消息队列 RocketMQ 版

开发指南

腾讯云

【联系我们】

我们致力于为您提供个性化的售前购买咨询服务，及相应的技术售后服务，任何问题请联系 4009100100或95716。

文档目录

开发指南
消息类型
普通消息

最近更新时间：2024-10-14 11:34:25

普通消息是一种基础的消息类型，由生产投递到指定 Topic 后，被订阅了该 Topic 的消费者所消费。普通消息的 Topic 中无顺序的概念，可以使用多个分区数来提升消息的生产和消费效率，在吞吐量巨大时其性能最好。
普通消息区别于有特性的定时与延迟消息、顺序消息和事务消息。这四种类型的消息对应的 Topic 不能混用，只能用于收发相同类型的消息，例如普通消息的 Topic 只能用于收发普通消息，不能用于收发延迟消息、顺序消息和事务消息。

定时与延迟消息

最近更新时间：2024-10-14 12:43:51

本文主要介绍消息队列 TDMQ RocketMQ 版中定时与延迟消息的概念和使用方式。

相关概念

- **定时消息**：消息在发送至服务端后，实际业务并不希望消费端马上收到这条消息，而是推迟到某个时间点被消费，这类消息统称为定时消息。
- **延时消息**：消息在发送至服务端后，实际业务并不希望消费端马上收到这条消息，而是推迟一段时间后再被消费，这类消息统称为延时消息。

实际上，延时消息可以看成是定时消息的一种特殊用法，其实现的最终效果和定时消息是一致的。

使用方式

开源 Apache RocketMQ 没有提供让用户自由设定延时时间的 API 的，TDMQ RocketMQ 版为了保证向开源 RocketMQ Client 的兼容，设计了一种通过添加 message 的 property 键值对来指定消息发送时间的方法。该方法只要在需要定时发送消息的 `property` 属性中增加 `__STARTDELIVERTIME` 属性值，就能在一定范围内（40天）实现该消息在任意时间的定时发送。延时消息则可以先通过计算得到定时发送的时间点，再以定时消息的形式发送。

下面给出一段具体代码示例来展示如何使用 TDMQ RocketMQ 版的定时和延时消息，查看完整示例 >>

延时消息先通过 `System.currentTimeMillis() + delayTime` 计算得到定时发送的时间点，再以定时消息的方式发送。

4.x 客户端

```
Message msg = new Message("test-topic", ("message
content").getBytes(StandardCharsets.UTF_8));

// 设定消息在10秒之后被发送
long delayTime = System.currentTimeMillis() + 10000;
// 将 __STARTDELIVERTIME 设定到 msg 的属性中
msg.putUserProperty("__STARTDELIVERTIME",
String.valueOf(delayTime));

SendResult result = producer.send(msg);
```

```
System.out.println("Send delay message: " + result);
```

5.x 客户端

```
Duration messageDelayTime = Duration.ofSeconds(10);
final Message message = provider.newMessageBuilder()
// Set topic for the current message.
.setTopic(topic)
 // Message secondary classifier of message besides topic.
.setTag(tag)
// Key(s) of the message, another way to mark message besides
message id.
.setKeys("yourMessageKey-3ee439f945d7")
// Set expected delivery timestamp of message. 设置延迟消息的时间
.setDeliveryTimestamp(System.currentTimeMillis() +
messageDelayTime.toMillis())
.setBody(body)
.build();
```

使用限制

- 使用延时消息时，请确保客户端的机器时钟和服务端的机器时钟（所有地域均为 UTC+8 北京时间）保持一致，否则会有时差。
- 定时和延时消息在精度上会有1秒左右的偏差。
- 关于定时和延时消息的时间范围，根据不同的集群规格，有不同的限制，可以参考产品系列 。
- 使用定时消息时，设置的时刻在当前时刻以后才会有定时效果，否则消息将被立即发送给消费者。

顺序消息

最近更新时间：2025-02-28 11:35:22

顺序消息是消息队列 RocketMQ 提供的一种高级消息类型，对于一个指定的Topic，消息严格按照先进先出（FIFO）的原则进行消息发布和消费，即先发送的消息先消费，后发送的消息后消费。
顺序消息适用于对消息发送和消费顺序有严格要求的情况。

使用场景

顺序消息和普通消息的对比如下：

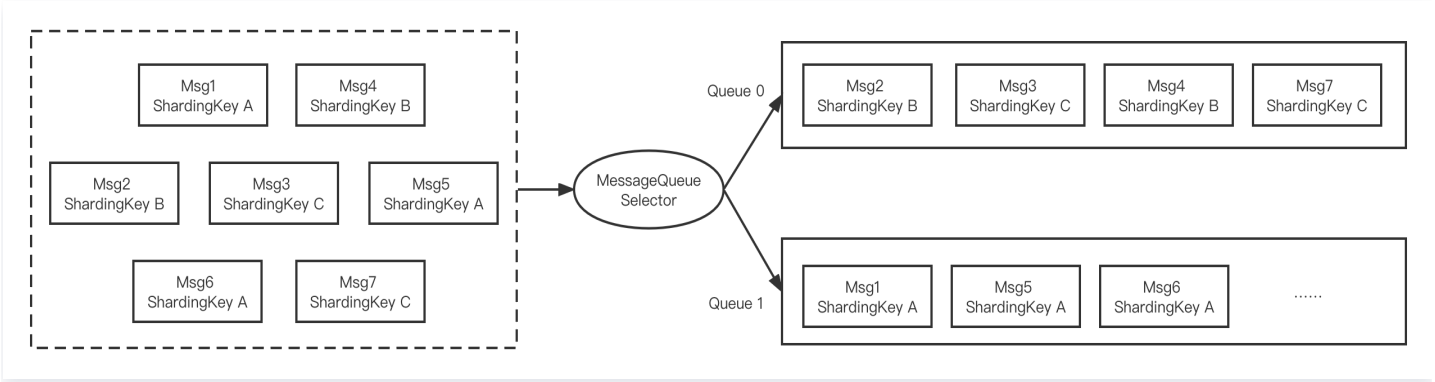
消息类型	消费顺序	性能	适用场景
普通消息	无顺序	高	适用于对吞吐量要求高，且对生产和消费顺序无要求
顺序消息	指定的 Topic 内的消息遵循先入先出（FIFO）规则	一般	吞吐量要求一般，但是要求特定的 Topic 严格地按照 FIFO 原则进行消息发布和消费的场景

对应到具体的业务场景，顺序消息可以被用在以下场景中：

- **订单创建场景：**在一些电商系统中，同一个订单相关的创建订单消息、订单支付消息、订单退款消息、订单物流消息必须严格按照先后顺序来进行生产或者消费，否则消费中传递订单状态会发生紊乱，影响业务的正常进行。因此，该订单的消息必须按照一定的顺序在客户端和消息队列中进行生产和消费，同时消息之间有先后的依赖关系，后一条消息需要依赖于前一条消息的处理结果。
- **日志同步场景：**在有序事件处理或者数据实时增量同步的场景中，顺序消息也能发挥较大的作用，如同步 mysql 的 binlog 日志时，需要保证数据库的操作是有顺序的。
- **金融场景：**在一些撮合交易的场景下，例如某些证券交易，在价格相同的情况下，先出价者优先处理，则需要按照FIFO的方式生产和消费顺序消息。

实现原理

在 RocketMQ 中支持顺序消息的原理如下图所示。我们可以按照某一个标准对消息进行分区（例如图中的 ShardingKey），同一个ShardingKey 的消息会被分配到同一个队列中，并按照顺序被消费。



生产顺序消息

顺序消息的代码如下所示：

```
public class Producer {
    public static void main(String[] args) throws
UnsupportedEncodingException {
        try {
            // 实例化消息生产者Producer
            DefaultMQProducer producer = new
DefaultMQProducer(groupName,
                // ACL权限
            new AclClientRPCHook(new SessionCredentials(ACCESS_KEY,
SECRET_KEY)), true, null);
            // 设置NameServer的地址
            producer.setNamesrvAddr("rmq-
xxx.rocketmq.xxxtencenttdmq.com:8080");
            // 启动Producer实例
            producer.start();

            String[] tags = new String[] {"TagA", "TagB", "TagC",
"TagD", "TagE"};
            for (int i = 0; i < 100; i++) {
                int orderId = i % 10;
                Message msg =
                    new Message("TopicTest", tags[i % tags.length],
"KEY" + i,
                        ("Hello RocketMQ " +
i).getBytes(RemotingHelper.DEFAULT_CHARSET));
                SendResult sendResult = producer.send(msg, new
MessageQueueSelector() {
```

```
                    @Override
                    public MessageQueue select(List<MessageQueue> mqs,
Message msg, Object arg) {
                        Integer id = (Integer) arg;
                        int index = id % mqs.size();
                        return mqs.get(index);
                    }
                }, orderId);

                System.out.printf("%s%n", sendResult);
            }

            producer.shutdown();
        } catch (MQClientException | RemotingException |
MQBrokerException | InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

这里的区别主要是调用了 `SendResult send(Message msg, MessageQueueSelector selector, Object arg)` 方法，`MessageQueueSelector` 是队列选择器，arg 是一个 Java Object 对象，可以传入作为消息发送分区的分类标准。

`MessageQueueSelector` 的接口如下：

```
public interface MessageQueueSelector {
    MessageQueue select(final List<MessageQueue> mqs, final Message msg,
final Object arg);
}
```

其中 mqs 是可以发送的队列，msg 是消息，arg 是上述 send 接口中传入的 Object 对象，返回的是该消息需要发送到的队列。上述例子里，是以 orderId 作为分区分类标准，对所有队列个数取余，来对将相同 orderId 的消息发送到同一个队列中。

生产环境中建议选择最细粒度的分区键进行拆分，例如，将订单ID、用户ID作为分区键关键字，可实现同一终端用户的消息按照顺序处理，不同用户的消息无需保证顺序。

> **注意：**
> 为了保证消息的高可用，目前TDMQ RocketMQ版不支持单队列的 “全局顺序消息”（已经创建了全局顺序消息的用户可以正常使用）；如果您想保证全局的顺序性，您可以通过使用一致的 ShardingKey 来实现。

消费顺序消息

顺序消费代码如下：

```
/**
 * Description: 顺序消费者
 */
public class OrderConsumer {
    /**
     * topic名称
     */
    private static final String TOPIC_NAME = "order_topic";

    /**
     * 消费者组名称
     */
    private static final String GROUP_NAME = "group2";
    public static void main(String[] args) throws Exception {
        // 创建消息消费者
        // 实例化消费者
        DefaultMQPushConsumer consumer = new
DefaultMQPushConsumer(GROUP_NAME,
                new AclClientRPCHook(new SessionCredentials("accessKey",
"secretKey")),
                new AllocateMessageQueueAveragely(), true, null);
        // 设置NameServer的地址
        consumer.setNamesrvAddr("rmq-
xxx.rocketmq.xxxtencenttdmq.com:8080");

consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_OFFSET)
;
        /*
         * 设置Consumer第一次启动是从队列头部开始消费还是队列尾部开始消费<br>
         * 如果非第一次启动，那么按照上次消费的位置继续消费
         */

consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_OFFSET)
;
        // 订阅topic中的所有的信息
        consumer.subscribe(TOPIC_NAME, "*");
        consumer.registerMessageListener(new MessageListenerOrderly() {
            @Override
```

```
            public ConsumeOrderlyStatus consumeMessage(List<MessageExt>
msgs, ConsumeOrderlyContext context) {
                context.setAutoCommit(true);
                for (MessageExt msg : msgs) {
                    // 可以看到每个queue有唯一的consume线程来消费，订单对每个
queue(分区)有序
                    System.out.println("consumeThread=" +
Thread.currentThread().getName() + ", queueId=" + msg.getQueueId() + ",
msgId=" + msg.getMsgId() + ", content:" + new String(msg.getBody()));
                }
                try {
                    // 模拟业务逻辑处理中...
                    TimeUnit.SECONDS.sleep(1);
                } catch (Exception e) {
                    e.printStackTrace();
                }
                return ConsumeOrderlyStatus.SUCCESS;
            }
        });
        consumer.start();
        System.out.println("Consumer Started.");
    }
}
```

事务消息

最近更新时间：2025-06-11 11:56:12

本文主要介绍消息队列 TDMQ RocketMQ 版中事务消息的概念、技术原理、应用场景和使用方式。

相关概念

事务消息是 RocketMQ 提供的一种高级特性消息，通过将二阶段提交的动作和本地事务绑定，来保障分布式场景下消息生产和本地事务的最终一致性，相比普通消息，主要是扩展了二次确认和本地事务状态回查补偿的机制。

1. 半消息 half message：

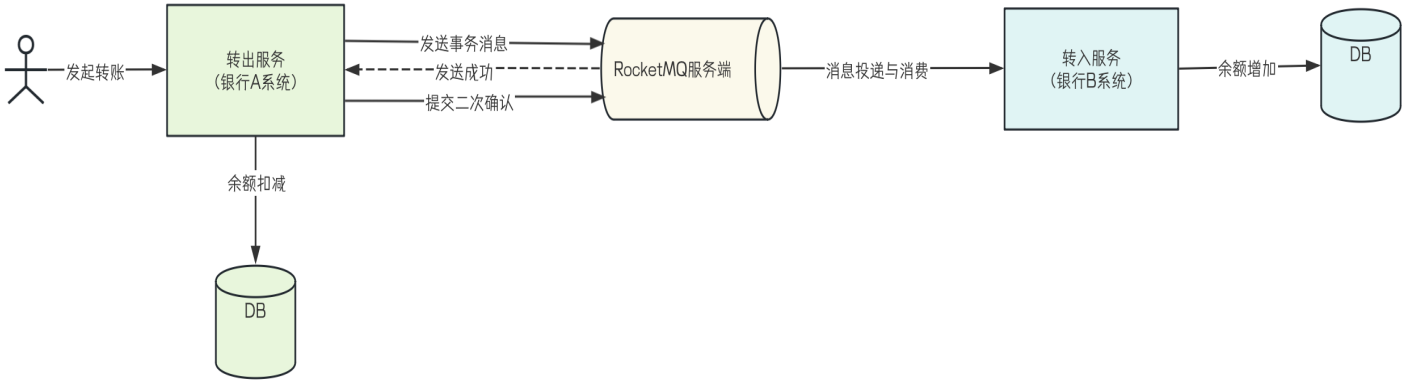
   生产者发送事务消息到 RocketMQ 服务端后，消息会被持久化并标记为“暂不可投递”的状态，直到本地事务执行完成并确认后，消息才会决定是否对消费者可见，此状态下的消息，称为半消息（half message）。
2. 二阶段提交

   实现事务最终一致性的关键机制，一阶段为发送 half message，二阶段为生产者执行本地事务，并根据执行结果向 RocketMQ 服务端提交 commit（允许投递）或 rollback（丢弃消息）的确认结果，以此来决定 half message 的去留。
3. OP 消息：

   用于给消息状态打标记，没有对应 OP 消息的 half message，就说明二阶段确认状态未知，需要 RocketMQ 服务端进行本地事务状态主动回查，OP 消息的内容为对应的 half message 的存储的Offset。
4. 相关 topic：
   - real topic：业务真实 topic，生产者发消息时指定的 topic 值。
   - half topic：系统 topic，topic名称为 RMQ_SYS_TRANS_HALF_TOPIC，用于存储 half message。
   - op topic：系统 topic，topic名称为 MQ_SYS_TRANS_OP_HALF_TOPIC，用于存储 OP 消息， half message 二次状态确认后，不管是 commit 还是 rollback，都会写入一条对应的 OP 消息到这个 topic。

应用场景

例如在跨行转账这一典型的分布式场景中，必须严格保证转出方账户扣款与转入方账户入账的最终一致性，我们可以采用 TDMQ RocketMQ 版的事务消息来实现这一功能，具体分为以下三个阶段：



1. 阶段一：发送事务消息（准备转账）

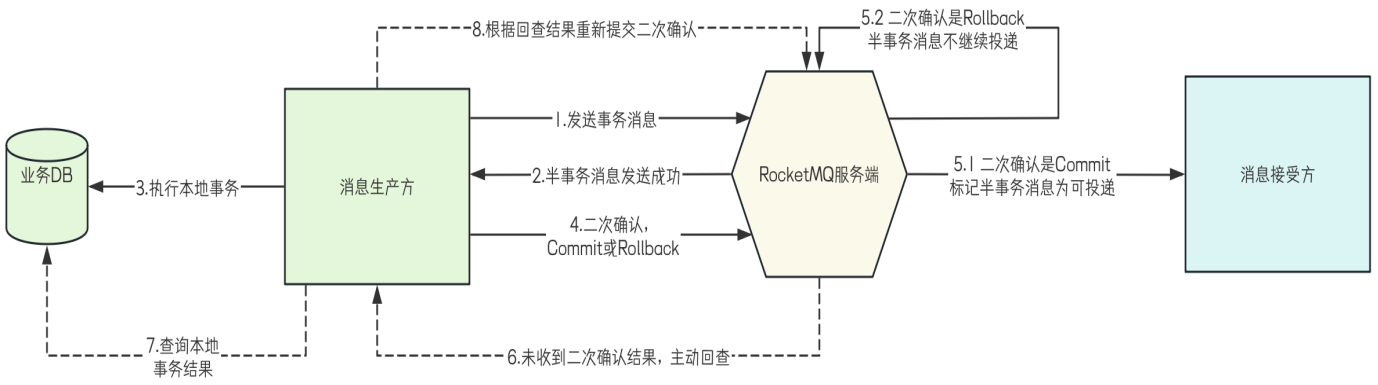
   用户 1 发起转账后，其所在的银行系统 A，向 RocketMQ 服务端对应的业务 topic 发送一条事务消息，内容包含 “用户 1 转出 1000 元至用户 2 账户”。此时，该消息对转入方的系统 B 不可见，避免在转出方本地事务确认前，转入方提前执行入账操作，确保资金流转安全性。
2. 阶段二：执行本地事务（转出账户扣减）

   事务消息发送成功后，系统 A 继续执行本地事务，扣减用户1 的账户余额，若扣减成功，则提交二次确认 commit 到 RocketMQ 服务端，消息被继续投递到下游，反之，提交 rollback，事务结束，双方账户余额保持不变。
3. 阶段三：下游服务消费（转入银行余额增加）

   转入方银行系统 B 预先订阅转账 topic，接收消息后执行用户 2 账户余额增加操作。若消费过程中因网络异常、账户状态问题导致失败，RocketMQ 将自动触发重试机制，若多次重试仍未成功，消息将转入死信队列，后续由人工介入核对，通过补偿流程保障资金最终准确入账。

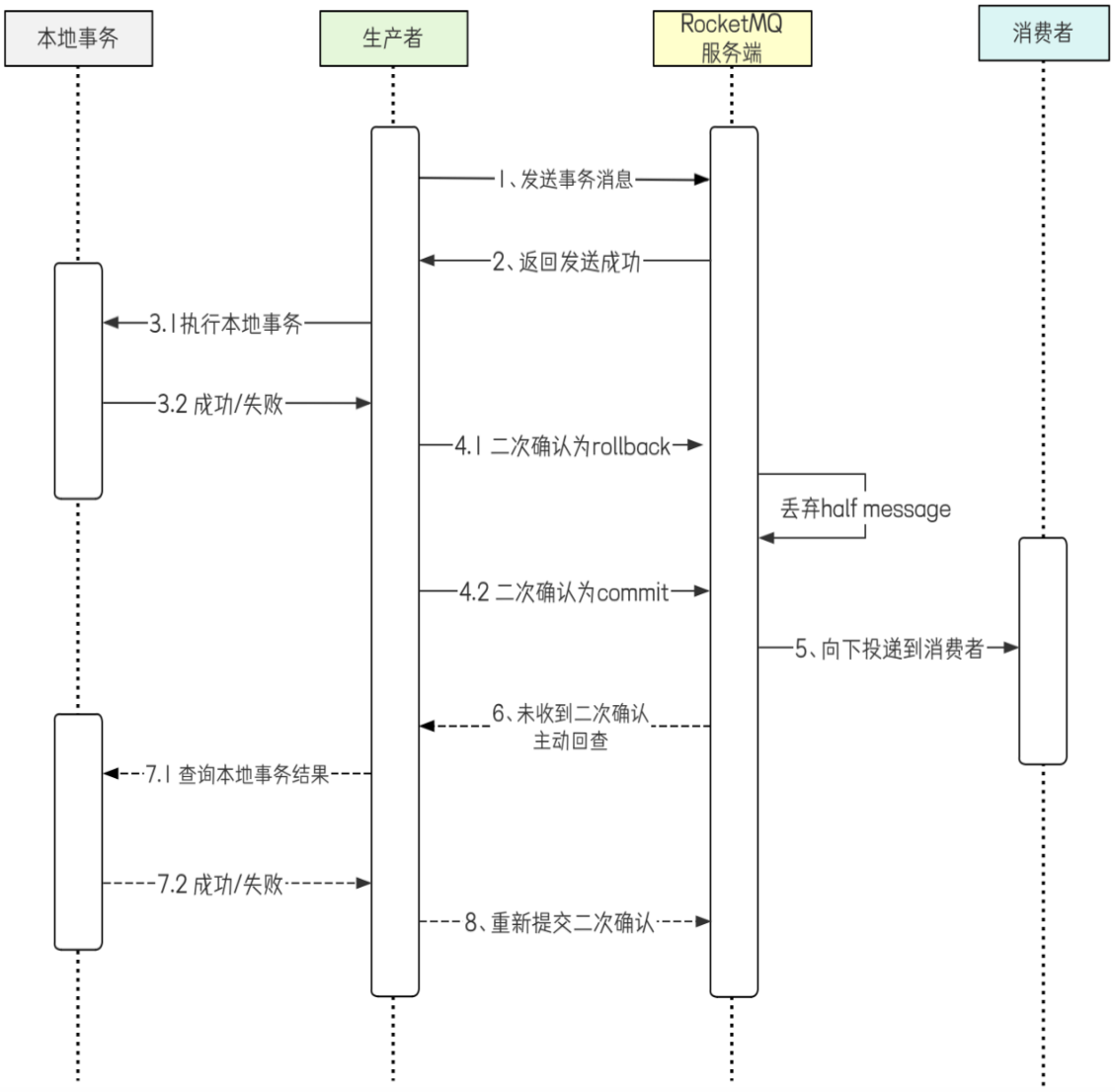
通过以上三个阶段，RocketMQ 事务消息机制在跨行转账场景中，成功实现了分布式事务的最终一致性。类似的，在电商订单支付与库存扣减、金融交易对账、企业多系统数据同步等场景中，RocketMQ 事务消息都能凭借其可靠的机制，保障跨服务操作的最终一致性。

实现流程



1. 生产者发送事务消息到 RocketMQ 服务端。
2. 服务端存储这条消息后返回发送成功的响应，此时消息对下游消费者不可见，处于 half message 状态。
3. 生产者收到半消息成功的响应后，继续往下执行本地事务（如更新业务数据库）。
4. 根据本地事务的执行结果，生产者会向 RocketMQ 服务端提交最终状态，也就是二次确认。
5. 确认结果为 commit 时，服务端会将事务消息继续向下投递给消费者，确认结果为 rollback 时，服务端将会丢弃该消息，不再向下投递。
6. 确认结果是 unknown 或一直没有收到确认结果时，一定时间后，将会触发事务状态主动回查。
7. 当生产者未提交最终状态或者二次确认的结果为 unknown 时，RocketMQ 服务端将会主动发起事务结果查询请求到生产者服务。
8. 生产者收到请求后提交二次确认结果，逻辑再次回到第5步，此时如果生产者服务暂时不可用，则 RocketMQ 服务端会在指定时间间隔后，继续主动发起回查请求，直到超过最大回查次数后，回滚消息。

如此，不管本地事务是否执行成功，都能实现事务状态的最终一致性。以上步骤，可用时序图直观体现为：



使用示例

这里以 TDMQ 版 RocketMQ 5.x 版本集群为例，演示事务消息的使用方式和效果。

1. 首先登录 RocketMQ 控制台，新建一个消息类型为事务消息的 topic。

2. 以 Java 语言为例，引入 5.x 对应版本的依赖。

```
  <dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-client-java</artifactId>
    <version>5.0.6</version>
</dependency>
```

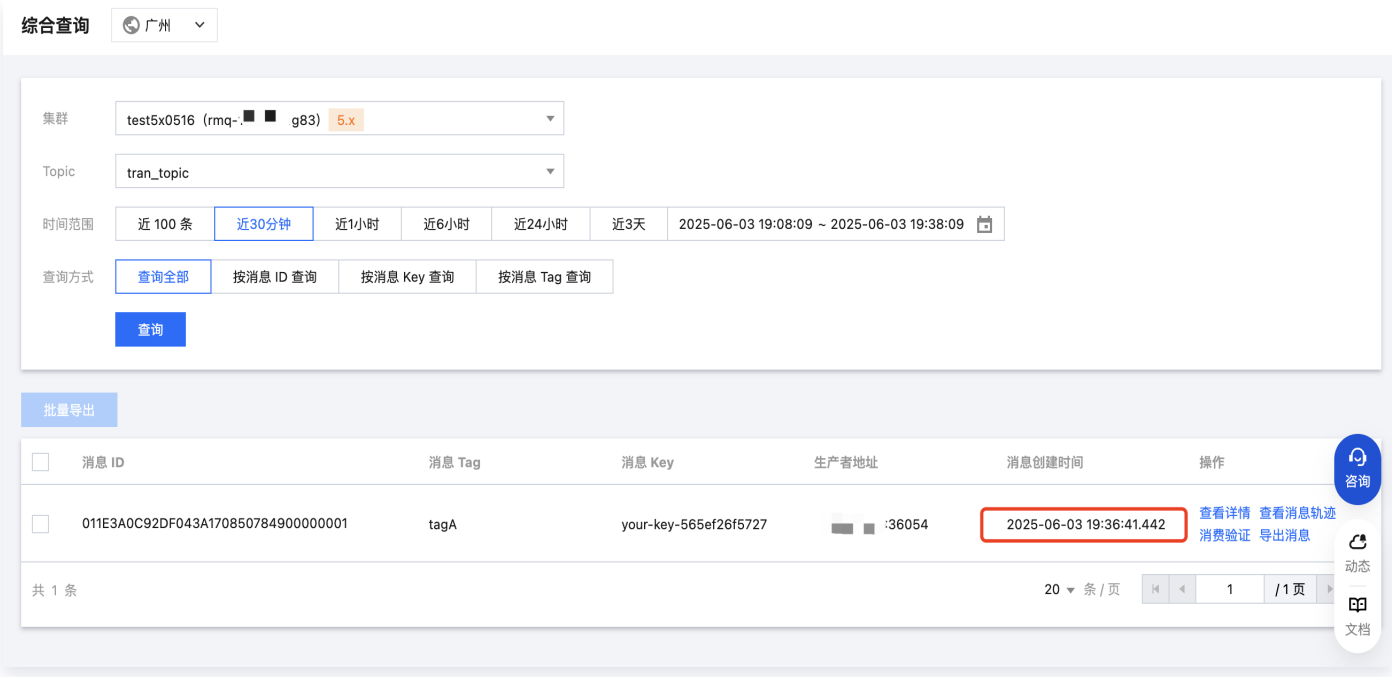
3. 启动生产者。

```
public class ProducerTransactionMessageDemo {

    private static final Logger log =
LoggerFactory.getLogger(ProducerTransactionMessageDemo.class);

    private static boolean executeLocalTransaction() {
        // 模拟本地事务（如数据库插入操作），这里假设执行成功
        return true;
    }

    private static boolean checkTransactionStatus(String orderId) {
        // 模拟查询本地事务执行结果，如查询订单ID是否已入库，查到则return true
        return true;
    }
```

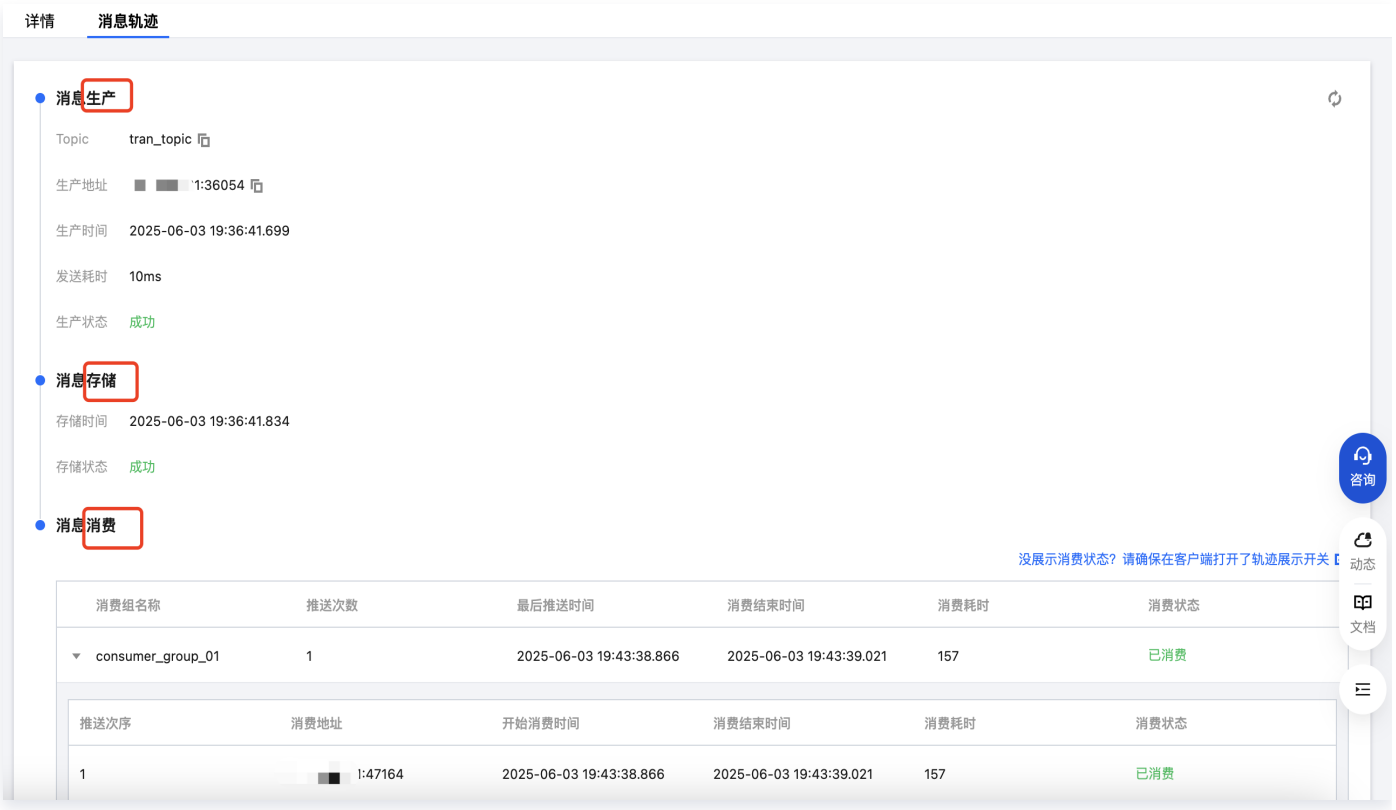
```
    public static void main(String[] args) throws ClientException {

        final ClientServiceProvider provider =
ClientServiceProvider.loadService();
        // 在控制台权限管理页面获取ak和sk
        String accessKey = "your-ak";
        String secretKey = "your-sk";
        SessionCredentialsProvider sessionCredentialsProvider =
            new StaticSessionCredentialsProvider(accessKey,
secretKey);

        // 在控制台获取并填写腾讯云提供的接入地址
        String endpoints = "https://your-endpoints";
        ClientConfiguration clientConfiguration =
ClientConfiguration.newBuilder()
            .setEndpoints(endpoints)
            .enableSsl(false)
            .setCredentialProvider(sessionCredentialsProvider)
            .build();

        String topic = "tran_topic";
        TransactionChecker checker = messageView -> {
            log.info("Receive transactional result check request,
message={}", messageView);
            // 服务端主动回查本地事务状态
            String orderId =
messageView.getProperties().get("orderId");
            boolean isSuccess = checkTransactionStatus(orderId);
            return isSuccess ? TransactionResolution.COMMIT :
TransactionResolution.ROLLBACK;
        };

        // 创建生产者并设置回查的checker对象
        Producer producer = provider.newProducerBuilder()
            .setClientConfiguration(clientConfiguration)
            .setTopics(topic)
            .setTransactionChecker(checker)
            .build();

        // 开启事务
        final Transaction transaction = producer.beginTransaction();
```

```
        byte[] body = "This is a transaction message for Apache
RocketMQ".getBytes(StandardCharsets.UTF_8);
        String tag = "tagA";
        final Message message = provider.newMessageBuilder()
            .setTopic(topic)
            .setTag(tag)
            .setKeys("your-key-565ef26f5727")
            //一般事务消息都会设置一个本地事务关联的唯一ID，用来做本地事务回查的
校验
            .addProperty("orderId", "0001")
            .setBody(body)
            .build();

        // 发送半消息
        try {
            final SendReceipt sendReceipt = producer.send(message,
transaction);
            log.info("Send transaction message successfully,
messageId={}", sendReceipt.getMessageId());
        } catch (Throwable t) {
            log.error("Failed to send message", t);
            return;
        }

        // 执行本地事务
        boolean localTxSuccess = executeLocalTransaction();
        if (localTxSuccess) {
            // 本地事务执行成功，二次确认为Commit
            transaction.commit();
        } else {
            // 本地事务执行失败，二次确认为Rollback
            transaction.rollback();
        }
        // producer.close();
    }
}
```

4. 运行代码后，在控制台的消息查询页面，可以看到已经有一条投递完成等待消费的消息。



5. 启动消费者，订阅这个topic，成功消费消息后，在腾讯云控制台查看消息轨迹：



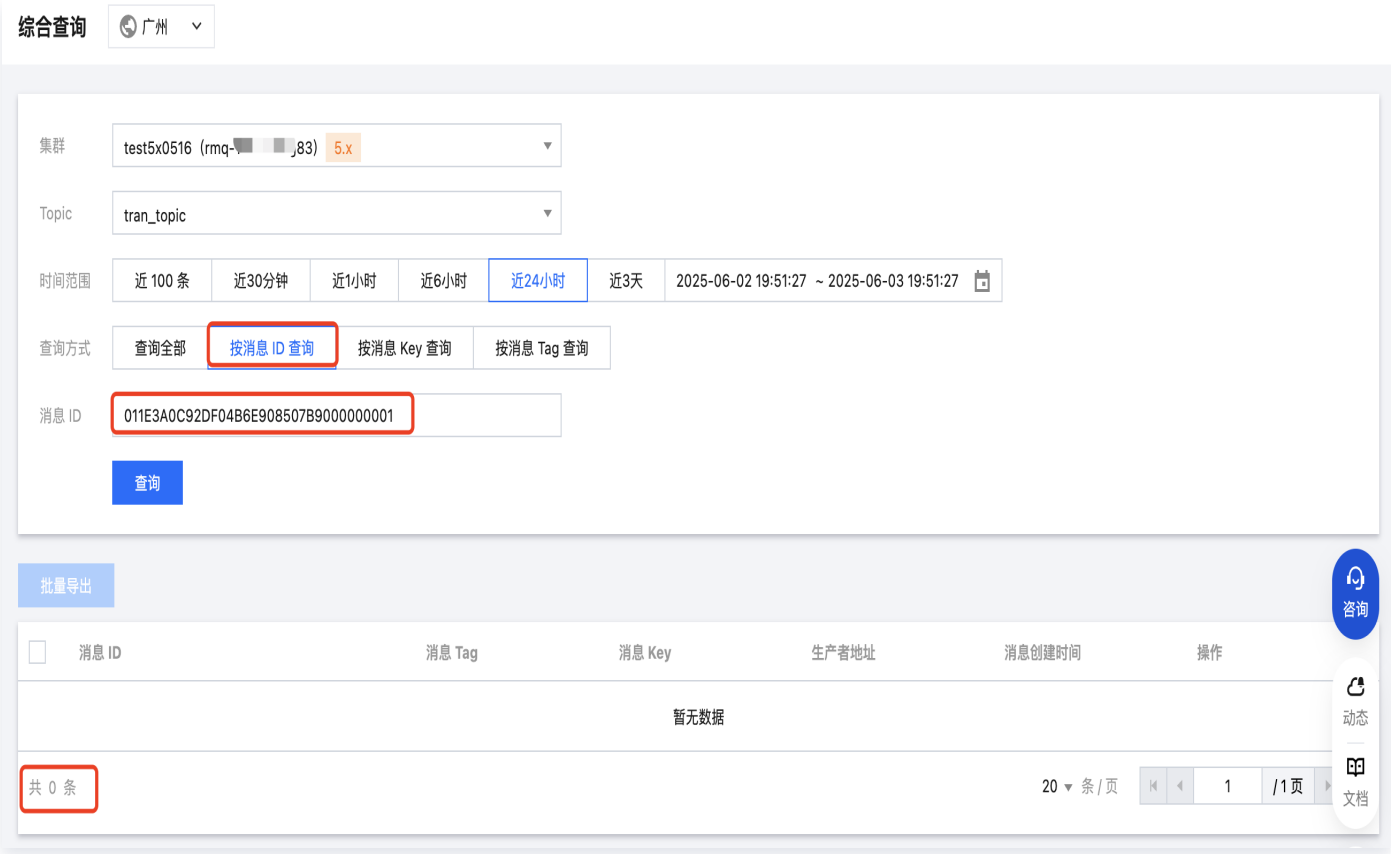
6. 修改代码，假设本地事务执行失败，使处于 half message 状态的事务消息回滚。

```
private static boolean executeLocalTransaction() {
    // 本地事务执行失败
    return false;
}
```

```
private static boolean checkTransactionStatus(String orderId) {
    // 回查结果自然也是rollback，返回false
    return false;
}
```

7. 此时，可以发现消息发送成功了，但在控制台的消息消息查询页面是不可见的，启动消费者也不能消费到这条消息。

```
19:50:40.661 [main] INFO com.llg.ProducerTransactionMessageDemo -- Send transaction message successfully, messageId=011E3A0C92DF04B6E908507B9000000001
```



注意事项

使用事务消息过程中，需注意以下几点：

1. topic 类型必须为事务 TRANSACTION，否则生产消息会报错，关键错误信息：current message type not match with topic accept message types。
2. 事务消息不支持延迟，若设置了延迟属性，在发送消息前会被清除延迟属性。
3. 如果本地事务执行较慢，此时服务端进行事务回查时，应返回 unknown，且如果确认本地事务执行耗时会很长，应修改第一次事务回查的时间，以避免产生大量结果未知的事务。

消息过滤

最近更新时间：2025-06-10 16:17:03

本文主要介绍 TDMQ RocketMQ 版中消息过滤的功能、应用场景和使用方式。

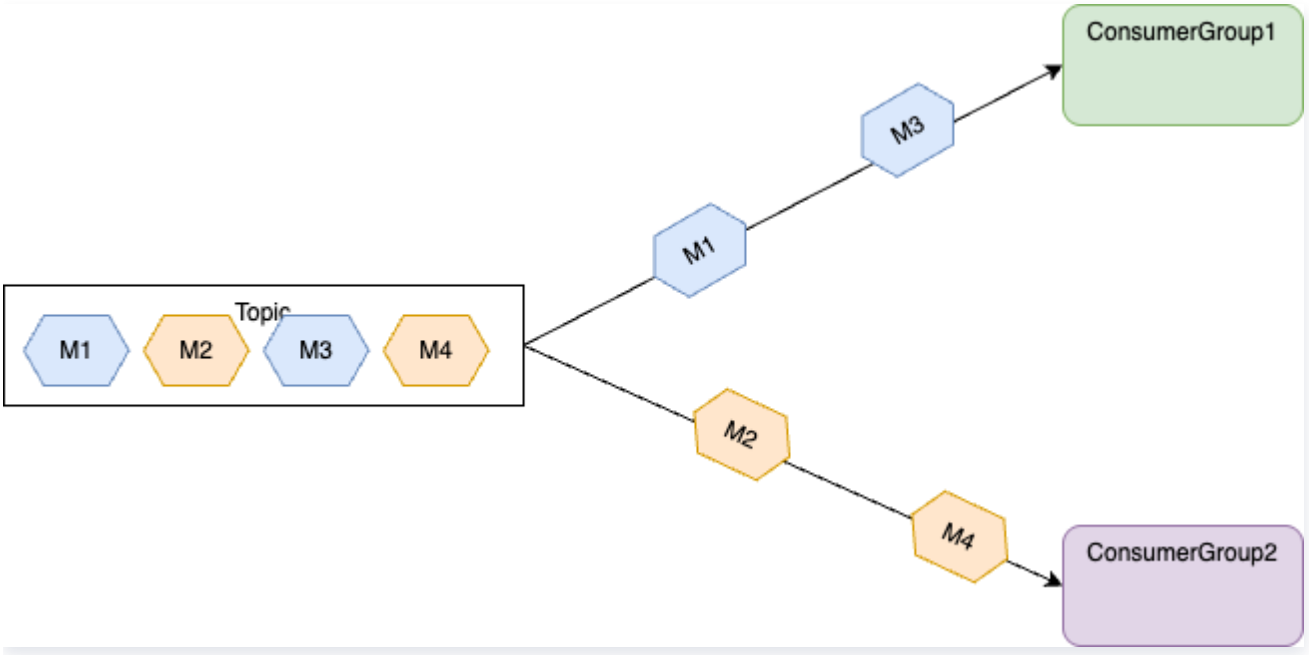
功能介绍

消息过滤功能指消息生产者向 Topic 中发送消息时，设置消息属性对消息进行分类，消费者订阅 Topic 时，根据消息属性设置过滤条件对消息进行过滤，只有符合过滤条件的消息才会被投递到消费端进行消费。
消费者订阅 Topic 时若未设置过滤条件，无论消息发送时是否有设置过滤属性，Topic 中的所有消息都将被投递到消费端进行消费。

应用场景

通常，一个 Topic 中存放的是相同业务属性的消息，例如交易流水 Topic 包含了下单流水、支付流水、发货流水等，这些消息会发送到同一个交易流水 Topic 中，业务若只想消费者其中一种类别的消息，可在客户端进行过滤。

- 账单系统：只需订阅支付消息。
- 物流系统：只需订阅物流消息。
- 库存系统：只需订阅下单消息。



使用方式

目前消息过滤主要支持两种过滤方式，分别是 SQL 过滤和 TAG 过滤。其核心逻辑都是在发送消息的时候，设置一些自定义字段，然后通过消费组订阅的时候指定对应的过滤表达式，消息在服务端进行过滤后，才被消费组消费。

TAG 过滤

发送消息

> **说明：**
> 发送消息时，每条消息必须指明 Tag。

```
 String tag = "yourMessageTagA";
        final Message message = provider.newMessageBuilder()
            // Set topic for the current message.
            .setTopic(topic)
            // Message secondary classifier of message besides
topic.
            .setTag(tag)
            // Key(s) of the message, another way to mark message
besides message id.
            .setKeys("yourMessageKey-1c151062f96e")
            .setBody(body)
            .build();
```

订阅消息

- 订阅所有 Tag：消费者如需订阅某 Topic 下所有类型的消息，Tag 用星号（*）表示。

```
  String consumerGroup = "yourConsumerGroup";
        String topic = "yourTopic";
        String tag = "*";
        FilterExpression filterExpression = new
FilterExpression(tag, FilterExpressionType.TAG);
        // In most case, you don't need to create too many
consumers, singleton pattern is recommended.
        PushConsumer pushConsumer =
provider.newPushConsumerBuilder()
            .setClientConfiguration(clientConfiguration)
            // Set the consumer group name.
            .setConsumerGroup(consumerGroup)
            // Set the subscription for the consumer.

.setSubscriptionExpressions(Collections.singletonMap(topic,
filterExpression))
            .setMessageListener(messageView -> {
```

```
                // Handle the received message and return consume
result.
                log.info("Consume message={}", messageView);
                return ConsumeResult.SUCCESS;
            })
            .build();
```

- **订阅单个 Tag：消费者如需订阅某 Topic 下某一种类型的消息，请明确标明 Tag。**

```
  String consumerGroup = "yourConsumerGroup";
        String topic = "yourTopic";
        String tag = "TAGA";
        FilterExpression filterExpression = new
FilterExpression(tag, FilterExpressionType.TAG);
        // In most case, you don't need to create too many
consumers, singleton pattern is recommended.
        PushConsumer pushConsumer =
provider.newPushConsumerBuilder()
            .setClientConfiguration(clientConfiguration)
            // Set the consumer group name.
            .setConsumerGroup(consumerGroup)
            // Set the subscription for the consumer.

.setSubscriptionExpressions(Collections.singletonMap(topic,
filterExpression))
            .setMessageListener(messageView -> {
                // Handle the received message and return consume
result.
                log.info("Consume message={}", messageView);
                return ConsumeResult.SUCCESS;
            })
            .build();
```

- **订阅多个 Tag：消费者如需订阅某 Topic 下多种类型的消息，请在多个 Tag 之间用两个竖线 `||` 分隔。**

```
 String consumerGroup = "yourConsumerGroup";
        String topic = "yourTopic";
        String tag = "TAGA || TAGB";
        FilterExpression filterExpression = new
FilterExpression(tag, FilterExpressionType.TAG);
        // In most case, you don't need to create too many
consumers, singleton pattern is recommended.
```

```
        PushConsumer pushConsumer =
provider.newPushConsumerBuilder()
            .setClientConfiguration(clientConfiguration)
            // Set the consumer group name.
            .setConsumerGroup(consumerGroup)
            // Set the subscription for the consumer.

.setSubscriptionExpressions(Collections.singletonMap(topic,
filterExpression))
            .setMessageListener(messageView -> {
                // Handle the received message and return consume
result.
                log.info("Consume message={}", messageView);
                return ConsumeResult.SUCCESS;
            })
            .build();
```

使用限制

- 发送消息只能设置一个 TAG。
- 多个 Tag 之间为或的关系，不同 Tag 间使用两个竖线（||）隔开。例如 Tag1||Tag2||Tag3，表示标签为 Tag1 或 Tag2 或 Tag3 的消息都满足匹配条件，都会被发送给消费者进行消费。
- 多个 Tag 的顺序也要保持一致，否则会导致订阅关系不一致，例如 Tag1||Tag2 和 Tag2||Tag1 就是不同的。

SQL 过滤

发送消息

发送代码和简单的消息没有区别。主要是在构造消息体的时候，带上自定义属性，允许多个。

```
final Message message = provider.newMessageBuilder()
            // Set topic for the current message.
            .setTopic(topic)
            // Message secondary classifier of message besides
topic.
            // Key(s) of the message, another way to mark message
besides message id.
            .setKeys("yourMessageKey-1c151062f96e")
            .setBody(body)
```

```
            //一些用于sql过滤的信息
            .addProperty("key1", "value1")
            .build();
```

订阅消息

对于消费消息，订阅时需带上相应的 SQL 表达式，其余与普通的消费消息流程无区别。

```
  String consumerGroup = "yourConsumerGroup";
        String topic = "yourTopic";
        String sql = "key1 IS NOT NULL AND key1='value1'";
        //sql表达式
        FilterExpression filterExpression = new
FilterExpression(sql, FilterExpressionType.SQL92);
        //如果是订阅所有
        //FilterExpression filterExpression =
FilterExpression.SUB_ALL;
        // In most case, you don't need to create too many
consumers, singleton pattern is recommended.
        PushConsumer pushConsumer =
provider.newPushConsumerBuilder()
            .setClientConfiguration(clientConfiguration)
            // Set the consumer group name.
            .setConsumerGroup(consumerGroup)
            // Set the subscription for the consumer.

.setSubscriptionExpressions(Collections.singletonMap(topic,
filterExpression))
            .setMessageListener(messageView -> {
                // Handle the received message and return consume
result.
                log.info("Consume message={}", messageView);
                return ConsumeResult.SUCCESS;
            })
            .build();
```

> **说明**
> 上述是对消息的发布和订阅方式的简单介绍。更多操作可参见 GitHub Demo 或 RocketMQ 官方文档 。

使用限制

由于 SQL 属性过滤是生产者定义消息属性，消费者设置 SQL 过滤条件，计算之后，可能有不同的结果，因此服务端的处理方式如下：
异常情况处理：如果过滤条件的表达式计算抛异常，消息默认被过滤，不会被投递给消费者。例如比较数字和非数字类型的值。

- 空值情况处理：如果过滤条件的表达式计算值为 null 或不是布尔类型（true 和 false），则消息默认被过滤，不会被投递给消费者。例如发送消息时不存在某个属性，在订阅时过滤条件中直接使用该属性，则过滤条件的表达式计算结果为 null。
- 类型不符处理：如果消息自定义属性为浮点型，但过滤条件中使用整数进行判断，则消息默认被过滤，不会被投递给消费者。

虽然这种方式是灵活的，但是在消息头中还是不建议设置太多的值，因为总的消息头部属性有大小限制（32K），内置的已经占用了不少。超长之后，可能导致消息发送或者消费异常。

使用建议

合理划分主题和 Tag 标签。
从消息的过滤机制和主题的原理机制可以看出，业务消息的拆分可以基于主题进行筛选，也可以基于主题内消息的 Tag 标签及属性进行筛选。关于拆分方式的选择，应注意以下问题：

- 消息类型是否一致：不同类型的消息，如顺序消息和普通消息需要使用不同的主题进行拆分，无法通过 Tag 标签进行分类。
- 业务域是否相同：不同业务域和部门的消息应该拆分不同的主题。例如物流消息和支付消息应该使用两个不同的主题；同样是一个主题内的物流消息，普通物流消息和加急物流消息则可以通过不同的 Tag 进行区分。
- 消息量级和重要性是否一致：如果消息的量级规模存在巨大差异，或者说消息的链路重要程度存在差异，则应该使用不同的主题进行隔离拆分。

消息重试

最近更新时间：2025-06-10 16:17:03

本文主要介绍消息队列 TDMQ RocketMQ 版中消息重试与使用方法。

功能介绍

当消息第一次被消费者消费后，没有得到正常的回应，或者用户主动要求服务端重投，TDMQ RocketMQ 版会通过消费重试机制自动重新投递该消息，直到该消息被成功消费，当重试达到一定次数后，消息仍未被成功消费，则会停止重试，将消息投递到死信队列中。
当消息进入到死信队列中，表示 TDMQ RocketMQ 版已经无法自动处理这批消息，一般这时就需要人为介入来处理这批消息。您可以通过编写专门的客户端来订阅死信 Topic，处理这批之前处理失败的消息。

> **说明：**
> 只有当消费模式为集群消费模式时，Broker 才会自动进行重试，广播消费模式下不会进行重试。

出现以下三种情况会按照消费失败处理并会发起重试：

- 消费者返回 ConsumeResult.FAILURE。
- 消费者返回 null。
- 消费者主动/被动抛出异常。

重试次数

当消息需要重试时，TDMQ RocketMQ 中配置了如下的 messageDelayLevel 参数来设置重试次数与时间间隔。

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

由于 SDK 中第一次 delayLevel 为3，所以重试次数与重试时间间隔关系如下：

第几次重试	距离上一次重试的时间间隔	第几次重试	距离上一次重试的时间间隔
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

使用方式

5.x SDK

无需特别处理，5.0的 SDK 遵循上述规则。

4.x SDK

如果用户需要自行调整这个重试次数。可以通过设置 consumer 的参数来决定。

```
pushConsumer.setMaxReconsumeTimes(3);
```

POP 消费模式（5.x）

最近更新时间：2025-06-10 16:17:03

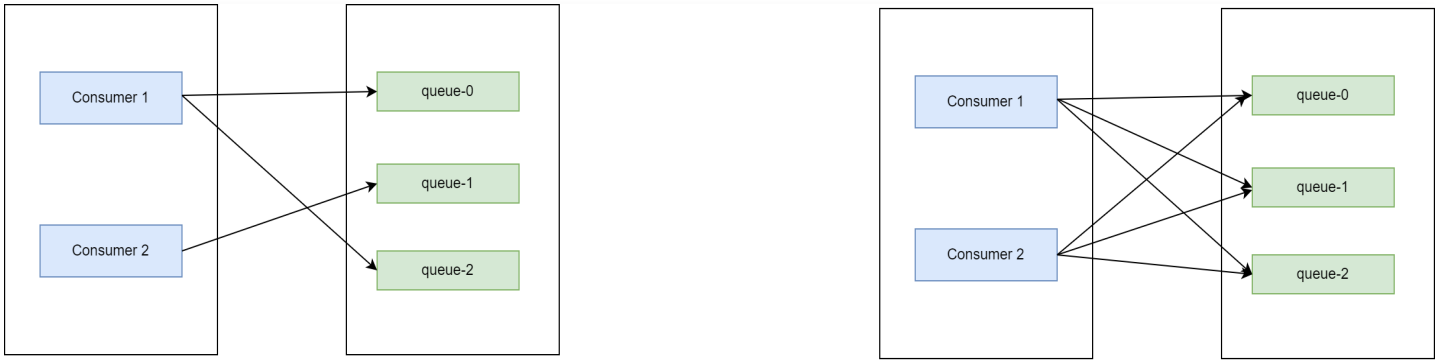
问题背景

RocketMQ 以其高性能，低延迟和抗积压的特性被不少客户和开发者熟知，但是在 RocketMQ 4.x 客户端 SDK 的使用中，不少客户会反馈消费者客户端在实际消费消息过程中，4.x 的客户端（例如常用的 Push Comsumer）在消费的过程中遇到一些问题：

- SDK 承担了太多功能，例如拉消息，负载均衡，消息位点管理和新增客户端时的 Rebalance 等等，这点对多语言开发开发者不友好。
- 队列独占的负载均衡策略容易导致消费瓶颈：Broker 上的每个队列只能分配到相同 Group 的一台消费者客户端上。因此当队列数固定时，单纯增加消费者客户端的数量并不能提升消费性能。假设某个 Topic 共有10个队列，Group 最多有 10 个客户端进行消费（即最多每个客户端消费一个队列）。在业务高峰期，即使客户想增加新的客户端去消费消息，新上线的第11个客户端也无法消费消息。
- 单个客户端异常导致堆积。假设单个客户端因为异常 “hang 机” 时，由于和服务端的心跳没有断开，因此该客户端会被分配到队列进行消费，而此时因为客户端异常实际并不能消费机器，导致异常堆积产生，且因为上一条的原因，单纯加客户端数量并不能解决问题。

解决方案

鉴于以上原因，5.x 推出了 POP 消费模式。
POP 模式下，消费位点由服务端进行管理，因此多个客户端可以消费同一个队列。使用 POP 消费模式的客户端，每个客户端都会从所有的队列去拉消息，因此解决了上述的单个客户端异常和消费瓶颈的问题。
同时，服务端维护消费信息，使得客户端 SDK 更加轻量，方便进行多语言移植。



Before　　After

代码示例

那么我们如何使用 POP 消费模式呢?
需要使用 5.x 的 gRPC SDK，引入相关依赖如下：

```
<dependencies>
        <dependency>
            <groupId>org.apache.rocketmq</groupId>
            <artifactId>rocketmq-client-java</artifactId>
            <version>5.0.6</version>
        </dependency>
</dependencies>
```

同时参考开源社区的 DEMO 如下所示（以 Java 代码为例）：

```
/*
 * 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.
 */

package org.apache.rocketmq.client.java.example;

import java.time.Duration;
import java.util.Collections;
import java.util.List;
import org.apache.rocketmq.client.apis.ClientConfiguration;
import org.apache.rocketmq.client.apis.ClientException;
import org.apache.rocketmq.client.apis.ClientServiceProvider;
import org.apache.rocketmq.client.apis.SessionCredentialsProvider;
import org.apache.rocketmq.client.apis.StaticSessionCredentialsProvider;
import org.apache.rocketmq.client.apis.consumer.FilterExpression;
import org.apache.rocketmq.client.apis.consumer.FilterExpressionType;
import org.apache.rocketmq.client.apis.consumer.SimpleConsumer;
```

```
import org.apache.rocketmq.client.apis.message.MessageId;
import org.apache.rocketmq.client.apis.message.MessageView;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class SimpleConsumerExample {
    private static final Logger log =
LoggerFactory.getLogger(SimpleConsumerExample.class);

    private SimpleConsumerExample() {
    }

    @SuppressWarnings({"resource", "InfiniteLoopStatement"})
    public static void main(String[] args) throws ClientException {
        final ClientServiceProvider provider =
ClientServiceProvider.loadService();

        // Credential provider is optional for client configuration.
        String accessKey = "用户ak";
        String secretKey = "用户sk";
        SessionCredentialsProvider sessionCredentialsProvider =
            new StaticSessionCredentialsProvider(accessKey, secretKey);

        String endpoints = "腾讯云页面接入点";
        ClientConfiguration clientConfiguration =
ClientConfiguration.newBuilder()
            .setEndpoints(endpoints)
            // On some Windows platforms, you may encounter SSL
compatibility issues. Try turning off the SSL option in
            // client configuration to solve the problem please if SSL
is not essential.
            // .enableSsl(false)
            .setCredentialProvider(sessionCredentialsProvider)
            .build();
        String consumerGroup = "消费组";
        // 默认消费时间，30s，也就是说，对于拉到的消息，30s内如果消费没完成，该条消
息会被别的客户端重新拉到，
        // 需要用户根据自己的场景配置
        Duration awaitDuration = Duration.ofSeconds(30);
        String tag = "*";
        String topic = "topic名字";
        FilterExpression filterExpression = new FilterExpression(tag,
FilterExpressionType.TAG);
```

```
        // In most case, you don't need to create too many consumers,
singleton pattern is recommended.
        SimpleConsumer consumer = provider.newSimpleConsumerBuilder()
            .setClientConfiguration(clientConfiguration)
            // Set the consumer group name.
            .setConsumerGroup(consumerGroup)
            // set await duration for long-polling.
            .setAwaitDuration(awaitDuration)
            // Set the subscription for the consumer.
            .setSubscriptionExpressions(Collections.singletonMap(topic,
filterExpression))
            .build();
        // Max message num for each long polling.
        int maxMessageNum = 16;
        // Set message invisible duration after it is received.
        Duration invisibleDuration = Duration.ofSeconds(15);
        // Receive message, multi-threading is more recommended.
        do {
            final List<MessageView> messages =
consumer.receive(maxMessageNum, invisibleDuration);
            log.info("Received {} message(s)", messages.size());
            for (MessageView message : messages) {
                final MessageId messageId = message.getMessageId();
                try {
                    consumer.ack(message);
                    log.info("Message is acknowledged successfully,
messageId={}", messageId);
                } catch (Throwable t) {
                    log.error("Message is failed to be acknowledged,
messageId={}", messageId, t);
                }
            }
        } while (true);
        // Close the simple consumer when you don't need it anymore.
        // You could close it manually or add this into the JVM shutdown
hook.
        // consumer.close();
    }
}
```

在这种情况下，同一个消费组内，一个消费者将不会再和队列一一绑定，也能最大程度避免之前4.x中单个消费者阻塞导致队列堆积的问题。

集群消费与广播消费

最近更新时间：2025-06-10 16:17:03

本文主要介绍消息队列 TDMQ RocketMQ 版中集群消费和广播消费的相关功能和应用场景。

功能介绍

- 集群消费：当使用集群消费模式时，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，每条消息会被推送给集群内所有注册过的消费者，保证消息至少被每个消费者消费一次。

应用场景

- 集群消费：适用于每条消息只需要被处理一次的场景。
- 广播消费：适用于每条消息需要被集群下每一个消费者处理的场景。

代码示例

- **集群订阅**

5.x SDK

5.0 SDK 默认为集群消费模式，无需特殊设置。

> **说明：**
> 请确保同一个 Group ID 下所有 Consumer 实例的订阅关系保持一致。

4.x SDK

- 同一个 Group ID 所标识的所有 Consumer 平均分摊消费消息。例如某个 Topic 有9条消息，一个 Group ID 有3个 Consumer 实例，那么在集群消费模式下每个实例平均分摊，只消费其中的3条消息。

```
// 集群订阅方式设置（不设置的情况下，默认为集群订阅方式）。
properties.put(PropertyKeyConst.MessageModel,
PropertyValueConst.CLUSTERING);
```

- **广播订阅**

5.x SDK

5.0 SDK 已经不支持广播消费，如果需要使用，可以给每个消费者创建一个单独的订阅组实现类似的功能。

> **说明：**
> 请确保同一个 Group ID 下所有 Consumer 实例的订阅关系保持一致。

4.x SDK

- 同一个 Group ID 所标识的所有 Consumer 都会各自消费某条消息一次。例如某个 Topic 有9条消息，一个 Group ID 有3个 Consumer 实例，那么在广播消费模式下每个实例都会各自消费9条消息。

```
// 广播订阅方式设置。
properties.put(PropertyKeyConst.MessageModel,
PropertyValueConst.BROADCASTING);
```

> **说明**
> 请确保同一个 Group ID 下所有 Consumer 实例的订阅关系保持一致。

订阅关系一致性

最近更新时间：2025-06-10 16:17:03

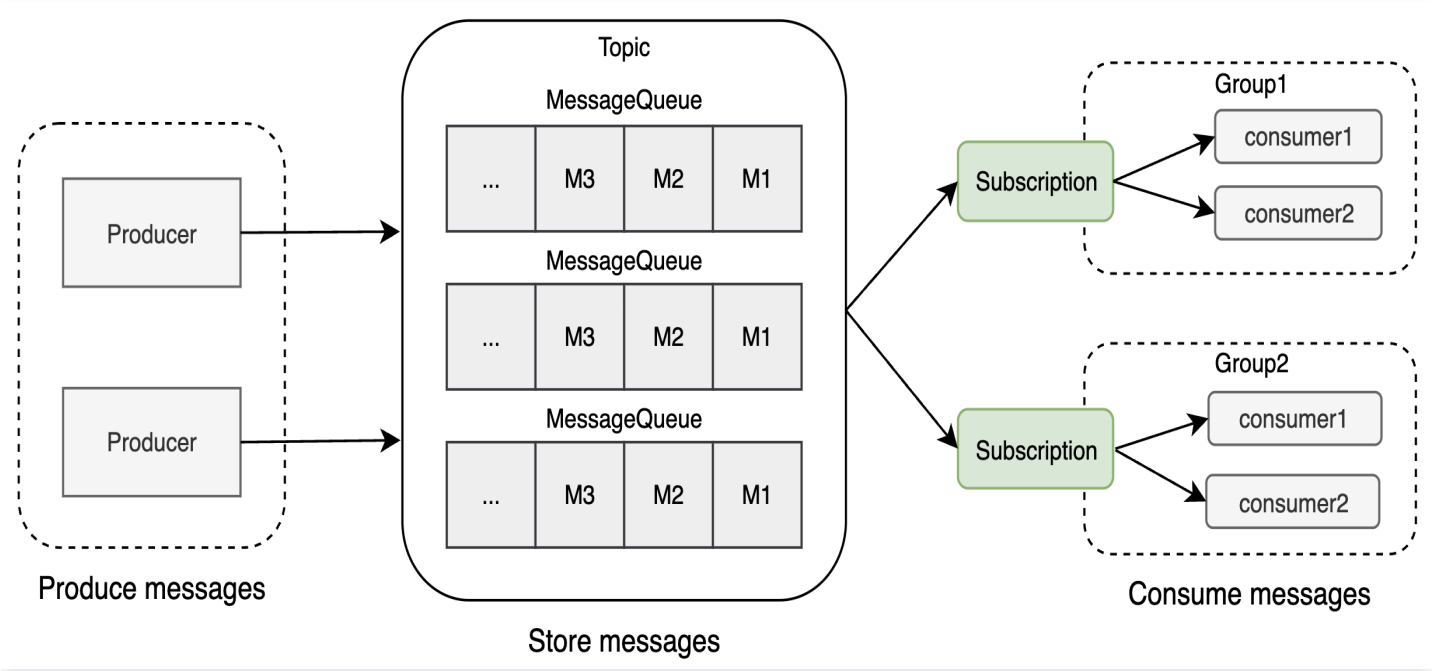
本文主要介绍订阅关系一致性的核心要点，从定义与约束机制，到底层实现原理与优化实践，再结合真实案例分享 TDMQ RocketMQ 版针对订阅关系不一致问题的解决方案，帮助开发者快速定位问题根源，构建稳定可靠的消息系统。

订阅关系定义

订阅关系是 RocketMQ 系统中消费者获取消息、处理消息的规则和状态配置，订阅关系由消费者组动态注册到服务端，并在后续的消息传输中按照订阅关系定义的过滤规则进行消息匹配和消费进度维护。
通过配置订阅关系，可控制如下消费行为：

- 消息过滤规则：用于控制消费者在消费消息时，选择主题内的哪些消息进行消费，设置消费过滤规则可以高效地过滤消费者需要的消息集合，灵活根据不同的业务场景设置不同的消息接收范围。
- 消费状态：RocketMQ 服务端默认提供订阅关系持久化的能力，即消费者分组在服务端注册订阅关系后，当消费者离线并再次上线后，可以获取离线前的消费进度并继续消费。

在 RocketMQ 的领域模型中，订阅关系的位置和流程如下：



1. 消息由生产者初始化并发送到 RocketMQ 服务端。
2. 消息按照到达 RocketMQ 服务端的顺序存储到主题的指定队列中。
3. 消费者按照指定的订阅关系从 RocketMQ 服务端中获取消息并消费。

订阅关系一致性约束

订阅关系一致性要求同一消费者组内的所有消费者实例所订阅的主题必须和过滤规则完全一致。这里涉及三个约束条件，具体来看：

- **消费组必须一致**

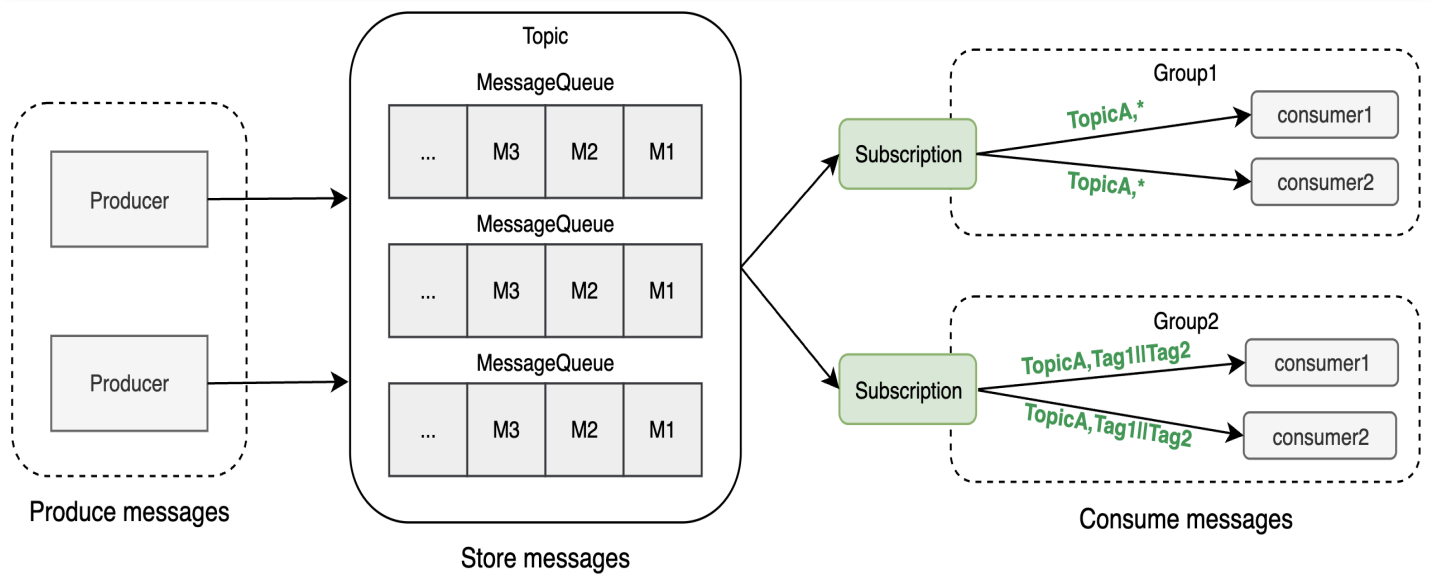
  对于大多数分布式应用来说，一个消费组下通常会挂载多个 Consumer 实例，订阅关系一致性的约束范围就是同一个消费组下的所有消费者。
- **订阅的主题必须一致**

  同一个消费组下的所有消费者订阅的主题必须一致，例如：consumer1 订阅 TopicA 和 TopicB，consumer2 也必须订阅 TopicA 和 TopicB，不能只订阅 TopicA、只订阅 TopicB 或订阅 TopicA 和 TopicC。
- **过滤规则必须一致**

  同一个消费组下的所有消费者过滤规则必须一致，包括 Tag 的数量和 Tag 的顺序，例如：consumer1 订阅 TopicB 且 Tag 为 Tag1||Tag2，consumer2 订阅 TopicB 的 Tag 也必须是 Tag1||Tag2，不能只订阅 Tag1、只订阅 Tag2 或者订阅 Tag2||Tag1。

订阅关系一致的示例

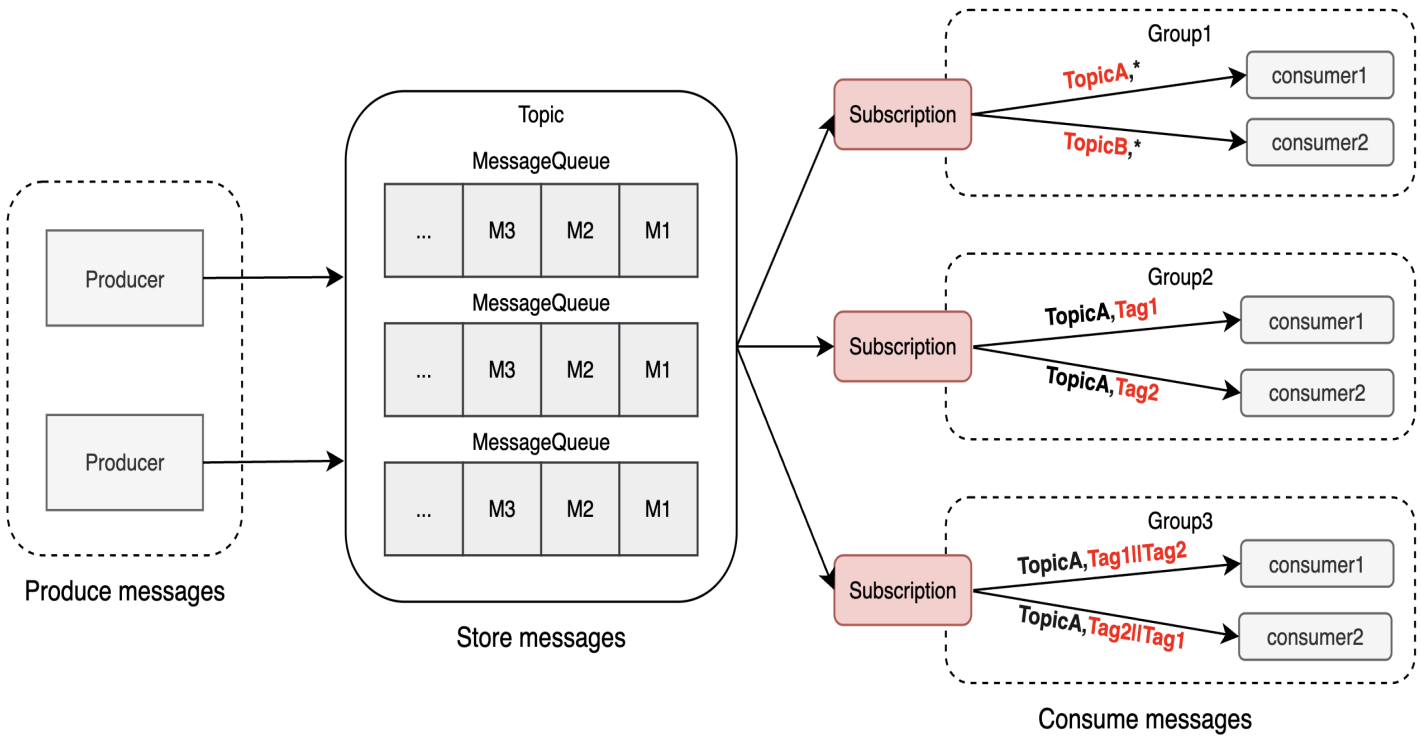
下图展示了常见的两种正确的订阅关系，分别对应两种情况：



正确示例一：单 Topic 单 Tag 订阅
如图中 Group 1 的 consumer1 和 consumer2 都订阅 TopicA 中所有消息。
正确示例二：单 Topic 多 Tag 订阅
如图中 Group 2 的 consumer1 和 consumer2 都订阅 TopicA 中 Tag 为 Tag1 或 Tag2 的消息，且顺序都是 Tag1||Tag2。

订阅关系不一致的示例

下图展示了三种典型错误的订阅关系，分别对应三种情况：



错误示例一：订阅 Topic 不同

如图中 Group 1 的 consumer1 和 consumer2 分别订阅了不同的 Topic。

错误示例二：订阅 Topic 相同但 Tag 不同

如图中 Group 2 的 consumer1 订阅 TopicA 的 Tag1，consumer2 订阅 TopicA 的 Tag2。

错误示例三：订阅 Topic 和 Tag 都相同但 Tag 顺序不同

如图中 Group 3 的 consumer1 订阅 TopicA 的 Tag1||Tag2，consumer2 订阅 TopicA 的 Tag2||Tag1，这里虽然订阅 Tag 都相同但顺序不同，也不符合订阅一致性约束。

订阅关系不一致的影响

如果订阅关系不一致，可能导致消息消费逻辑混乱，消息被重复消费或遗漏。

如下示例，这里我们启动两个 consumer，它们都属于消费组 Group1，都订阅了主题 TopicA，但是 consumer1 订阅的是 Tag1 的消息，consumer2 订阅的是 Tag2 的消息。

```
String topic = "TopicA";
String consumerGroup = "Group1";
FilterExpression filterExpressionTag1 = new FilterExpression("Tag1",
FilterExpressionType.TAG);
PushConsumer consumer1 = provider.newPushConsumerBuilder()
    .setConsumerGroup(consumerGroup)
    .setSubscriptionExpressions(Collections.singletonMap(topic,
filterExpressionTag1))
    .build();
FilterExpression filterExpressionTag2 = new FilterExpression("Tag2",
FilterExpressionType.TAG);
```

```
PushConsumer consumer2 = provider.newPushConsumerBuilder()
    .setConsumerGroup(consumerGroup)
    .setSubscriptionExpressions(Collections.singletonMap(topic,
filterExpressionTag2))
    .build();
```

这种情况下两个客户端分别会有什么表现呢?

- consumer1 消费者无法消费 Tag 值为 Tag1 的消息，因为 consumer1 消费者在拉取消息时，服务端该消费组的订阅信息中 Tag 值为 Tag2，经过服务端过滤后，consumer1 消费者拉取到的消息的 Tag 值都是 Tag2 , 但消费者在收到消息后也会进行过滤，这部分消息也都被过滤掉了。
- consumer2 消费者只能消费部分 Tag 值为 Tag2 的消息，因为只有部分队列分配给了consumer2。

但是在服务端同一个消费组内的各个消费者客户端的订阅信息会相互被覆盖，所以这种消费状态非常混乱，上面示例中 consumer1 和 consumer2 的消费情况可能也会发生切换。

腾讯云优化实践

订阅关系不一致会直接导致消息消费异常，需快速定位并修复。TDMQ RocketMQ 版控制台提供可视化检测能力，无需人工逐个排查日志或配置，通过控制台 3 步即可完成问题的发现、定位、修复，降低运维复杂度。

1. **一键检测**

   自动对比消费组内所有客户端的订阅配置，高亮显示出不一致的订阅关系。

2. **精准定位**

   点击不一致详情，直接关联到具体客户端实例，可快速判断出非预期的订阅关系所在的实例。

3. **闭环验证**

修订后实时同步订阅关系一致性状态，确保消费组订阅关系符合预期。

常见问题

哪些典型场景会出现订阅关系不一致?

- 环境未完全隔离，非生产环境和生产环境使用了相同的 Group 订阅不同的 Topic。
- 业务代码修改了订阅关系，在灰度和版本发布过程中，新旧版本消费者共存。

使用建议

订阅关系一致性是 RocketMQ 消息系统消费行为正确的核心保障：

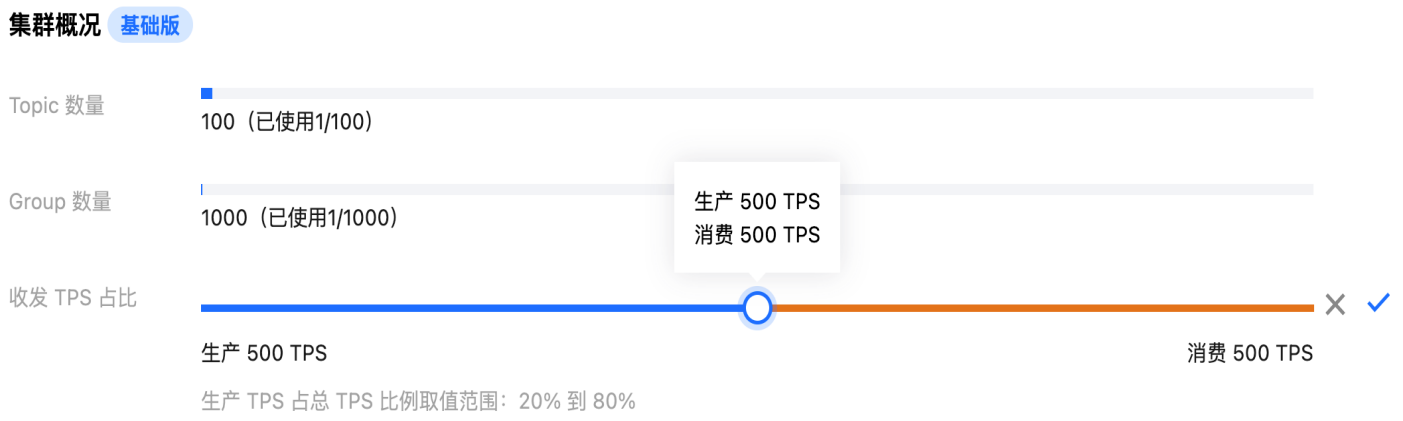
1. 核心约束：同一消费组内的所有消费需严格遵循主题一致、过滤规则一致（含 Tag 顺序）的原则，任一环节的差异均可能导致漏消费消息。
2. 腾讯云能力：依托控制台的一键检测、精准定位、闭环验证功能，开发者可快速识别异常实例并修复，将传统人工排查耗时从小时级缩短至分钟级。
3. 最佳实践：建议通过消费组隔离、动态配置强校验、多 Topic 场景分层治理等策略，从源头规避订阅关系不一致风险。

限流

最近更新时间：2025-02-20 16:38:42

概述

腾讯云消息队列 RocketMQ 版为各类大规模、低延时、高可用性要求的在线业务提供消息服务，客户端通过 SDK 与 RocketMQ 集群建立长连接并进行消息收发，同时消耗集群机器节点的计算、存储、网络带宽等资源。为了提供高质量的消息服务，我们需要控制集群在高并发、大流量情况下的负载水位，以保障系统的稳定性与可靠性。因此，服务端会**根据集群规格限制客户端每秒能够发送和消费的最大折算消息条数**（Transaction Per Second/TPS），具体计算规则可参见 计费概述 - 计算规格 。为了兼具隔离性和灵活性，发送消息与消费消息的 TPS 配额不共享，同时支持自定义配额比例（默认配额比例为1：1 ）。



限流行为说明

腾讯云消息队列 RocketMQ 版采用基于快速失败 (Fail-Fast) 限流策略，即**当客户端请求速率达到上限后，服务端会立即响应错误**。通常在线业务都对响应时间敏感，快速失败可以让客户端感知限流事件并及时介入处理，避免业务消息端到端耗时恶化。

以1000TPS 规格的基础集群为例，假设配额收发 TPS 比例为1:1，相关的触发限流行为说明如下：

说明	发送消息限流	消费消息限流
触发限流情景	所有连接该集群的发送客户端每秒最多可发送折算消息的总和为 500 条，发送速率达到限制后，超限的发送请求会失败。	所有连接该集群的消费客户端每秒最多可消费折算消息的总和为 500 条，消费速率达到限制后，消息的消费延迟会增加。
触发限流时 SDK 日志关键词	Rate of message sending reaches limit, please take a control or upgrade the resource specification。	Rate of message receiving reaches limit, please take a control or upgrade the resource specification。

触发限流时 SDK 重试机制	不同协议的 SDK 处理有差异： • 5.x SDK 会根据指数退避策略进行重试发送，最大重试次数可在初始化 Producer 时自定义，默认值为 2 次；达到最大重试次数仍未成功的发送请求会抛出异常。 • 4.x SDK 直接抛出异常，不会进行重试。	SDK 拉消息线程会自动退避重试。

客户端实践教程

规划集群

腾讯云消息队列 RocketMQ 版集群限流的目的是保障服务稳定可靠，防止在集群高负载时出现服务响应时间变长、请求成功率下降等问题，从而避免业务受损。因此，在您接入腾讯云消息队列 RocketMQ 版时，合理规划集群非常重要，建议您：

- 依据当前规模和未来趋势预测来充分评估业务 TPS， 如果业务流量具有波动特性，应以峰值 TPS 为准。此外，评估时建议您预留一部分 TPS 配额（例如 30%）来应对可能出现的突发流量。
- 对稳定性要求较高的业务，建议您使用多套 RocketMQ 集群加强隔离性。例如，将核心链路（如交易系统）与非核心链路（如日志系统）隔离，以及生产环境与开发测试环境进行隔离等。

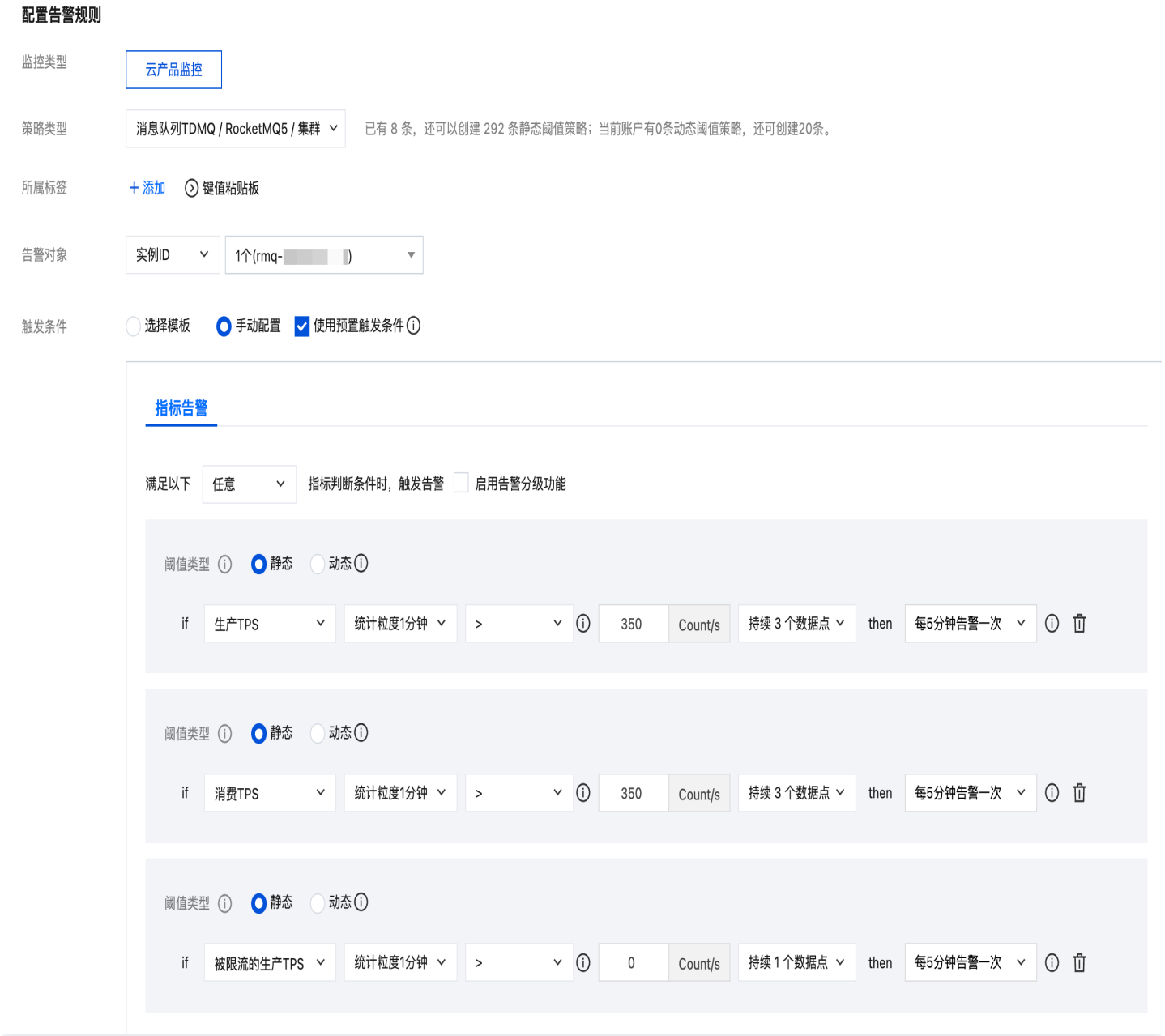
监控负载

您可以利用腾讯云消息队列 RocketMQ 版控制台的监控告警能力实现对集群负载的实时观测，提前发现 TPS 水位风险并及时操作升配，保证资源充足，避免触发限流。具体操作可参见 监控告警，告警策略建议如下：

- 发送和消费 TPS 水位超过容量的 70% 时触发告警，提醒进行升配评估。
- 出现发送限流时触发告警，警告业务发送消息可能失败风险。

示例

以 1000TPS 规格的基础集群为例，TPS 告警策略如下：



TPS 告警配置

代码异常处理

业务代码通过 RocketMQ SDK 发送消息时，需要捕获包括限流错误在内的异常，并保存必要的上下文信息，以便人工介入恢复业务。不同协议的 SDK 重试机制有差异，相关处理示例代码如下：

- 4.x SDK 不会对限流错误进行自动重试，因此业务代码需要捕获异常并进行处理，示例代码如下：

```
import java.io.UnsupportedEncodingException;
import java.nio.charset.StandardCharsets;
import org.apache.rocketmq.acl.common.AclClientRPCHook;
import org.apache.rocketmq.acl.common.SessionCredentials;
import org.apache.rocketmq.client.exception.MQBrokerException;
import org.apache.rocketmq.client.exception.MQClientException;
```

```
import org.apache.rocketmq.client.producer.DefaultMQProducer;
import org.apache.rocketmq.client.producer.SendResult;
import org.apache.rocketmq.common.message.Message;
import org.apache.rocketmq.common.message.MessageClientIDSetter;
import org.apache.rocketmq.logging.InternalLogger;
import org.apache.rocketmq.logging.InternalLoggerFactory;

public class ProducerExample {
    private static final InternalLogger log =
InternalLoggerFactory.getLogger(ProducerExample.class);

    public static void main(
        String[] args) throws MQClientException, InterruptedException,
UnsupportedEncodingException {

        String nameserver = "Your_Nameserver";
        String accessKey = "Your_Access_Key";
        String secretKey = "Your_Secret_Key";
        String topicName = "Your_Topic_Name";
        String producerGroupName = "Your_Producer_Group_Name";

        // 实例化消息生产者 Producer
        DefaultMQProducer producer = new DefaultMQProducer(
            producerGroupName, // 生产者组名字
            new AclClientRPCHook(new SessionCredentials(accessKey,
secretKey)) // ACL权限, accessKey 和 secretKey 可以在控制台集群权限页面获取
        );

        // 设置 Nameserver 地址, 可以在控制台集群基本信息页面获取
        producer.setNamesrvAddr(nameserver);

        // 启动 Producer 实例
        producer.start();

        // 创建消息实例, 设置 topic 和消息内容
        Message message = new Message(topicName,
"Your_Biz_Body".getBytes(StandardCharsets.UTF_8));

        // 最大尝试发送次数, 请根据业务情况设置
        final int maxAttempts = 3;
        // 重试间隔时间, 请根据业务情况设置
        final int retryIntervalMillis = 200;
```

```
        // 发送消息
        int attempt = 0;
        do {
            try {
                SendResult sendResult = producer.send(message);
                log.info("Send message successfully, {}", sendResult);
                break;
            } catch (Throwable t) {
                attempt++;
                if (attempt >= maxAttempts) {
                    // 达到最大次数
                    log.warn("Failed to send message finally, run out
of attempt times, attempt={}, maxAttempts={}, msgId={}",
                        attempt, maxAttempts,
MessageClientIDSetter.getUniqID(message), t);
                    // 记录发送失败的消息（或记录到其他业务系统，比如数据库等）
                    log.warn(message.toString());
                    break;
                }
                int waitMillis;
                if (t instanceof MQBrokerException &&
((MQBrokerException) t).getResponseCode() == 215 /* FLOW_CONTROL */) {
                    // 限流异常，采用退避重试
                    waitMillis = (int) Math.pow(2, attempt - 1) *
retryIntervalMillis; // 重试间隔: 200ms, 400ms, ......
                } else {
                    // 其他异常
                    waitMillis = retryIntervalMillis;
                }
                log.warn("Failed to send message, will retry after
{}ms, attempt={}, maxAttempts={}, msgId={}",
                    waitMillis, attempt, maxAttempts,
MessageClientIDSetter.getUniqID(message), t);
                try {
                    Thread.sleep(waitMillis);
                } catch (InterruptedException ignore) {
                }
            }
        }
        while (true);

        producer.shutdown();
    }
```

```
}
```

- 5.x SDK 会对发送异常进行自动重试，业务代码可以自定义最大重试次数，示例代码如下：

```
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import org.apache.rocketmq.client.apis.ClientConfiguration;
import org.apache.rocketmq.client.apis.ClientException;
import org.apache.rocketmq.client.apis.ClientServiceProvider;
import org.apache.rocketmq.client.apis.SessionCredentialsProvider;
import
org.apache.rocketmq.client.apis.StaticSessionCredentialsProvider;
import org.apache.rocketmq.client.apis.message.Message;
import org.apache.rocketmq.client.apis.producer.Producer;
import org.apache.rocketmq.client.apis.producer.SendReceipt;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class ProducerExample {
    private static final Logger log =
LoggerFactory.getLogger(ProducerExample.class);

    public static void main(String[] args) throws ClientException,
IOException {

        String nameserver = "Your_Nameserver";
        String accessKey = "Your_Access_Key";
        String secretKey = "Your_Secret_Key";
        String topicName = "Your_Topic_Name";

        // ACL权限, accessKey 和 secretKey 可以在控制台集群权限页面获取
        SessionCredentialsProvider sessionCredentialsProvider = new
StaticSessionCredentialsProvider(accessKey, secretKey);

        ClientConfiguration clientConfiguration =
ClientConfiguration.newBuilder()
            .setEndpoints(nameserver) // 设置 NameServer 地址, 可以在控制
台集群基本信息页面获取
            .setCredentialProvider(sessionCredentialsProvider)
            .build();

        // 启动 Producer 实例
```

```
        ClientServiceProvider provider =
ClientServiceProvider.loadService();
        Producer producer = provider.newProducerBuilder()
            .setClientConfiguration(clientConfiguration)
            .setTopics(topicName) // 预声明消息发送的 topic，建议设置
            .setMaxAttempts(3) // 最大尝试发送次数，请根据业务情况设置
            .build();

        // 创建消息实例，设置 topic 和消息内容
        byte[] body =
"Your_Biz_Body".getBytes(StandardCharsets.UTF_8);
        final Message message = provider.newMessageBuilder()
            .setTopic(topicName)
            .setBody(body)
            .build();

        try {
            final SendReceipt sendReceipt = producer.send(message);
            log.info("Send message successfully, messageId={}",
sendReceipt.getMessageId());
        } catch (Throwable t) {
            log.warn("Failed to send message", t);
            // 记录发送失败的消息（或记录到其他业务系统，比如数据库等）
            log.warn(message.toString());
        }

        producer.close();
    }
}
```

常见问题

触发限流后会不会丢消息?

发送消息触发限流后服务端不会存储该条消息，客户端需要捕获异常并做降级处理；消费触发限流后会出现消费延迟，但已经发送成功的消息不会丢。

为什么监控页面的 TPS 比消息条数大?

TPS 是折算消息数量，如果业务使用了高级消息（顺序、延迟、事务等）或消息体比较大，那么一条业务消息会被统计为多条折算消息，具体的折算逻辑可参见 计算规格 。此外，消息条数指标统计的是一分钟内的秒级平均值，而 TPS 指标统计的是一分钟内的秒级峰值。

集群偶尔出现短暂的消费被限流，是否有影响?

一般没有影响。在客户端重启、服务端重启、控制台扩容主题队列等操作期间，都有可能因为消费组瞬间堆积而触发短暂的消费限流，通常稳定后很快会恢复。

如何判断集群是否出现了限流?

除了通过识别 SDK 发送接口抛出的异常或 SDK 日志记录的信息外，您还可以查看 腾讯云 RocketMQ 控制台 > 监控大盘 的**被限流的生产 TPS(Count/s)** 和**被限流的消费 TPS(Count/s)**。