭订单的场景，在订单创建时会发送一条延时消息。这条消息将会在 30 分钟以后投递给消费者，消费者收到此消息后需要判断对应的订单是否已完成支付。 如支付未完成，则关闭订单。如已完成支付则忽略。

核心的代码部分

整个代码见 delayOrder 包

1、 保存订单 SaveOrder.java 的时候，作为生产者往消息队列里推入订单，核心 RocketMQProducer，这个类当然是要继承 IDelayOrder，同时也是 RocketMQ 的生产者。

2、 消息队列会把延时的订单发给消费者 MessageListenerImpl，它是一个 RocketMQ 的消费者监听，它来负责检查订单是否过期，有消息过来，证明消息订单过期了，则把订单状态修改为过期订单。

RocketMQ 本身又如何保证可用性和伸缩性？这个就需要 RocketMQ 的主从同步(HA 机制)。

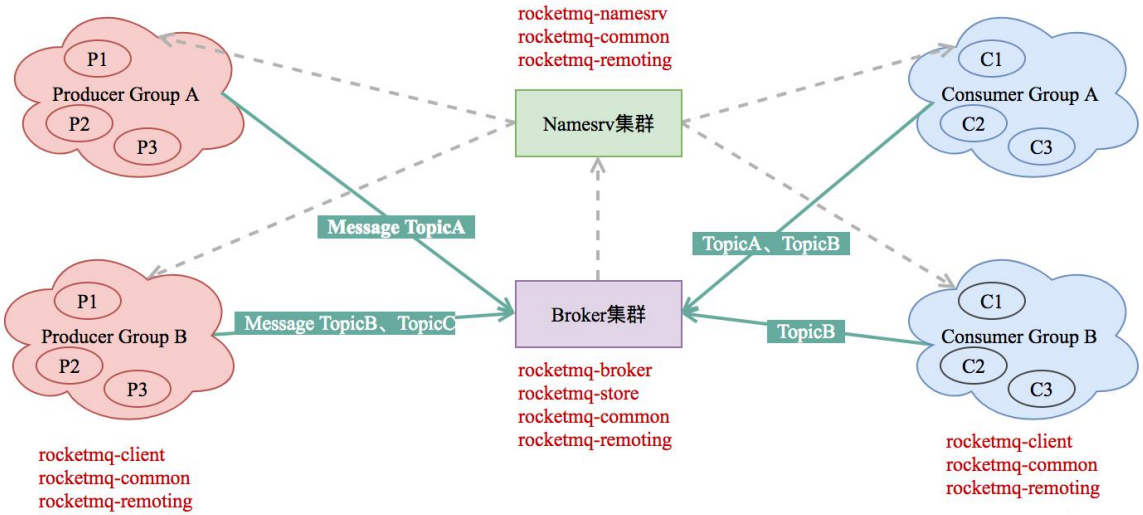
RocketMQ 源码分析

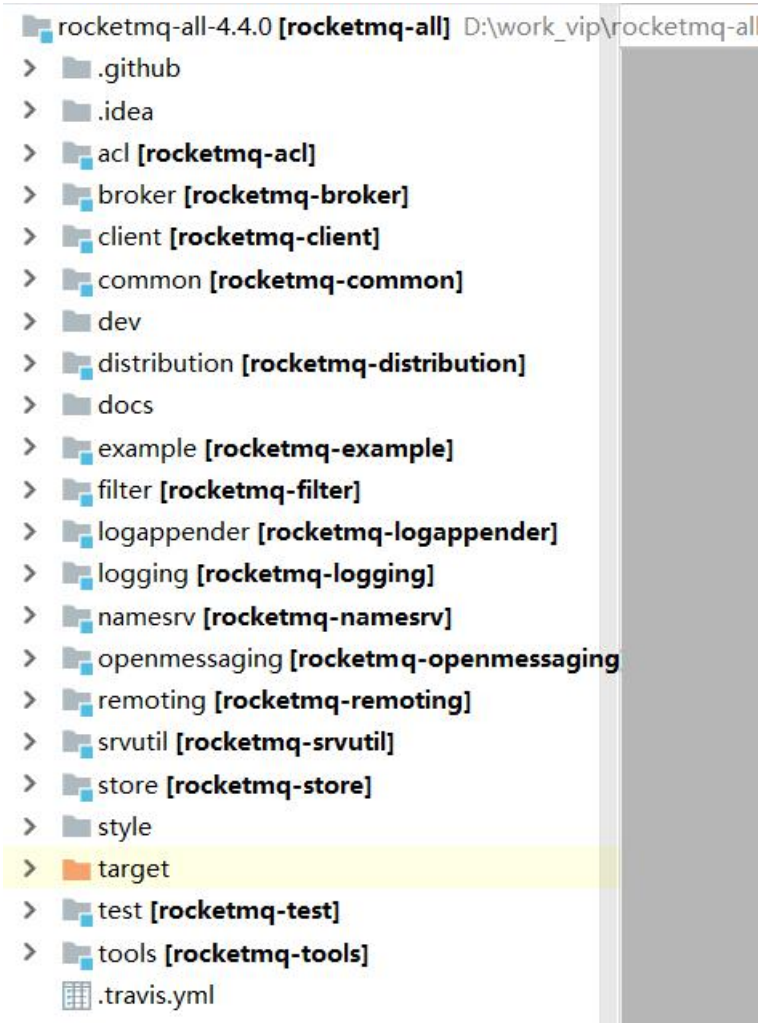
使用的 RocketMQ 版本是 4.4.0,鉴于 RocketMQ 通信模块的底层源码是 Netty 实现的，在学习 RocketMQ 的源码之前，建议读者先对 Netty 的多线程模型、JAVA NIO 模型均有一定的了解，以便快速理解 RocketMQ 源码。

RocketMQ 源码安装和调试见《RocketMQ 源码安装和调试.docx》

RocketMQ 整体架构

RocketMQ 主要的功能集中在 NameServer、rocketmq-broker、rocketmq-remoting、rocketmq-store 4 个模块


rocketmq-namesrv
rocketmq-common
rocketmq-remoting
P1
Producer Group A
P2
P3
Namesrv集群
C1
Consumer Group A
C2
C3
Message TopicA
TopicA、TopicB
P1
Producer Group B
P2
P3
Message TopicB、TopicC
Broker集群
TopicB
C1
Consumer Group B
C2
C3
rocketmq-broker
rocketmq-store
rocketmq-common
rocketmq-remoting
rocketmq-client
rocketmq-common
rocketmq-remoting
rocketmq-client
rocketmq-common
rocketmq-remoting



图中名字对应每一个工程中的 artifactId。

整体模块如下：

1. **rocketmq-namesrv：**
   命名服务，更新和路由发现 broker 服务。
   NameServer 要作用是为消息生产者、消息消费者提供关于主题 Topic 的路由信息，NameServer 除了要存储路由的基础信息，还要能够管理 Broker 节点，包括路由注册、路由删除等功能
2. **rocketmq-broker：**
   mq 的核心。
   它能接收 producer 和 consumer 的请求，并调用 store 层服务对消息进行处理。
   HA 服务的基本单元，支持同步双写，异步双写等模式。
3. **rocketmq-store：**
   存储层实现，同时包括了索引服务，高可用 HA 服务实现。
4. **rocketmq-remoting：**
   基于 netty 的底层通信实现，所有服务间的交互都基于此模块。
5. **rocketmq-common：**
   一些模块间通用的功能类，比如一些配置文件、常量。
6. **rocketmq-client：**
   java 版本的 mq 客户端实现
7. **rocketmq-filter：**
   消息过滤服务，相当于在 broker 和 consumer 中间加入了一个 filter 代理。
8. **rocketmq-srvutil：**
   解析命令行的工具类 ServerUtil。
9. **rocketmq-tools：**
   mq 集群管理工具，提供了消息查询等功能

NameServer

RocketMQ 服务启动

这个类是服务启动时执行，初始化了发送消息、消费消息、清理过期请求等各种线程池和监听事件。

了解了 mq 服务启动的过程，接下来，我们按照一条消息从客户端发出，最终到服务端的存储层并如何落盘，这一条调用链来分析源码，了解一条消息是怎么处理的。

源码分析之消息的来龙去脉

RocketMQ 是一个消息中间件，消息中间件最大的功能就是处理消息，所以我们从消息的角度来做一次源码分析，分析消息的来龙去脉。因为 RocketMQ 本身解耦的，我们从两个独立的部分，消息的生产和消息的消费两大部分入手。

消息的生产

Client 中的消息发送

源码跟踪

前面讲过生产者发送有，单向发送，可靠同步发送和可靠异步发送，我们分析消息可靠同步发送的接口代码。

DefaultMQProducer.send() -> DefaultMQProducerImpl.send() -> DefaultMQProducerImpl.sendDefaultImpl()

核心分析 sendDefaultImpl 方法：

1. 获取主题路由相关信息
2. for 循环发送(发送次数由 retryTimesWhenSendFailed+1 来决定)
3. 调用 sendKernelImpl 方法，下面详细分析 sendKernelImpl 方法

   3.1 获取路由表信息（如果没有则会从 NameServer 中获取）

3.2 通过判断发送类型设置不同的入参，但是最终都调用了 MQClientAPIImpl 类的 sendMessage 方法。下面详细分析同步调用的 sendMessage 方法

3.2.1 sendMessageSync ->NettyRemotingClient.invokeSync()方法完成发送。

在 NettyRemotingAbstract 中的 invokeSyncImpl 里面会大量使用 Netty 进行调用（Netty 的版本是 4.0.42.Final）

4. 不同发送方式的 sendResult 处理不同

核心关键点

消息重试：为什么 RocketMQ 中的消息重试是 2?

就是消息一般情况下发送三次。（King 老师个人认为跟中国文化有关，事不过三）

Netty: 这块 Netty 发送的 详情见《网络协议和 Netty》专题

为何要使用 Netty 作为高性能的通信库？

（1）Netty 的编程 API 使用简单，开发门槛低，无需编程者去关注和了解太多的 NIO 编程模型和概念；

（2）对于编程者来说，可根据业务的要求进行定制化地开发，通过 Netty 的 ChannelHandler 对通信框架进行灵活的定制化扩展；

（3）Netty 框架本身支持拆包/解包，异常检测等机制，让编程者可以从 JAVA NIO 的繁琐细节中解脱，而只需要关注业务处理逻辑；

（4）Netty 解决了（准确地说应该是采用了另一种方式完美规避了）JDK NIO 的 Bug（Epoll bug，会导致 Selector 空轮询，最终导致 CPU 100%）；

（5）Netty 框架内部对线程，selector 做了一些细节的优化，精心设计的 reactor 多线程模型，可以实现非常高效地并发处理；

（6）Netty 已经在多个开源项目（Hadoop 的 RPC 框架 avro 使用 Netty 作为通信框架）中都得到了充分验证，健壮性/可靠性比较好。

总结

客户端发送消息流程比较简单，首先封装消息，然后根据 NameServer 返回的路由信息，然后把这些组成一个整体，最后调用 Remoting 模块使用 Netty 把消息发送给 Broker。在里面包含了多种发送方式，同时也有消息重新发送机制。

Broker 中消息的生产

因为在 Broker 启动流程中涉及到了非常复杂的封装，这里没有必要进行讲解，我们简单想一想，Broker 最核心的功能就是接收到消息然后把消息进行存储，那么我们就 Broker 中对于消息的处理流程进行分析。

源码跟踪

发送的消息到达 broker,调用 org.apache.rocketmq.broker.processor.SendMessageProcessor 类的 processRequest()方法,processRequest()调用 sendMessag()

1.非批次发送消息 sendMessag()

2.消息存储（注意，这里都只存储 commitlog），调用 DefaultMessageStore.putMessage()方法

2.1 这里进行 commitlog 的提交，调用 CommitLog.putMessage()

2.1.1 在 MappedFile 类中，处理存储都是使用 MappedFile 这个类进行处理的，最终调用 appendMessage 方法。

appendMessagesInner 方法中，这里进行文件的追加（AppendMessageCallback 接口的实现 DefaultAppendMessageCallback 在 CommitLog 类中，是一个内部类）

2.2 在 commitlog 类中 doAppend 方法中进行 commitlog 的处理，还是基于 byteBuffer 的操作

2.3 在 commitlog 类中 doAppend 方法中进行返回，将消息写入的内存的位置信息、写入耗时封装为 AppendMessageResult 对象返回

核心关键点

Commitlog，RocketMQ 接收到消息后，首先是写入 Commitlog 文件，按照顺序进行写入，使用 NIO 技术。

在 Commitlog 中 putMessage 最后通过判断配置文件的主从同步类型和刷盘类型，进行刷盘。

总结

借助 java NIO 的力量，使得 I/O 性能十分高。当消息来的时候，顺序写入 CommitLog。

RocketMQ 下主要有三类大文件：commitlog 文件、Index 文件，consumequeue 文件，对于三类大文件，使用的就是 NIO 的 MappedByteBuffer 类来提高读写性能（主要是刷盘方法中）。这个类是文件内存映射的相关类，支持随机读和顺序写。在 RocketMQ 中，被封装成了 MappedFile 类

Broker 中更新消息队列和索引文件

消息进入 Commitlog 文件还不够，因为对于消费者来说，他们必须要看到 ConsumeQueue 和 IndexFile（ConsumeQueue 是因为消费要根据队列进行消费，另外没有索引文件 IndexFile，消息的查找会出现很大的延迟）。

所以 RocketMQ 通过开启一个线程 ReputMessageService 来监听 CommitLog 文件更新事件，如果有新的消息，则及时更新 ComsumeQueue、IndexFile 文件。

源码跟踪

1. DefaultMessageStore 类中的内部类 ReputMessageService 专门处理此项任务
2. ReputMessageService 类的 run()，默认 1 毫秒处理一次（文件从 CommitLog 到 ComsumeQueue 和 Index）
3. ReputMessageService 类的 doReput（）方法。
4. ReputMessageService 类的 doReput（）方法中， doDispatch，最终会构建（构建消息消费队 ）和（构建索引文件）

核心关键点

定时任务来处理的消息存储转换。处理核心类是 DefaultMessageStore 类中的内部类 ReputMessageService

总结

定时任务来处理的消息存储转换。处理核心类是 DefaultMessageStore 类中的内部类 ReputMessageService

消息的消费

消息消费分为推和拉两种模式。这里重点分析推模式。

Client 中的消费者启动流程

DefaultMQPushConsumerImpl#start()方法，其中重点就是 consumer.start()

DefaultMQPushConsumer.start() ->DefaultMQPushConsumerImpl.start()

源码跟踪

1.检查配置信息

2.加工订阅信息(同时，如果消息消费模式为集群模式，还需要为该消费组创建一个重试主题。)

3.创建 MQClientInstance 实例，

4.负载均衡

5.队列默认分配算法

6.pullAPIWrapper 拉取消息

7.消费进度存储

8.加载消息进度(

9.判断是顺序消息还是并发消息

10.消息消费服务并启动

11.注册消费者

12.MQClientInstance 启动

12.1 定时任务 startScheduledTask（）

12.1.1 每隔 2 分钟尝试获取一次 NameServer 地址

12.1.2 每隔 30S 尝试更新主题路由信息

12.1.3 每隔 30S 进行 Broker 心跳检测

12.1.4 默认每隔 5 秒持久化 ConsumeOffset

12.1.5 默认每隔 1S 检查线程池适配

12.2 开启拉消息服务（线程）

12.3 负载均衡服务（线程）

13.更新 TopicRouteData

14.检测 broker 状态

15.发送心跳

16.重新负载

核心关键点

在 RocketMQ 中，推模式还是使用拉模式进行消息的处理的。在 MQClientInstance 启动过程中启动了哪些定时任务

定时任务中 12 步中，包括了消费过程中的各种信息，这些信息都是定时去处理的。

总结

在 RocketMQ 中，推模式对比拉模式封装了非常多的功能，比如负载均衡、队列分配、消费进度存储、顺序消息、心跳检测等。

消息的拉取

分析一下 PUSH 模式下的集群模式消息拉取代码。

同一个消费组内有多个消费者，一个 topic 主题下又有多个消费队列，那么消费者是怎么分配这些消费队列的呢，从上面的启动的代码中是不是还记得在 org.apache.rocketmq.client.impl.factory.MQClientInstance#start 中，启动了 pullMessageService 服务线程，这个服务线程的作用就是拉取消息，我们去看下它的 run 方法：

源码跟踪

1. MQClientInstance. start() -> PullMessageService. start()
2. PullMessageService. pullMessage() -> DefaultMQPushConsumerImpl. pullMessage()方法
3. pullMessage 方法中包含了消息拉取的核心部分，包括处理暂停、流量控制、方法回调等。

核心关键点

流量控制

1.当 processQueue 没有消费的消息的数量达到（默认 1000 个）会触发流量控制

2.当 processQueue 中没有消费的消息体总大小 大于(默认 100m)时，触发流控

3.消息的最大位置和最小位置的差值如果大于默认值 2000，那么触发流控

总结

消息进行拉取时的核心是流量控制，这个也是解决客户端与服务端消费能力不对等的一种方案。

消息的消费

消息拉取到了之后，消费者要进行消息的消费，消息的消费主要是 consumeMessageService 线程做的，我们先看下 consumeMessageService 的构造函数

源码跟踪

1. ConsumeMessageConcurrentlyService 构造函数，在这个构造函数中，new 了一个名字叫 consumeExecutor 的线程池，在并发消费的模式下，这个线程池也就是消费消息的方式

2. 通过回调方式的模式，提交到 consumeMessageService 中（ConsumeMessageConcurrentlyService 实现类）,进入 submitConsumeRequest 方法，这个就是提交偏移量的处理。

3. 再直接进入 ConsumeMessageConcurrentlyService 中的内部类 ConsumeRequest 中的 run 方法

3.1 判断 processQueue 的 dropped 属性

3.2：拿到业务系统定义的消息监听 listener

3.3 判断是否有钩子函数，执行 before 方法

3.4 调用 resetRetryTopic 方法设置消息的重试主题

3.5 执行 listener.consumeMessage，业务系统具体去消费消息，

3.6 对消费结果的处理，进入 processConsumeResult 方法

3.6.1 集群模式下

3.6.2 失败的消息进入一个 List

3.6.3 消费失败的数据会重新建立一个数据，使用一个定时任务，再次到 Client 中的消费者启动流程 ．源码跟踪 step 6(重试消息的时候会创建一个条新的消息，而不是用老的消息)

核心关键点

消费失败的数据会重新建立一个数据，使用一个定时任务，重试消息的时候会创建一个条新的消息，而不是用老的消息。

总结

不管是消费成功还是消费失败的消息，都会更新消费进度，首先从 processQueue 中移除所有消费成功的消息并返回 offset，这里要注意一点，就是这个 offset 是 processQueue 中的 msgTreeMap 的最小的 key，为什么要这样做呢？因为消费进度的推进是 offset 决定的，因为是线程池消费，不能保证先消费的是 offset 大的那条消息，所以推进消费进度只能取最小的那条消息的 offset，这样在消费端重启的时候就可能会导致消息重复消费。