RocketMQ消息重试

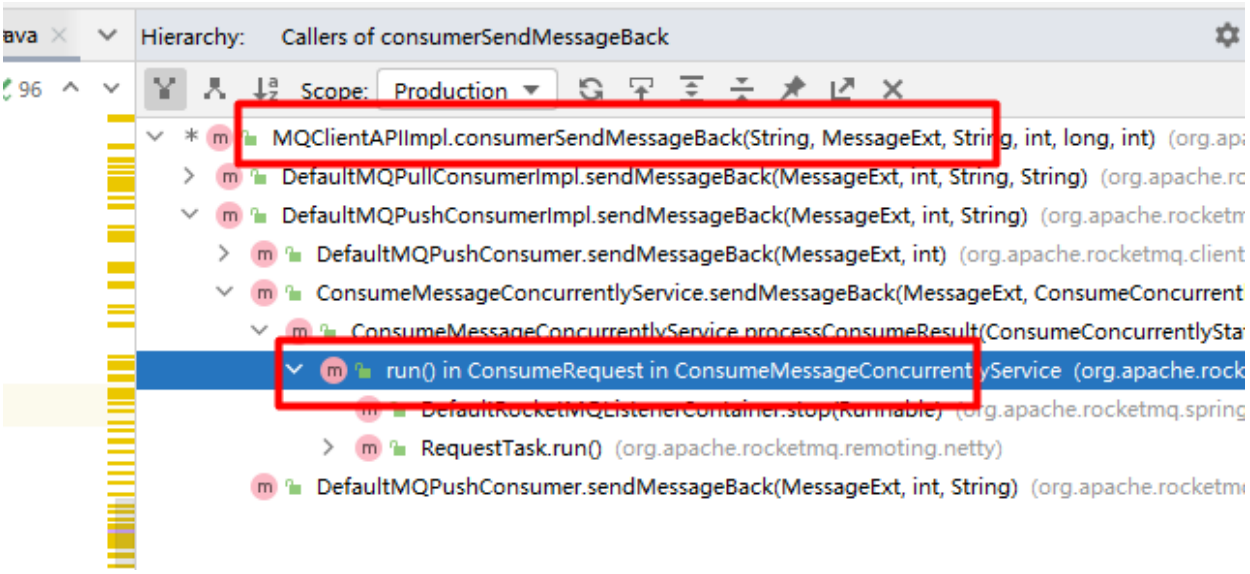
RocketMQ为什么广播消息不会重试

我们需要明确，只有当消费模式为 MessageModel.CLUSTERING(集群模式) 时，Broker 才会自动进行重试，对于广播消息是不会重试的。

为什么广播消息不会消息重试

注意：消费端的消息重试机制一定要在集群消费模式下才有效，广播消费模式下，RMQ是不会进行重试机制的，广播模式下，消息只消费一次，不管你有没有成功！！！

RocketMQ消息消费之后 不管消息是否消费成功，消费者client都会给broker发送一个请求，这个请求的代码就是RequestCode.*CONSUMER_SEND_MSG_BACK*:



在org.apache.rocketmq.client.impl.MQClientAPIImpl#consumerSendMessageBack方法中会床架你一个code为*CONSUMER_SEND_MSG_BACK 请求发送给Broker*

```
    */
    public void consumerSendMessageBack(
        final String addr,
        final MessageExt msg,
        final String consumerGroup,
        final int delayLevel,
        final long timeoutMillis,
        final int maxConsumeRetryTimes
    ) throws RemotingException, MQBrokerException, InterruptedException {
        ConsumerSendMsgBackRequestHeader requestHeader = new ConsumerSendMsgBackRequestHeader();
        RemotingCommand request = RemotingCommand.createRequestCommand(RequestCode.CONSUMER_SEND_MSG_BACK, requestHeader);

        /**
```

而consumerSendMessageBack方法是在ConsumeRequest的run方法中被调用的。

我们知道RokcetMQ中有两个ConsumeRequest对象

1，ConsumeMessageConcurrentlyService中的ConsumeRequest

在run方法中首先调用listener处理消息，处理完消息之后会执行ConsumeMessageConcurrentlyService对象的processConsumeResult方法，同时将ConsumeRequest对象传递过去

ConsumeMessageConcurrentlyService.this.processConsumeResult(status, context, consumeRequest对象);

```java
/***
 * org.apache.rocketmq.client.impl.consumer.
 * ConsumeMessageConcurrentlyService#processConsumeResult(org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlyStatus, org.apache.rocketmq.client.consumer.list
 *
 * @param status
 * @param context
 * @param consumeRequest
 */
public void processConsumeResult(
    final ConsumeConcurrentlyStatus status,
    final ConsumeConcurrentlyContext context,
    final ConsumeRequest consumeRequest
) {

    /**
     * ：根据消息监听器返回的结果 ， 计算 ackIndex ，如果返回 CONSUME_SUCCESS,
     * ackIndex 设置为 msgs.size()-1 ，如果返回 RECONSUME _LATER， ackIndex =-1 ， 这是为下
     * 文发送 msg back （ ACK ）消息做准备的 。
     *
     */
    int ackIndex = context.getAckIndex();

    if (consumeRequest.getMsgs().isEmpty())
        return;


    switch (status) {
        case CONSUME_SUCCESS:
            if (ackIndex >= consumeRequest.getMsgs().size()) {
```

```
                ackIndex = consumeRequest.getMsgs().size() - 1;
            }
            int ok = ackIndex + 1;
            int failed = consumeRequest.getMsgs().size() - ok;
            this.getConsumerStatsManager().incConsumeOKTPS(consumerGroup, consumeRequest.getMessageQueue().getTopic(), ok);
            this.getConsumerStatsManager().incConsumeFailedTPS(consumerGroup, consumeRequest.getMessageQueue().getTopic(), failed);
            break;
        case RECONSUME_LATER:
            ackIndex = -1;
            this.getConsumerStatsManager().incConsumeFailedTPS(consumerGroup, consumeRequest.getMessageQueue().getTopic(),
                consumeRequest.getMsgs().size());
            break;
        default:
            break;
    }

    /**
     *   * 消息消费者在消费一批消息后，需要记录该批消息已经消费完毕，否则 当消费者重新
     *                  * 启动时又得从消息消费队列的开始消费，这显然是不能接受的 。 从 5.6.1 节也可以看到，一
     *                  * 次消息消费后会从 ProcesQueue 处理队列中移除该批消息，返回 ProceeQueue 最小偏移量 ，
     *                  * 并存入消息进度表中 。 那消息进度文件存储在哪合适呢？
     *                  *
     *                  * 广播模式 ： 同一个消费组的所有消息消费者都需要消费主题下的所有消息，也就是同
     *                  * 组内的消费者的消息消费行为是对立的，互相不影响，故消息进度需要独立存储，最理想
     *                  * 的存储地方应该是与消费者绑定 。
     *                  * 集群模式：同一个消费组内的所有消息消费者共享消息主题下的所有消息， 同 一条消
     *                  * 息（同一个消息消费队列）在同一时间只会被消费组内的一个消费者消费，并且随着消费队
     *                  * 列的动态变化重新负载，所以消费进度需要保存在一个每个消费者都能访问到的地方。
     *
     *
     *                  ======================
     * RocketMQ消息重试
     * 我们需要明确，只有当消费模式为 MessageModel.CLUSTERING(集群模式) 时，Broker 才会自动进行重试，对于广播消息是不会重试的。
     * 为什么广播消息不会消息重试
     * 注意：消费端的消息重试机制一定要在集群消费模式下才有效，广播消费模式下，RMQ是不会进行重试机制的，广播模式下，消息只消费一次，不管你有没有成功！！！
     *
     *
     */
    switch (this.defaultMQPushConsumer.getMessageModel()) {
        case BROADCASTING:
            /**
             * 如果是广播模式， 业务方返回 RECONSUME_LATER ，消息并不会重新被消
             * 费，只是以警告级别输出到日志文件
             */
            for (int i = ackIndex + 1; i < consumeRequest.getMsgs().size(); i++) {
                MessageExt msg = consumeRequest.getMsgs().get(i);
                log.warn("BROADCASTING, the message consume failed, drop it, {}", msg.toString());
            }
            break;
        case CLUSTERING:
            /**
             *  如果是集群模式， 消息消费成功，由于 acklndex =consumeRequest.getMsgs().size()-1， 故 i=acklndex + 1 等于 consumeRequest.getMsgs().size() ，
             * 并不会执行 sendMessageBack 。
             */
            List<MessageExt> msgBackFailed = new ArrayList<MessageExt>(consumeRequest.getMsgs().size());
            /**
             * 消息消费成功的时候 acklindex=msgs.size()-1, i=actIndex+1=msgs.size 所以并不会执行下面的for方法
             *
             *  只有在业务方返回 RECONSUME_LATER 时（此时ackIndex=-1），该批消息都
             * 需要发 ACK 消息，如果消息发送 ACK 失败，则直接将本批 ACK 消费发送失败的消息再
             * 次封装为 ConsumeRequest ，然后延迟 5s后重新消费 。 如果 ACK 消息发送成功，则该消息
             * 会延迟消费.
             *
             * 如果消息监听器返回的消费结果为 RECONSUME LATER ，则需要将这些消息发送
             * 给 Broker 延迟消息 。 如果发送 ACK 消息失败，将延迟 5s 后提交线程池进行消费。 ACK
             * 消息发送的网络客户端入口： MQClientAPIImpl#consumerSendMessageBack
             *
             */
            for (int i = ackIndex + 1; i < consumeRequest.getMsgs().size(); i++) {
                MessageExt msg = consumeRequest.getMsgs().get(i);
                boolean result = this.sendMessageBack(msg, context);
                if (!result) {
                    msg.setReconsumeTimes(msg.getReconsumeTimes() + 1);
                    msgBackFailed.add(msg);
                }
            }

            if (!msgBackFailed.isEmpty()) {
                consumeRequest.getMsgs().removeAll(msgBackFailed);

                this.submitConsumeRequestLater(msgBackFailed, consumeRequest.getProcessQueue(), consumeRequest.getMessageQueue());
            }
            break;
        default:
            break;
    }

    /**
     *
     * 从 Process Queue 中 移除这批 消息， 这里 返回的偏移量是移除该批消息后最
     * 小的偏移量，然后用该偏移量更新消息消 费进度 ，以便在消费者重启后能从上一次的消费
     * 进度开始消费，避免消息重复消费 。 值得重点注意的是当消息监听器返回 RECONSUME_LATER ，消息消 费进度也会向前推进，用 Proces sQueue 中 最小的队列偏移量调用消息消费
     * 进度存储器 OffsetStore 更新消费进度，这是因为当返回 RECONSUME_LATER, RocketMQ
     * 会创建一条与原先消息属性相同的消息，拥有一个唯一 的新 msgId ，并存储原消息 ID ，该
     * 消息会存入到 commitlog 文件中，与原先的消息没有任何关联，那该消息当然也会进入到
     * ConsuemeQueue 队列中，将拥有一个全新的队列偏移量 。
     *
     * 对消息消费的其中两个重要步骤进行详细分析， ACK 消息发送与消息消费进度存储
     *
     */
    long offset = consumeRequest.getProcessQueue().removeMessage(consumeRequest.getMsgs());
    if (offset >= 0 && !consumeRequest.getProcessQueue().isDropped()) {
        this.defaultMQPushConsumerImpl.getOffsetStore().updateOffset(consumeRequest.getMessageQueue(), offset, true);
    }
}
```

在这里对集群和广播两种模式判断

这个地方会发送消息是否消费成功的通知请求给Broker

```
public boolean sendMessageBack(final MessageExt msg, final ConsumeConcurrentlyContext context) {
    int delayLevel = context.getDelayLevelWhenNextConsume();

    // Wrap topic with namespace before sending back message.
    msg.setTopic(this.defaultMQPushConsumer.withNamespace(msg.getTopic()));
    try {
        this.defaultMQPushConsumerImpl.sendMessageBack(msg, delayLevel, context.getMessageQueue().getBrokerName());
        return true;
    } catch (Exception e) {
        log.error("sendMessageBack exception, group: " + this.consumerGroup + " msg: " + msg.toString(), e);
    }

    return false;
}
```

```java
public void sendMessageBack(MessageExt msg, int delayLevel, final String brokerName)
    throws RemotingException, MQBrokerException, InterruptedException, MQClientException {
    try {
        String brokerAddr = (null != brokerName) ? this.mQClientFactory.findBrokerAddressInPublish(brokerName)
            : RemotingHelper.parseSocketAddressAddr(msg.getStoreHost());
        this.mQClientFactory.getMQClientAPIImpl().consumerSendMessageBack(brokerAddr, msg,
            this.defaultMQPushConsumer.getConsumerGroup(), delayLevel,  timeoutMillis: 5000, getMaxReconsumeTimes());
    } catch (Exception e) {
        log.error("sendMessageBack Exception, " + this.defaultMQPushConsumer.getConsumerGroup(), e);

        Message newMsg = new Message(MixAll.getRetryTopic(this.defaultMQPushConsumer.getConsumerGroup()), msg.getBody());

        String originMsgId = MessageAccessor.getOriginMessageId(msg);
        MessageAccessor.setOriginMessageId(newMsg, UtilAll.isBlank(originMsgId) ? msg.getMsgId() : originMsgId);

        newMsg.setFlag(msg.getFlag());
        MessageAccessor.setProperties(newMsg, msg.getProperties());
        MessageAccessor.putProperty(newMsg, MessageConst.PROPERTY_RETRY_TOPIC, msg.getTopic());
        MessageAccessor.setReconsumeTime(newMsg, String.valueOf(msg.getReconsumeTimes() + 1));
        MessageAccessor.setMaxReconsumeTimes(newMsg, String.valueOf(getMaxReconsumeTimes()));
        MessageAccessor.clearProperty(newMsg, MessageConst.PROPERTY_TRANSACTION_PREPARED);
        newMsg.setDelayTimeLevel(3 + msg.getReconsumeTimes());

        this.mQClientFactory.getDefaultMQProducer().send(newMsg);
    } finally {
        msg.setTopic(NamespaceUtil.withoutNamespace(msg.getTopic(), this.defaultMQPushConsumer.getNamespace()));
    }
}
```

```java
     * @throws InterruptedException
     */
    public void consumerSendMessageBack(
        final String addr,
        final MessageExt msg,
        final String consumerGroup,
        final int delayLevel,
        final long timeoutMillis,
        final int maxConsumeRetryTimes
    ) throws RemotingException, MQBrokerException, InterruptedException {
        ConsumerSendMsgBackRequestHeader requestHeader = new ConsumerSendMsgBackRequestHeader();
        RemotingCommand request = RemotingCommand.createRequestCommand(RequestCode.CONSUMER_SEND_MSG_BACK, requestHeader);

        /**
         * : 消费组名 。
         */
        requestHeader.setGroup(consumerGroup);
        /**
         * : 消息主题。
         */
        requestHeader.setOriginTopic(msg.getTopic());
        /**
         * 消息物理偏移量
         */
        requestHeader.setOffset(msg.getCommitLogOffset());
        /**
         * : 延迟级别， RcketMQ 不支持精确的定时消息调度，而是提供几个延时
         * 级别， Messages toreConfig# messageD已 layLevel = " ls Ss  10s 30s  lm 2m  3m 4m Sm  6m 7m 8m
         * 9m  lOm 20m 30m lh 2h "，如果 delayLevel= I 表示延迟缸，delayLevel=2 则表示延迟 1 Os 。
         */
        requestHeader.setDelayLevel(delayLevel);
        /**
         * 消息 ID 。
         */
        requestHeader.setOriginMsgId(msg.getMsgId());
        /**
         *  最大重新消费次数，默认为 16 次。
         */
        requestHeader.setMaxReconsumeTimes(maxConsumeRetryTimes);
```

Broker收到这个消息之后如何处理呢?

Broker注册了针对这个code使用什么处理器处理SendMessageProcessor

```
this.remotingServer.registerProcessor(RequestCode.CONSUMER_SEND_MSG_BACK, sendProcessor, this.sendMessageExecutor);
```

最终会调用 SendMessageProcessor处理器的asyncConsumerSendMsgBack方法处理，在这个方法中会判断消息的重试次数，如果大于最大重试次数就放入死信队列。如果没有大于最大重试次数，则进入重试队列

```
private CompletableFuture<RemotingCommand> asyncConsumerSendMsgBack(ChannelHandlerContext ctx,
                                                                    RemotingCommand request) throws RemotingCommandException {
    final RemotingCommand response = RemotingCommand.createResponseCommand(null);
    final ConsumerSendMsgBackRequestHeader requestHeader =
            (ConsumerSendMsgBackRequestHeader)request.decodeCommandCustomHeader(ConsumerSendMsgBackRequestHeader.class);
    String namespace = NamespaceUtil.getNamespaceFromResource(requestHeader.getGroup());
    if (this.hasConsumeMessageHook() && !UtilAll.isBlank(requestHeader.getOriginMsgId())) {
        ConsumeMessageContext context = buildConsumeMessageContext(namespace, requestHeader, request);
        this.executeConsumeMessageHookAfter(context);
    }
    /**
     * 获取消 费组的订阅配置信息， 如果配置信息为空返回配置组信息不存在错误 ，
     * 如果重试队列数量小于 1 ，则直接返回成功 ， 说明该消费组不支持重试。
     */
    SubscriptionGroupConfig subscriptionGroupConfig =
        this.brokerController.getSubscriptionGroupManager().findSubscriptionGroupConfig(requestHeader.getGroup());
    if (null == subscriptionGroupConfig) {
        response.setCode(ResponseCode.SUBSCRIPTION_GROUP_NOT_EXIST);
        response.setRemark("subscription group not exist, " + requestHeader.getGroup() + " "
            + FAQUrl.suggestTodo(FAQUrl.SUBSCRIPTION_GROUP_NOT_EXIST));
        return CompletableFuture.completedFuture(response);
    }
    if (!PermName.isWriteable(this.brokerController.getBrokerConfig().getBrokerPermission())) {
        response.setCode(ResponseCode.NO_PERMISSION);
        response.setRemark("the broker[" + this.brokerController.getBrokerConfig().getBrokerIP1() + "] sending message is forbidden");
        return CompletableFuture.completedFuture(response);
    }

    if (subscriptionGroupConfig.getRetryQueueNums() <= 0) {
        response.setCode(ResponseCode.SUCCESS);
        response.setRemark(null);
        return CompletableFuture.completedFuture(response);
    }

    /**
     * 创建重试主题，重试主题名称 ： %RETRY%+消费组名称，并从重试队列中随
     * 机选择一个队列 ，并构建 TopicConfig 主题配置信息 。
     */
    String newTopic = MixAll.getRetryTopic(requestHeader.getGroup());
    int queueIdInt = Math.abs(this.random.nextInt() % 99999999) % subscriptionGroupConfig.getRetryQueueNums();
    int topicSysFlag = 0;
    if (requestHeader.isUnitMode()) {
        topicSysFlag = TopicSysFlag.buildSysFlag(false, true);
    }

    TopicConfig topicConfig = this.brokerController.getTopicConfigManager().createTopicInSendMessageBackMethod(
        newTopic,
        subscriptionGroupConfig.getRetryQueueNums(),
        PermName.PERM_WRITE | PermName.PERM_READ, topicSysFlag);
    if (null == topicConfig) {                                  重试队列
        response.setCode(ResponseCode.SYSTEM_ERROR);
        response.setRemark("topic[" + newTopic + "] not exist");
        return CompletableFuture.completedFuture(response);
    }

    if (!PermName.isWriteable(topicConfig.getPerm())) {
        response.setCode(ResponseCode.NO_PERMISSION);
        response.setRemark(String.format("the topic[%s] sending message is forbidden", newTopic));
        return CompletableFuture.completedFuture(response);
    }
    /**
     * 根据消息物理偏移量从 commitlog 文件中 获取消息， 同时将消息的主题存入属
     * 性中
     */
    MessageExt msgExt = this.brokerController.getMessageStore().lookMessageByOffset(requestHeader.getOffset());
    if (null == msgExt) {
        response.setCode(ResponseCode.SYSTEM_ERROR);
        response.setRemark("look message by offset failed, " + requestHeader.getOffset());
        return CompletableFuture.completedFuture(response);
    }

    final String retryTopic = msgExt.getProperty(MessageConst.PROPERTY_RETRY_TOPIC);
    if (null == retryTopic) {
        MessageAccessor.putProperty(msgExt, MessageConst.PROPERTY_RETRY_TOPIC, msgExt.getTopic());
    }
    msgExt.setWaitStoreMsgOK(false);
```

```
        int delayLevel = requestHeader.getDelayLevel();

        /**
         *   /**
         *          * 设置消息重试次数， 如果消息 已 重试次数超过 maxReconsumeTimes ，再次改变
         *          * newTopic 主题为 DLQ （” %DLQ%”），该主题的权限为只写，说明消息一旦进入到 DLQ 队
         *          * 列中， RocketMQ 将不负责再次调度进行消费了， 需要人工干预。
         *
         */
        int maxReconsumeTimes = subscriptionGroupConfig.getRetryMaxTimes();
        if (request.getVersion() >= MQVersion.Version.V3_4_9.ordinal()) {
            maxReconsumeTimes = requestHeader.getMaxReconsumeTimes();
        }

        if (msgExt.getReconsumeTimes() >= maxReconsumeTimes
            || delayLevel < 0) {
            newTopic = MixAll.getDLQTopic(requestHeader.getGroup());
            queueIdInt = Math.abs(this.random.nextInt() % 99999999) % DLQ_NUMS_PER_GROUP;

            topicConfig = this.brokerController.getTopicConfigManager().createTopicInSendMessageBackMethod(newTopic,
                    DLQ_NUMS_PER_GROUP,
                    PermName.PERM_WRITE, 0);                                    死信队列
            if (null == topicConfig) {
                response.setCode(ResponseCode.SYSTEM_ERROR);
                response.setRemark("topic[" + newTopic + "] not exist");
                return CompletableFuture.completedFuture(response);
            }
        } else {
            if (0 == delayLevel) {
                delayLevel = 3 + msgExt.getReconsumeTimes();
            }
            msgExt.setDelayTimeLevel(delayLevel);
        }


        /**
         *  根据原先的消息创建一个新的消息对象，重试消息会拥有自 己 的唯一消息 ID
         *（ ms g ld ）并存入到 comm i tlo g 文件 中，并不会去更新原先消息 ， 而是会将原先的主题、 消息
         * ID 存入消息的属性 中 ， 主题名 称为重试主题 ， 其他属性与原先消息保持相同 。
         */
        MessageExtBrokerInner msgInner = new MessageExtBrokerInner();
        msgInner.setTopic(newTopic);
        msgInner.setBody(msgExt.getBody());
        msgInner.setFlag(msgExt.getFlag());
        MessageAccessor.setProperties(msgInner, msgExt.getProperties());
        msgInner.setPropertiesString(MessageDecoder.messageProperties2String(msgExt.getProperties()));
        msgInner.setTagsCode(MessageExtBrokerInner.tagsString2tagsCode(null, msgExt.getTags()));

        msgInner.setQueueId(queueIdInt);
        msgInner.setSysFlag(msgExt.getSysFlag());
        msgInner.setBornTimestamp(msgExt.getBornTimestamp());
        msgInner.setBornHost(msgExt.getBornHost());
        msgInner.setStoreHost(msgExt.getStoreHost());
        msgInner.setReconsumeTimes(msgExt.getReconsumeTimes() + 1);

        String originMsgId = MessageAccessor.getOriginMessageId(msgExt);
        MessageAccessor.setOriginMessageId(msgInner, UtilAll.isBlank(originMsgId) ? msgExt.getMsgId() : originMsgId);
        /**
         * ：将消息存入到 CommitLog 文件中 ,这里的MessageStore的asyncPutMessage 会调用 commitLog.putMessage(msg);, 这里
         * 想再重点突 出 一个机制，消息重试机制依托于定时任务实现，
         *
         * ACK 消息存入 CommitLog 文件后 ，将依托 RocketMQ 定时消息机制在延迟时间到期
         * 后再次将消息拉取，提交消费线程池，
         */
        CompletableFuture<PutMessageResult> putMessageResult = this.brokerController.getMessageStore().asyncPutMessage(msgInner);
        return putMessageResult.thenApply((r) -> {
            if (r != null) {
                switch (r.getPutMessageStatus()) {
                    case PUT_OK:
                        String backTopic = msgExt.getTopic();
                        String correctTopic = msgExt.getProperty(MessageConst.PROPERTY_RETRY_TOPIC);
                        if (correctTopic != null) {
                            backTopic = correctTopic;
                        }
                        this.brokerController.getBrokerStatsManager().incSendBackNums(requestHeader.getGroup(), backTopic);
                        response.setCode(ResponseCode.SUCCESS);
                        response.setRemark(null);
                        return response;
                    default:
                        break;
                }
                response.setCode(ResponseCode.SYSTEM_ERROR);
                response.setRemark(r.getPutMessageStatus().name());
                return response;
            }
            response.setCode(ResponseCode.SYSTEM_ERROR);
            response.setRemark("putMessageResult is null");
            return response;
        });
    }
```

2.ConsumeMessageOrderlyService中的ConsumeRequest

事实上只有在ConsumeMessageConcurrentlyService 处理消息的时候才会发送 code为 *CONSUMER_SEND_MSG_BACK的请求给Broker。*

如果是顺序消费，使用了ConsumeMessageOrderlyService，OrderlyService会直接将消息发送到奥重试队列；并发模式下是Broker将消息发送到重试队列

OrderlyService中有一个sendMessageBack方法，这个方法会被ConsumeRequest调用

```
public boolean sendMessageBack(final MessageExt msg) {
    try {
        // max reconsume times exceeded then send to dead letter queue.
        Message newMsg = new Message(MixAll.getRetryTopic(this.defaultMQPushConsumer.getConsumerGroup()), msg.getBody());
        String originMsgId = MessageAccessor.getOriginMessageId(msg);
        MessageAccessor.setOriginMessageId(newMsg, UtilAll.isBlank(originMsgId) ? msg.getMsgId() : originMsgId);
        newMsg.setFlag(msg.getFlag());
        MessageAccessor.setProperties(newMsg, msg.getProperties());
        MessageAccessor.putProperty(newMsg, MessageConst.PROPERTY_RETRY_TOPIC, msg.getTopic());
        MessageAccessor.setReconsumeTime(newMsg, String.valueOf(msg.getReconsumeTimes()));
        MessageAccessor.setMaxReconsumeTimes(newMsg, String.valueOf(getMaxReconsumeTimes()));
        MessageAccessor.clearProperty(newMsg, MessageConst.PROPERTY_TRANSACTION_PREPARED);
        newMsg.setDelayTimeLevel(3 + msg.getReconsumeTimes());

        this.defaultMQPushConsumer.getDefaultMQPushConsumerImpl().getmQClientFactory().getDefaultMQProducer().send(newMsg);
        return true;
    } catch (Exception e) {
        log.error("sendMessageBack exception, group: " + this.consumerGroup + " msg: " + msg.toString(), e);
    }

    return false;
}
```

创建一个消息，topic为重试队列

将消息发送到重试队列