TDMQ for RocketMQ

Development Guide

Product Documentation

Contents

- Development Guide
  - Message Type
    - General Message
    - Scheduled Message and Delayed Message
    - Sequential Message
    - Transactional Message
  - Message Filtering
  - Message Retry
  - Dead Letter Message
  - Consumption Mode
    - Cluster Consumption
    - Broadcast Consumption
  - POP Consumption Mode (5.X)
  - Cluster Consumption and Broadcast Consumption
  - Subscription Relationship Consistency
  - Throttling

Development Guide
Message Type
General Message

Last updated: 2025-07-15 17:46:16

A general message is a basic type of message that is delivered to a specified topic by production and consumed by consumers who subscribe to the topic. There is no order in the topic of general messages, and multiple partitions can be used to improve the efficiency of message production and consumption. The performance of general messages is optimal when the throughput is huge.
General messages are different from scheduled and delayed messages, sequential messages, and transactional messages. The topics corresponding to these four message types cannot be mixed and can only be used for sending and receiving messages of the same type. For instance, the topic of a general message can only be used to send and receive general messages and cannot be used to send and receive delayed, sequential, and transactional messages.

Scheduled Message and Delayed Message

Last updated：2025-07-15 17:46:16

This document primarily describes the concepts and usage of scheduled messages and delayed messages in TDMQ for RocketMQ.

Concepts

- **Scheduled message**: In actual business, after a message is sent to the server, the consumer is expected to receive it at a later time point rather than immediately. This type of message is called scheduled message.
- **Delayed message**: In actual business, after a message is sent to the server, the consumer is expected to receive it after a period of time rather than immediately. This type of message is called delayed message.

  Actually, the delayed message can be regarded as a special type of scheduled message, which is essentially the same thing.

Instructions

Apache RocketMQ does not provide an API for you to freely set the delay time. In order to ensure compatibility with the open-source RocketMQ client, TDMQ for RocketMQ has designed a method to specify the message sending time by adding the property key-value pair to the message. You only need to add the `__STARTDELIVERTIME` property value to the `property` of the message that needs to be sent at a scheduled time (within 40 days). For delayed messages, you can first calculate the time point for scheduled sending and then send them as scheduled messages.

A code sample is given below to show how to use scheduled messages and delayed messages in TDMQ for RocketMQ. You can also view the complete code sample >>.

Delayed messages first calculate the scheduled sending time via `System.currentTimeMillis() + delayTime`, then send them as scheduled messages.

4.X Client

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

// Set the message to be sent after 10 seconds
long delayTime = System.currentTimeMillis() + 10000;
// Set __STARTDELIVERTIME into the property of msg.
```

```
msg.putUserProperty("__STARTDELIVERTIME",
String.valueOf(delayTime));

SendResult result = producer.send(msg);
System.out.println("Send delay message: " + result);
```

5.X Client

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

Use Restrictions

- When using delayed messages, make sure that the time on the client is in sync with the time on the server (UTC+8 Beijing time in all regions). Otherwise, there will be a time difference.
- There is a precision deviation of about one second for scheduled and delayed messages.
- The time ranges of scheduled and delayed messages varies based on different cluster specifications. For details, see Product Series.
- When using scheduled messages, you need to set a time point after the current time. Otherwise, the message will be sent to the consumer immediately.

Sequential Message

Last updated: 2025-07-15 17:46:17

Sequential message is an advanced type of message offered by TDMQ for RocketMQ. For a specified Topic, messages are published and consumed in strict accordance with the principle of First-In-First-Out (FIFO). That is, messages sent first are consumed first, and messages sent later are consumed later.

Sequential messages are suitable for scenarios that have strict requirements on the sequence of message sending and consumption.

Use Cases

The comparison between sequential messages and general messages is as follows:

Message Type	Consumption Sequence	Performance	Use Cases
General message	No sequence	High	Suitable for scenarios with high demands for throughput and no requirement for production and consumption sequence.
Sequential message	Messages in a specified Topic following the FIFO rule	Average	Scenarios having average demands for throughput and requiring publishing and consuming all messages in a specified Topic in strict accordance with the FIFO rule

Sequential messages are often used in the following actual business scenarios:

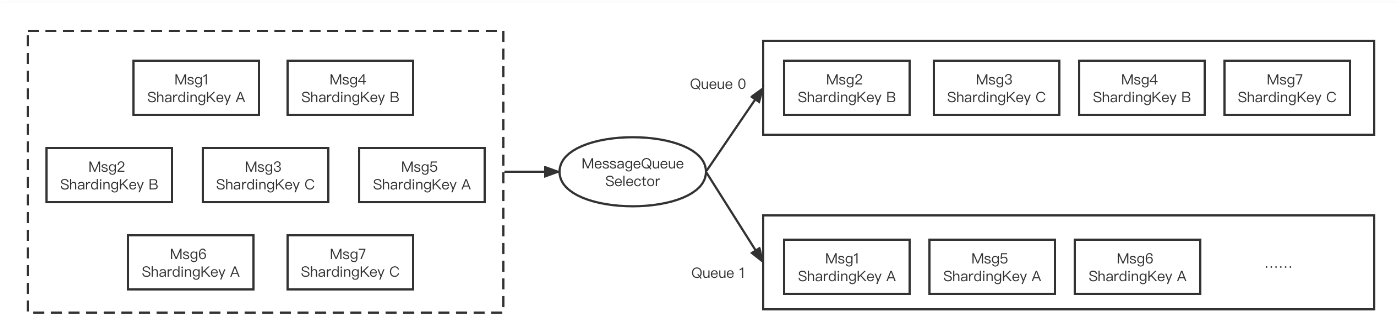
- **Order creation:** In some e-commerce systems, the creation, payment, refund, and logistics messages of an order must be produced or consumed in strict sequence. Otherwise, the order status delivery will be messed up, affecting normal businesses. Therefore, the messages of this order must be produced and consumed in a certain sequence in the client and message queue. In addition, the messages are sequentially dependent, and the processing of the next message depend on the processing result of the preceding message.
- **Log synchronization:** In the scenario of sequential event processing or real-time incremental data

synchronization, sequential messages can also play a vital role. For example, it is necessary to ensure that database operations are in sequence when MySQL binlogs are synchronized.

- **Financial scenarios:** In some matchmaking transaction scenarios such as certain securities trading, priority is given to those who bid first case of the same biding price, so it is necessary to produce and consume sequential messages in a FIFO manner.

How It Works

In TDMQ for RocketMQ, the principle of sequential messages is shown in the figure below. You can partition messages according to a certain standard, such as ShardingKey in the figure. Messages of the same ShardingKey will be assigned to the same queue and consumed in sequence.



The code for sequential messages is as follows:

```
public class Producer {
    public static void main(String[] args) throws
UnsupportedEncodingException {
        try {
            DefaultMQProducer producer = new
DefaultMQProducer("please_rename_unique_group_name");
            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();
        }
    }
}
```

The main difference here is that the `SendResult send(Message msg, MessageQueueSelector selector, Object arg)` method is called. `MessageQueueSelector` is a queue selector and arg is a Java object, which can be passed in as the classification standard for message sending partition.
The `MessageQueueSelector` API is as follows:

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

Where, mqs is the queue that can be sent, msg is the message, arg is the object passed in the preceding send API, and the queue to which the message needs to be sent is returned. In the preceding sample, orderId is used as the partition classification standard, and the remainder of the number of all queues is obtained to send messages with the same orderId to the same queue.

In the production environment, it is recommended that you select the most fine-grained partition key for splitting. For example, when the order ID and user ID are used as the partition key keywords, the messages of the same end user will be processed in sequence, while those of different users will not.

Note:
To ensure high availability of messages, TDMQ for RocketMQ currently does not support "globally sequential messages" in a single queue (users who have already created globally sequential messages can continue to use them normally). If you want to ensure global sequence, you can use a consistent ShardingKey.

Consuming Sequential Messages

Sequential consumption code is as follows:

```
/**
 * Description: Sequential consumer
 */
public class OrderConsumer {
    /**
     *topic name
     */
    private static final String TOPIC_NAME = "order_topic";

    /**
     * consumer group name
     */
    private static final String GROUP_NAME = "group2";
    public static void main(String[] args) throws Exception {
        create message consumer
        // Instantiate the consumer.
        DefaultMQPushConsumer consumer = new
DefaultMQPushConsumer(GROUP_NAME,
                new AclClientRPCHook(new SessionCredentials("accessKey",
"secretKey")),
                new AllocateMessageQueueAveragely(), true, null);
        // Set the NameServer address.
        consumer.setNamesrvAddr("rmq-
xxx.rocketmq.xxxtencenttdmq.com:8080");

consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_OFFSET)
```



```
;
        /*
         Set whether the Consumer starts consuming from the head or tail
of the queue on first startup
         if not first startup, then resume consumption from last
position
         */

consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_OFFSET)
;
        // subscribe to all messages in the topic
        consumer.subscribe(TOPIC_NAME, "*");
        consumer.registerMessageListener(new MessageListenerOrderly() {
            @Override
            public ConsumeOrderlyStatus consumeMessage(List<MessageExt>
msgs, ConsumeOrderlyContext context) {
                context.setAutoCommit(true);
                for (MessageExt msg : msgs) {
                    // You can see each queue has a unique consume
thread to consume, order is ordered per queue (partition)
                    System.out.println("consumeThread=" +
Thread.currentThread().getName() + ", queueId=" + msg.getQueueId() + ",
msgId=" + msg.getMsgId() + ", content:" + new String(msg.getBody()));
                }
                try {
                    // Simulating business logic processing...
                    TimeUnit.SECONDS.sleep(1);
                } catch (Exception e) {
                    e.printStackTrace();
                }
                return ConsumeOrderlyStatus.SUCCESS;
            }
        });
        consumer.start();
        System.out.println("Consumer Started.");
    }
}
```

Transactional Message

Last updated：2025-07-15 17:46:17

This document describes the concept, technical principle, use cases, and application scenarios of transactional messages in TDMQ for RocketMQ.

Concepts

Transactional message is an advanced feature message provided by RocketMQ. It guarantees final transaction consistency in distributed scenarios by binding two-phase submission with local transaction operations. Compared with regular messages, it mainly extends the mechanism of secondary confirmation and compensatory status check for local transactions.

1. half message

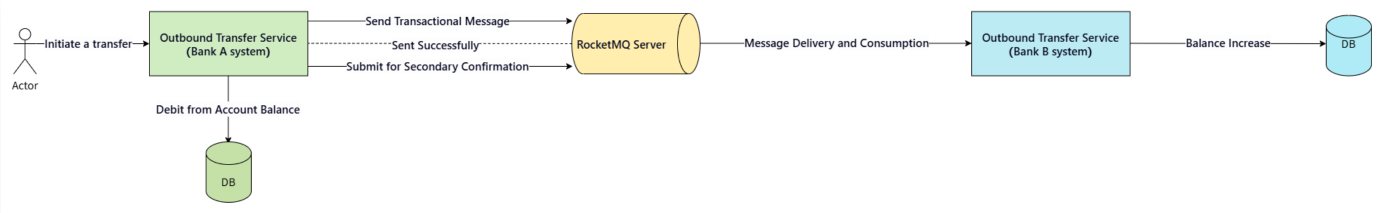
   After the producer sends a transactional message to the RocketMQ server, the message will be persisted and marked with a "temporary not deliverable" status until the local transaction is completed and confirmed. Only then will it determine whether the message is visible to the consumer. Messages in this status are called half messages.
2. Two-phase submission

   The key mechanism to achieve final consistency of transactions involves two phases. In the first phase, a half message is sent. In the second phase, the producer executes the local transaction and submits a confirmation result (commit for delivery or rollback to discard the message) to the RocketMQ server based on the execution result, determining the fate of the half message.
3. OP message:

   Used to tag message status. If a half message has no corresponding OP message, it indicates the two-phase confirmation status is unknown, and the RocketMQ server must proactively check the local transaction status. The OP message content corresponds to the stored Offset of the half message.
4. Related topic:
   - real topic: the actual business topic specified by the producer when sending messages.
   - half topic: a system topic named RMQ_SYS_TRANS_HALF_TOPIC, used to store half messages.
   - op topic: a system topic named MQ_SYS_TRANS_OP_HALF_TOPIC, used to store OP messages. After the second status confirmation of a half message, whether commit or rollback, a corresponding OP message is written to this topic.

Use Cases

For example, in a typical distributed scenario like interbank transfer, it is strictly required to ensure final consistency between the deduction from the transferring party's account and the credit to the transferee's account. We can use TDMQ for RocketMQ's transactional message feature to achieve this, divided into the following three stages:



1. Stage 1: Send a transaction message (prepare fund transfer)

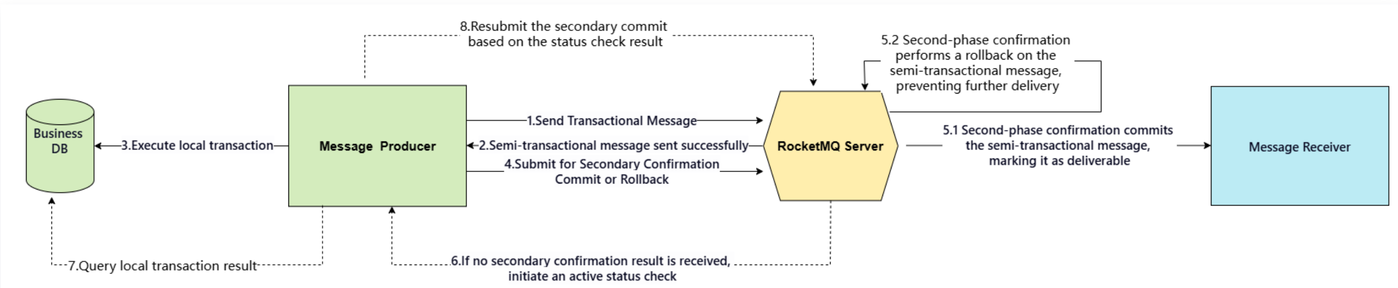
   After user 1 triggers A fund transfer, the banking system where it resides sends A transactional message to the corresponding business topic on the RocketMQ server. The message contains "User 1 transfers 1,000 CNY to user 2's account." At this point, the message is invisible to the transferee's system to avoid executing the posting operation in advance before the transferring party confirms the transaction locally, ensuring capital flow security.
2. Phase 2: Execute local transaction (deduct transfer account)

   The message is sent successfully. System A proceeds to execute the local transaction, deducting the account balance of User1. If the deduction succeeds, it submits a secondary confirmation (commit) to the RocketMQ server, and the message continues delivery downstream. Conversely, if it submits a rollback, the transaction ends, and both parties' account balances remain unchanged.
3. Phase 3: Downstream service consumption (transfer to bank balance addition)

   The transferee's banking system B subscribes to the fund transfer topic in advance. Upon message reception, it performs the operation of adding to User2's account balance. If the consumption process fails due to network issues or account status problems, RocketMQ will automatically trigger the retry mechanism. If multiple retries still fail, the message will be transferred to the DLQ for follow-up manual intervention and verification. The compensatory process ensures the fund is accurately credited eventually.

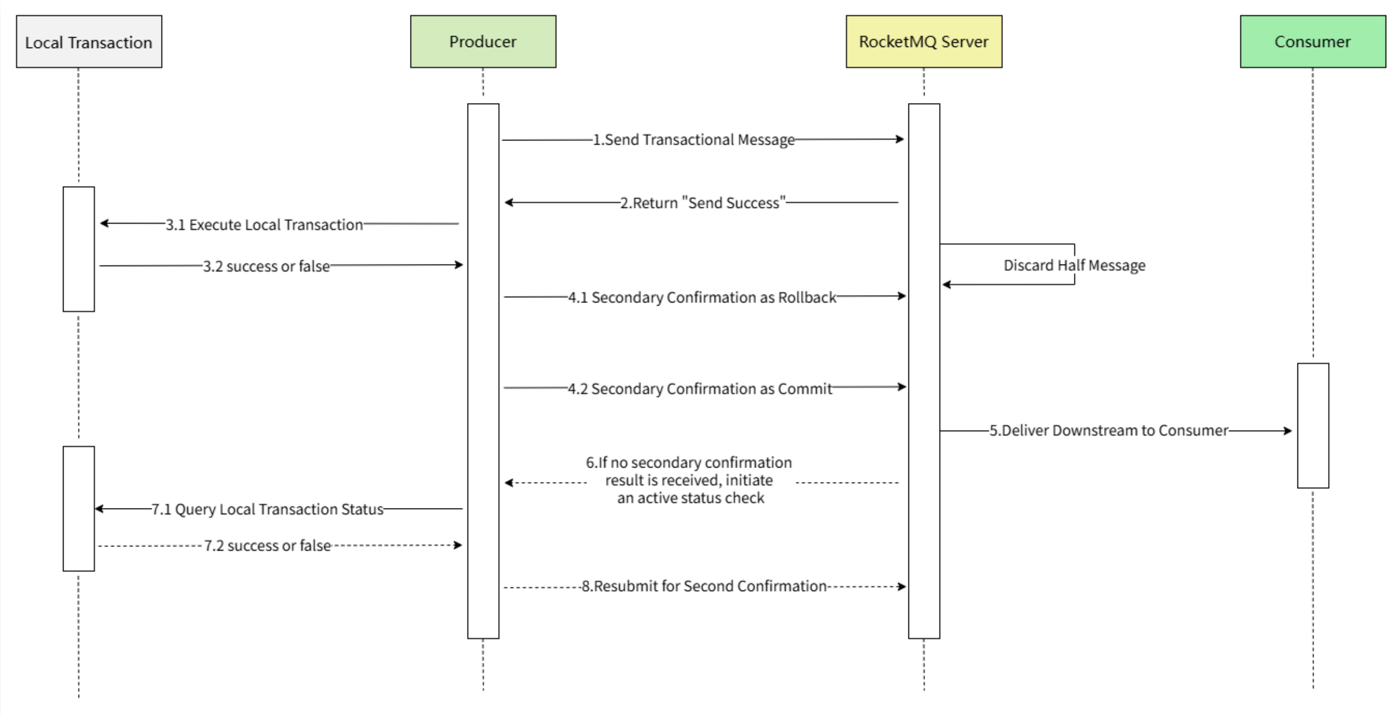
Through these three phases, RocketMQ's transactional message mechanism successfully implements final transaction consistency in cross-line transfer scenarios. Similarly, in other scenarios such as e-commerce order payment and inventory deduction, financial transaction reconciliation, and corporate multi-system data synchronization, RocketMQ transactional messages can guarantee final consistency in cross-service operations based on their reliable mechanism.

Implementation Process



1. The producer sends a transactional message to the RocketMQ server.
2. The server stores this message and returns a sent successfully response. At this point, the message is invisible to downstream consumers and is in half message status.
3. The producer receives the half message response successfully and proceeds to execute the local transaction (such as updating the business database).
4. Based on the outcome of the local transaction, the producer submits the final status to the RocketMQ server, that is, the secondary confirmation.
5. When the confirmation result is commit, the server will deliver the transaction message to consumers. When the confirmation result is rollback, the server will discard the message and no longer deliver it.
6. If the confirmation result is unknown or fails to receive confirmation within a specified period, the transaction status will be proactively checked.
7. When the producer fails to submit the final status or the secondary confirmation result is unknown, the RocketMQ server will actively initiate a transaction result query request to the producer service.
8. After receiving the request, the producer submits the secondary confirmation result, and the logic returns to step 5. At this point, if the producer service is temporarily unavailable, the RocketMQ server will continue to actively initiate a query request after a specified interval until exceeding the maximum number of retries, then roll back the message.

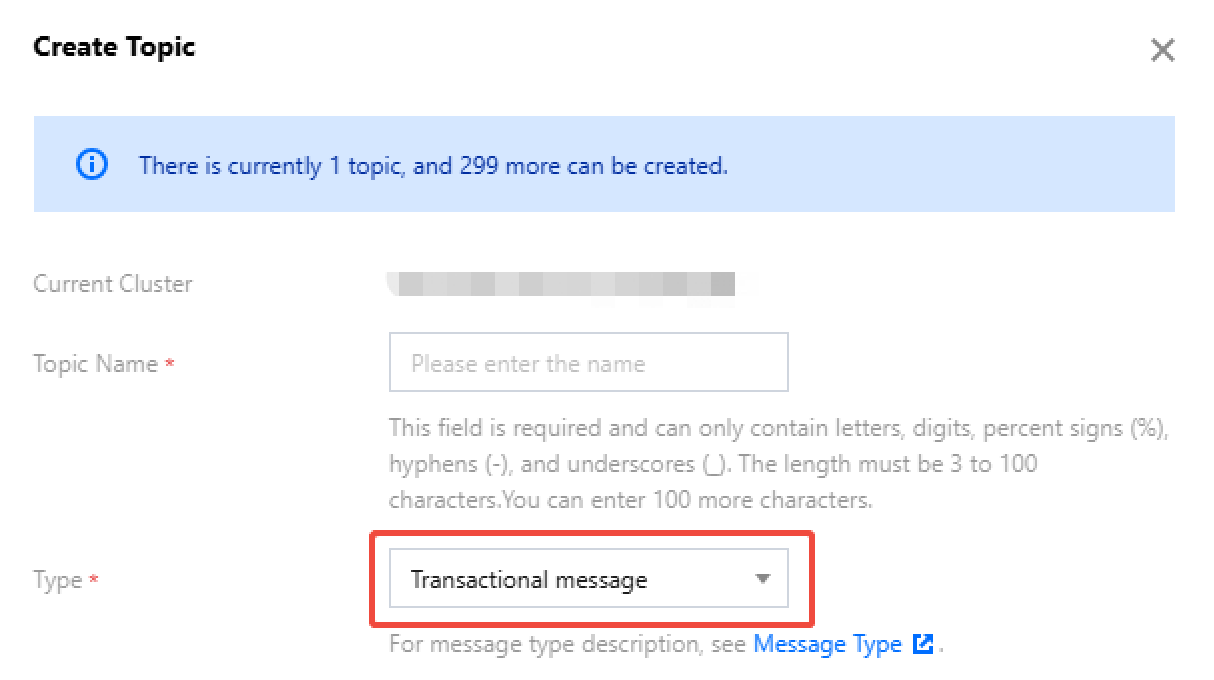
Thus, regardless of the execution success of local transaction operations, final consistency of the transaction status can be achieved. The above procedure can be intuitively represented by a time sequence diagram:



Usage Examples

Here we take the TDMQ version of RocketMQ 5.x Version Cluster as an example to demonstrate the usage method and effect of transaction message.

1. First, log in to the RocketMQ console and create a new topic with the message type set as transaction message.



2. Taking Java language as an example, introduce the dependency for the corresponding 5.x version.

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

3. Start the producer.

```
public class ProducerTransactionMessageDemo {

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

    private static boolean executeLocalTransaction() {
        // Simulate a local transaction (such as a database insert
operation), assume execution successful here
        return true;
    }

    private static boolean checkTransactionStatus(String orderId) {
        // Simulate local query of transaction execution result, for
example, query whether the order ID has been stored, return true if
```

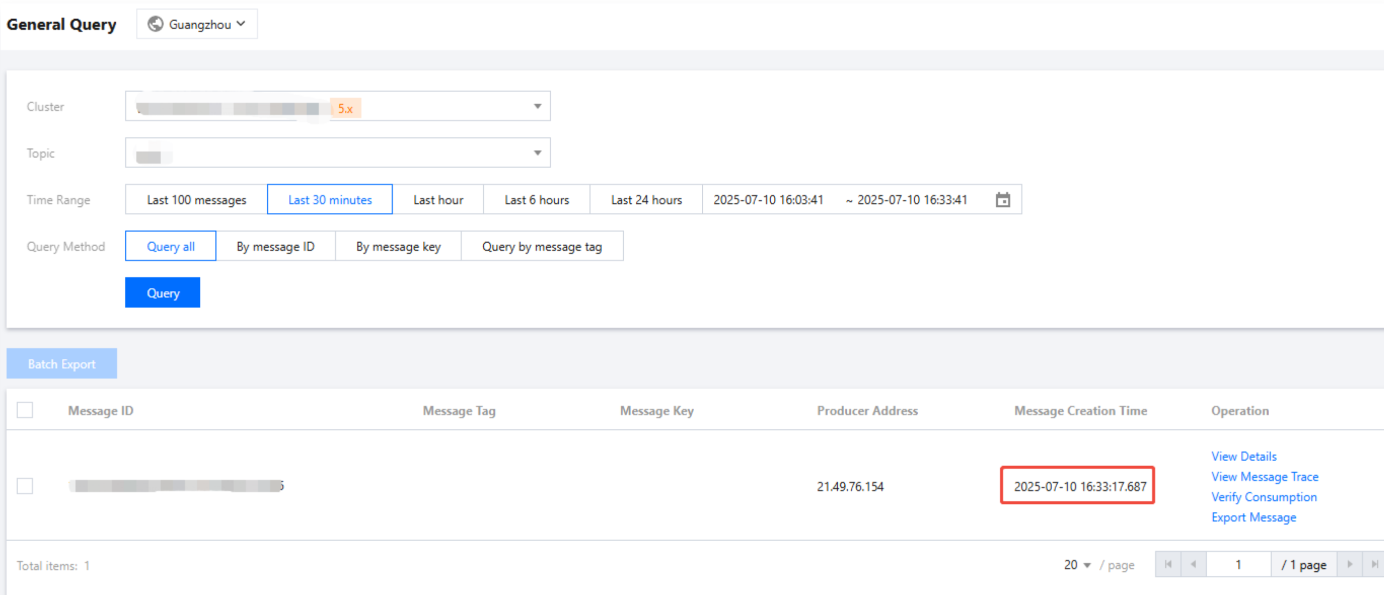
```
found
        return true;
    }


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

        final ClientServiceProvider provider =
ClientServiceProvider.loadService();
        // Get ak and sk in the console permission management page
        String accessKey = "your-ak";
        String secretKey = "your-sk";
        SessionCredentialsProvider sessionCredentialsProvider =
            new StaticSessionCredentialsProvider(accessKey,
secretKey);

        // Get the access address provided by Tencent Cloud from
console and fill in
        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);
            // Server proactively checks local transaction status
            String orderId =
messageView.getProperties().get("orderId");
            boolean isSuccess = checkTransactionStatus(orderId);
            return isSuccess ? TransactionResolution.COMMIT :
TransactionResolution.ROLLBACK;
        };

        Create a producer and set the checker object
        Producer producer = provider.newProducerBuilder()
```

```
                .setClientConfiguration(clientConfiguration)
                .setTopics(topic)
                .setTransactionChecker(checker)
                .build();

        execute a transaction enabling statement
        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")
                //Transactional messages usually set a unique ID
associated with the local transaction, used to validate local
transaction operations.
                .addProperty("orderId", "0001")
                .setBody(body)
                .build();

        // Send half message
        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;
        }

        // Execute local transaction
        boolean localTxSuccess = executeLocalTransaction();
        if (localTxSuccess) {
            // Local transaction is successfully executed, secondary
confirmation is Commit
            transaction.commit();
        } else {
```

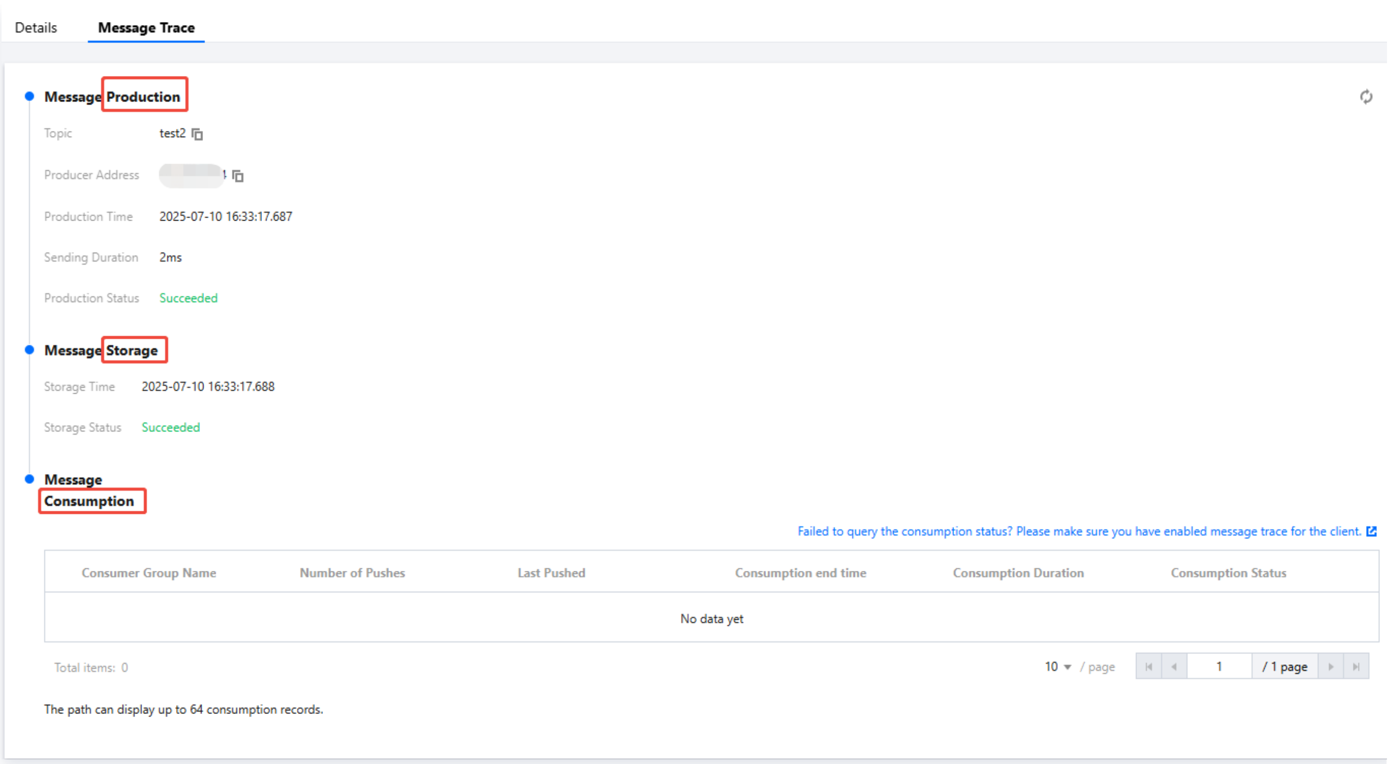


```
                // Local transaction execution failed, secondary
confirmation is Rollback
                transaction.rollback();
            }
            // producer.close();
        }
}
```

4. After running the code, on the console's message query page, you can see a delivered message pending consumption.



5. Start the consumer, subscribe to this topic, and after successfully consuming the message, view the message trace in the Tencent Cloud console.

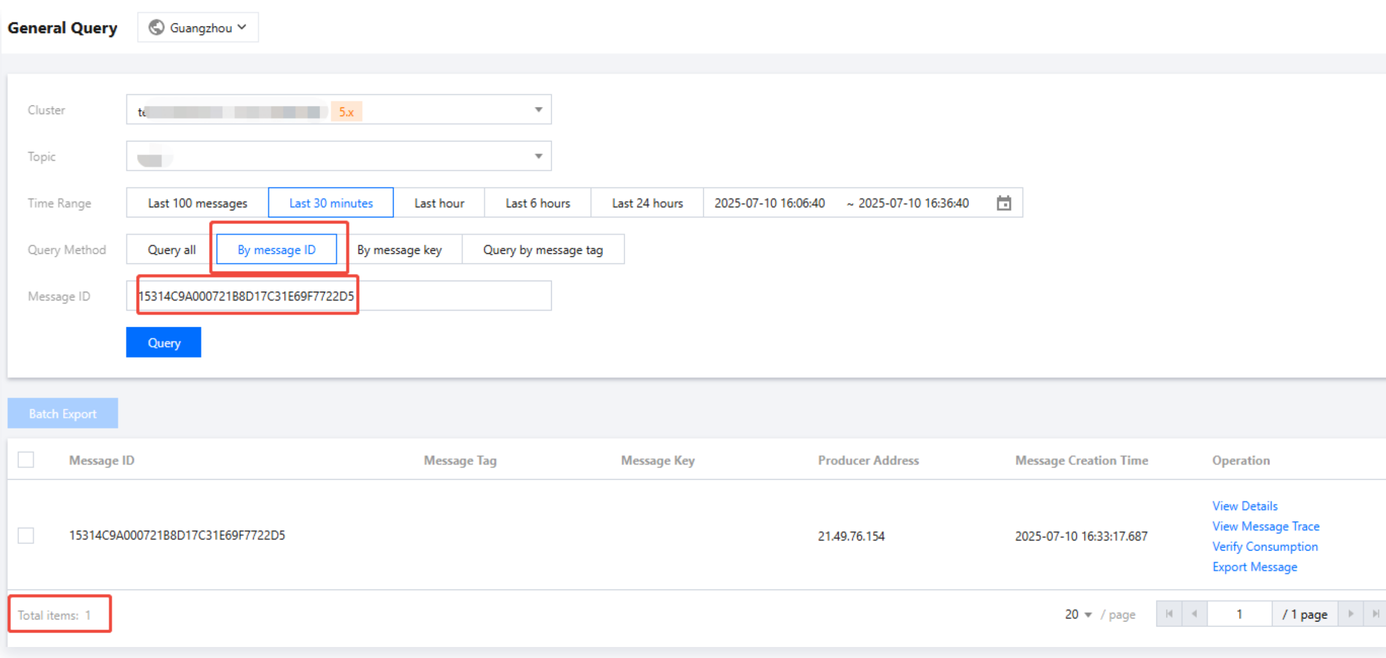


6. Modify the code, assume the local transaction fails to execute, and roll back the transaction message in half message state.

```
private static boolean executeLocalTransaction() {
    // Local transaction execution failed
    return false;
}

private static boolean checkTransactionStatus(String orderId) {
    // The result is also rollback, return false
    return false;
}
```

7. At this point, you can see that the message was sent successfully, but it is invisible on the console's message query page, and starting the consumer cannot consume this message.

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

Precautions

When using transactional messages, note the following points:

1. The topic type must be TRANSACTION, otherwise producing messages will report an error. Critical error information: current message type does not match with topic accept message types.
2. Transactional messages do not support delay. If a delay property is set up, it will be cleared before sending messages.
3. If the local transaction execution is slow, the server should return unknown during transaction check. If the local transaction execution time is confirmed to be long, modify the first transaction check time to avoid a large number of transactions with unknown results.

Message Filtering

Last updated：2025-07-15 17:46:17

This document describes the features, application scenarios, and usage instructions of message filtering in TDMQ for RocketMQ.

Feature Description

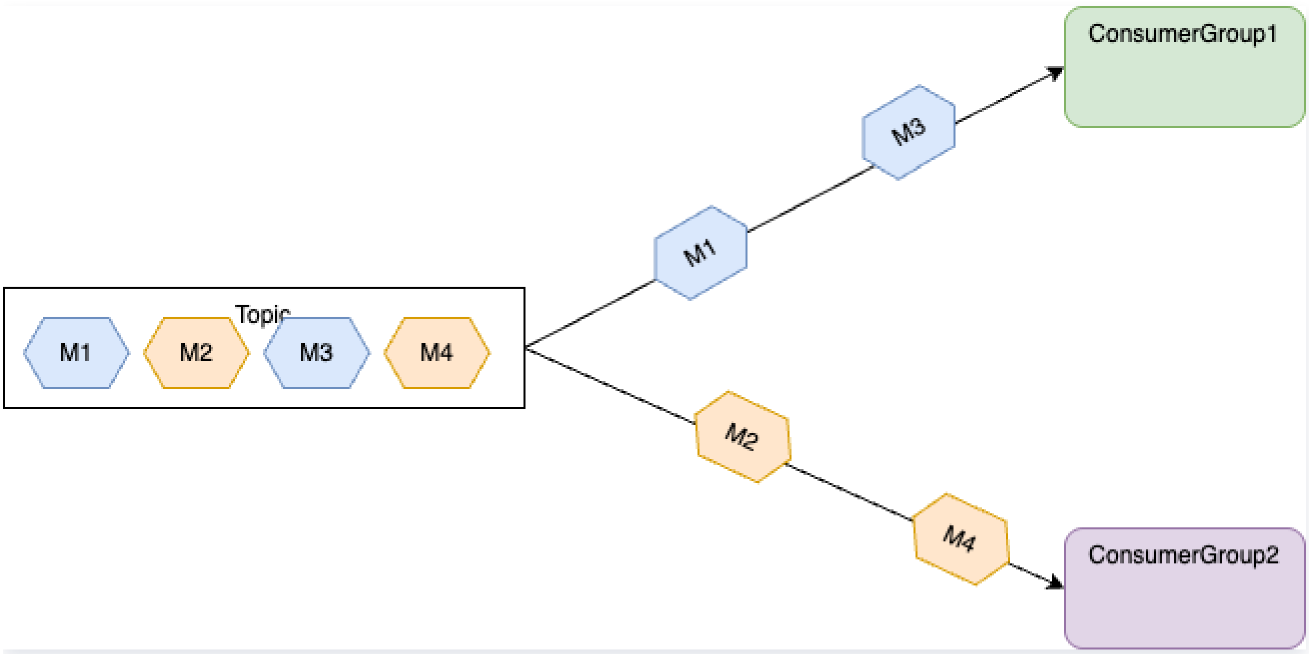
Message filtering indicates that messages are filtered by the message attribute configured by the message producer when the producer sends messages to the topic. The consumer that subscribes to the topic can filter messages based on their attributes so that only eligible messages are delivered to the consumer for consumption.

If a consumer configures no filter conditions when subscribing to a topic, no matter whether filter attributes are configured during message sending, all messages in the topic will be delivered to the consumer for consumption.

Use Cases

Generally, messages with the same business attributes are stored in the same topic. For example, when an order transaction topic contains messages of order placements, payments, and deliveries. These messages are sent to the same transaction statement topic. To obtain a certain type of message, users can filter it on the client.

- Billing system: just subscribe to payment messages.
- Logistics system: just subscribe to logistics messages.
- Inventory system: just subscribe to order messages.



Instructions

Currently, message filtering mainly supports two types of filtering: SQL filtering and TAG filtering. The core logic involves setting custom fields when sending messages, then specifying the corresponding filter expression when subscribing via a consumption group. Messages are filtered on the server side before being consumed by the group.

Filtering by Tag

Sending Messages

> **Note:**
> During message sending, tags must be clearly specified for each message.

```
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();
```

Subscribing to Messages

- Subscribing to all tags:
- If a consumer wants to subscribe to all types of messages under a topic, an asterisk (*) can be used to represent all tags.

```
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();
```

- Subscribing to one tag:
- If a consumer wants to subscribe to a certain type of messages under a topic, the tag should be specified clearly.

```
  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();
```

- Subscribing to multiple tags:
- If a consumer wants to subscribe to multiple types of messages under a topic, two vertical bars (`||`) should be added between the two tags for separation.

```
 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();
```

Filtering by SQL

Sending Messages

The message sending code here is basically the same as the code for sending simple messages. A message is allowed to carry multiple user-defined attributes when you construct the message body.

```
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)
            // Some information for SQL filtering
            .addProperty("key1", "value1")
            .build();
```

Subscribing to Messages

The message consumption code here is basically the same as the code for consuming simple messages. However, a message needs to be carried with the corresponding SQL expression when being subscribed to.

```
  String consumerGroup = "yourConsumerGroup";
        String topic = "yourTopic";
        String sql = "key1 IS NOT NULL AND key1='value1'";
        // SQL expression
        FilterExpression filterExpression = new
FilterExpression(sql, FilterExpressionType.SQL92);
        // If all is subscribed to
        //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();
```

Note:

The preceding sections provide introduction to the use instructions of message publishing and subscription. For more operations, see GitHub Demo or RocketMQ Official Documentation.

Use Limits

Since SQL attribute filtering involves producers defining message attributes and consumers setting SQL filter conditions, the computation may yield different results. Therefore, the server's processing method is as follows:

Exception handling: If the expression calculation of the filter condition throws an exception, the message is filtered by default and will not be delivered to consumers. For example, comparing values of numeric and non-numeric types.

- Null value handling: If the expression calculation of the filter condition is null or not a boolean type (true and false), the message is filtered by default and will not be delivered to consumers. For example, when sending a message, if a certain attribute does not exist and the filter condition directly uses this attribute during subscription, the expression calculation result will be null.
- Type mismatch handling: If the message custom attribute is floating-point but the filter condition uses an integer to judge, the message is filtered by default and will not be delivered to consumers.

Although this approach is flexible, it is not recommended to set too many values in the message header because the total size limit of message header properties is 32K, and the built-in ones already occupy a significant portion. If it becomes excessively long, it may cause message sending or consumption to be abnormal.

Usage recommendations

Divide topics and tags reasonably.

From the message filtering mechanism and the principles behind topics, you can see that business messages can be filtered based on topics or by tags and attributes within a topic. For the selection of split methods, note the following problems.

- Message type consistency: Different types of messages, such as sequential messaging and standard messages, require the use of different topics for splitting and cannot be grouped by Tag.
- Whether the business domain is the same: Messages from different business domains and departments should be split into different topics. For example, logistics messages and payment messages should use two different topics. For logistics messages within the same topic, ordinary logistics messages and urgent logistics messages can be distinguished by different Tags.
- Message scale and severity consistency: If there is a huge difference in message scale or severity, different topics should be used to isolate and split them.

Message Retry

Last updated：2025-07-15 17:46:17

This document describes the mechanism of message retry and its usage in TDMQ for RocketMQ.

Feature Description

When a message is consumed for the first time by a consumer and does not receive a normal response, or when users request the server to deliver it again, TDMQ for RocketMQ will automatically attempt to deliver this message again through the message retry mechanism until it is consumed successfully. When the number of retries reaches the specified value but the message is still not consumed successfully, retry will stop, and the message will be delivered to the dead letter queue.
After the message enters the dead letter queue, TDMQ for RocketMQ can no longer process it automatically. In this situation, human intervention is generally required. You can write a dedicated client to subscribe to the dead letter queue to process such failed messages.

> **Note:**
> The broker will automatically retry in the cluster consumption mode but not the broadcasting consumption mode.

The following results are considered as consumption failure, and the message will be retried accordingly:
- The consumer returns ConsumeResult.FAILURE.
- The consumer returns null.
- The consumer actively/passively throws an exception.

Maximum Number of Retries

When a message needs to be retried in TDMQ for RocketMQ, set the messageDelayLevel parameter as follows to configure the number of retries and retry intervals:

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

The number of retries and retry intervals have the following relationships:

Retry No.	Time Interval Since Last Retry	Retry No.	Time Interval Since Last Retry
1	1 second	10	6 minutes
2	5 seconds	11	7 minutes

3	10 seconds	12	8 minutes
4	30 seconds	13	9 minutes
5	1 minute	14	10 minutes
6	2 minutes	15	20 minutes
7	3 minutes	16	30 minutes
8	4 minutes	17	1 hour
9	5 minutes	18	2 hours

Instructions

5.x SDK

No special processing is required, the 5.0 SDK follows the preceding rules.

4.x SDK

If users need to adjust the number of retries by their own, it can be determined by setting the consumer parameter.

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

Dead Letter Message

Last updated：2025-07-15 17:46:17

This document describes the dead letter queue in TDMQ for RocketMQ and how to use it.

Feature Description

- When a message is consumed for the first time by a consumer and does not receive a normal response, or when users request the server to deliver it again, TDMQ for RocketMQ will automatically attempt to deliver this message again through the message retry mechanism until it is consumed successfully. When the number of retries reaches the specified value but the message is still not consumed successfully, retry will stop, and the message will be delivered to the dead letter queue.
- If messages enter the dead letter queue, TDMQ for RocketMQ can no longer automatically process them. In this situation, human intervention is required. You can confirm it via message export, or via specified message resending on the control console.

Feature Description

- Different from the retry queue, which supports automatic consumption, messages within the dead letter queue require manual intervention.
- The validity of messages also adheres to the rule of deletion after three days by default.
- The dead letter queue starts with `%DLQ%` and corresponds to the consumer group one by one. Therefore, a dead letter queue contains all dead letter messages corresponding to the group ID, regardless of their originating topic.

Consumption Mode

Cluster Consumption

Last updated：2025-07-15 17:46:17

Introduction to Cluster Consumption Mode

When the cluster consumption mode is used, any message only needs to be processed by just one consumer within the same subscription group.

Use Cases

This is suitable to scenarios where each message only needs to be processed once.

How to Use

The 5.0 SDK uses the cluster consumption mode by default, requiring no special configuration.

> **Note:**
> Please ensure consistency in the subscription relationships of all consumer instances under the same group ID.

Broadcast Consumption

Last updated：2025-07-15 17:46:17

Introduction to Broadcasting Consumption Mode

When the broadcasting consumption mode is used, each message is pushed to all registered consumers within the cluster, ensuring that each message is consumed at least once by every consumer.

Use Cases

This mode is suitable for scenarios where every message needs to be processed by every consumer within the cluster.

How to Use

The 5.0 SDK no longer supports broadcasting consumption. However, a similar feature can be achieved by creating a distinct subscription group for every consumer if it is required.

> **Note:**
> Please ensure consistency in the subscription relationships of all consumer instances under the same group ID.

POP Consumption Mode (5.X)

Last updated：2025-07-15 17:46:17

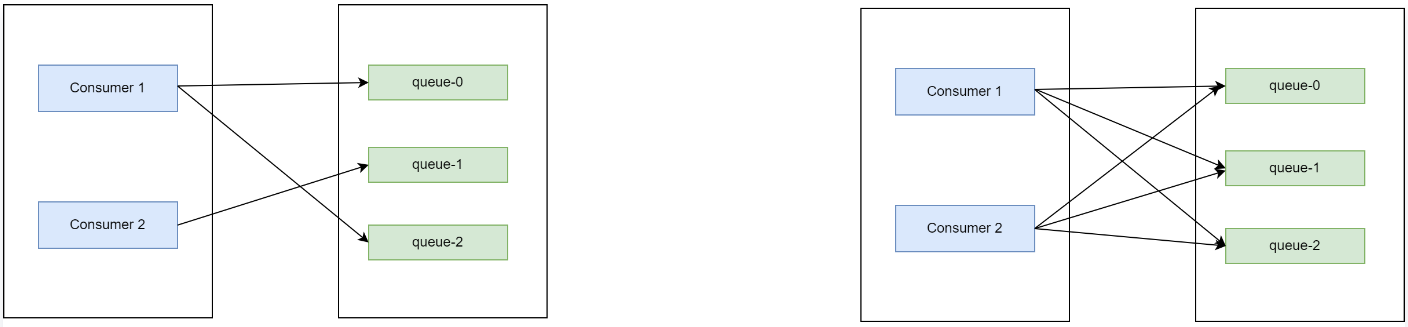
Issue Background

RocketMQ is known by many customers and developers for its high performance, low delay, and backlog resistance characteristics. However, during the usage of the RocketMQ 4.x client SDK, some customers report that the consumer client encounters issues while consuming messages, such as the commonly used Push Consumer.

- The SDK takes on too many functions, such as pulling messages, load balancing, message offset management, and Rebalance when adding new clients, which is unfriendly to multi-language developers.
- The queue-exclusive load balancing policy can cause consumption bottlenecks: Each queue on the Broker can only be assigned to one consumer client in the same Group. So when the queue count is fixed, simply adding more consumer clients won't enhance consumption performance. Assuming a Topic has 10 queues, the Group can have up to 10 clients consuming messages (meaning each client can consume at most one queue). During peak business periods, even if the customer wants to add a new client to consume messages, the newly added 11th client will be unable to consume messages.
- A single client exception leads to accumulation. If a client hangs due to an exception, since the heartbeat with the server is not disconnected, the client will still be allocated to the queue for consumption. However, because the client is actually unable to consume messages, exceptions accumulate. Due to the previous reason, simply increasing the number of clients does not resolve the issue.

Solution

Given the above reasons, 5.x introduced the POP consumption mode.
In POP mode, the consumer offset is managed by the server, so multiple clients can consume the same queue. Clients using POP mode pull messages from all queues, addressing the issues of single client exceptions and consumption bottlenecks.
Meanwhile, the server maintains consumption information, making the client SDK more lightweight and easier to port to multiple languages.



Before

After

Sample Code

How to use POP consumption mode?

Require the use of gRPC SDK 5.x with the following dependencies:

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

Meanwhile, refer to the open-source community DEMO as follows (for example, Java code):

```
/*
 * 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 = "User A ak";
        String secretKey = "User A sk";
        SessionCredentialsProvider sessionCredentialsProvider =
```



```
            new StaticSessionCredentialsProvider(accessKey, secretKey);

        String endpoints = "Tencent Cloud webpage access point";
        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 = "consumption group";
        // Default consumption time, 30s, which means that for pulled
messages, if consumption is not completed within 30s, the message will
be pulled again by another client.
        // users are advised to configure based on own scenario
        Duration awaitDuration = Duration.ofSeconds(30);
        String tag = "*";
        String topic = "topic name";
        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();
    }
}
```

In such cases, consumers in the same consumption group will not be bound to queues one-to-one, and the problem of queue accumulation due to a single consumer blocking, as seen in earlier 4.x versions, can be avoided to the maximum extent.

Cluster Consumption and Broadcast Consumption

Last updated：2025-07-15 17:46:17

This document describes the features and use cases of cluster consumption and broadcast consumption in the RocketMQ.

Feature Introduction

- Cluster consumption: In the cluster consumption mode, any message only needs to be processed by any consumer in the cluster.
- Broadcast consumption: In the broadcast consumption mode, each message will be pushed to all registered consumers in the cluster, ensuring that each message is consumed at least once by each consumer.

Use Cases

- Cluster consumption: Suitable for scenarios where each message needs to be processed only once.
- Broadcast consumption: Suitable for scenarios where each message needs to be processed by every consumer in the cluster.

Sample Code

- **Cluster subscription**

5.x SDK

The SDK 5.0 defaults to cluster consumption mode with no need for special settings.

Notes:
Make sure that all consumer instances under the same group ID have consistent subscription configurations.

4.x SDK

All consumers identified by the same Group ID share consumption messages equally. For example, if a Topic has 9 messages and a Group ID has 3 Consumer instances, then in cluster consumption mode, each instance evenly distributes and only consumes 3 of them.

```
// Cluster subscription mode (If not explicitly set, cluster mode is
used by default).
properties.put(PropertyKeyConst.MessageModel,
PropertyValueConst.CLUSTERING);
```

- **Broadcast subscription**

5.x SDK

5.0 SDK does not support broadcast consumption. If required, you can create a separate subscription group for each consumer to implement similar functionality.

> **Notes:**
> Make sure that all consumer instances under the same group ID have consistent subscription configurations.

4.x SDK

All consumers identified by the same Group ID consume each message once. For example, if a Topic has 9 messages and a Group ID has 3 Consumer instances, then in broadcast consumption mode, each instance consumes all 9 messages.

```
// Broadcast subscription mode.
properties.put(PropertyKeyConst.MessageModel,
PropertyValueConst.BROADCASTING);
```

> **Note**
> Make sure that all consumer instances under the same group ID have consistent subscription configurations.

Subscription Relationship Consistency

Last updated：2025-07-15 17:46:17

This document describes the key points of subscription relationship consistency, from definition and constraint mechanisms to underlying implementation principles and optimization practices. It also combines actual case sharing of TDMQ RocketMQ's solution for inconsistent subscription relationships, helping developers quickly locate the root cause and build a stable and reliable distributed messaging system.

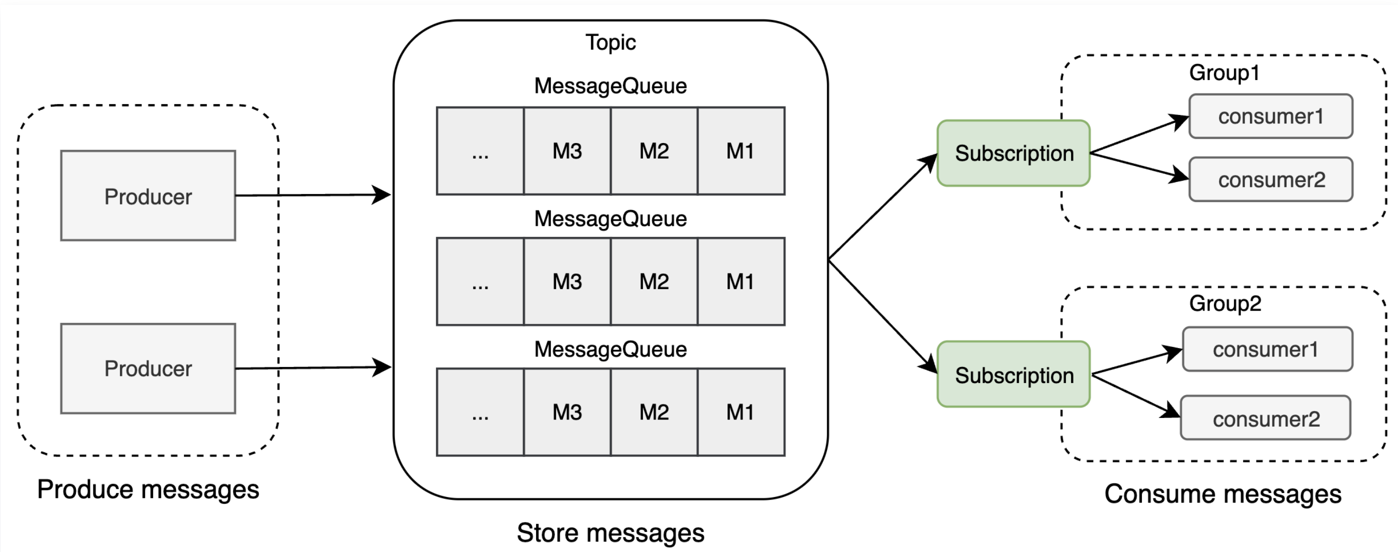
Defining Subscription Relationships

The subscription relationship is the rule and status configuration for consumers to obtain and process messages in the RocketMQ system. It is dynamically registered to the server by the consumer group and subsequently matches messages and maintains consumption progress according to the filtering rules defined in the subscription relationship during message transmission.

By configuring the subscription relationship, you can control the following consumption behavior:

- Message filter rule: Used to control which messages within a topic consumers process when consuming messages. Setting a filter rule enables efficient filtering of required message collections and flexible adjustment of the reception range based on different business scenarios.
- Consumption status: The RocketMQ server provides subscription relationship persistence by default. After a consumer group registers its subscription relationship with the server, it can obtain the pre-offline consumption progress and resume consumption post-launch.

In the domain model of RocketMQ, the location and process of the subscription relationship are as follows:



1. The message is initialized by the producer and sent to the RocketMQ server.
2. Messages are stored in the specified queue of the topic in the order they arrive at the RocketMQ server.
3. Consumers obtain messages from the RocketMQ server according to the specified subscription

relationship and consume them.

Subscription Relationship Consistency Constraints

Subscription relationship consistency requires that all consumer instances in the same consumer group must have identical subscribed topics and filter rules. Here, three constraints are involved:

- **Consumption groups must be consistent**

  For most distributed applications, a consumption group typically mounts multiple Consumer instances, and the constraint range for subscription relationship consistency is all consumers in the same consumption group.

- **Subscribed topics must be consistent**

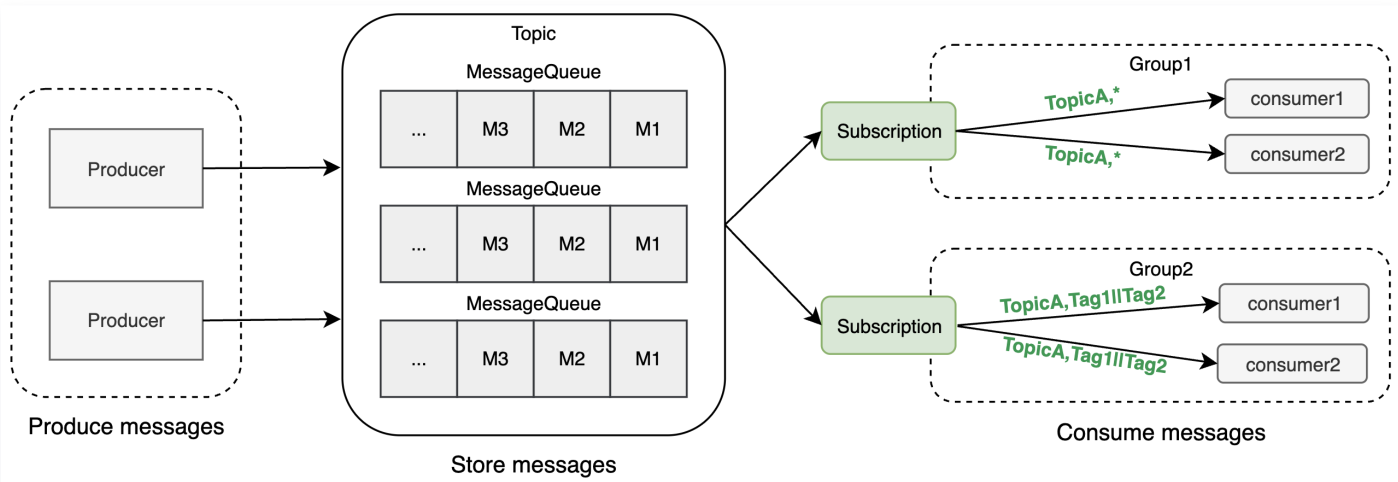
  All consumers in the same consumption group must subscribe to the same topics. For example, if consumer1 subscribes to TopicA and TopicB, consumer2 must also subscribe to TopicA and TopicB. It cannot subscribe only to TopicA, only to TopicB, or to TopicA and TopicC.

- **Filter rules must be consistent**

  All consumers in the same consumption group must have consistent filter rules, including the number of tags and the order in which they appear. For example, if consumer1 subscribes to TopicB with tags Tag1||Tag2, consumer2 must also subscribe to TopicB with tags Tag1||Tag2. It cannot subscribe only to Tag1, only to Tag2, or to Tag2||Tag1.

Examples of Consistent Subscription Relationship

The following diagram shows two common correct subscription relationships, corresponding to two cases:



Correct example 1: Single Topic with Single Tag subscription

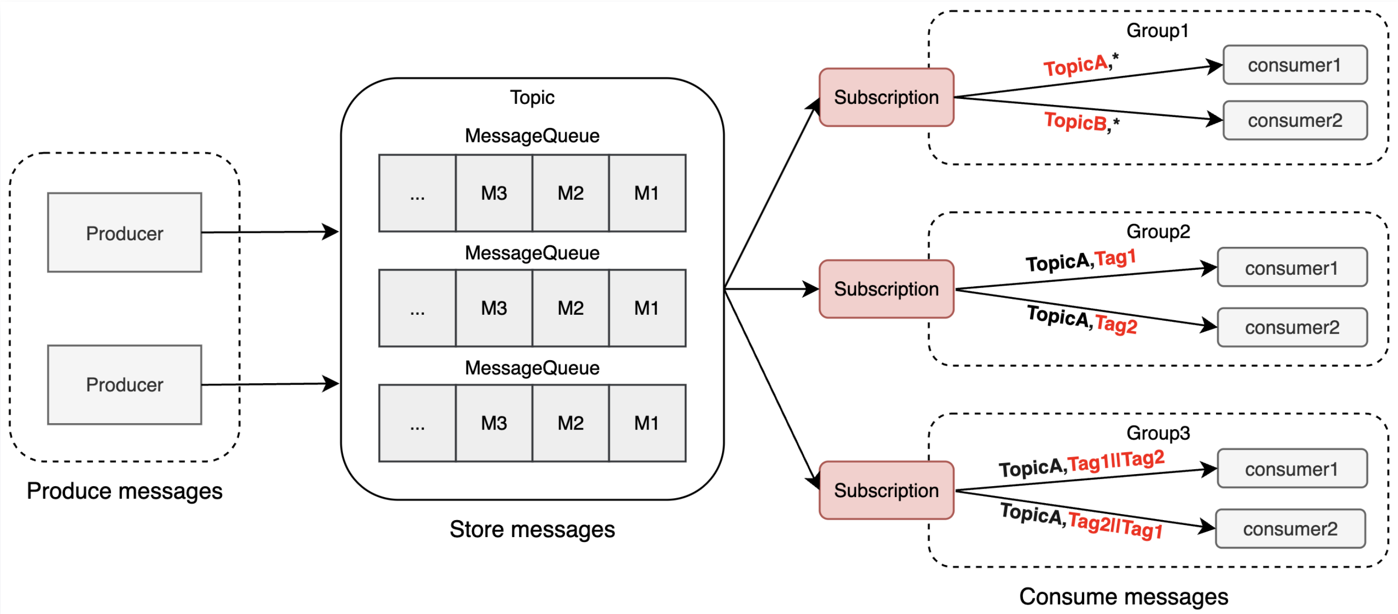
As shown in figure, both consumer1 and consumer2 in Group 1 subscribe to all messages in TopicA.

Correct example 2: Single Topic with multiple Tag subscriptions

As shown in figure, both consumer1 and consumer2 in Group 2 subscribe to messages in TopicA with Tag1 or Tag2, and the sequence is Tag1||Tag2.

Examples of Inconsistent Subscription Relationship

The following diagram shows three typical incorrect subscription relationships, corresponding to three cases:



Incorrect example 1: Different subscription topics
As shown in figure, consumer1 and consumer2 in Group 1 subscribe to different topics.
Incorrect example 2: The subscription Topic is identical but the Tag is different
As shown in figure, consumer1 in Group 2 subscribes to Tag1 in TopicA, while consumer2 subscribes to Tag2 in TopicA.
Incorrect example 3: The subscription Topic and Tag are the same but the Tag sequence is different
As shown in figure, consumer1 in Group 3 subscribes to Tag1||Tag2 in TopicA, while consumer2 subscribes to Tag2||Tag1 in TopicA. Although the subscribed tags are the same, the sequential order is different, which does not meet the subscription consistency constraint.

Impact of Inconsistent Subscription Relationship

If the subscription relationship is inconsistent, it may cause message consumption logic chaos, duplicate consumption, or message loss.
In the following example, here we start two consumers, they belong to consumption group Group1, both subscribe to topic TopicA, but consumer1 subscribes to messages of Tag1, consumer2 subscribes to messages of Tag2.

```
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();
```

In such cases, what are the performance of the two clients?

- Consumer1 is unable to consume messages with Tag1 because when pulling messages, the subscription information of the consumption group on the server side has Tag2. After server-side filtering, all messages pulled by consumer1 have Tag2. However, the consumer also filters messages after receiving them, so some messages are filtered.
- Consumer2 can only consume part of messages with Tag2 because only part of the queue is assigned to consumer2.

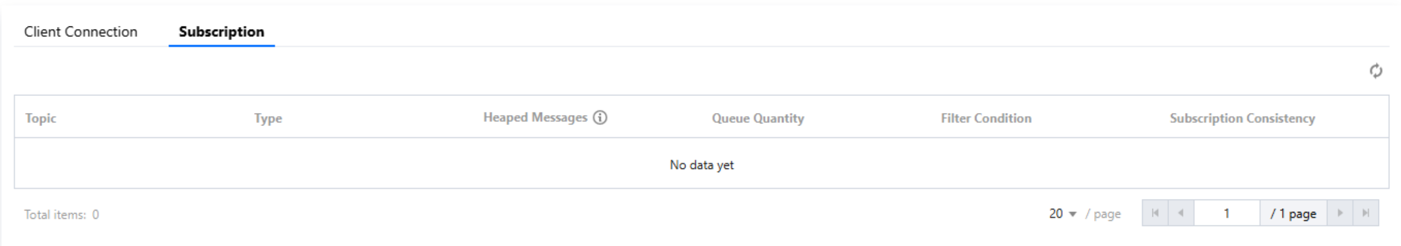
However, on the server side, the subscription information of each consumer client in the same consumption group may be overwritten mutually, so this kind of consumption status is super chaotic. In the example above, the consumption of consumer1 and consumer2 may also switch.

Tencent Cloud Optimization Practice

Inconsistent subscription relationship may directly cause message consumption exception, requiring rapid positioning and fix. TDMQ RocketMQ Console provides visual inspection capacity without manual intervention to check logs or configuration one by one. Via console, you can complete issue discovery, locating, and fix in 3 steps to reduce Ops complexity.

1. **One-click scan**

   Automatically compare subscription configurations of all clients in the consumption group and highlight inconsistent subscription relationships.



2. **Precise positioning**

   Click the inconsistency details to directly associate with the specific client instance and quickly determine the instance of unexpected subscription relationships.

Client ID	Client Address	Subscribe to Topic	Subscription mode	Subscription rules
[redacted] Pro.local@35568@0@byjpbag848	[redacted]:51965	TopicA	TAG	Tag1
[redacted] Pro.local@35588@0@byjpxdi6of	[redacted] 73	TopicA	TAG	Tag2

3. **Closed-loop verification**

   Revised real-time synchronization of subscription relationship status to ensure the consumption group behaves as expected.

FAQs

In which typical scenarios does inconsistent subscription relationship occur?

- The environment is not completely isolated. The non-production and production environments use the same Group to subscribe to different Topics.
- Business code modifies the subscription relationship. New and old versions of consumers coexist during grayscale and version release.

Usage recommendations

Subscription Relationship Consistency is the core guarantee for correct consumption behavior in the RocketMQ message system.

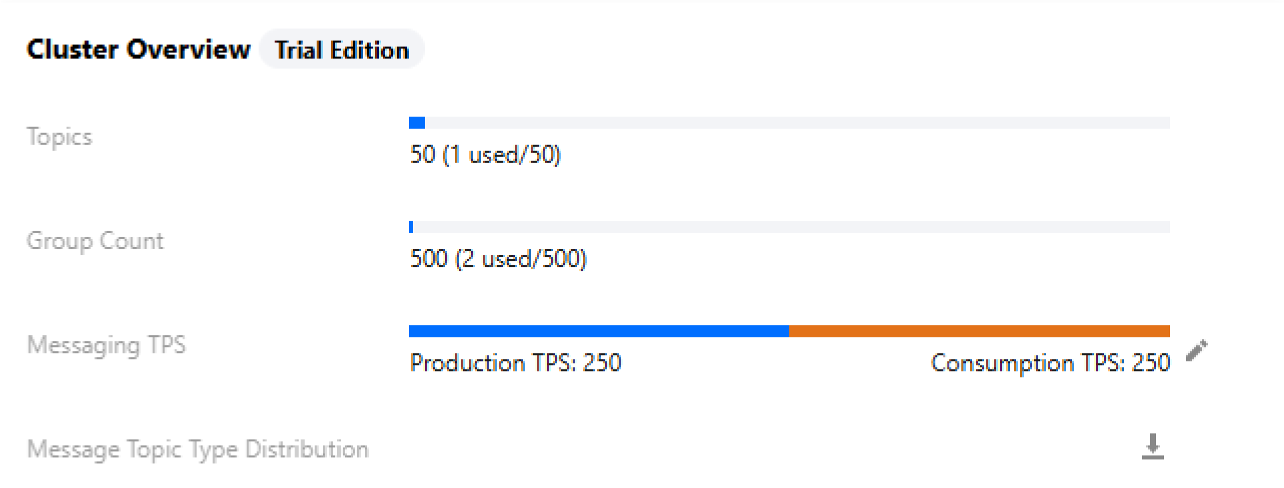
1. Core constraints: All consumption within the same consumption group must strictly adhere to the principle of identical topic and filter rule (including Tag sequence). Any variation may cause message loss.
2. Tencent Cloud capacity: Rely on the console's one-click scan, precise positioning, and closed-loop verification feature, developers can rapidly identify anomalies in instances and fix them, shortening traditional manual troubleshooting duration from hour-level to minute-level.
3. Best practice: It is recommended to pass strategies such as consumption group isolation, strict verification of dynamic configuration, and hierarchical governance in multi-Topic scenarios to avoid the risk of inconsistent subscription relationship from the source.

Throttling

Last updated：2025-07-15 17:46:18

Overview

TDMQ for RocketMQ provides message service for various large-scale, low-latency online services with high availability requirements. Clients establish persistent connections with RocketMQ clusters via SDK to send and receive messages, consuming compute, storage, and network bandwidth resources of cluster machines. To deliver high-quality message service, we need to control the load watermarks of clusters under high-concurrency and heavy traffic to ensure system stability and reliability. Therefore, the server **limits the maximum converted message count per second (Transaction Per Second/TPS) that clients can send and consume based on cluster specifications**. For specific calculation rules, see Billing Overview – Compute Specification. To balance isolation and flexibility, TPS quotas for sending and consuming messages are not shared, while supporting customizable quota ratios (default ratio is 1:1).



Throttling Behavior Description

TDMQ for RocketMQ implements a quick failure (Fail-Fast) throttling policy, where **when client request rate reaches the limit, the server immediately responds with an error**. Normally, online services are sensitive to response time. Quick failure enables clients to detect throttling events and intervene promptly, preventing prolonged end-to-end message duration in business operations.
For a base cluster with a 1000 TPS specification, assume the send and receive TPS quota ratio is 1:1. The related throttling trigger behavior description is as follows:

Description	Message Throttling for Sending	Message Throttling for Consumption
Throttling Trigger Scenario	All connections to the cluster can send a total of 500 converted messages per second. Requests will	All consumer clients connected to the cluster can consume a total of 500 converted messages per

	fail if the sending rate reaches the limit.	second. Message consumption delay will increase if the consumption rate reaches the limit.
Log Keyword When Trigger Throttling	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.
Retry Mechanism When Trigger Throttling	SDKs handle differences across protocols: • 5.x SDK retries sending with exponential backoff policy. The maximum number of retries can be customized during Producer initialization, with a default value of 2. Failed requests that reach the maximum retry limit will throw an exception. • 4.x SDK directly throws exception and will not retry.	The SDK pull message thread automatically performs backoff retry.

Client practical tutorial

Planning a Cluster

The purpose of cluster rate limiting in TDMQ for RocketMQ is to ensure service stability and reliability, prevent issues such as increased service response time and decreased request success rate in case of high load, thereby avoiding business impact. Therefore, when integrating TDMQ for RocketMQ, reasonably planning the cluster is very important. We recommend that you:

- Assess business TPS fully based on current scale and future trend prediction. If business traffic fluctuates, use peak TPS as the benchmark. In addition, we recommend reserving some TPS quota (such as 30%) to handle possible traffic spikes.
- For businesses with high stability requirements, we recommend using multiple RocketMQ clusters to enhance isolation. For example, isolate core links (such as transaction systems) from non-core links (such as log systems), and separate production environments from development and testing environments.

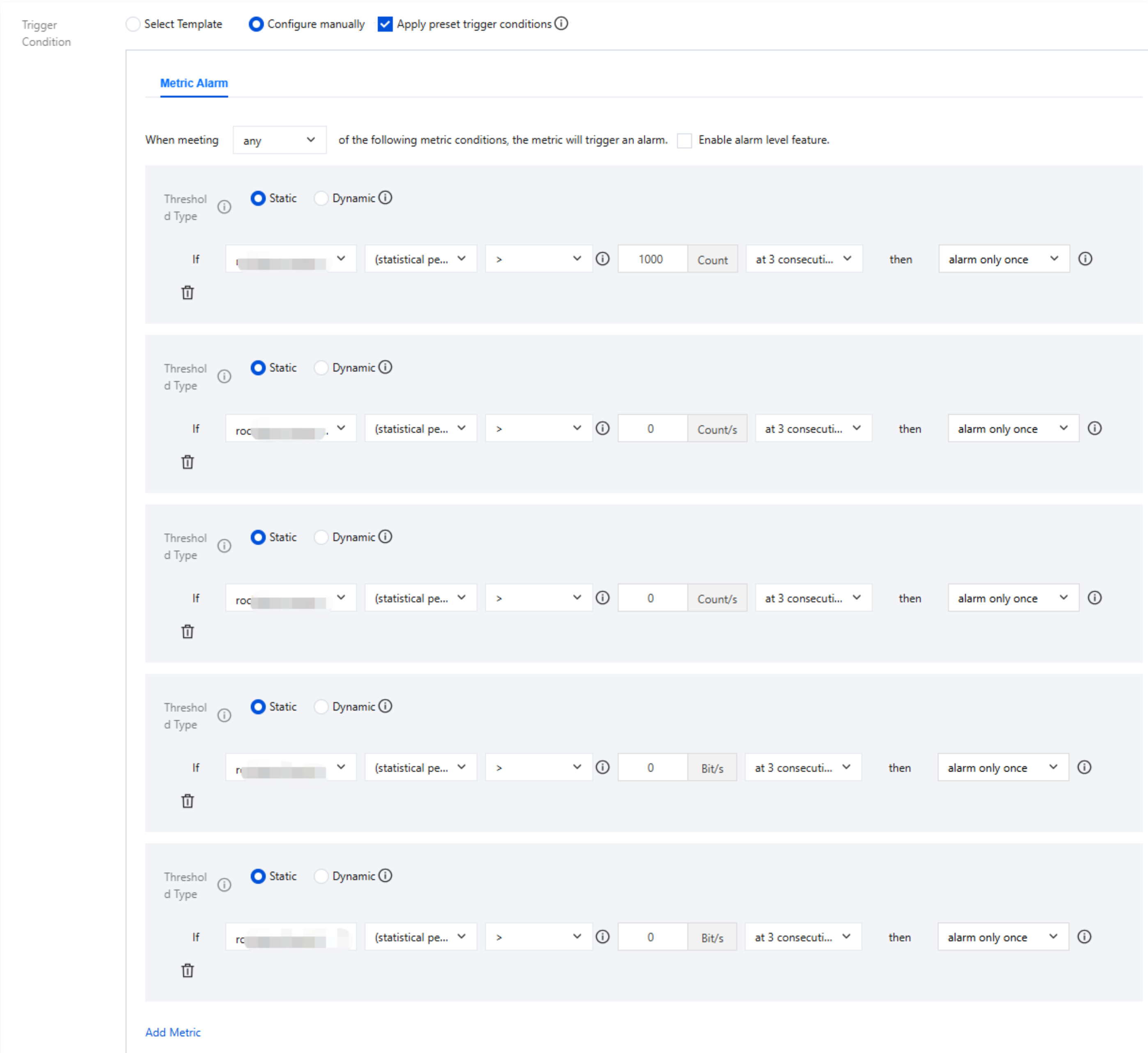
Monitoring Workload

You can use the monitoring and alerting capabilities of the TDMQ for RocketMQ console to observe cluster load in real-time, discover TPS water level risks in advance, and perform timely upgrades to ensure adequate resources and avoid triggering throttling. For specific operations, see Monitoring Alarm. Alarm policy recommendations are as follows:

- Send and consumption TPS water level exceeding 70% of capacity triggers alarm to remind perform upgrade estimation.
- Trigger alarm when traffic throttling occurs, warning that business sending may experience failures.

Example

For a base cluster with 1000 TPS specification, the TPS alarm policy is as follows:



Handling Code Exceptions

When sending a message via the RocketMQ SDK, the business code must capture exceptions including traffic throttling errors and save necessary context information for human intervention to resume business. Retry mechanisms vary across SDKs of different protocols. Sample code for related handling:

- 4.x SDK does not automatically retry traffic throttling errors, so business code must catch exception and handle it. Sample code:

```
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";

        // Instantiate the message producer Producer
        DefaultMQProducer producer = new DefaultMQProducer(
            producerGroupName, // Producer group name
            new AclClientRPCHook(new SessionCredentials(accessKey,
secretKey)) // ACL permission, accessKey and secretKey can be obtained
```

```
on the console cluster permission page
        );

        // Set Nameserver address, can be obtained on the console
cluster basic information page
        producer.setNamesrvAddr(nameserver);

        Start the Producer instance
        producer.start();

        // Create message instance, set topic and message content
        Message message = new Message(topicName,
"Your_Biz_Body".getBytes(StandardCharsets.UTF_8));

        // Maximum send count, set based on business needs
        final int maxAttempts = 3;
        // retry interval, set based on business needs
        final int retryIntervalMillis = 200;

        // Send the message.
        int attempt = 0;
        do {
            try {
                SendResult sendResult = producer.send(message);
                log.info("Send message successfully, {}",
sendResult);
                break;
            } catch (Throwable t) {
                attempt++;
                if (attempt >= maxAttempts) {
                    // Reached the maximum number of times
                    log.warn("Failed to send message finally, run out
of attempt times, attempt={}, maxAttempts={}, msgId={}",
                        attempt, maxAttempts,
MessageClientIDSetter.getUniqID(message), t);
                    // Log messages that failed to send (or record in
other services, such as database etc.)
                    log.warn(message.toString());
                    break;
                }
```

```
                int waitMillis;
                if (t instanceof MQBrokerException &&
((MQBrokerException) t).getResponseCode() == 215 /* FLOW_CONTROL */)
{
                    // Rate limit exceeded, use backoff retry
                    waitMillis = (int) Math.pow(2, attempt - 1) *
retryIntervalMillis; // Retry interval: 200ms, 400ms, ......
                } else {
                    other exceptions
                    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 automatically retries sending exceptions. The business code can customize the maximum number of retries. Sample code:

```
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 permission, accessKey and secretKey can be obtained on
the console cluster permission page
        SessionCredentialsProvider sessionCredentialsProvider = new
StaticSessionCredentialsProvider(accessKey, secretKey);

        ClientConfiguration clientConfiguration =
ClientConfiguration.newBuilder()
            .setEndpoints(nameserver) // Set NameServer address, can
be obtained on the console cluster basic information page
            .setCredentialProvider(sessionCredentialsProvider)
            .build();

        Start the Producer instance
        ClientServiceProvider provider =
ClientServiceProvider.loadService();
        Producer producer = provider.newProducerBuilder()
            .setClientConfiguration(clientConfiguration)
            .setTopics(topicName) // Pre-declare the topic for
message sending, recommend setting
            .setMaxAttempts(3) // Maximum send count, set based on
business needs
            .build();
```



```
        // Create message instance, set topic and message content
        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);
            // Log messages that failed to send (or record in other
services, such as databases)
            log.warn(message.toString());
        }

        producer.close();
    }
}
```

FAQs

Trigger Throttling and Message Loss

When message sending triggers throttling, the server does not store the message. The client must catch the exception and perform downgrade processing. If consumption triggers throttling, a delay in message consumption occurs, but successfully sent messages are not lost.

Why Is TPS on Monitoring Page Higher Than Message Count

TPS converts message count. If business operations use advanced messages (sequential, transactional, delayed, etc.) or the message body is relatively large, one business message will be counted as multiple converted messages. For specific conversion logic, see Billable Items. In addition, the message count metric calculates the per-second mean within one minute, while the TPS metric calculates the per-second peak value within one minute.

Cluster Occasionally Experiences Brief Rate Limited Consumption, Is There Any Impact

Generally no impact. During operations such as client restart, server restart, or console expansion of topic queues, temporary consumption throttling may be triggered due to accumulation in the consumption group. It usually recovers quickly after stabilization.

How to Determine Whether Cluster Traffic Throttling Is Displayed

In addition to identifying exceptions thrown by the SDK Sending Interface or logging info from the SDK, you can also check the Tencent Cloud RocketMQ Console > Monitoring Dashboard for **rate-limited production TPS (Count/s)** and **rate-limited consumption TPS (Count/s)**.