RocketMQ高阶使用

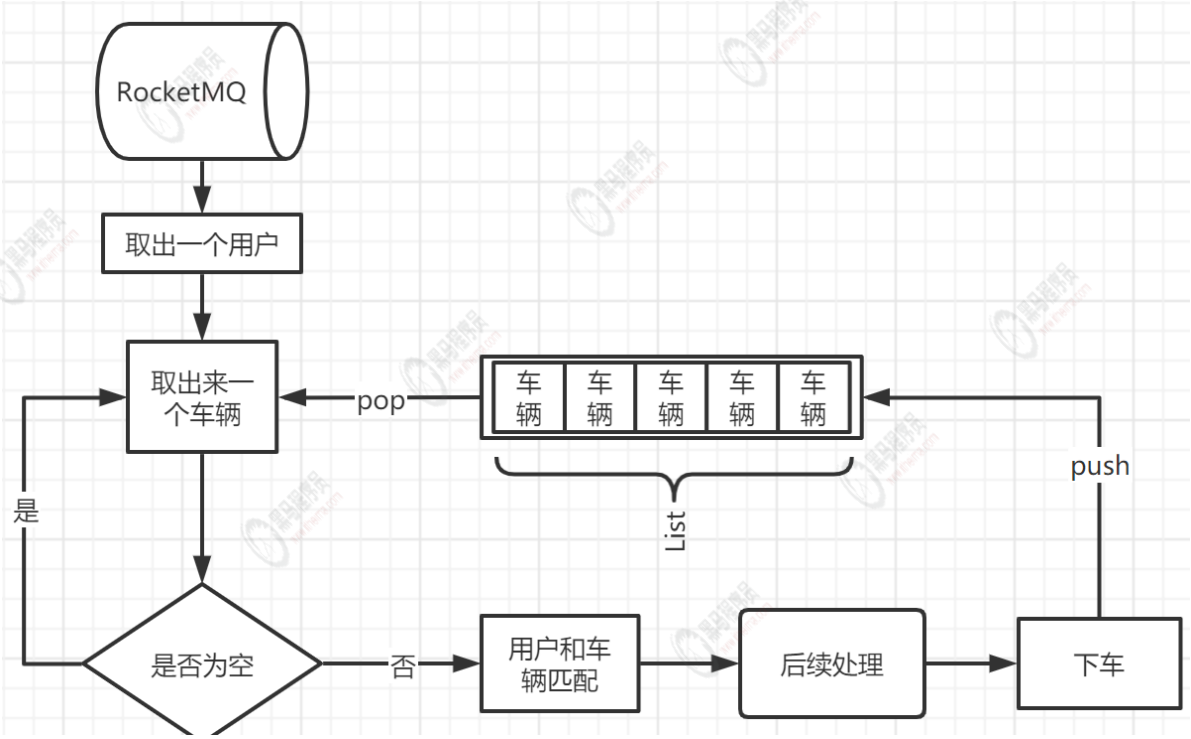
1 能力目标

- 能够理解和使用RocketMQ的顺序消息
- 能够理解和使用RocketMQ的生产者保证策略
- 能够理解和使用RocketMQ的消息投递策略
- 能够理解和使用RocketMQ的消费者的重试策略

2 1. 车辆调度

2.1 1.1 业务分析

2.1.1 1.1.1 车辆调度分析



用户打车从派单服务到调度服务，首先将消息以顺序方式扔到RocketMQ中，然后消费的事务就会严格按照放入的顺序进行消费，用户首先拿到从RocketMQ推送的顺序消息，然后保持住，开始轮询检查Redis中的List中是否存在车辆，存在两种情况：

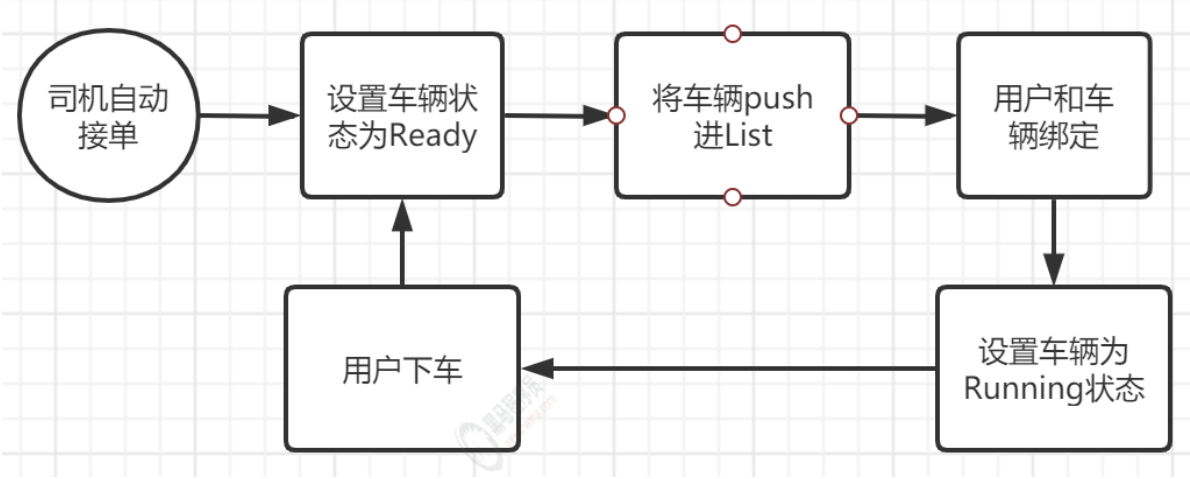
2.1.1.1 1.1.1.1 没有拉取到车辆

如果没有拉取到车辆，然后会延时一段时间，继续进行拉取，一直拉取不到的话一直进行自旋，一直等到拿到车辆才退出自旋。

2.1.1.2 1.1.1.2 拉取到车辆

如果拉取车辆就会将用户和拿到的车辆绑定到一起，开始后续操作，比如下订单等。

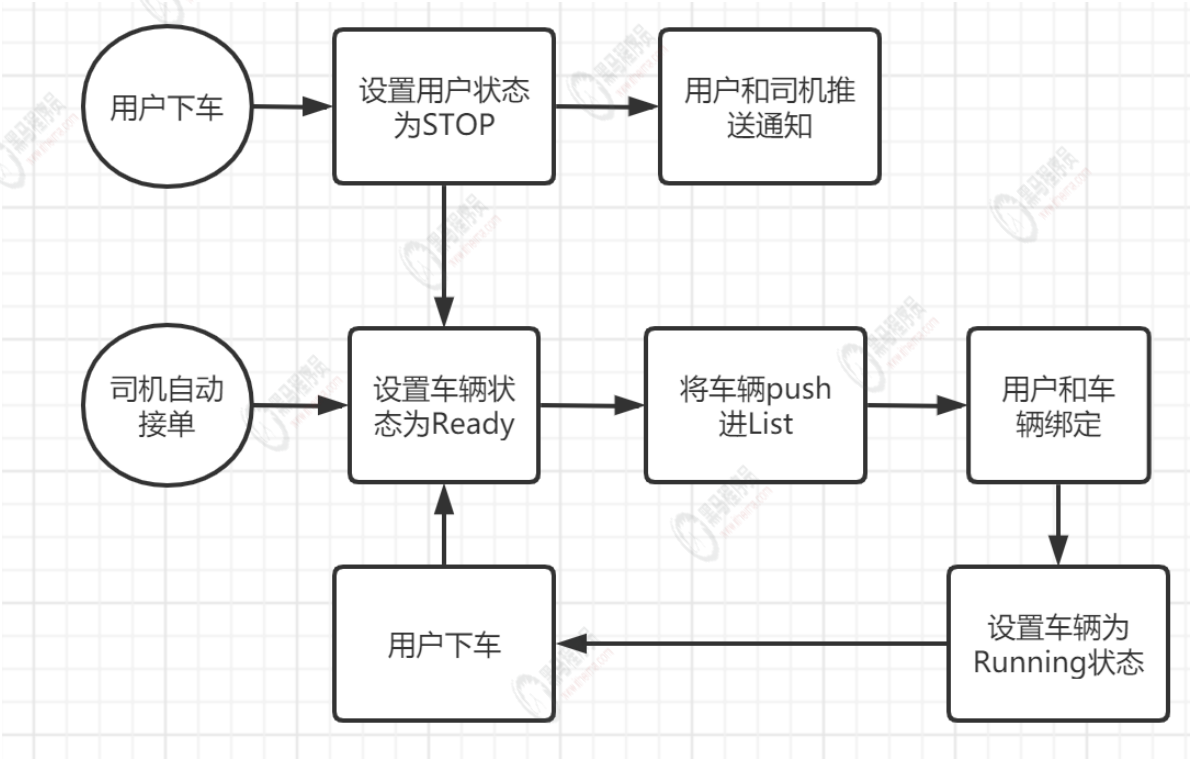
2.1.2 1.1.2 司机自动接单



> 当司机上线后，开启自动接单后，主题流程图下

1. 会先将车辆状态设置为在Ready状态，
2. 当车辆接到用户后会将车辆设置为Running状态，
3. 用户下车后，会将车辆继续设置为Ready状态，并将车辆push进list

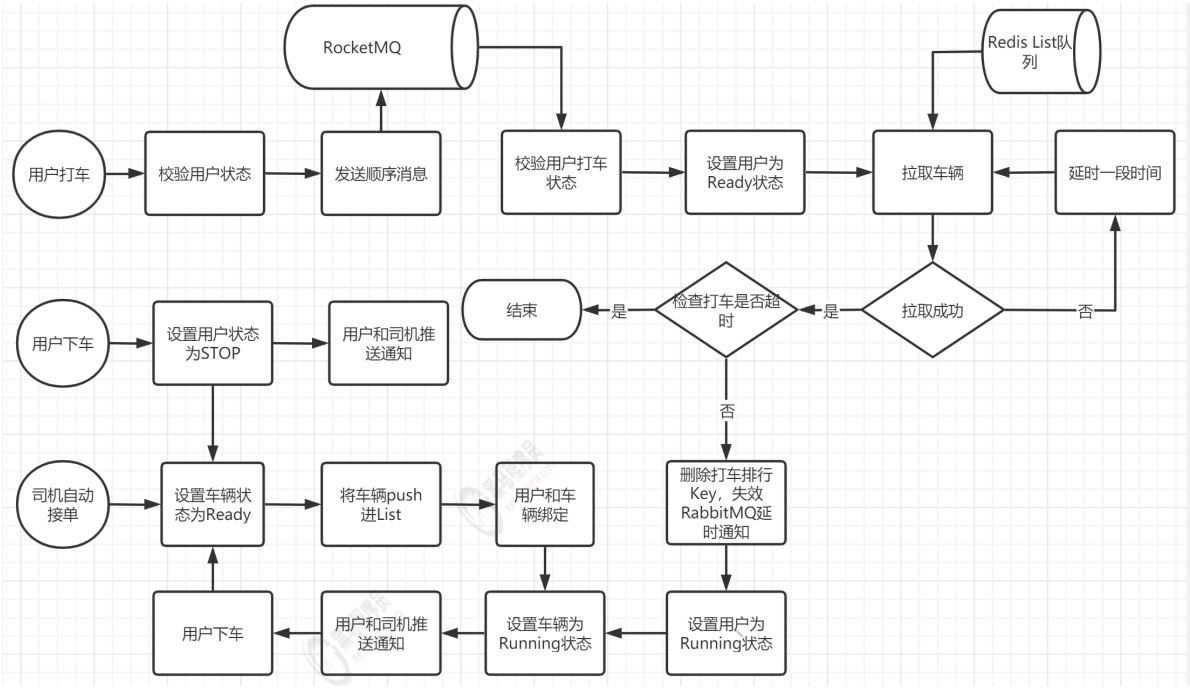
2.1.3 1.1.3 用户下车



> 如果用户点击下车，主体流程如下

1. 会先将用户状态设置为Stop状态
2. 然后会解除车辆和用户的绑定，
3. 之后车辆会将会push到list的尾端，让其他的用户可以拉取到车辆信息。

2.1.4 1.1.4 用户打车



> 用户上车后流程如下

1. 校验用户状态，然后将发送顺序消息到RabbitMQ
2. 消费者获取到用户消息，开始轮询来拉取车辆信息，如果拉取不到休眠一会继续拉取，一直到拉取到
3. 拉取到后校验是否超时，如果超时直接结束打车，否则删除RabbitMQ的超时检测Key，失效超时通知
4. 设置用户状态为Running，后续就到了司机自动接单的流程了

2.2 1.2 技术分析

2.2.1 1.2.1 RocketMQ顺序消息

> 打车需要排队，我们需要让前面的人能够被消费到，不能让这个顺序乱掉，这就需要用到RocketMQ的顺序消息

2.2.2 1.2.2 Redis 轮询队列

> 我们要让车辆在队列中，从MQ拿到一个车辆后，需要再从队列中拿取一个车辆如果拿不到则需要不断的轮询，一直到拿到车辆为止，如果打车玩完成还是需要将车辆归还队列，让其他的用户来打车，将一辆车重复利用起来

3 2 顺序消息

3.1 2.1 顺序类型

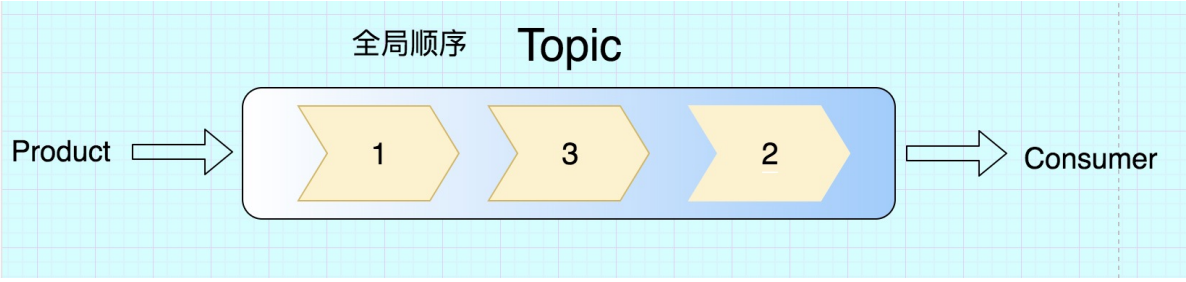
3.1.1 2.1.1 无序消息

> 无序消息也指普通的消息，Producer 只管发送消息，Consumer 只管接收消息，至于消息和消息之间的顺序并没有保证。

- Producer 依次发送 orderId 为 1、2、3 的消息
- Consumer 接到的消息顺序有可能是 1、2、3，也有可能是 2、1、3 等情况，这就是普通消息。

3.1.2 2.1.2 全局顺序

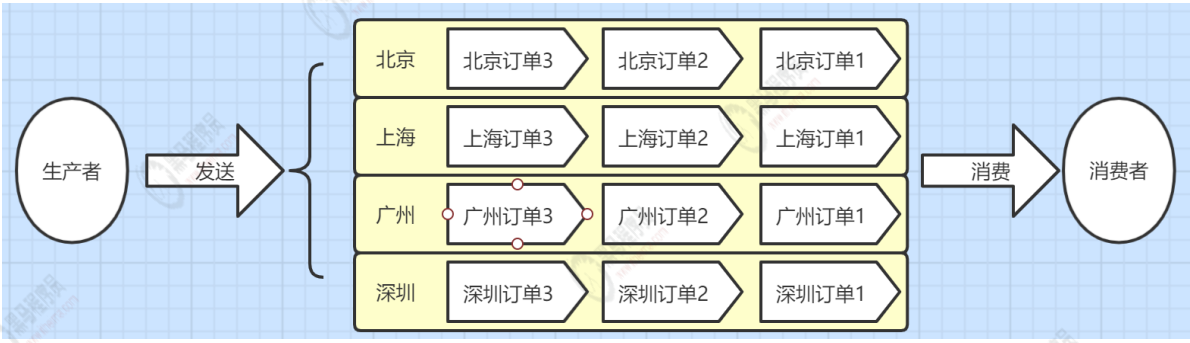
> 对于指定的一个 Topic，所有消息按照严格的先入先出（FIFO）的顺序进行发布和消费



比如 **Producer** 发送orderId 1,3,2 的消息, 那么 **Consumer** 也必须要按照 1,3,2 的顺序进行消费。

3.1.3 2.1.3 局部顺序

> 在实际开发有些场景中，我并不需要消息完全按照完全按的先进先出，而是某些消息保证先进先出就可以了。



就好比一个打车涉及到不同地区 `北京`， `上海` 、 `广州` 、 `深圳` 。我不用管其它的订单，只保证 `同一个地区的订单ID能保证这个顺序` 就可以了。

3.2 2.2 Rocket顺序消息

> RocketMQ可以严格的保证消息有序，但这个顺序，不是全局顺序，只是分区（queue）顺序，要全局顺序只能一个分区。

之所以出现你这个场景看起来不是顺序的，是因为发送消息的时候，消息发送默认是会采用轮询的方式发送到不通的queue（分区）

3.2.1 2.2.1 实现原理

我们知道 **生产的message最终会存放在Queue中**，如果一个Topic关联了4个Queue,如果我们不指定消息往哪个队列里放，那么默认是平均分配消息到4个queue，

好比有10条消息，那么这10条消息会平均分配在这4个Queue上，那么每个Queue大概放2个左右。这里有一点很重的是：**同一个queue，存储在里面的message 是按照先进先出的原则**

这个时候思路就来了，我们让不同的地区用不同的queue。只要保证同一个地区的订单把他们放到同一个Queue那就保证消费者先进先出了。

这就保证局部顺序了，即同一订单按照先后顺序放到同一Queue,那么取消息的时候就可以保证先进先取出。

3.2.2 2.2.2 如何保证集群有序

> 这里还有很关键的一点，在一个消费者集群的情况下，消费者1先去Queue拿消息，它拿到了 **北京订单1**，它拿完后，消费者2去queue拿到的是 **北京订单2**。

拿的顺序是没毛病了，但关键是先拿到不代表先消费完它。会存在虽然你消费者1先拿到**北京订单1**，但由于网络等原因，消费者2比你真正的先消费消息。这是不是很尴尬了。

3.2.2.1 2.2.2.1 分布式锁

> **Rocker采用的是分段锁，它不是锁整个Broker而是锁里面的单个Queue**，因为只要锁单个Queue就可以保证局部顺序消费了。
>
> 所以最终的消费者这边的逻辑就是

- 消费者1去Queue拿 **北京订单1**，它就锁住了整个Queue，只有它消费完成并返回成功后，这个锁才会释放。
- 然后下一个消费者去拿到 **北京订单2** 同样锁住当前Queue,这样的一个过程来真正保证对同一个Queue能够真正意义上的顺序消费，而不仅仅是顺序取出。

3.2.3 2.2.3 消息类型对比

> 全局顺序与分区顺序对比

Topic消息类型	支持事务消息	支持定时/延时消息	性能
无序消息（普通、事务、定时/延时）	是	是	最高
分区顺序消息	否	否	高
全局顺序消息	否	否	一般

> 发送方式对比

Topic消息类型	支持可靠同步发送	支持可靠异步发送	支持Oneway发送
无序消息（普通、事务、定时/延时）	是	是	是
分区顺序消息	是	否	否
全局顺序消息	是	否	否

3.2.4 2.2.4 注意事项

1. 顺序消息暂不支持广播模式。
2. 顺序消息不支持异步发送方式，否则将无法严格保证顺序。
3. 建议同一个 Group ID 只对应一种类型的 Topic，即不同时用于顺序消息和无序消息的收发。
4. 对于全局顺序消息，建议创建broker个数 >=2。

3.3 2.3 代码示例

3.3.1 2.31 队列选择器

```
public class SelectorFactory {
    /**
     * 工厂模式获取MessageQueueSelector
     *
     * @param value
     * @return
     */
    public static MessageQueueSelector getMessageQueueSelector(String value) {
        //如果value不为空使用hash选择器
        if (StringUtils.isNotEmpty(value)) {
            return new SelectMessageQueueByHash();
        }
        //如果value为空使用随机选择器
        return new SelectMessageQueueByRandom();
    }
}
```

3.3.2 2.3.2 消息发送者

```
@Component
public class MQProducer {

    @Autowired
    DefaultMQProducer defaultMQProducer;

    /**
     * 同步发送消息
     * @param taxiBO
     */
    public void send(TaxiBO taxiBO) {
        if (null == taxiBO) {
            return;
        }
        SendResult sendResult = null;

        try {
            //获取消息对象
            Message message =
RocketMQHelper.buildMessage(DispatchConstant.SEQ_TOPIC, taxiBO);
            //根据区域编码获取队列选择器
            MessageQueueSelector selector =
SelectorFactory.getMessageQueueSelector(taxiBO.getAreaCode());
            //发送同步消息
            sendResult = defaultMQProducer.send(message, selector,
taxiBO.getAreaCode(), 10000);
        } catch (MQClientException e) {
```

```
            e.printStackTrace();
        } catch (RemotingException e) {
            e.printStackTrace();
        } catch (MQBrokerException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        if (null != sendResult) {
            System.out.println(sendResult.toString());
        }
    }
}
```

3.3.3 2.3.3 消息消费者

> 消费者真正要达到消费顺序，需要分布式锁，所以这里需要将`MessageListenerOrderly`替换之前的**MessageListenerConcurrently**，因为它里面实现了分布式锁。

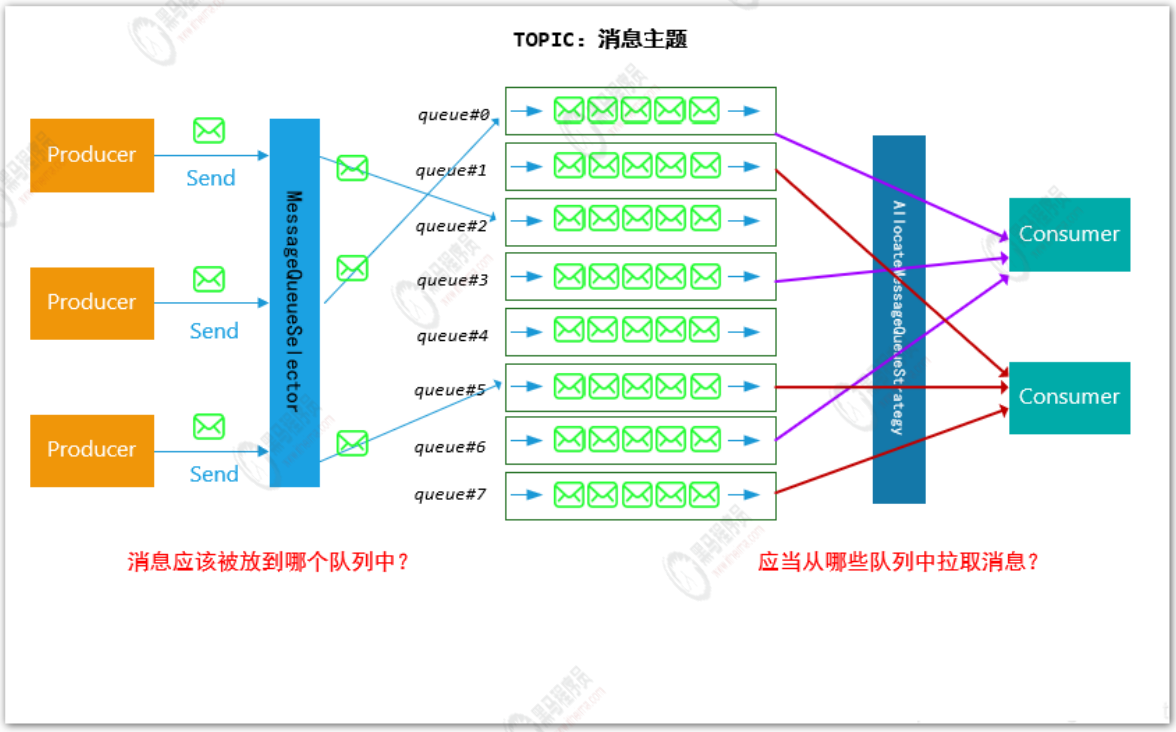
```
/**
 * 消费消息
 */
public abstract class MQConsumeMessageListenerProcessor implements
MessageListenerOrderly {
    public static final Logger logger =
LoggerFactory.getLogger(MQConsumeMessageListenerProcessor.class);

    /**
     * 消费有序消息
     *
     * @param list
     * @param consumeOrderlyContext
     * @return
     */
    @Override
    public ConsumeOrderlyStatus consumeMessage(List<MessageExt> list,
ConsumeOrderlyContext consumeOrderlyContext) {

        if (CollectionUtils.isEmpty(list)) {
            logger.info("MQ接收消息为空，直接返回成功");
            return ConsumeOrderlyStatus.SUCCESS;
        }
        //消费消息
        for (MessageExt messageExt : list) {
            try {
                String topic = messageExt.getTopic();
                String tags = messageExt.getTags();
                String body = new String(messageExt.getBody(), "utf-8");
                //调用具体消费流程
                processMessage(topic, tags, body);
                logger.info("MQ消息topic={}, tags={}, 消息内容={}", topic, tags,
body);
            } catch (Exception e) {
                logger.error("获取MQ消息内容异常{}", e);
                //暂停当前队列
                return ConsumeOrderlyStatus.SUSPEND_CURRENT_QUEUE_A_MOMENT;
            }
```

```
        }

        // TODO 处理业务逻辑
        return ConsumeOrderlyStatus.SUCCESS;
    }

    /**
     * 处理消息
     *
     * @param body
     */
    public abstract void processMessage(String topic, String tags, String body);
}
```

> 上面我们介绍了顺序消息，它主要将相同的消息投递到一个队列中的，具体如何投递呢

4 3 消息投递策略

> 上面我们介绍了顺序消息，但是RocketMQ还支持那些投递策略呢、
>
> RocketMQ 的消息模型整体并不复杂，如下图所示：



一个`Topic(消息主题)`可能对应多个实际的消息队列(MessgeQueue)

在底层实现上，为了提高MQ的可用性和灵活性，一个Topic在实际存储的过程中，采用了多队列的方式，具体形式如上图所示。每个消息队列在使用中应当保证**先入先出**（FIFO,First In First Out）的方式进行消费。

> 那么，基于这种模型，就会引申出两个问题：

- **生产者** 在发送相同Topic的消息时，消息体应当被放置到哪一个消息队列(MessageQueue)中?
- **消费者** 在消费消息时，应当从哪些消息队列中拉取消息?

4.1 3.1 生产者投递策略

> 生产者投递策略就是讲如何讲一个消息投递到不同的queue中

4.1.1 3.1.1 轮询算法投递

> 默认投递方式：基于`Queue队列`轮询算法投递

默认情况下，采用了最简单的轮询算法，这种算法有个很好的特性就是，保证每一个`Queue队列`的消息投递数量尽可能均匀，算法如下图所示：

4.1.2 3.1.2 顺序投递策略

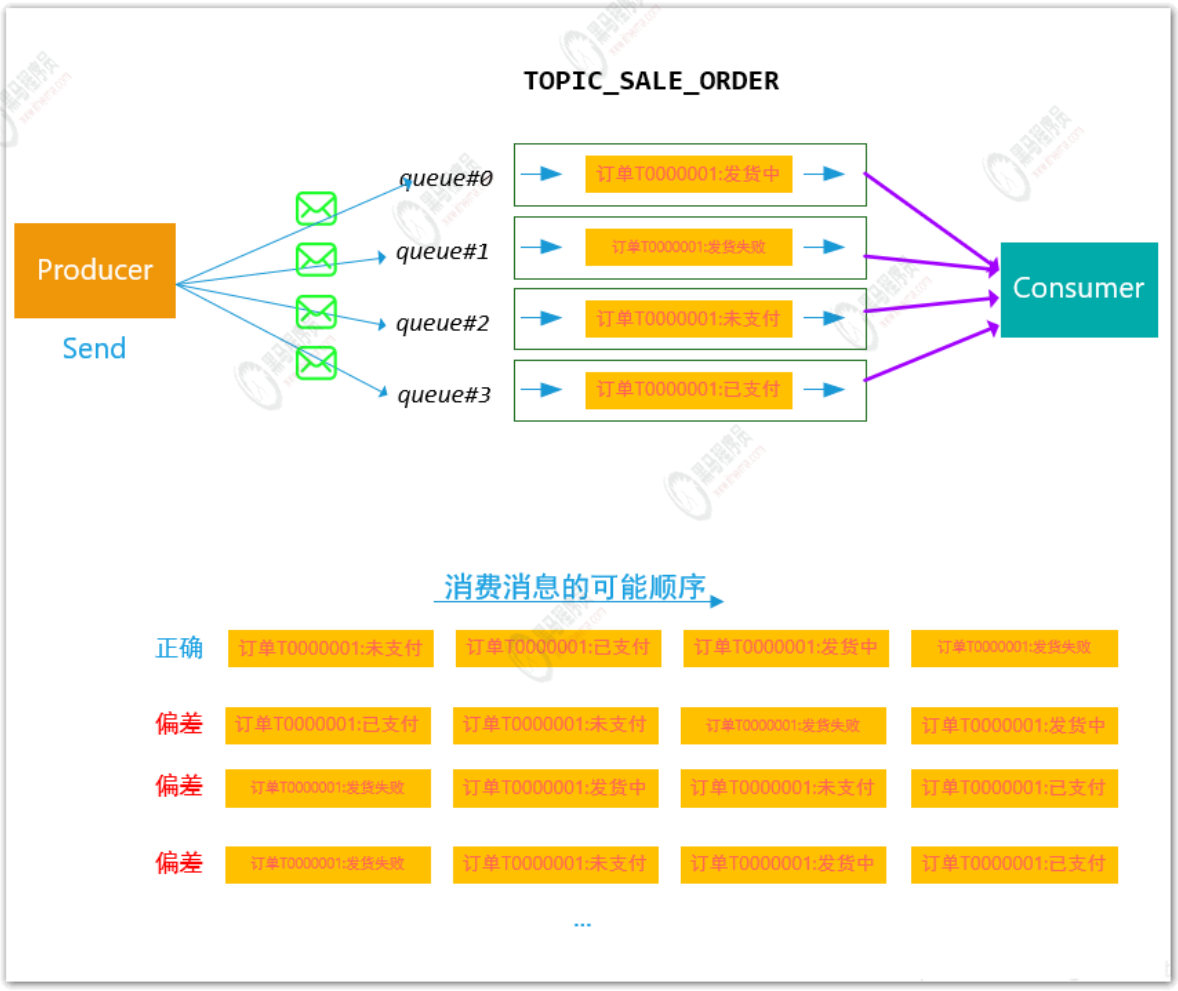
> 在有些场景下，需要保证同类型消息投递和消费的顺序性。

例如，假设现在有TOPIC `topicTest`,该 Topic下有4个`Queue队列`，该Topic用于传递订单的状态变迁，假设订单有状态：`未支付`、`已支付`、`发货中(处理中)`、`发货成功`、`发货失败`。

> 在时序上，生产者从时序上可以生成如下几个消息：

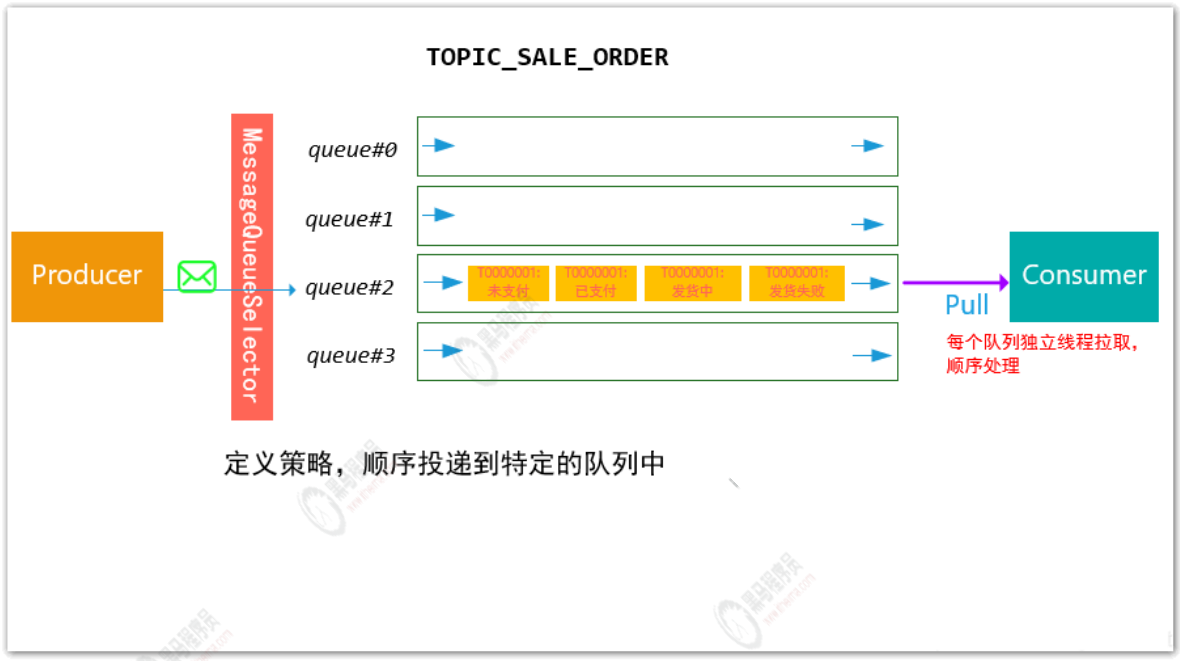
```
订单T0000001:未支付 --> 订单T0000001:已支付 --> 订单T0000001:发货中(处理中) --> 订单T0000001:发货失败
```

> 消息发送到MQ中之后，可能由于轮询投递的原因，消息在MQ的存储可能如下：



这种情况下，我们希望`消费者`消费消息的顺序和我们发送是一致的，然而，有上述MQ的投递和消费机制，我们无法保证顺序是正确的，对于顺序异常的消息，`消费者`即使有一定的状态容错，也不能完全处理好这么多种随机出现组合情况。

基于上述的情况，`RockeMQ`采用了这种实现方案：**对于相同订单号的消息，通过一定的策略，将其放置在一个`queue队列中`，然后`消费者`再采用一定的策略(一个线程独立处理一个`queue`,保证处理消息的顺序性)，能够保证消费的顺序性**



生产者在消息投递的过程中，使用了`MessageQueueSelector`作为队列选择的策略接口，其定义如下：

```
public interface MessageQueueSelector {
        /**
         * 根据消息体和参数，从一批消息队列中挑选出一个合适的消息队列
         * @param mqs  待选择的MQ队列选择列表
         * @param msg  待发送的消息体
         * @param arg  附加参数
         * @return  选择后的队列
         */
        MessageQueue select(final List<MessageQueue> mqs, final Message msg,
final Object arg);
}
```

4.1.3 3.1.3 自带实现类

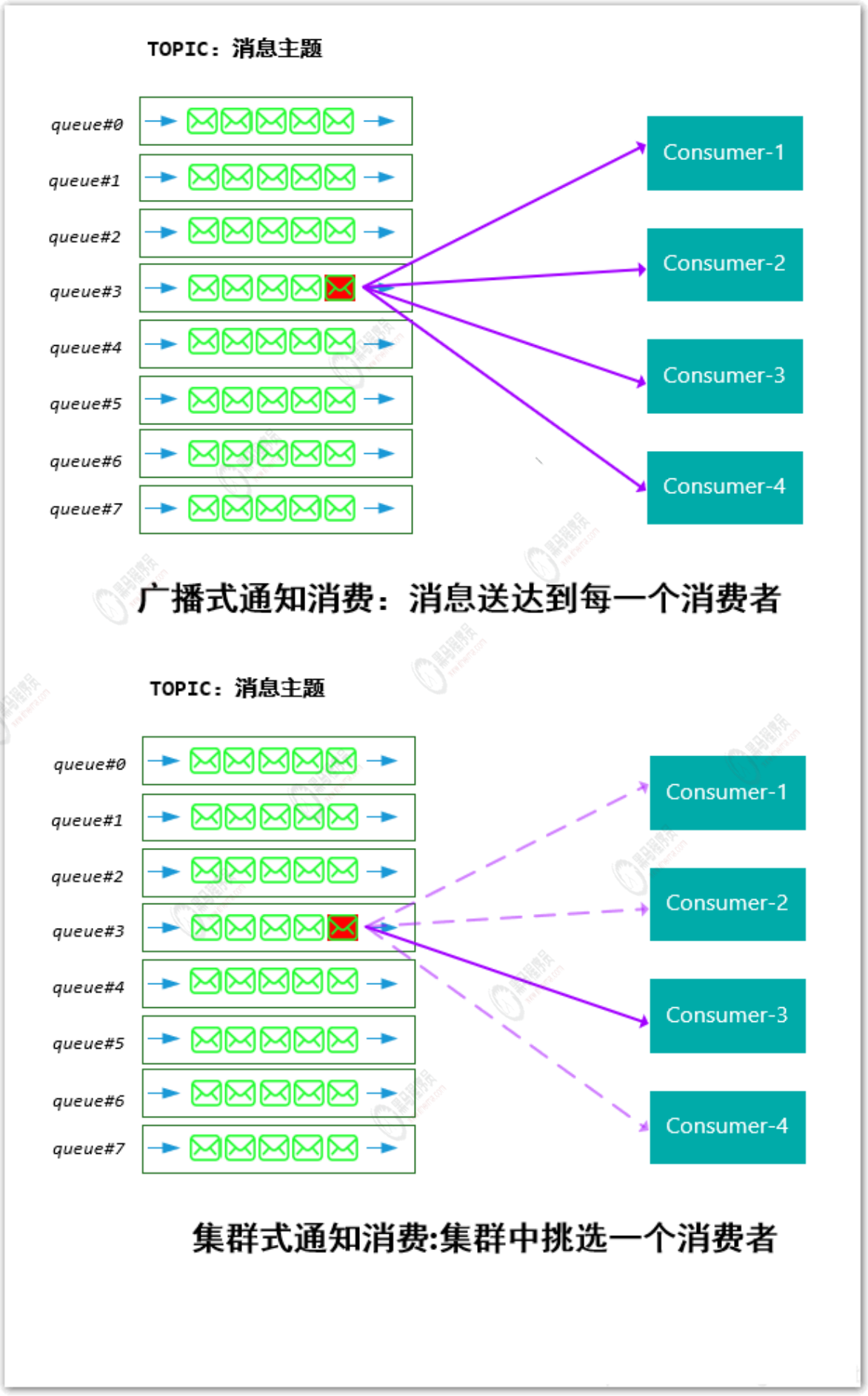
投递策略	策略实现类	说明
随机分配策略	SelectMessageQueueByRandom	使用了简单的随机数选择算法
基于Hash分配策略	SelectMessageQueueByHash	根据附加参数的Hash值，按照消息队列列表的大小取余数，得到消息队列的index
基于机器机房位置分配策略	SelectMessageQueueByMachineRoom	开源的版本没有具体的实现，基本的目的应该是机器的就近原则分配

4.2 3.2 消费者分配队列

> RocketMQ对于消费者消费消息有两种形式：

- `BROADCASTING`:广播式消费，这种模式下，一个消息会被通知到每一个`消费者`
- `CLUSTERING`：集群式消费，这种模式下，一个消息最多只会被投递到一个`消费者`上进行消费

模式如下：



对于使用了消费模式为`MessageModel.CLUSTERING`进行消费时，需要保证一个消息**在整个集群中只需要被消费一次。**实际上，在RoketMQ底层，消息指定分配给消费者的实现，是通过queue队列分配给消费者的方式完成的：也就是说，消息分配的单位是消息所在的queue队列

> 将queue队列指定给特定的消费者后，queue队列内的所有消息将会被指定到消费者进行消费。

RocketMQ定义了策略接口AllocateMessageQueueStrategy，对于给定的消费者分组,和消息队列列表、消费者列表，当前消费者应当被分配到哪些queue队列，定义如下：

```
**
 * 为消费者分配queue的策略算法接口
 */
public interface AllocateMessageQueueStrategy {

    /**
     * Allocating by consumer id
     *
     * @param consumerGroup 当前 consumer群组
     * @param currentCID 当前consumer id
     * @param mqAll 当前topic的所有queue实例引用
     * @param cidAll 当前 consumer群组下所有的consumer id set集合
     * @return 根据策略给当前consumer分配的queue列表
     */
    List<MessageQueue> allocate(
        final String consumerGroup,
        final String currentCID,
        final List<MessageQueue> mqAll,
        final List<String> cidAll
    );

    /**
     * 算法名称
     *
     * @return The strategy name
     */
    String getName();
}
```

> 相应地，RocketMQ提供了如下几种实现：

算法名称	含义
AllocateMessageQueueAveragely	平均分配算法
AllocateMessageQueueAveragelyByCircle	基于环形平均分配算法
AllocateMachineRoomNearby	基于机房临近原则算法
AllocateMessageQueueByMachineRoom	基于机房分配算法
AllocateMessageQueueConsistentHash	基于一致性hash算法
AllocateMessageQueueByConfig	基于配置分配算法

> 为了讲述清楚上述算法的基本原理，我们先假设一个例子，下面所有的算法将基于这个例子讲解。

假设当前同一个topic下有queue队列 10 个，消费者共有 4 个,如下图所示：

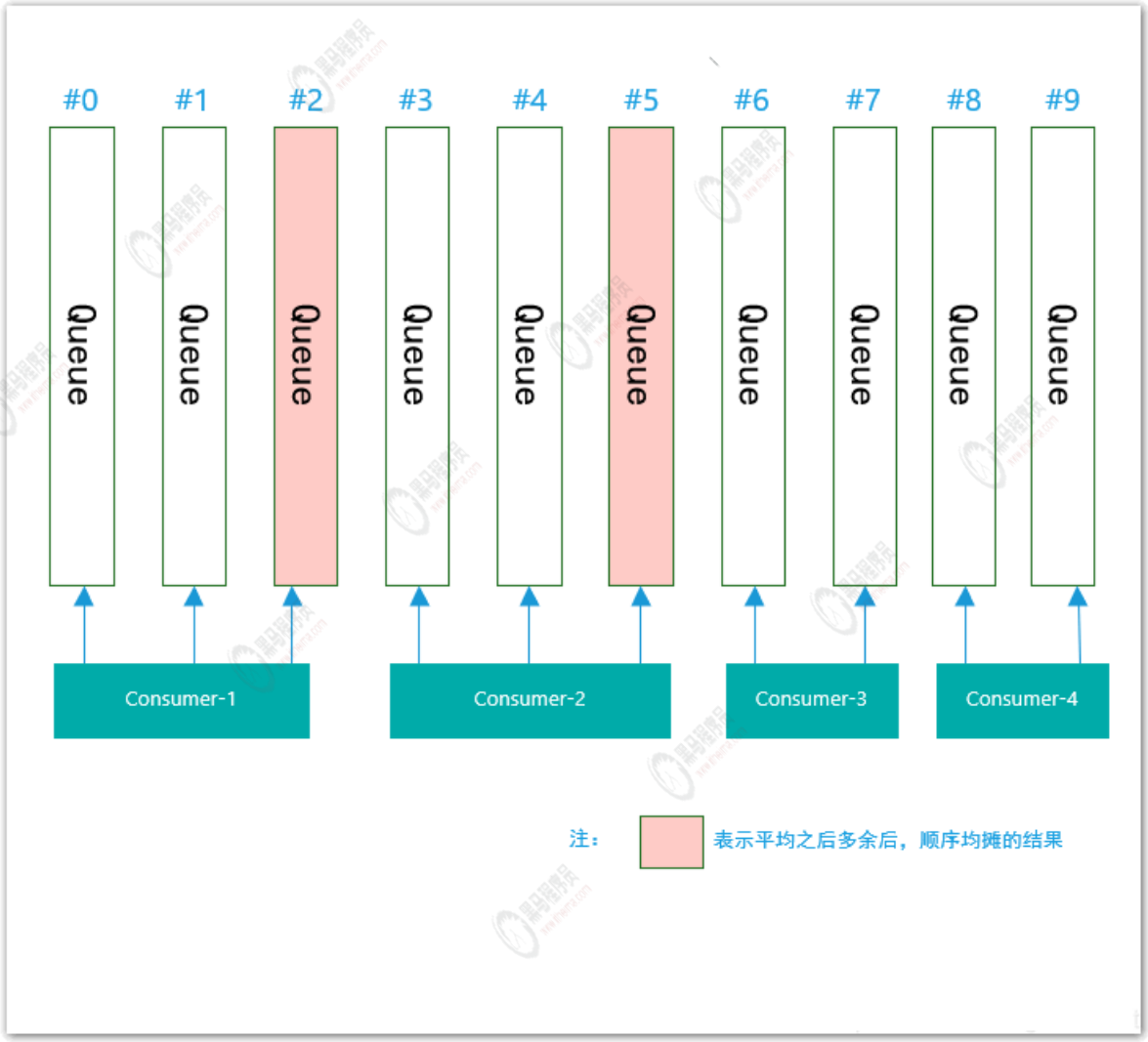
4.2.1 3.2.1 平均分配算法

这里所谓的平均分配算法，并不是指的严格意义上的完全平均，如上面的例子中，10个queue，而消费者只有4个，无法是整除关系，除了整除之外的多出来的queue,将依次根据消费者的顺序均摊。

按照上述例子来看，`10/4=2`，即表示每个`消费者`平均均摊2个queue；而`10%4=2`，即除了均摊之外，多出来2个`queue`还没有分配，那么，根据消费者的顺序`consumer-1`、`consumer-2`、`consumer-3`、`consumer-4`,则多出来的2个`queue`将分别给`consumer-1`和`consumer-2`。

> 最终，分摊关系如下：

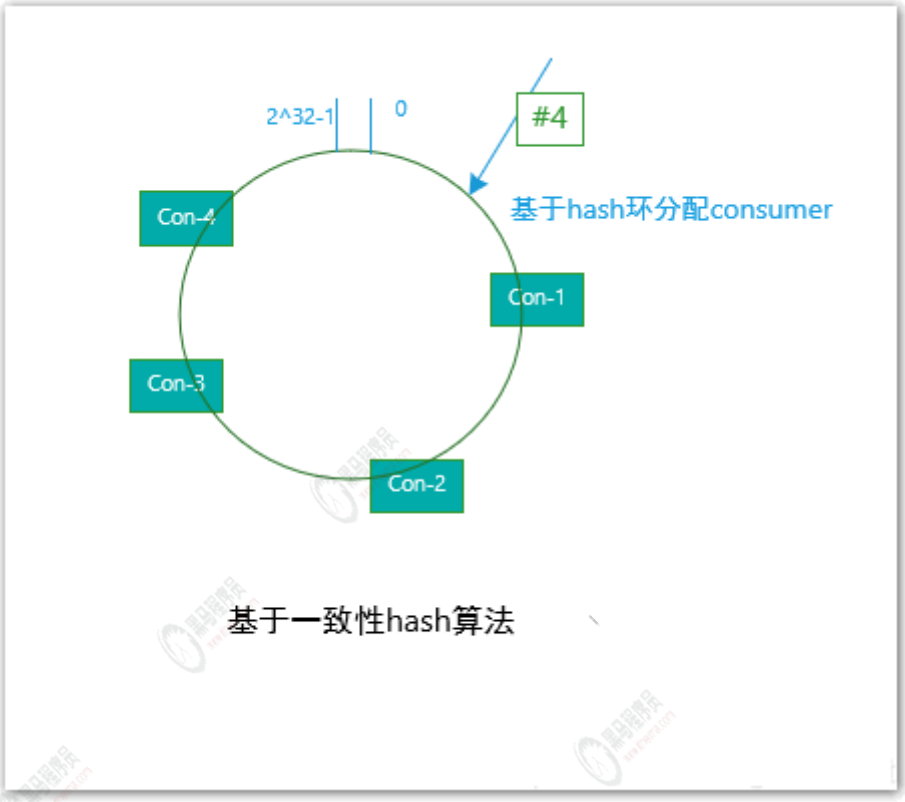
- `consumer-1`:3个
- `consumer-2`:3个
- `consumer-3`:2个
- `consumer-4`:2个



4.2.2 3.2.2 一致性hash分配算法

使用这种算法，会将`consumer消费者`作为Node节点构造成一个hash环，然后`queue队列`通过这个hash环来决定被分配给哪个`consumer消费者`。

> 其基本模式如下：



> 一致性hash算法用于在分布式系统中，保证数据的一致性而提出的一种基于hash环实现的算法

4.2.3 3.2.3 使用方式

> 默认消费者使用使用了`AllocateMessageQueueAveragely`平均分配策略

如果需要使用其他分配策略，使用方式如下

```
//创建一个消息消费者，并设置一个消息消费者组，并指定使用一致性hash算法的分配策略
DefaultMQPushConsumer consumer = new
DefaultMQPushConsumer(null,"rocket_test_consumer_group",null,new
AllocateMessageQueueConsistentHash());
.....
```

5 4 RocketMQ消息保障

> 下面我们详细说下如何保障消息不丢失以及消息幂等性问题

5.1 4.1 生产端保障

> 生产端保障需要从一下几个方面来保障

1. 使用可靠的消息发送方式
2. 注意生产端重试
3. 生产禁止自动创建topic

5.1.1 4.1.1 消息发送保障

5.1.1.1 4.1.1.1 同步发送

> 发送者向MQ执行发送消息API时，同步等待，直到消息服务器返回发送结果，会在收到接收方发回响应之后才发下一个数据包的通讯方式，这种方式只有在消息完全发送完成之后才返回结果，此方式存在需要同步等待发送结果的时间代价。

简单来说，同步发送就是指 producer 发送消息后，会在接收到 broker 响应后才继续发下一条消息的通信方式。

使用场景

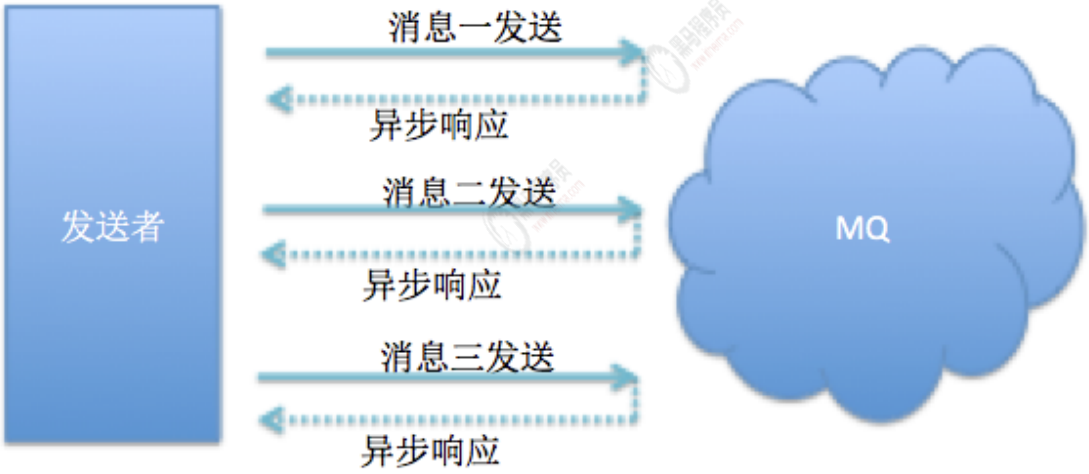
由于这种同步发送的方式确保了消息的可靠性，同时也能及时得到消息发送的结果，故而适合一些发送比较重要的消息场景，比如说重要的通知邮件、营销短信等等。在实际应用中，这种同步发送的方式还是用得比较多的。

注意事项

这种方式具有内部重试机制，即在主动声明本次消息发送失败之前，内部实现将重试一定次数，默认为2次（`DefaultMQProducer#getRetryTimesWhenSendFailed`）。 发送的结果存在同一个消息可能被多次发送给broker，这里需要应用的开发者自己在消费端处理幂等性问题。

5.1.1.2 4.1.1.2 异步发送

> 异步发送是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的通讯方式。 MQ 的异步发送，需要用户实现异步发送回调接口（`SendCallback`）



异步发送是指 producer 发出一条消息后，不需要等待 broker 响应，就接着发送下一条消息的通信方式。需要注意的是，不等待 broker 响应，并不意味着 broker 不响应，而是通过回调接口来接收 broker 的响应。所以要记住一点，异步发送同样可以对消息的响应结果进行处理。

使用场景

由于异步发送不需要等待 broker 的响应，故在一些比较注重 RT（响应时间）的场景就会比较适用。比如，在一些视频上传的场景，我们知道视频上传之后需要进行转码，如果使用同步发送的方式来通知启动转码服务，那么就需要等待转码完成才能发回转码结果的响应，由于转码时间往往较长，很容易造成响应超时。此时，如果使用的是异步发送通知转码服务，那么就可以等转码完成后，再通过回调接口来接收转码结果的响应了。

注意事项

注意：RocketMQ内部只对同步模式做了重试，异步发送模式是没有自动重试的，需要自己手动实现

5.1.2 4.1.2 消息发送总结

5.1.2.1 4.1.2.1 发送方式对比

发送方式	发送 TPS	发送结果反馈	可靠性	适用场景
同步发送	一般	有	不丢失	、重要的通知场景
异步发送	快	有	不丢失	比较注重 RT（响应时间）的场景
单向发送	最快	无	可能丢失	可靠性要求并不高的场景

5.1.2.2 4.1.2.2 使用场景对比

> 在实际使用场景中，利用何种发送方式，可以总结如下：
>
> - 当发送的消息不重要时，采用 `one-way` 方式，以提高吞吐量；
> - 当发送的消息很重要是，且对响应时间不敏感的时候采用 `sync` 方式；
> - 当发送的消息很重要，且对响应时间非常敏感的时候采用 `async` 方式；

5.1.3 4.1.3 发送状态

> 发送消息时，将获得包含SendStatus的SendResult。首先，我们假设Message的isWaitStoreMsgOK = true（默认为true），如果没有抛出异常，我们将始终获得SEND_OK，以下是每个状态的说明列表：

5.1.3.1 4.1.3.1 FLUSH_DISK_TIMEOUT

如果设置了 `FlushDiskType=SYNC_FLUSH` (默认是 ASYNC_FLUSH)，并且 Broker 没有在 `syncFlushTimeout` （默认是 5 秒）设置的时间内完成刷盘，就会收到此状态码。

5.1.3.2 4.1.3.2 FLUSH_SLAVE_TIMEOUT

如果设置为 `SYNC_MASTER`，并且 slave Broker 没有在 `syncFlushTimeout` 设定时间内完成同步，就会收到此状态码。

5.1.3.3 4.1.3.3 SLAVE_NOT_AVAILABLE

如果设置为 `SYNC_MASTER`，并没有配置 slave Broker，就会收到此状态码。

5.1.3.4 4.1.3.4 SEND_OK

这个状态可以简单理解为，没有发生上面列出的三个问题状态就是SEND_OK。需要注意的是，`SEND_OK` 并不意味着**可靠**，如果想严格确保没有消息丢失，需要开启 **SYNC_MASTER or SYNC_FLUSH**。

5.1.3.5 4.1.3.5 注意事项

如果收到了 `FLUSH_DISK_TIMEOUT, FLUSH_SLAVE_TIMEOUT`，意味着消息会丢失，有2个选择，一是无所谓，适用于消息不关紧要的场景，二是重发，但可能产生消息重复，这就需要consumer进行去重控制。如果收到了 `SLAVE_NOT_AVAILABLE` 就要赶紧通知管理员了。

5.1.4 4.2.4 MQ发送端重试保障

> 如果由于网络抖动等原因，Producer程序向Broker发送消息时没有成功，即发送端没有收到Broker的ACK，导致最终Consumer无法消费消息，此时RocketMQ会自动进行重试。

DefaultMQProducer可以设置消息发送失败的最大重试次数，并可以结合发送的超时时间来进行重试的处理，具体API如下：

```
//设置消息发送失败时的最大重试次数
public void setRetryTimesWhenSendFailed(int retryTimesWhenSendFailed) {
   this.retryTimesWhenSendFailed = retryTimesWhenSendFailed;
}

//同步发送消息，并指定超时时间
public SendResult send(Message msg,
                       long timeout) throws MQClientException, RemotingException,
MQBrokerException, InterruptedException {
   return this.defaultMQProducerImpl.send(msg, timeout);
```

5.1.4.1 4.2.4.1 重试问题

超时重试针对网上说的超时异常会重试的说法都是错误的

是因为下面测试代码的超时时间设置为5毫秒，按照正常肯定会报超时异常，但设置1次重试和3000次的重试，虽然最终都会报下面异常，但输出错误时间报显然不应该是一个级别。但测试发现无论设置的多少次的重试次数，报异常的时间都差不多。

测试代码

```
public class RetryProducer {
    public static void main(String[] args) throws UnsupportedEncodingException,
InterruptedException, RemotingException, MQClientException, MQBrokerException {
        //创建一个消息生产者，并设置一个消息生产者组
        DefaultMQProducer producer = new
DefaultMQProducer("rocket_test_consumer_group");

        //指定 NameServer 地址
        producer.setNamesrvAddr("127.0.0.1:9876");
        //设置重试次数(默认2次)
        producer.setRetryTimesWhenSendFailed(300000);
        //初始化 Producer，整个应用生命周期内只需要初始化一次
        producer.start();
        Message msg = new Message(
                /* 消息主题名 */
                "topicTest",
                /* 消息标签 */
                "TagA",
                /* 消息内容 */
                ("Hello Java demo RocketMQ
").getBytes(RemotingHelper.DEFAULT_CHARSET));
        //发送消息并返回结果,设置超时时间 5ms 所以每次都会发送失败
        SendResult sendResult = producer.send(msg, 5);
```

```
        System.out.printf("%s%n", sendResult);
        // 一旦生产者实例不再被使用则将其关闭，包括清理资源，关闭网络连接等
        producer.shutdown();
    }
}
```

揭晓答案

> 针对这个疑惑，需要查看源码，发现只有同步发送才会重试，并且超时是不重试的

```
/**
  * 说明 抽取部分代码
  */
private SendResult sendDefaultImpl(Message msg, final CommunicationMode
communicationMode, final SendCallback sendCallback, final long timeout) {

    //1、获取当前时间
    long beginTimestampFirst = System.currentTimeMillis();
    long beginTimestampPrev ;
    //2、去服务器看下有没有主题消息
    TopicPublishInfo topicPublishInfo =
this.tryToFindTopicPublishInfo(msg.getTopic());
    if (topicPublishInfo != null && topicPublishInfo.ok()) {
        boolean callTimeout = false;
        //3、通过这里可以很明显看出 如果不是同步发送消息 那么消息重试只有1次
        int timesTotal = communicationMode == CommunicationMode.SYNC ? 1 +
this.defaultMQProducer.getRetryTimesWhenSendFailed() : 1;
        //4、根据设置的重试次数，循环再去获取服务器主题消息
        for (times = 0; times < timesTotal; times++) {
            MessageQueue mqSelected =
this.selectOneMessageQueue(topicPublishInfo, lastBrokerName);
            beginTimestampPrev = System.currentTimeMillis();
            long costTime = beginTimestampPrev - beginTimestampFirst;
            //5、前后时间对比 如果前后时间差 大于 设置的等待时间 那么直接跳出for循环了 这就
说明连接超时是不进行多次连接重试的
            if (timeout < costTime) {
                callTimeout = true;
                break;

            }
            //6、如果超时直接报错
            if (callTimeout) {
                throw new RemotingTooMuchRequestException("sendDefaultImpl call
timeout");
            }
        }
    }
```

5.1.4.2 4.1.4.2 重试总结

> 通过这段源码很明显可以看出以下几点

1. 如果是**异步发送**那么重试次数只有1次
2. 对于同步而言，**超时异常也是不会再去重试。**
3. 如果发生重试是在一个for 循环里去重试，所以它是立即重试而不是隔一段时间去重试。

5.1.5 4.1.5 禁止自动创建topic

5.1.5.1 4.1.5.1 自动创建TOPIC流程

> `autoCreateTopicEnable`设置为true 标识开启自动创建topic

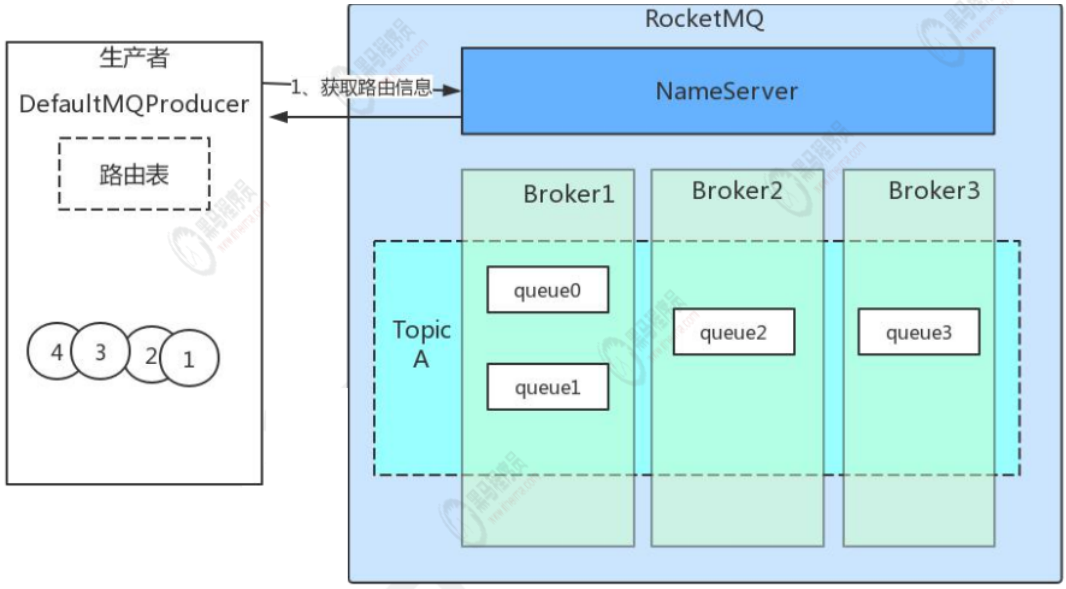
1. 消息发送时如果根据topic没有获取到 路由信息，则会根据默认的topic去获取，获取到路由信息后选择一个队列进行发送，发送时报文会带上默认的topic以及默认的队列数量。
2. 消息到达broker后，broker检测没有topic的路由信息，则查找默认topic的路由信息，查到表示开启了自动创建topic，则会根据消息内容中的默认的队列数量在本broker上创建topic，然后进行消息存储。
3. **broker创建topic后并不会马上同步给namesrv**，而是每30进行汇报一次，更新namesrv上的topic路由信息，producer会每30s进行拉取一次topic的路由信息，更新完成后就可以正常发送消息。更新之前一直都是按照默认的topic查找路由信息。

5.1.5.2 4.1.5.2 为什么不能开启自动创建

上述 broker 中流程会有一个问题，就是在producer更新路由信息之前的这段时间，如果消息只发送到了broker-a，则broker-b上不会创建这个topic的路由信息，broker互相之间不通信。当producer更新之后，获取到的broker列表只有broker-a，就永远不会轮询到broker-b的队列(因为没有路由信息)，所以我们生产通常关闭自动创建broker，而是采用手动创建的方式。

5.1.6 4.1.5.6 发端端规避

> 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
>
> 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。

5.2 4.2 消费端保障

5.2.1 4.2.1 注意幂等性

> 应用程序在使用RocketMQ进行消息消费时必须支持幂等消费，即同一个消息被消费多次和消费一次的结果一样。这一点在使用RoketMQ或者分析RocketMQ源代码之前再怎么强调也不为过。

“至少一次送达”的消息交付策略，和消息重复消费是一对共生的因果关系。要做到不丢消息就无法避免消息重复消费。原因很简单，试想一下这样的场景：客户端接收到消息并完成了消费，在消费确认过程中发生了通讯错误。从Broker的角度是无法得知客户端是在接收消息过程中出错还是在消费确认过程中出错。为了确保不丢消息，重发消息是唯一的选择。

有了消息幂等消费约定的基础，RocketMQ就能够有针对性地采取一些性能优化措施，例如：并行消费、消费进度同步机制等，这也是RocketMQ性能优异的原因之一。

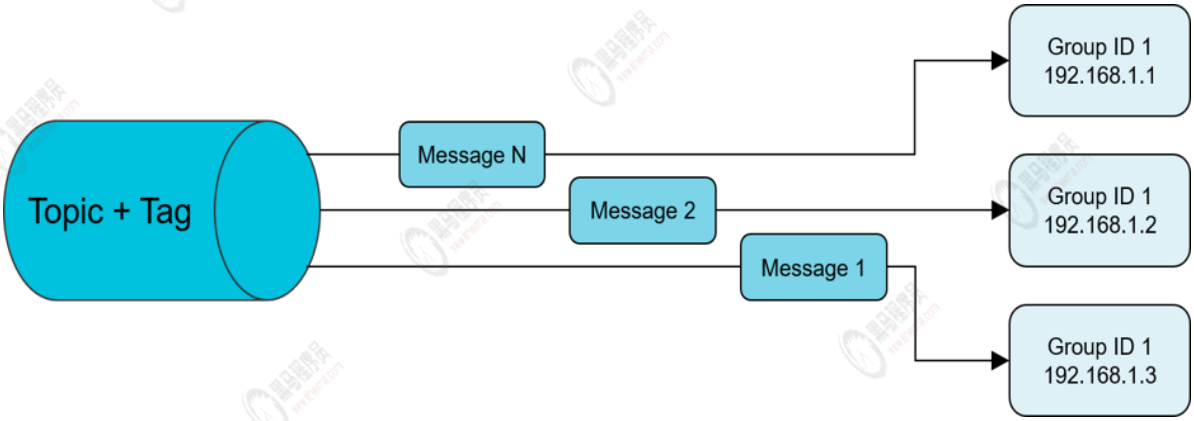
5.2.2 4.2.2 消息消费模式

> 从不同的维度划分，Consumer支持以下消费模式：

- 广播消费模式下，消息消费失败不会进行重试，消费进度保存在Consumer端；
- 集群消费模式下，消息消费失败有机会进行重试，消费进度集中保存在Broker端。

5.2.2.1 4.2.2.1 集群消费

> 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

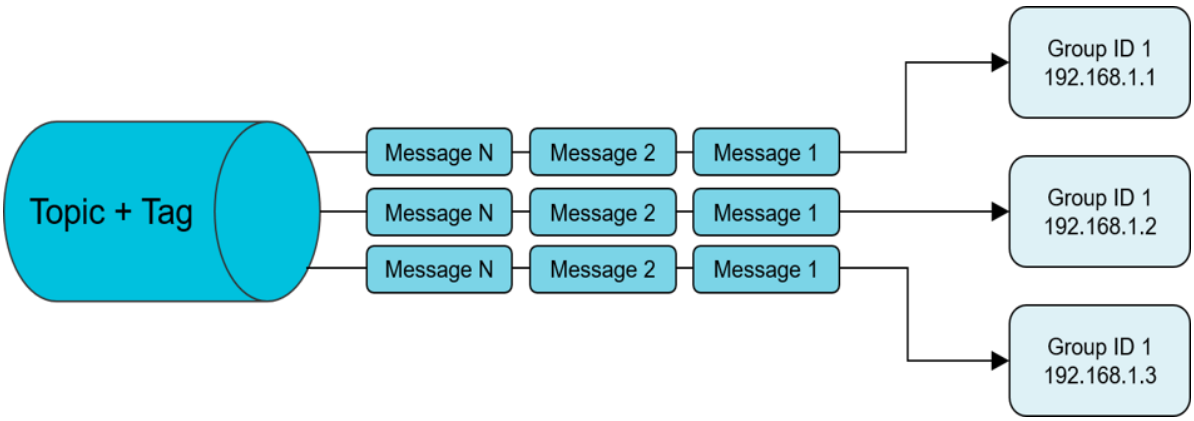


> 注意事项

- 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。

5.2.2.2 4.2.2.2 广播消费

> 广播消费指的是：一条消息被多个consumer消费，即使这些consumer属于同一个ConsumerGroup,消息也会被ConsumerGroup中的每个Consumer都消费一次，广播消费中ConsumerGroup概念可以认为在消息划分方面无意义。

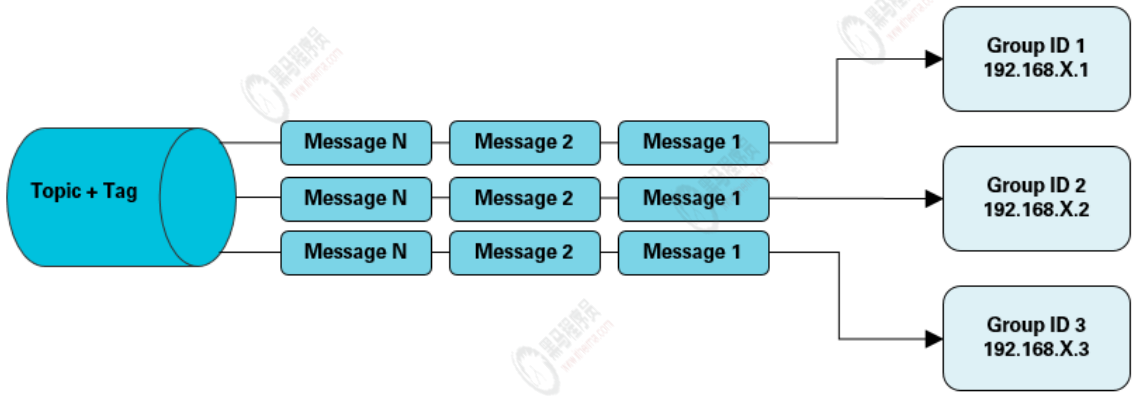


> 注意事项

- 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。

5.2.2.3 4.2.2.3 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。



> 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。

5.2.3 4.2.3 消息消费模式

> RocketMQ消息消费本质上是基于的拉（pull）模式，consumer主动向消息服务器broker拉取消息。

- 推消息模式下，消费进度的递增是由RocketMQ内部自动维护的；
- 拉消息模式下，消费进度的变更需要上层应用自己负责维护，RocketMQ只提供消费进度保存和查询功能。

5.2.3.1 4.2.3.1 推模式(PUSH)

> 我们上面使用的消费者都是PUSH模式，也是最常用的消费模式

由消息中间件（MQ消息服务器代理）主动地将消息推送给消费者；采用Push方式，可以尽可能实时地将消息发送给消费者进行消费。但是，在消费者的处理消息的能力较弱的时候(比如，消费者端的业务系统处理一条消息的流程比较复杂，其中的调用链路比较多导致消费时间比较久。概括起来地说就是**“慢消费问题”**)，而MQ不断地向消费者Push消息，消费者端的缓冲区可能会溢出，导致异常。

> 实现方式，代码上使用 **DefaultMQPushConsumer**

consumer把轮询过程封装了，并注册MessageListener监听器，取到消息后，唤醒MessageListener的consumeMessage()来消费，对用户而言，感觉消息是被推送（push）过来的。主要用的也是这种方式。

5.2.3.2 4.2.3.2 拉模式(PULL)

> RocketMQ的PUSH模式是由PULL模式来实现的

由消费者客户端主动向消息中间件（MQ消息服务器代理）拉取消息；采用Pull方式，如何设置Pull消息的频率需要重点去考虑，举个例子来说，可能1分钟内连续来了1000条消息，然后2小时内没有新消息产生（概括起来说就是**“消息延迟与忙等待”**）。如果每次Pull的时间间隔比较久，会增加消息的延迟，即消息到达消费者的时间加长，MQ中消息的堆积量变大；若每次Pull的时间间隔较短，但是在一段时间内MQ中并没有任何消息可以消费，那么会产生很多无效的Pull请求的RPC开销，影响MQ整体的网络性能。

5.2.3.3 4.2.3.3 注意事项

> **注意：RocketMQ 4.6.0版本后将弃用DefaultMQPullConsumer**
>
> DefaultMQPullConsumer方式需要手动管理偏移量，官方已经被废弃，将在2022年进行删除

DefaultLitePullConsumer

> 该类是官方推荐使用的手动拉取的实现类，偏移量提交由RocketMQ管理，不需要手动管理

5.2.4 4.2.4 消息确认机制

> consumer的每个实例是靠队列分配来决定如何消费消息的。那么消费进度具体是如何管理的，又是如何保证消息成功消费的？（RocketMQ有保证消息肯定消费成功的特性,失败则重试）

为了保证数据不被丢失，RocketMQ支持消息确认机制，即ack。发送者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。

5.2.4.1 4.2.4.1 确认消费

> 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。

```
consumer.registerMessageListener(new MessageListenerConcurrently() {
    @Override
    public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs,
ConsumeConcurrentlyContext context) {
        System.out.println(Thread.currentThread().getName() + " Receive New
Messages: " + msgs);
        execute();//执行真正消费
        return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
    }
})
```

5.2.4.2 4.2.3.2 消费异常

> 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。

```
consumer.registerMessageListener(new MessageListenerConcurrently() {
    @Override
    public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs,
ConsumeConcurrentlyContext context) {
        System.out.println(Thread.currentThread().getName() + " Receive New
Messages: " + msgs);
        execute();//执行真正消费
        return ConsumeConcurrentlyStatus.RECONSUME_LATER
    }
})
```

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而是这个消费租的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。

5.2.5 4.2.4 消息重试机制

5.2.5.1 4.2.4.1 顺序消息的重试

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

5.2.5.2 4.2.4.2 无序消息的重试

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

5.2.5.3 4.2.4.3 重试次数

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

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

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

5.2.5.4 4.2.4.4 和生产端重试区别

> 消费者和生产者的重试还是有区别的，主要有两点

- 默认重试次数：**Product默认是2次，而Consumer默认是16次。**
- 重试时间间隔：**Product是立刻重试，而Consumer是有一定时间间隔的。**它照`1S,5S,10S,30S,1M,2M····2H`进行重试。

注意：Product在**异步**情况重试失效，而对于Consumer在**广播**情况下重试失效。

5.2.5.5 4.2.4.5 重试配置方式

需要重试

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

- 方式1：返回RECONSUME_LATER（推荐）
- 方式2：返回Null
- 方式3：抛出异常

无需重试

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

```
//注册消息监听器
consumer.registerMessageListener(new MessageListenerConcurrently() {
    public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> list,
ConsumeConcurrentlyContext context) {
        //消息处理逻辑抛出异常，消息将重试。
        try {
            doConsumeMessage(list);
        }catch (Exception e){
            //捕获消费逻辑中的所有异常，并返回Action.CommitMessage;
            return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
        }
        //业务方正常消费
        return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
    }
});
```

5.3 4.3 死信队列

> 在正常情况下无法被消费(超过最大重试次数)的消息称为死信消息(Dead-Letter Message)，存储死信消息的特殊队列就称为死信队列(Dead-Letter Queue)

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

5.3.1 4.3.1 死信特性

5.3.1.1 4.3.1.1 死信消息特性

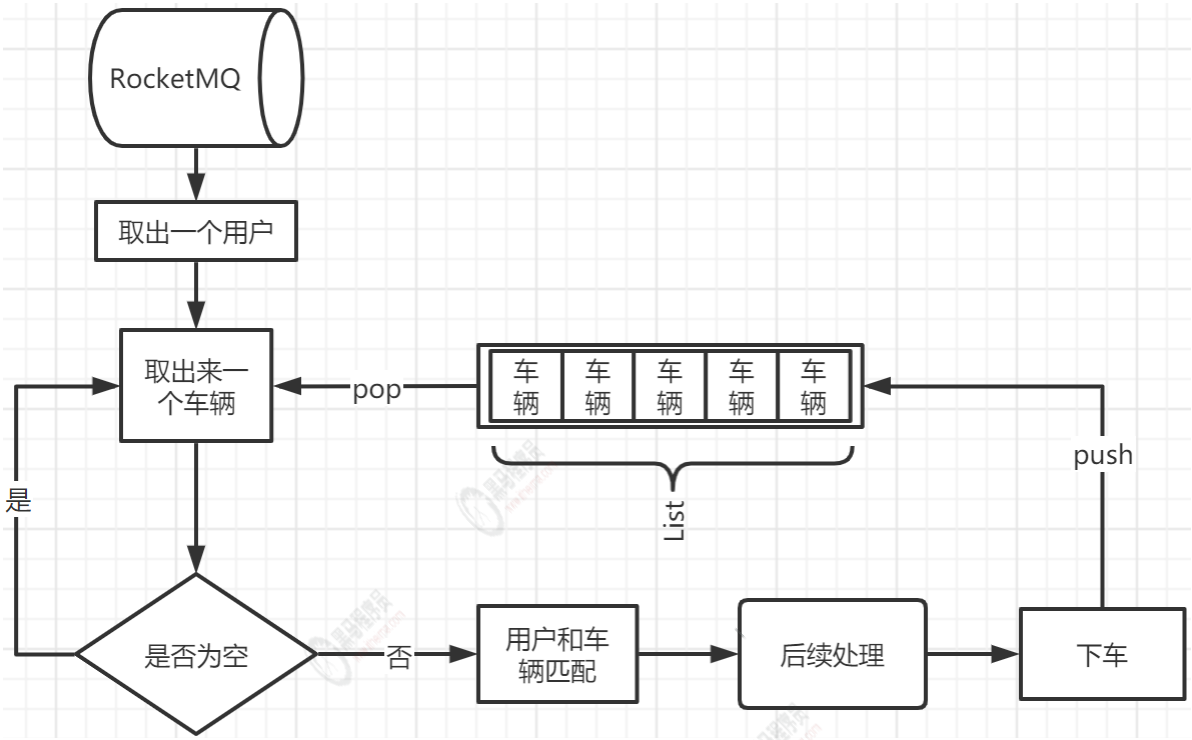
- 不会再被消费者正常消费
- 有效期与正常消息相同，均为 3 天，3 天后会被自动删除。故死信消息应在产生的 3 天内及时处理

5.3.1.2 4.3.1.2 死信队列特性

- 一个死信队列对应一个消费者组，而不是对应单个消费者实例
- 一个死信队列包含了对应的 Group ID 所产生的所有死信消息，不论该消息属于哪个 Topic
- 若一个 Group ID 没有产生过死信消息，则 RocketMQ 不会为其创建相应的死信队列

6 5 Redis 轮询队列

> redis队列中存放车辆信息，调度系统从队列中获取车辆信息，打车完成后再将车辆信息放回队列中



6.1 5.1 相关代码

6.1.1 5.1.1 redis获取车辆

> 从list左侧弹出一个车辆

```
/**
  * 从Redis List列表中拿取一个车辆ID
  * 如果没有获取到延时10S
  *
  * @return
  */
public String takeVehicle() {
    //从Redis List列表中拿取一个车辆ID
    return redisTemplate.opsForList().leftPop(DispatchConstant.VEHICLE_QUEUE, 1,
TimeUnit.SECONDS);
}
```

6.1.2 5.1.2 redis压入车辆

> 检查车辆状态，并从右侧压入车辆

```
/**
  * 设置车辆状态为Ready
  *
  * @param vehicleId
  */
public void readyDispatch(String vehicleId) {
    //检查车辆状态
    DispatchConstant.DispatchType vehicleDispatchType =
taxiVehicleStatus(vehicleId);
    //如果车辆时运行状态
    if (vehicleDispatchType.isRunning() || vehicleDispatchType.isReady()) {
        redisTemplate.opsForValue().set(DispatchConstant.VEHICLE_STATUS_PREFIX +
vehicleId, DispatchConstant.DispatchType.READY.toString());
```

```
        //从右侧压入车辆
        redisTemplate.opsForList().rightPush(DispatchConstant.VEHICLE_QUEUE,
vehicleId);
    }
}
```