TDMQ for RocketMQ

Practical Tutorial

Product Documentation

Contents

Practical Tutorial
Naming Conventions for Common Concepts of RocketMQ

Last updated: 2025-07-15 18:35:42

This document introduces the naming conventions and usage norms for common concepts in RocketMQ.

Naming Conventions

topic

- It must not be empty, and can only contain letters, digits, (-), and (_), 3 to 64 characters.
- Suggested format: String.format (tp_%s_%s, system name, and business name)
- For example: tp_order_check

tag

- Can be empty, as long as it is a string, used for secondary message filtering under a topic.
- Suggested format: String.format (tag_%s, business action or category)
- For example: tag+business action, e.g., the tag for order creation is tag_create; the tag for order closure is tag_close

keys

- Can be empty, recommended setting, as long as it is a string or an array of strings, used for querying messages or message traces in the console.
- Suggested format: String.format (%s and unique business ID)
- For example: order ID, transaction ID or serial number, TraceID, and other unique IDs

producer group

- It must not be empty, 3 to 64 characters, and can only contain letters, digits, (-), and (_).
- Suggested format: String.format (pg_%s_%s, system name, and business name)
- For example: pg_transfer_check

consumer group

- It must not be empty, 3 to 64 characters, and can only contain letters, digits, (-), and (_).
- Suggested format: String.format (cg_%s_%s, system name, and business name)
- For example: cg_transfer_check

role

- It must not be empty, supports only digits, upper and lower case letters, and delimiters (_, –), and cannot exceed 32 characters.
- Markers for different business read and write permissions, suggested format: business name + send/consume
- For example: role_order_send, role_order_consume, and role_order_all

clientId

clientId represents a client instance ID, which must be unique across different clients. clientId cannot be directly set on the client. instanceName is an optional component of clientId, which can modify the clientId by adjusting instanceName.

Classification	Generating Strategy
Producer	${currentIP}@${instanceName}
Cluster Consumer	${currentIP}@${instanceName}
Broadcast Consumer	${currentIP}@${instanceName}

```
public String buildMQClientId() {
        StringBuilder sb = new StringBuilder();
        sb.append(this.getClientIP());

        sb.append("@");
        sb.append(this.getInstanceName());
        if (!UtilAll.isBlank(this.unitName)) {
            sb.append("@");
            sb.append(this.unitName);
        }

        if (enableStreamRequestType) {
            sb.append("@");
            sb.append(RequestType.STREAM);
        }

        return sb.toString();
    }
```

instanceName

Instance names, under default scenarios, do not require special settings by the user. The system will generate a unique value randomly through the following code by default.

```
  public void changeInstanceNameToPID() {
        if (this.instanceName.equals("DEFAULT")) {
            this.instanceName = UtilAll.getPid() + "#" +
System.nanoTime();
        }
    }
```

Broadcast consumers, upon every startup, need to keep the instance name constant to read the local progress file on the client. It is required to set the instanceName explicitly, and broadcast consumption must ensure the current Client IP remains unchanged before and after startup. If deploying in a container, it is necessary to set a fixed pod IP for scheduling; otherwise, broadcast messages during the restart period will be missed.
Suggested format: String.format (instance-%s-%s, system name, and business name).

Usage Guidelines

Producer

- [Mandatory] A **domain** service can only have one topic.
- [Mandatory] When sending messages, a **domain** service must set the tag according to the business action.
- [Mandatory] The keys must be set when the producer sends a message.
- [Mandatory] Logs must be printed whether a message sending succeeded or failed, with both the SendResult and key fields required to be printed.
- [Recommended] After a message-sending failure, it is recommended to store the message in a database, and then use a timer-like thread for periodic retries to ensure the message can be delivered to the broker.
- [Recommended] For business scenarios where reliability is not a high requirement, the oneway messages can be used.
- [Mandatory] When creating a new producer, a producer group must be specified.

Consumer

- [Mandatory] When creating a new consumer, a producer group must be specified.
- [Mandatory] Message consumers cannot avoid duplicate messages, so the business service needs to

ensure idempotent message consumption.

- [Recommended] To improve consumption parallelism, you can start multiple Consumer instances under the same ConsumerGroup or increase the parallel consumption capacity of a single Consumer by modifying ConsumeThreadMin and ConsumeThreadMax.
- [Recommended] To increase business throughput, you can batch the consume messages by setting the consumer's consumeMessageBatchMaxSize.
- [Recommended] In the event of message accumulation, if the business does not have high data requirements, you can choose to discard unimportant messages.
- [Recommended] If the message volume is small, it is recommended to print messages and consumption time, etc. at the entry point of consumption to facilitate subsequent troubleshooting.

RocketMQ Client Practice

Last updated：2025-07-15 18:35:42

Sender Practice

Choosing between Topic and Tags

A Topic is a message Topic used to distinguish business messages of the same type. A Tag is a special attribute or sub-type of a message, which can be accessed by the downstream for efficient consumption filtering on the Broker side using a filter expression.
Choose between Topic and Tag based on the following factors:

1. Message type consistency: Currently, regular messages, scheduled messages, transaction messages, and sequential messages cannot be mixed. If they are different, you must select different Topics.
2. Business scenario consistency: For example, order messages and payment messages, it is recommended to choose different Topics. If order messages require region filtering for different provinces downstream, you can mark the city as a Tag.
3. Message volume consistency: Since RocketMQ consumption progress is maintained by Topic, if there is a very large difference in scale between different types earlier, small-scale Tags need to filter out a massive amount of useless messages before consumption, resulting in invalid resource consumption and increased latency.

Comprehensively, we recommend using Topic to distinguish messages with no business association and Tag to identify key attributes within the same Topic, enabling efficient filtering of some messages for different downstream consumers.

Keys Usage

It is recommended to set the unique business identifier of each message at the business level in the key field. The server will create an index (hash index) for each message. Users can query this message content as well as the message trace by Topic and Key to troubleshoot problems.

Choosing Appropriate Sending Method

RocketMQ supports three sending methods:

- Synchronous sending: suitable for scenarios with relatively high reliability requirements, such as payment messages and SMS notifications.
- Async sending: applicable to time-sensitive business scenarios where the sender cannot tolerate long waits for the Broker's response.
- One-way sending: suitable for scenarios where the sending result is not a major concern, such as log sending.

send(msg) synchronously sends and only returns the result if sent successfully, but it blocks synchronously, increasing the sending duration. If there are performance requirements, you can use the asynchronous method: send(msg, callback), and check the sending result in the callback. For specific applications that do not require confirmation of whether the message was sent successfully, such as log collection category applications, directly use the sendOneWay method to send messages.

Sending Retry Policy

For applications where messages cannot be lost, there must be a message resend mechanism. The Producer's send method itself supports internal retry.

1. Retry up to 2 times (2 for synchronous sending, 0 for async sending).
2. If sending fails, switch to the next Broker. The total duration of this method is no more than the set sendMsgTimeout value, default 3s.
3. If sending a message to the broker causes a timeout exception, it will not be retried.

For key business operations, if message-sending failure occurs, it is advisable to store the message in a db and use a timer-like thread for periodic retries to resend the message to downstream consumers after RocketMQ service resumption.

Printing Message Sending Log

Regardless of whether the message is sent successfully or fails, it is recommended to log attributes such as sendresult, msgid, key, and tag in logs for troubleshooting.

Consumer Practical Tutorial

Using Consumption Groups to Isolate Downstream Businesses

Different businesses can use consumption groups to independently consume the same topic, and each consumer maintains its own consumption offset (offsets), ensuring every consumer in the same group subscribes to the same topic and uses the same filtering rules.

Idempotent Processing

RocketMQ ensures at-least-once consumption in principle and cannot avoid message duplication. For example, network jitter during sending can trigger retries, consumption exceptions during delivery can also cause retries, and load balancing during consumer restarts may lead to duplicate messages. Therefore, if your business is super sensitive to duplicate consumption, you must perform deduplication at the business level. You can use distributed cache or relational database to deduplicate.
For idempotent processing, first determine the unique key of the message. It is not recommended to depend on msgid. Instead, use the unique business identification field set in the keys field, such as order Id. Before consumption, check whether the unique key exists in the distributed cache or relational database. If it does not exist, insert it and consume. Otherwise, skip or perform further deduplication according to business logic.

Retry on Consumption Failure

Concurrent consumption: Failed concurrent consumption uses a backoff retry mechanism. Messages that fail to be consumed are sent back to the system's delayed queue. Each time consumption fails, the delayLevel increases by 1. The default maximum number of retries is 16. Messages exceeding the maximum number will enter the dead letter queue. The time intervals for 16 retries correspond to delayed message levels 3 to 18, ranging from 10s to 2h. For details, see Message Retry.
Sequential consumption: To ensure sequence, sequential consumption does not send messages back to the server. It uses a local continuous retry mechanism with a default retry count of Integer.MAX_VALUE. The maximum number of retries is customizable. Messages exceeding the maximum number will enter the dead letter queue.
The maximum number of retries can also be configured through maxReconsumeTimes.

Enhancing Consumption Parallelism

Most message consumption behaviors are IO-intensive, which may involve database operations or RPC calls during the process. The consumption speed depends on the throughput of backend databases or external systems. Adding consumption parallelism can improve overall consumption throughput.

1. Increase the count of consumption nodes. Under the same ConsumerGroup, raise the degree of parallelism by adding the number of instance nodes (notably, Consumer instances exceeding the subscription queue count are invalid). This can be done through adding machines or starting multiple processes on existing machines.
2. Increase the number of threads for each consumer node by changing the Consumer parameters consumeThreadMin and consumeThreadMax to achieve higher concurrency.
3. Batch processing consumption – If specific business processes support batch processing, it can significantly raise consumption throughput. For example, in order deduction applications, processing one order takes 1s, while processing 10 orders may only take 2s. This way, throughput can be greatly improved. By setting the consumer's consumeMessageBatchMaxSize parameter, which is 1 by default (consuming only one message at a time), you can adjust it to N so each consumption handles up to N messages.
4. Skip non-critical messages. In case of message backlog, if business data requirements are not high, you can skip expired messages through time filtering or drop unimportant messages based on business selection to enhance message processing efficiency.

Subscription Relationship Consistency

A Consumer Group normally represents multiple nodes with consistent Consumer business logic. A consistent subscription relationship means all consumers in the same Consumer Group have identical subscription relationships. Otherwise, the consumption logic may become chaotic, even leading to message loss.

Try to ensure compatibility with subscription relationship changes. It is advisable not to alter the subscribed Topic. Only perform incremental subscriptions for filter expressions. During the process of publishing among different consumer nodes in the same Group, inconsistent filtering rules may occur between nodes. When the Broker filters messages, different consumer nodes may apply different filtering rules. Therefore, once the subscription relationship of the same Group changes, pay attention to its compatibility to avoid missing messages.
Tencent Cloud RocketMQ console provides diagnostics for inconsistent subscription relationships in the consumer detail page, where you can see which nodes have this occurrence.

Printing Consumption Message Log

Regardless of whether the message is sent successfully or fails, it is recommended to first print a log upon receiving the message, including attributes such as msgId, key, and tag, for troubleshooting. After message processing is completed, also print msgid, key, tag, retry count, and consumption result in logs.

Additional Consumption Recommendations

About Consumers and Subscriptions

The first thing to note is that different consumer groups can independently consume some Topics, and each consumer group has its own consumption offset. Please ensure each consumer in the same group maintains information consistency.

About Ordered Messages

The consumer will lock each message queue to ensure they are consumed one by one. Although this will result in performance degradation, it is useful when you care about message order. We do not recommend throwing an exception. You can go back to ConsumeOrderlyStatus.SUSPEND_CURRENT_QUEUE_A_MOMENT as an alternative.

About Concurrent Consumption

As the name suggests, the consumer will consume these messages concurrently. We recommend you use it to obtain good performance. We do not recommend throwing an exception. You can return ConsumeConcurrentlyStatus.RECONSUME_LATER as an alternative.

About Consumption Status

For a concurrent message listener, you can return RECONSUME_LATER to notify the consumer that this message cannot be consumed now and hope to reconsume it later. Then, you can continue to consume other messages. For an ordered message listener, since you care about its order, the message cannot be skipped. However, you can return SUSPEND_CURRENT_QUEUE_A_MOMENT to tell the consumer to wait a moment.

About Blocking

It is not recommended to block the listener because it may block the thread pool and eventually terminate the consumption process.

About Consumer Offset

When creating a new consumer group, you need to determine whether to CONSUME historical messages already present in the Broker. CONSUME_FROM_LAST_OFFSET will ignore historical messages and CONSUME any messages generated afterward. CONSUME_FROM_FIRST_OFFSET will CONSUME each message present in the Broker. You can also use CONSUME_FROM_TIMESTAMP to CONSUME messages generated after the specified TIMESTAMP.

Troubleshoot Practice

SDK Log

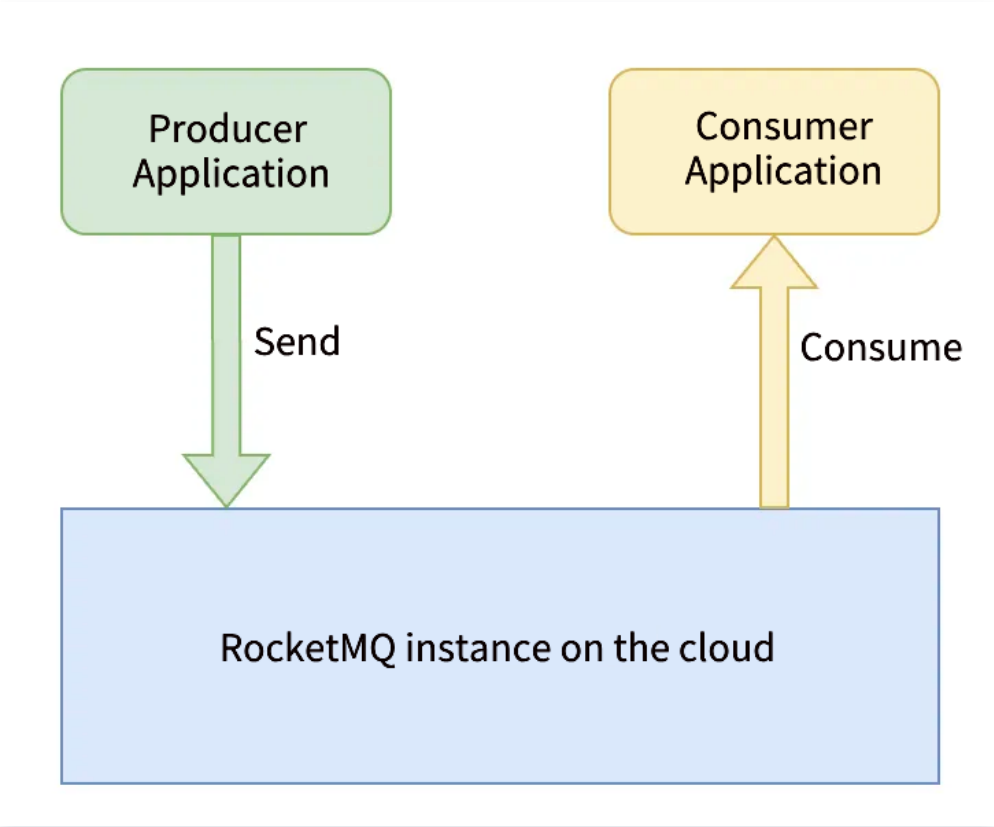
The communication protocol between RocketMQ client and server is complex. For example, the allocation of consumption queues and topic addressing are determined by the client SDK implementation. Important information about these processes is saved in SDK logs. Therefore, when production or consumption issues occur, SDK logs are one of the most important means to troubleshoot problems. Be sure to save these logs. Normally, SDK logs are not printed in the same file as business logs. We provide the default path for commonly used SDK logs as follows:

SDK Language	SDK Protocol	Path
Java	remoting	~/logs/rocketmqlogs/rocketmq_client.log
Java	grpc	~/logs/rocketmq/rocketmq_client.log
Go	remoting	/tmp/rocketmq-client.log
Go	grpc	~/logs/rocketmqlogs/rocketmq_client_go.log

RocketMQ Performance Stress Testing and Capacity Assessment

Last updated：2025-07-15 18:35:42

Performance testing and capacity assessment are basic methods to ensure overall system stability. However, performing an effective stress test is not a simple task. We recommend the following stress testing approach. Before conducting the test, first clarify the purpose of this stress test based on the business scenario. Then determine the specific test plan, including test objects, load testing scenario simulation, tool selection, and key metrics to monitor. Finally, analyze whether the test meets expectations and proceed to purchase suitable specifications.



Clarifying Stress Test Targets

As shown above, for RocketMQ business scenarios, the focus is generally on stress testing message sending or message consumption.
If stress testing message sending, mainly focus on sending rate, duration, and success rate, as well as application performance when traffic throttling is triggered at peak value.
If stress testing message consumption, mainly focus on consumption rate, duration, success rate, retry policy after failure, and the impact of accumulated message delay on business.

Analyzing Stress Test Objects

In the scenario of stress testing message sending, the main test object is the RocketMQ instance. Focus on the sending duration and success rate of the RocketMQ instance. Taking the RocketMQ 5.x instance as an example, distributed rate limiting is enabled by default to prevent overwhelming the cluster with excessive traffic. Therefore, pay attention to the impact of throttling on business.
In the scenario of stress testing message consumption, the main test object is the downstream consumer application. Focus on the message consumption capability of the business downstream. Key indicators include consumption processing time, concurrent thread count, whether consumption times out, whether retries occur due to exceptions, and whether a message backlog is generated.

Simulating Stress Test Scenarios

For the scenario of sending messages, there are usually two methods. The first method can use the built-in stress test script in Apache RocketMQ open-source code to generate performance testing traffic. The second method can also simulate business traffic by using the producer application with business logic code for end-to-end stress testing.
Normally, we recommend first using the built-in stress test script in the open-source code to quickly perform stress tests and get some benchmark indicators of the RocketMQ instance. This helps build confidence and ensures the RocketMQ instance itself meets standards. Then, combine it with the business model, simulate performance testing task traffic, set reasonable concurrency, and ensure the final stress test results meet business needs.
For the scenario of consuming messages, there are usually two methods. The first method can use the built-in stress test script in Apache RocketMQ open-source code to subscribe to the test topic and immediately acknowledge messages upon receipt, confirming the RocketMQ instance provides sufficient consumption capacity. The second method also involves end-to-end stress testing, where the upstream sender sends messages compliant with the consumer's business code requirements, ensuring the consumption logic is covered by testing, and even further propagating the stress test traffic downstream.

Analyzing Performance Testing Metrics

- Send metrics

  Focus on sending rate, duration, and success rate, whether throttling is triggered, its impact on business, or the retry policy.
- Consumption metrics

  Focus on consumption rate, business duration, delay in message consumption, and success rate, as well as the impact of message backlog and delay on business.

Common Issue Analysis

How to Troubleshoot Low Sending Rate

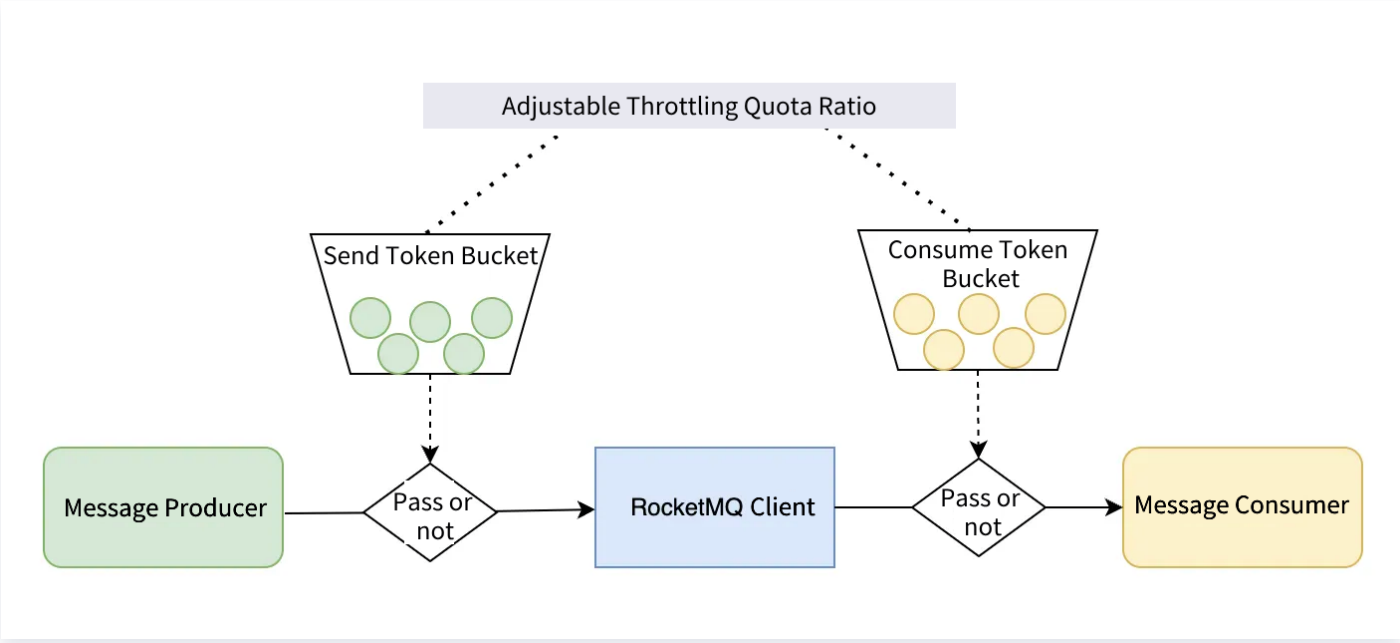
The core factors that determine the sending rate are two: sending duration and degree of concurrency. If the average sending duration is 5ms and the degree of concurrency is 1, the sending rate is 200 TPS. Therefore,

if the stress test target is not met, first confirm the sending duration—for example, whether the network uses a public network or goes through a proxy, resulting in relatively high sending duration. If the sending duration behaves as expected, focus on troubleshooting whether the degree of concurrency is met, whether the sender's parallel number of threads are sufficient, whether the sender's node workload is normal, or whether higher-level factors such as locks influence performance.

How to Efficiently Simulate Traffic Volume for Downstream Business Stress Testing

To achieve good stress test results, the test traffic needs to closely resemble actual business traffic. To simulate traffic, in addition to full-link stress testing, you can reset the consumer offset to replay historical messages, efficiently generating traffic for downstream businesses. This eliminates the need for upstream services to repeatedly send traffic.

How to Analyze the Causes for Trigger Throttling



Since RocketMQ 5.x instances have traffic throttling enabled by default, if throttling is triggered during stress testing, focus on the following reasons:

1. In scenarios with "micro bursts," for example, our monitoring has minute granularity, and all traffic may be concentrated in the first second. In fact, our rate-limiting token window updates every 10 seconds, which can cause minute-level monitoring to show no throttling threshold exceeded, while the 10-second granularity is actually rate-limited.
2. The message body is too large. Because throttling converts messages at 4KB per message, for example, a 100KB message will be converted into 25 messages for rate limiting. Therefore, the throttling value and the number of messages produced are not one-to-one.

3. The traffic throttling ratio is inappropriate. We provide the option to adjust the sending and consumption throttling quota ratio, which defaults to 5:5 and can be adjusted to a maximum of 2:8 or 8:2. Therefore, if throttling is triggered, check whether the configured throttling quota ratio is reasonable.

Access over HTTP
SDK for Java
Sending and Receiving General Messages

Last updated：2025-07-15 18:35:42

TDMQ for RocketMQ is fully compatible with the community edition HTTP SDK. If your client used the community edition HTTP SDK, you can switch to TDMQ for RocketMQ without the need for any client-side code modification.

Overview

If you currently use HTTP protocol for message sending and receiving, after introducing the open-source HTTP SDK in your client, TDMQ for RocketMQ supports user access through private network or public network using HTTP protocol.
This document describes how to use HTTP SDK to send and receive messages by using the SDK for Java as an example and helps you better understand the message sending and receiving processes.

Note:
- Currently, transactional message cannot be implemented over HTTP.
- If you use a 4.x cluster, when creating a group, the protocol type (TCP or HTTP, for details, see Group Management) needs to be set. Therefore, the same group (consumption group) does not support TCP and HTTP clients consuming simultaneously.

Prerequisites

- You have created the required resources as instructed in Resource Creation and Preparation.
- You have installed JDK 1.8 or later.
- You have installed Maven 2.5 or later.
- You have imported dependencies through Maven and added SDK dependencies of the corresponding programming language in the pom.xml file.

Retry Mechanism

A fixed retry interval is used in HTTP, which can't be customized currently.

Retry Interval	Maximum Number of Retries
5 minutes	You can customize the maximum number of retries by modifying the

	consumer group configuration. The default is 16.

Note:

- If the client acknowledges a message within the retry interval, the message consumption is successful and will not be retried.
- If the client doesn't acknowledge a message after the retry interval has expired, the message will become visible again, and the client will consume it again.
- The message handle consumed each time is only valid within the retry interval, and become invalid after that time period.

Directions

Step 1. Install the Java dependent library

Add dependency on the community edition HTTP SDK in the Java project.

Step 2. Produce messages

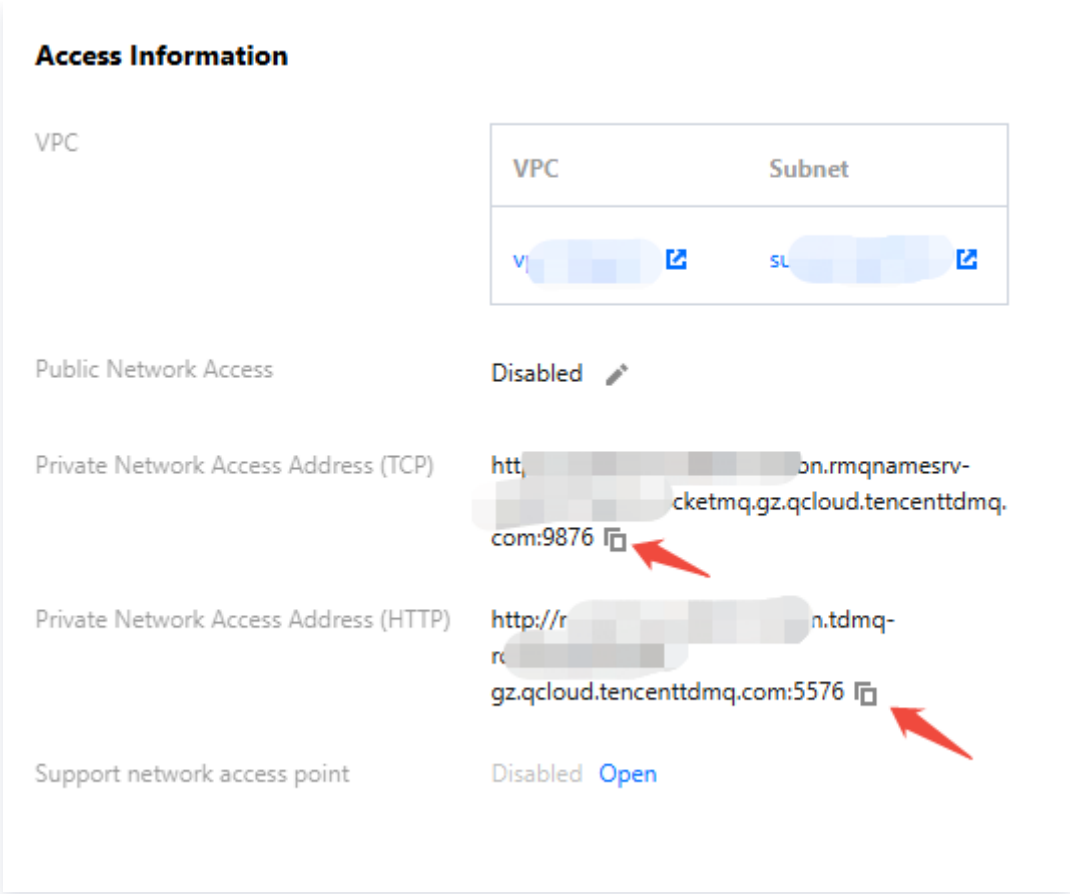
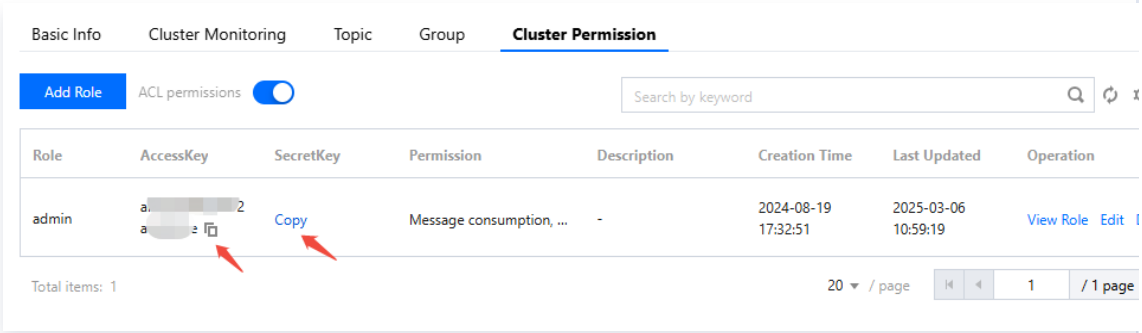
Creating a message producer

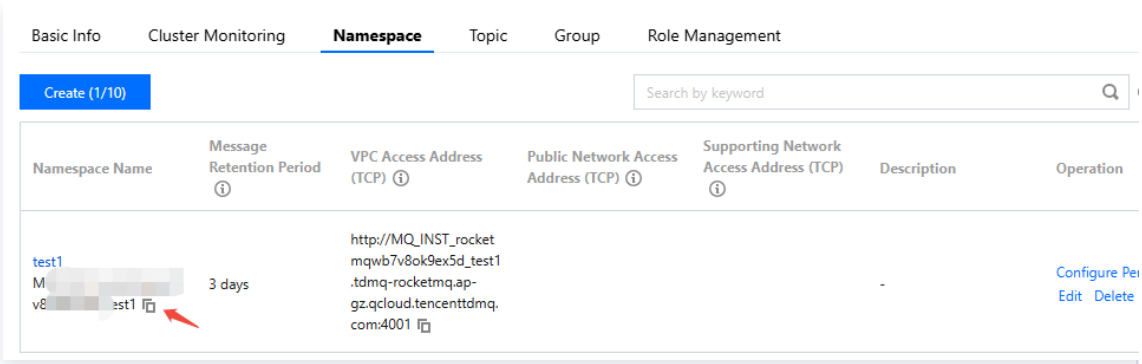
```
// Get the client
MQClient mqClient = new MQClient(endpoint, accessKey, secretKey);

// Get the topic producer
MQProducer producer = mqClient.getProducer(namespace, topicName);
```

Note:

The following parameters require logging in to the TDMQ RocketMQ Console to obtain.

Parameter	Description
endpoint	Get the cluster access address from the access information module on the console cluster basic information page.

	
accessKey	Role key, copied from the AccessKey column on the Cluster Permission page in the console. 
secretKey	Role name, copied from the SecretKey column on the Cluster Permission page in the console.
namespace	Copy the namespace name from the namespace page on the console. If you use a 4.x generic cluster or 5.x cluster, fill in the Cluster ID here.

	
topicName	Copy the topic name from the Topic page in the console

Sending a message

```
try {
    for (int i = 0; i < 10; i++) {
        TopicMessage pubMsg;
        pubMsg = new TopicMessage(
            ("Hello RocketMQ " + i).getBytes(),
            "TAG"
        );
        TopicMessage pubResultMsg = producer.publishMessage(pubMsg);
        System.out.println("Send mq message success. MsgId is: " +
pubResultMsg.getMessageId());
    }
} catch (Throwable e) {
    System.out.println("Send mq message failed.");
    e.printStackTrace();
}
```

Parameter	Description
TAG	Set the message tag.

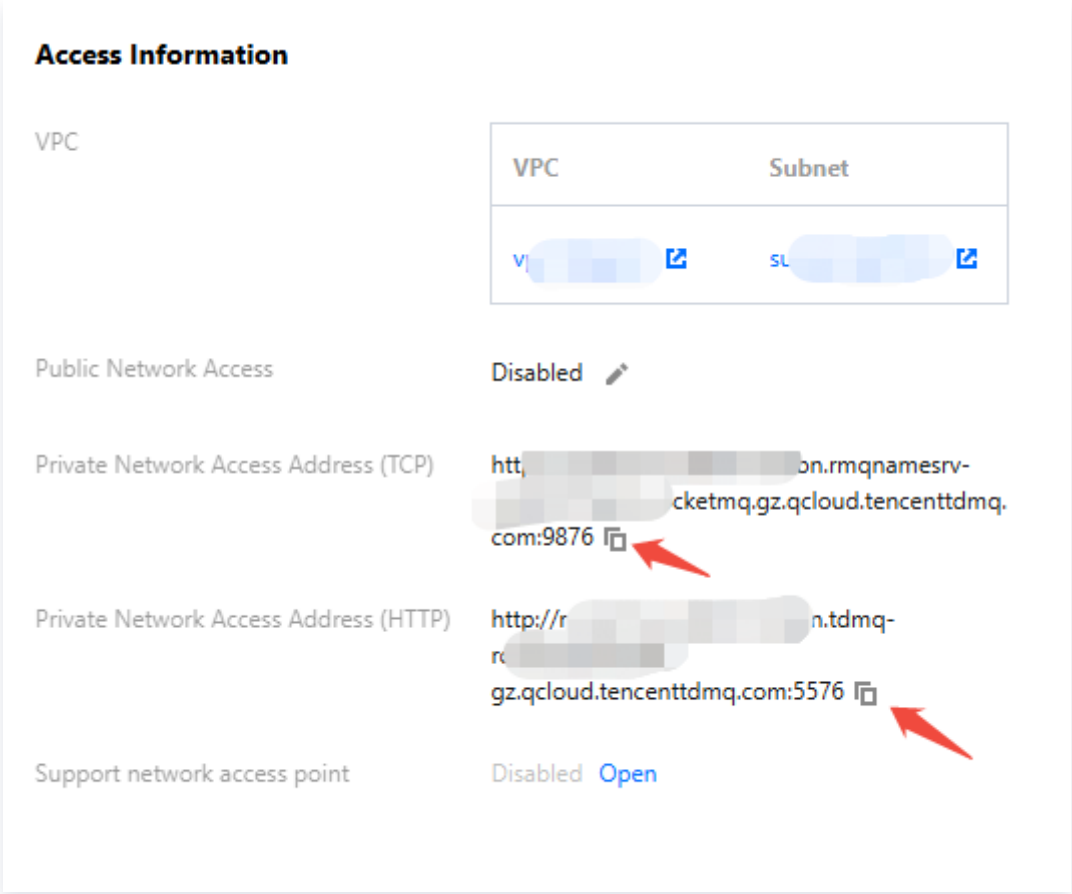
Step 3. Consume messages

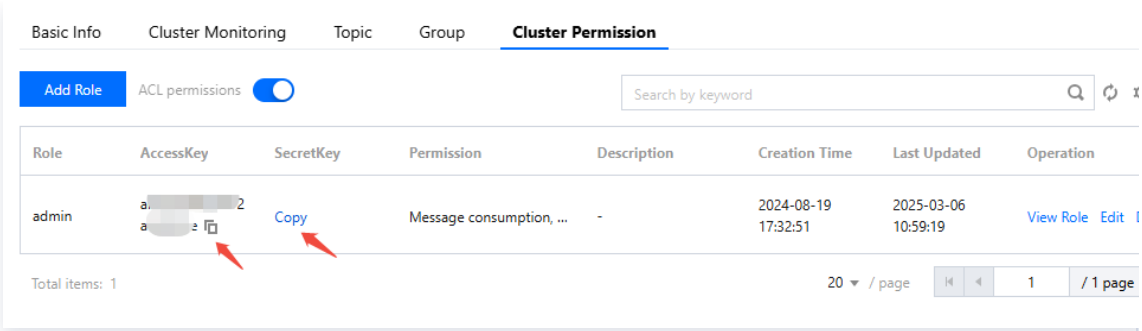
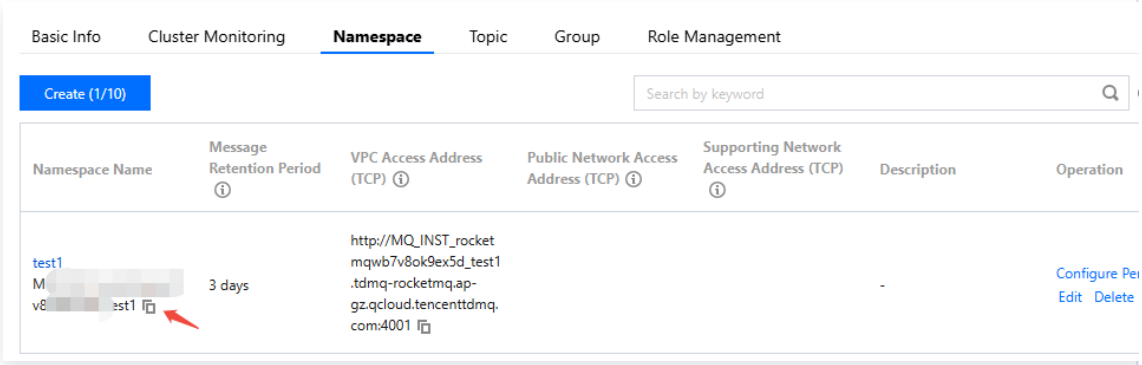
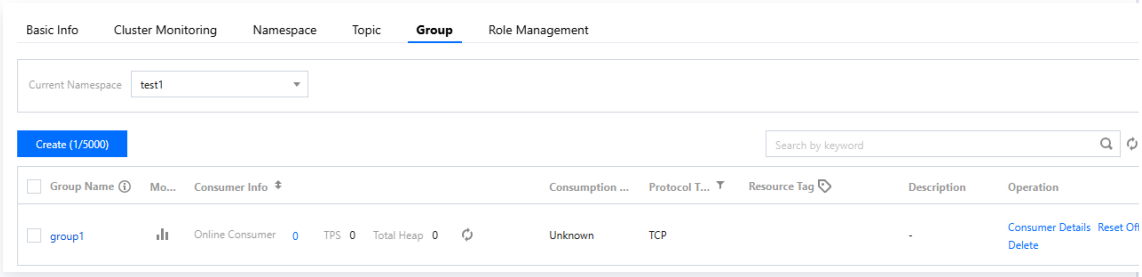
Creating a consumer

```
    // Get the client
    MQClient mqClient = new MQClient(endpoint, accessKey, secretKey);
```

```
    // Get the topic consumer
    MQConsumer consumer = mqClient.getConsumer(namespace, topicName,
groupName, "TAG");
```

Note:
The following parameters require logging in to the TDMQ RocketMQ Console to obtain.

Parameter	Description
endpoint	Get the cluster access address from the access information module on the console cluster basic information page. 
accessKey	Role key, copied from the AccessKey column on the Cluster Permission page in the console.

	
secretKey	Role name, copied from the SecretKey column on the Cluster Permission page in the console.
namespace	Copy the namespace name from the namespace page on the console. If you use a 4.x generic cluster or 5.x cluster, fill in the Cluster ID here. 
topicName	Copy the topic name from the Topic page in the console.
groupName	Copy the consumer group name from the Group Management page in the console. 

Subscribing to messages

```
do {
    List<Message> messages = null;

    try {
        // long polling of consumption messages
```

```
        // Long polling means that if the topic has no messages, the
request will wait on the server for 3 seconds. If there is a message
that can be consumed within 3 seconds, it will return immediately.
        messages = consumer.consumeMessage(
                Integer.parseInt(batchSize),
                Integer.parseInt(waitSeconds)
        );
    } catch (Throwable e) {
        e.printStackTrace();
    }
    if (messages == null || messages.isEmpty()) {
        System.out.println(Thread.currentThread().getName() + ": no new
message, continue!");
        continue;
    }

    for (Message message : messages) {
        System.out.println("Receive message: " + message);
    }

    {
        List<String> handles = new ArrayList<String>();
        for (Message message : messages) {
            handles.add(message.getReceiptHandle());
        }

        try {
            consumer.ackMessage(handles);
        } catch (Throwable e) {
            if (e instanceof AckMessageException) {
                AckMessageException errors = (AckMessageException) e;
                System.out.println("Ack message fail, requestId is:" +
errors.getRequestId() + ", fail handles:");
                if (errors.getErrorMessages() != null) {
                    for (String errorHandle
:errors.getErrorMessages().keySet()) {
                        System.out.println("Handle:" + errorHandle + ",
ErrorCode:" + errors.getErrorMessages().get(errorHandle).getErrorCode()
                                + ", ErrorMsg:" +
errors.getErrorMessages().get(errorHandle).getErrorMessage());
```

```
                    }
                }
                continue;
            }
            e.printStackTrace();
        }
    }
} while (true);
```

Parameter	Description
batchSize	The number of messages pulled at a time. Maximum value: 16.
waitSeconds	The polling waiting time for a message pull. Maximum value: 30 seconds.

Sending and Receiving Sequential Messages

Last updated：2025-07-15 18:35:42

TDMQ for RocketMQ is fully compatible with the community edition HTTP SDK. If your client used the community edition HTTP SDK, you can switch to TDMQ for RocketMQ without the need for any client-side code modification.

Overview

If you currently use HTTP protocol for message sending and receiving, after introducing the open-source HTTP SDK to your client, TDMQ for RocketMQ supports users to integrate through the private network or public network using HTTP protocol.
This document describes how to use HTTP SDK to send and receive messages by using the SDK for Java as an example and helps you better understand the message sending and receiving processes.

Note:

- Currently, transactional message cannot be implemented over HTTP.
- If you use a 4.x cluster, you need to set the protocol type (TCP or HTTP, for details, see Group Management) when creating a group. Therefore, the same group (consumption group) does not support TCP and HTTP client consuming simultaneously.

Prerequisites

- You have created the required resources as instructed in Resource Creation and Preparation.
- You have installed JDK 1.8 or later.
- You have installed Maven 2.5 or later.
- You have imported dependencies through Maven and added SDK dependencies of the corresponding programming language in the pom.xml file.

Retry Mechanism

A fixed retry interval is used in HTTP, which can't be customized currently.

Retry Interval	Maximum Number of Retries
5 minutes	You can customize the maximum number of retries by modifying the consumer group configuration, default 16.

Note:

- If the client acknowledges a message within the retry interval, the message consumption is successful and will not be retried.
- If the client doesn't acknowledge a message after the retry interval has expired, the message will become visible again, and the client will consume it again.
- The message handle consumed each time is only valid within the retry interval, and becomes invalid after that time period.

Directions

Step 1. Install the Java dependent library

Add the community edition HTTP SDK dependency in the Java project.

Step 2. Produce messages

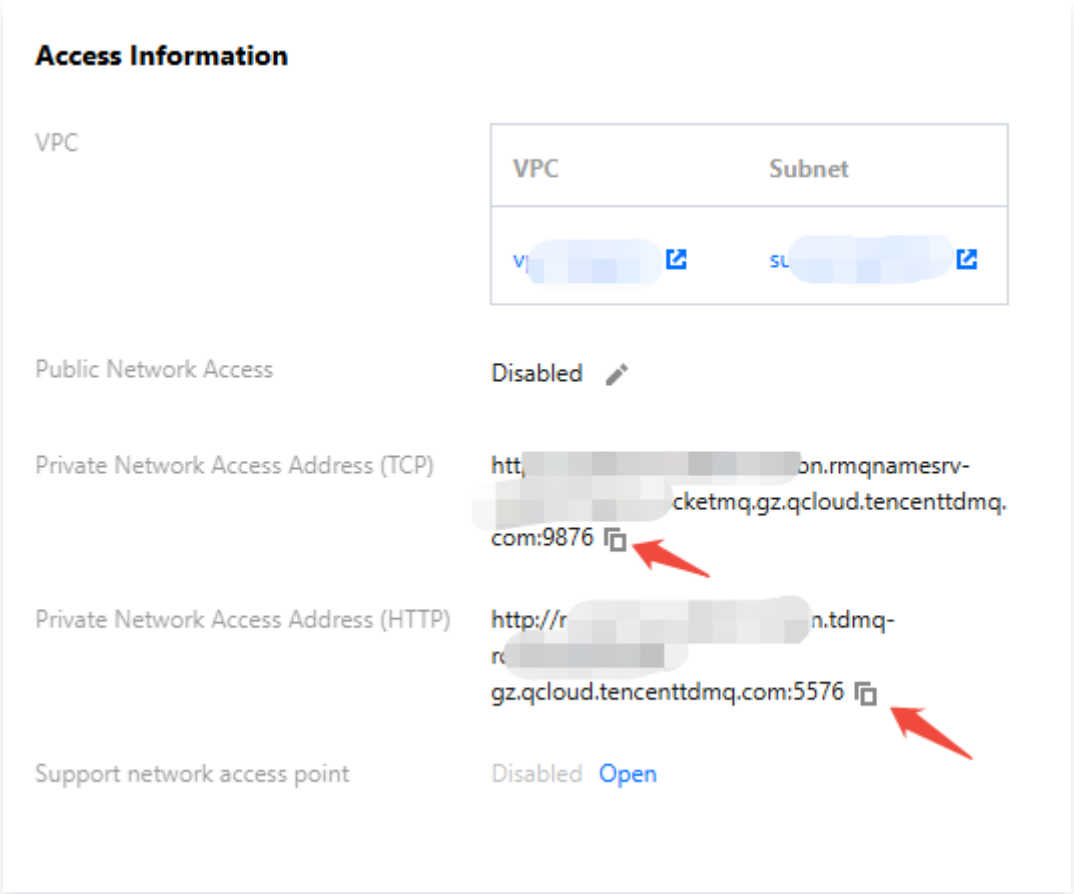
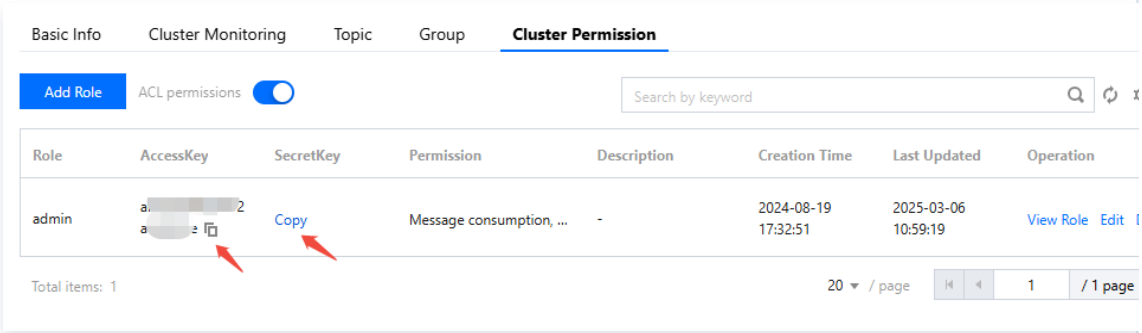
Creating a message producer

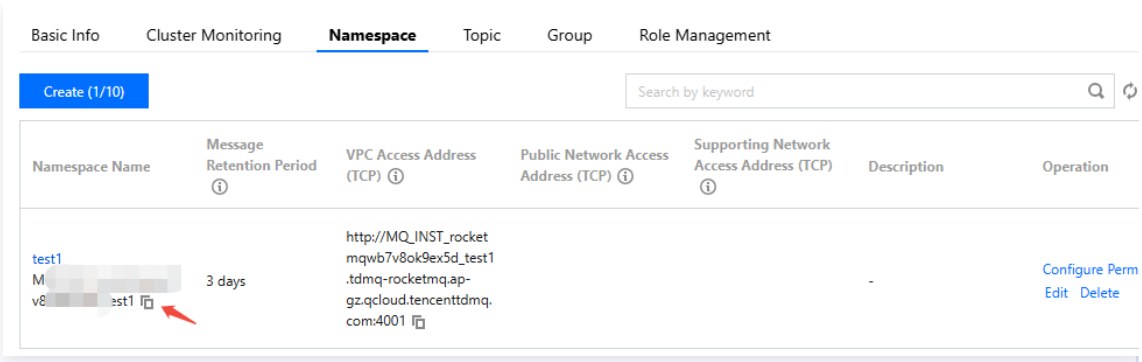
```
// Get the client
MQClient mqClient = new MQClient(endpoint, accessKey, secretKey);

// Get the topic producer
MQProducer producer = mqClient.getProducer(namespace, topicName);
```

Note:

The following parameters require logging in to the TDMQ RocketMQ Console to obtain.

Parameter	Description
endpoint	Get the cluster access address from the access information module on the console cluster basic information page.

	
accessKey	Role key, copied from the AccessKey column on the Cluster Permission page in the console. 
secretKey	Role name, copied from the SecretKey column on the Cluster Permission page in the console.
namespace	Copy the namespace name from the namespace page in the console. If you use a 4.x generic cluster or 5.x cluster, fill in the Cluster ID here.

	
topicName	Copy the topic name from the Topic page in the console.

Sending a message

```
try {
    for (int i = 0; i < 10; i++) {
        TopicMessage pubMsg;
        pubMsg = new TopicMessage(
            ("Hello RocketMQ " + i).getBytes(),
            "TAG"
        );
        // Set the ShardingKey of the partitionally sequential message
        pubMsg.setShardingKey(i % 3);
        TopicMessage pubResultMsg = producer.publishMessage(pubMsg);
        System.out.println("Send mq message success. MsgId is: " +
pubResultMsg.getMessageId());
    }
} catch (Throwable e) {
    System.out.println("Send mq message failed.");
    e.printStackTrace();
}
```

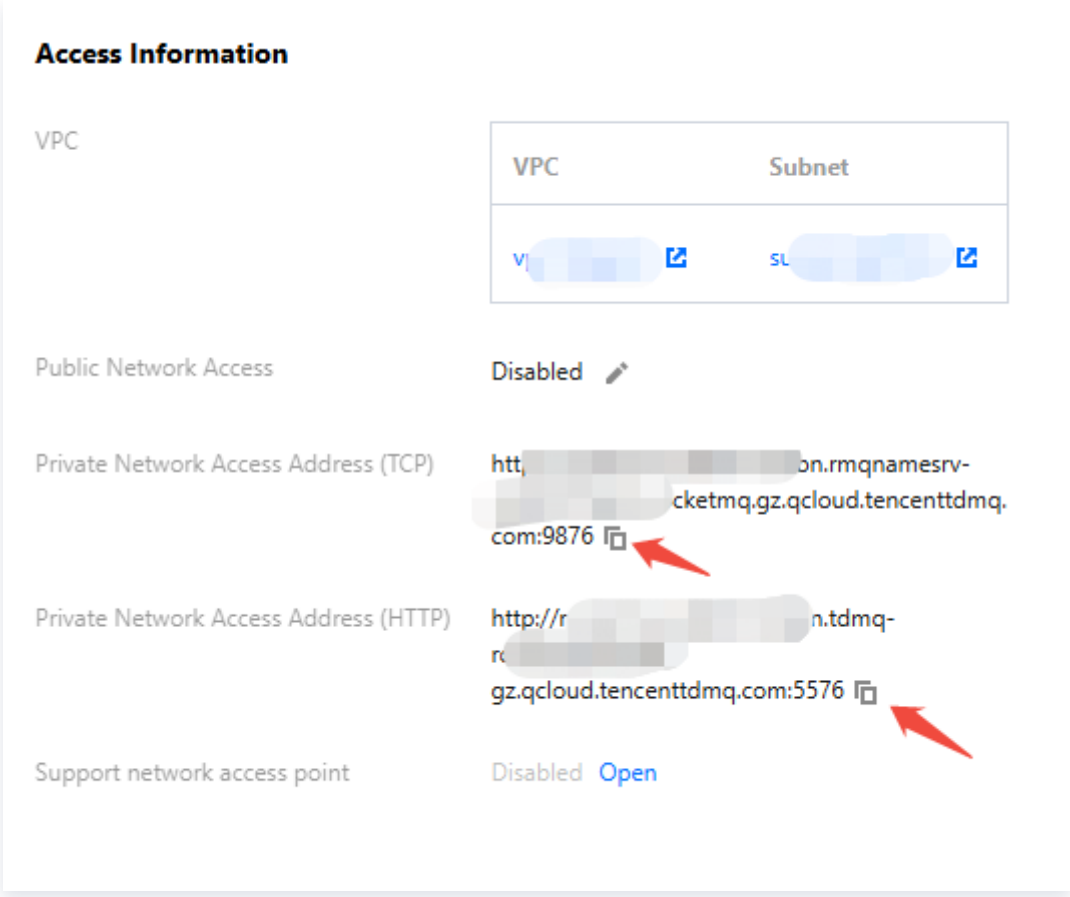
Parameter	Description
TAG	Set the message tag.
ShardingKey	A partition field of sequential messages. Messages with the same ShardingKey will be sent to the same partition.

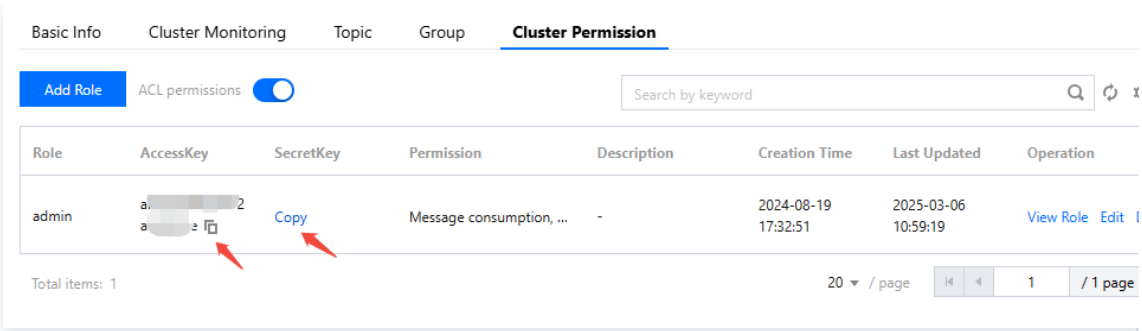
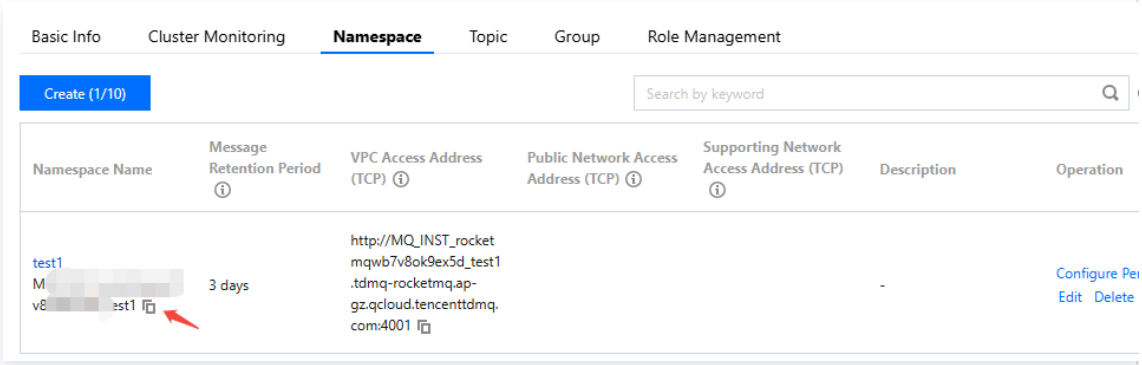
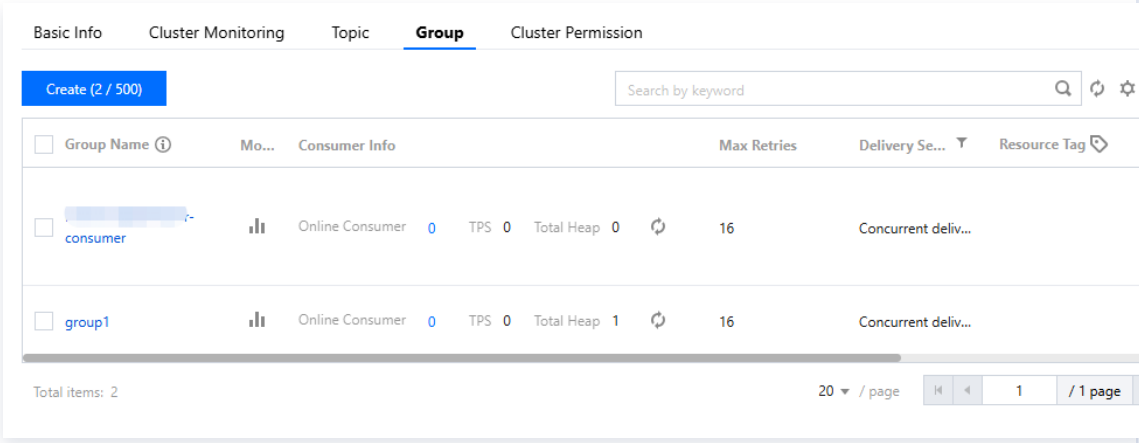
Step 3. Consume messages

Creating a consumer

```
    // Get the client
    MQClient mqClient = new MQClient(endpoint, accessKey, secretKey);

    // Get the topic consumer
    MQProducer consumer = mqClient.getConsumer(namespace, topicName,
groupName, "TAG");
```

Note:
The following parameters require logging in to the TDMQ RocketMQ Console to obtain.

Parameter	Description
endpoint	Get the cluster access address from the access information module on the console cluster basic information page. 
accessKey	Role key, copied from the AccessKey column on the Cluster Permission page in the console.

	
secretKey	Role name, copied from the SecretKey column on the Cluster Permission page in the console.
namespace	Copy the namespace name from the namespace page in the console. If you use a 4.x generic cluster or 5.x cluster, fill in the Cluster ID here. 
topicName	Copy the topic name from the Topic page in the console.
groupName	Copy the consumer group name from the Group management page on the console. 

Subscribing to messages

```
do {
    List<Message> messages = null;
```

```
    try {
        // Long polling consumes messages sequentially. Although the
messages obtained may be from multiple partitions, the messages in a
partition will definitely be sequential.
        // For sequential consumption, as long as a message in a
partition hasn't been acknowledged to be consumed successfully, it will
be consumed next time for this partition.
        // For a partition, the next batch of messages can only be
consumed after all previous messages are acknowledged to be consumed
successfully.
        messages = consumer.consumeMessageOrderly(
                Integer.parseInt(batchSize),
                Integer.parseInt(waitSeconds)
        );
    } catch (Throwable e) {
        e.printStackTrace();
    }
    if (messages == null || messages.isEmpty()) {
        System.out.println(Thread.currentThread().getName() + ": no new
message, continue!");
        continue;
    }

    for (Message message : messages) {
        System.out.println("Receive message: " + message);
    }

    {
        List<String> handles = new ArrayList<String>();
        for (Message message : messages) {
            handles.add(message.getReceiptHandle());
        }

        try {
            consumer.ackMessage(handles);
        } catch (Throwable e) {
            if (e instanceof AckMessageException) {
                AckMessageException errors = (AckMessageException) e;
```

```
                System.out.println("Ack message fail, requestId is:" +
errors.getRequestId() + ", fail handles:");
                if (errors.getErrorMessages() != null) {
                    for (String errorHandle
:errors.getErrorMessages().keySet()) {
                        System.out.println("Handle:" + errorHandle + ",
ErrorCode:" + errors.getErrorMessages().get(errorHandle).getErrorCode()
                                + ", ErrorMsg:" +
errors.getErrorMessages().get(errorHandle).getErrorMessage());
                    }
                }
                continue;
            }
            e.printStackTrace();
        }
    }
} while (true);
```

Parameter	Description
batchSize	The number of messages pulled at a time. Maximum value: 16.
waitSeconds	The polling waiting time for a message pull. Maximum value: 30 seconds.

PHP SDK
Sending and Receiving Normal Messages

Last updated：2025-07-15 18:35:42

TDMQ for RocketMQ is fully compatible with the community edition HTTP SDK. If your client used the community edition HTTP SDK, you can switch to TDMQ for RocketMQ without the need for any client-side code modification.

Scenarios

If you currently use HTTP protocol for message sending and receiving, after introducing the open-source HTTP SDK to your client, TDMQ for RocketMQ supports users to integrate via HTTP protocol through the private network or public network.
In this document, Java SDK call is used as an example to introduce the operation process of sending and receiving messages through the HTTP SDK, which helps you better understand the whole process of sending and receiving messages.

Note:
- Temporary not support transaction message implementation through HTTP protocol.
- If you use a 4.x cluster, you need to set the protocol type (TCP or HTTP, for details, see Group Management) when creating a group. Therefore, the same group (consumption group) does not support TCP and HTTP clients consuming simultaneously.

Prerequisites

- Complete resource creation and preparation
- Install PHP.
- For more examples, refer to the Demo in the open-source community.

Retry Mechanism

HTTP uses a fixed retry interval mechanism.

Retry Interval	Maximum number of retries
5 minutes	You can modify the consumer group configuration to set a custom maximum number of retries, default 16.

Note:

- The client ACKs this message within the retry interval, indicating successful consumption. No retry will be performed.
- If the client does not ACK the message upon retry interval expiry, it will reconsume this message.
- Each message handle is only valid within the retry interval and expires afterward.

Operation Steps

Step 1: Installing PHP Dependency Library

Support for importing related dependencies in the PHP project

```
{
  "require": {
     "aliyunmq/mq-http-sdk": ">=1.0.4"
  }
}
```

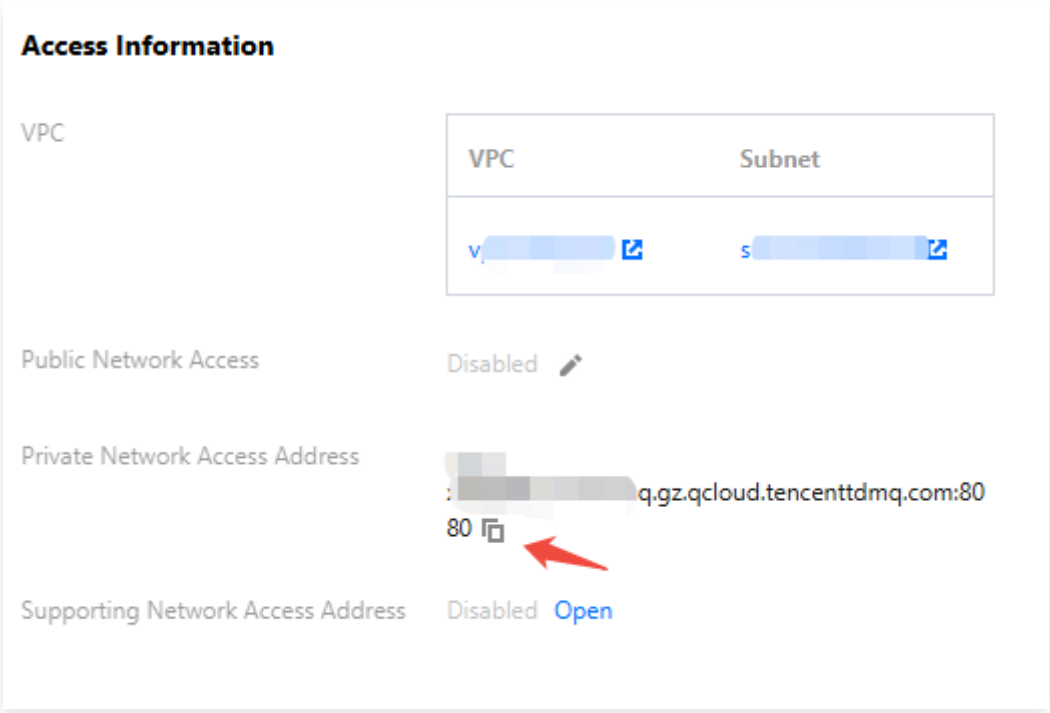
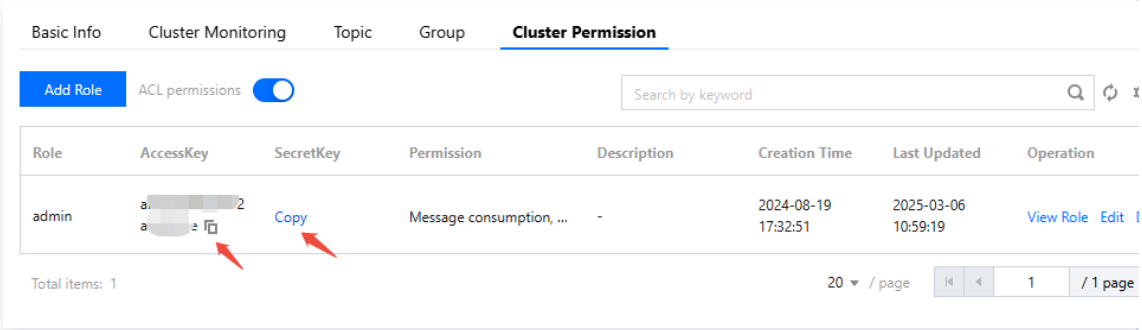
Step 2: Producing Messages

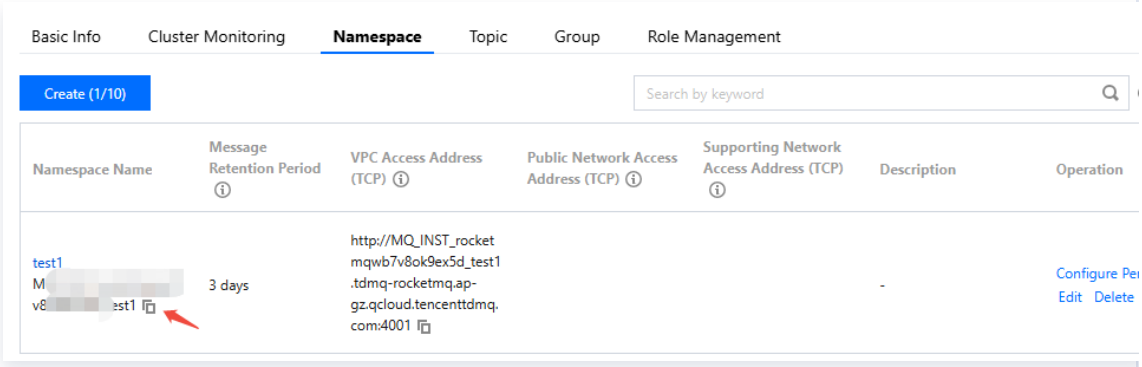
Creating Producers

```
    private $client;
    private $producer;

    public function __construct()
     {
         // Get client
         $this->client = new MQClient(
             endpoint,
             accessKey,
             secretKey
         );

         $topic = topicName;
         $instanceId = instanceId;

         // Get producer
         $this->producer = $this->client->getProducer($namespace,
$topic);
```

```
}
```

Note:
Note: The following parameters require logging in to the TDMQ RocketMQ Console to obtain.

Parameter	Description
endpoint	Get the cluster access address from the access information module on the console cluster basic information page. 
accessKey	Role key, copy from the AccessKey column on the cluster permissions page in the console. 
secretKey	Role name, copy from the SecretKey column on the cluster permissions page in the console.
namespace	Copy the namespace name from the Namespace page in the console. If you use a

	4.x generic cluster or 5.x cluster, fill in the Cluster ID here. 
topicName	Copy the topic name from the Topic page in the console.

Sending Messages

```
public function run()
    {
        try
        {
            for ($i=0; $i<8; $i++)
            {
                $publishMessage = new TopicMessage(
                    "hello mq!"
                );
                // Set attributes
                $publishMessage->putProperty("a", $i);
                $result = $this->producer-
>publishMessage($publishMessage);
                print "Send success. msgId is:" . $result-
>getMessageId() . ", bodyMD5 is:" . $result->getMessageBodyMD5() . "\n";
            }
        } catch (\Exception $e) {
            print_r($e->getMessage() . "\n");
        }
    }
```

Step 3: Consuming Messages

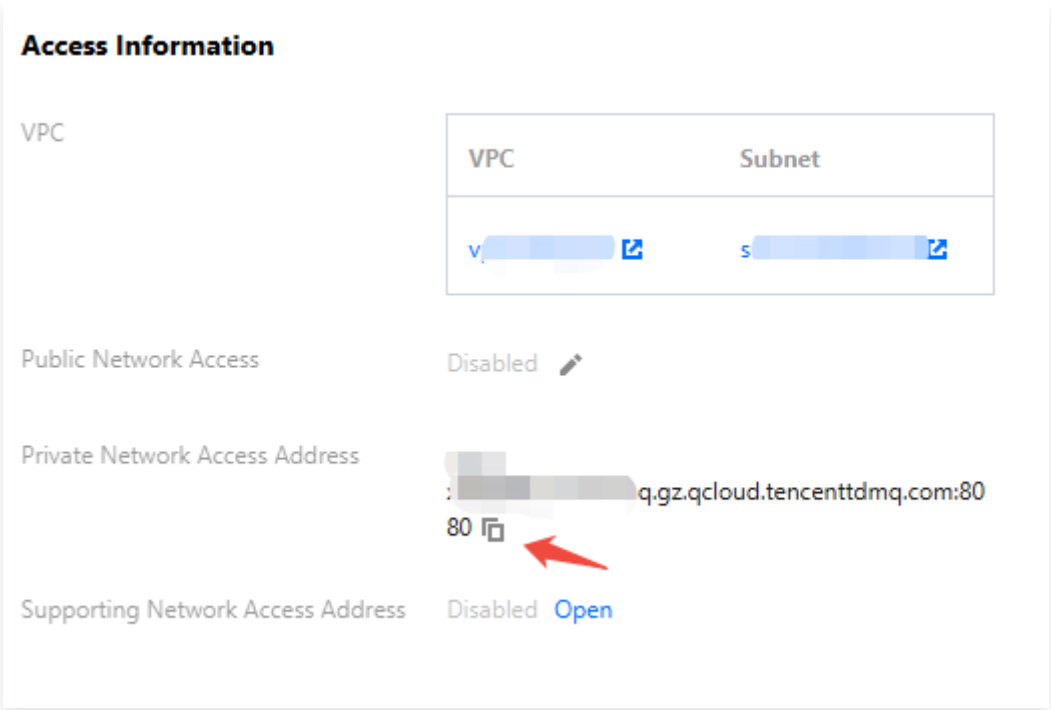
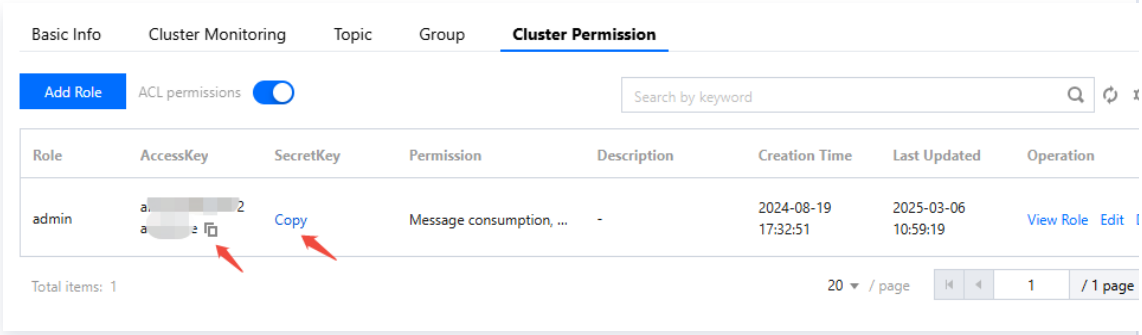
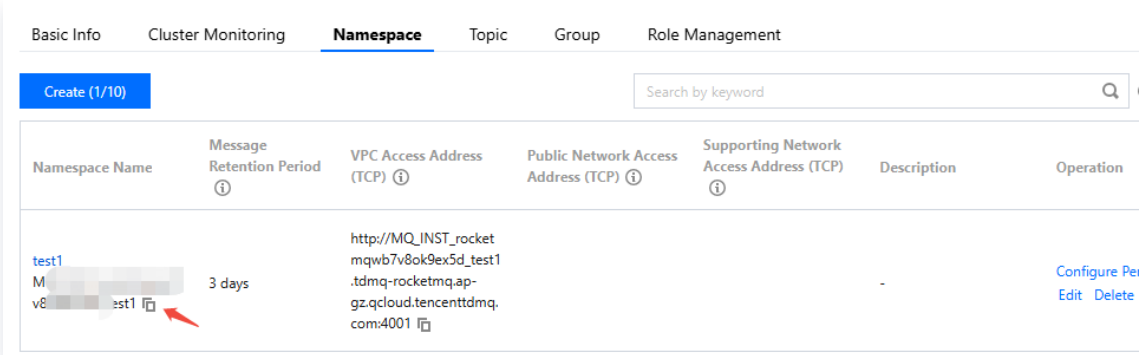
Creating Consumers

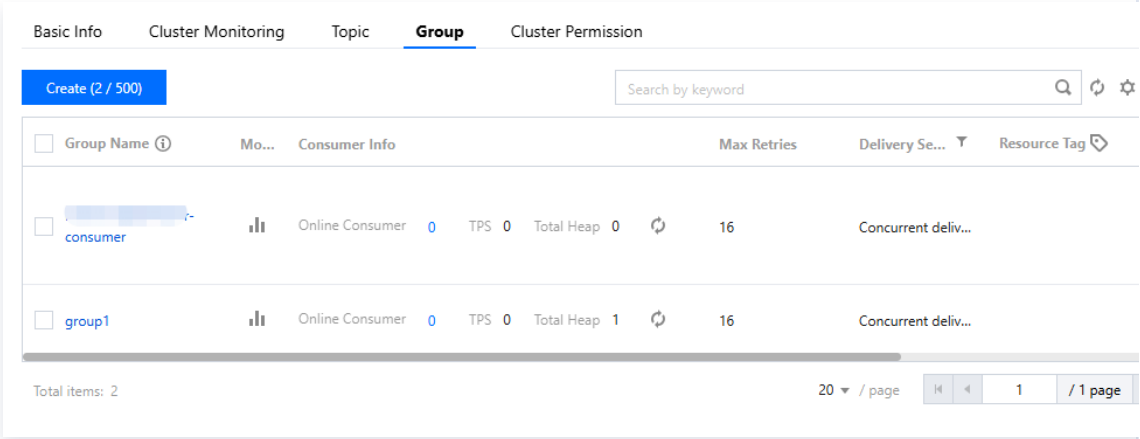
```
    private $client;
    private $consumer;

    public function __construct()
     {
         // Get client
         $this->client = new MQClient(
             endpoint,
             accessKey,
             secretKey
         );

         $topic = topicName;
         $groupId = groupName;
         $instanceId = namespace;

         // Get consumer
         $this->consumer = $this->client->getConsumer($namespace, $topic,
$groupId);
     }
```

Note:

Note: The following parameters require logging in to the TDMQ RocketMQ Console to obtain.

Parameter	Description
endpoint	Get the cluster access address from the access information module on the console cluster basic information page.

	
accessKey	Role key, copy from the AccessKey column on the cluster permissions page in the console. 
secretKey	Role name, copy from the SecretKey column on the cluster permissions page in the console.
namespace	Copy the namespace name from the Namespace page in the console. If you use a 4.x generic cluster or 5.x cluster, fill in the Cluster ID here. 

topicName	Copy the topic name from the Topic page in the console.
groupName	Copy the consumer group name from the Group management page on the console. 

Subscribing to Messages

```
public function run()
    {
        while (True) {
            try {
                // Consume messages with long polling
                // Long polling means if there is no message in the
topic, the request will wait on the server side, and if there is a
message to consume, it will return immediately
                // If sensitive to consumption delay, strongly recommend
pulling messages concurrently
                $messages = $this->consumer->consumeMessage(
                    batchSize,
                    waitSeconds
                );
            } catch (\Exception $e) {
                if ($e instanceof MQ\Exception\MessageNotExistException)
{
                    printf("No message, contine long
polling!RequestId:%s\n", $e->getRequestId());
                    continue;
                }

                print_r($e->getMessage() . "\n");
```

```
                sleep(1);
                continue;
            }

            print "consume finish, messages:\n";

            $receiptHandles = array();
            foreach ($messages as $message) {
                $receiptHandles[] = $message->getReceiptHandle();
                printf("MessageID:%s TAG:%s BODY:%s \nPublishTime:%d,
FirstConsumeTime:%d, \nConsumedTimes:%d,
NextConsumeTime:%d,MessageKey:%s\n",
                    $message->getMessageId(), $message->getMessageTag(),
$message->getMessageBody(),
                    $message->getPublishTime(), $message-
>getFirstConsumeTime(), $message->getConsumedTimes(), $message-
>getNextConsumeTime(),
                    $message->getMessageKey());
                print_r($message->getProperties());
                // Process the business
            }

            print_r($receiptHandles);
            try {
                $this->consumer->ackMessage($receiptHandles);
            } catch (\Exception $e) {
                if ($e instanceof MQ\Exception\AckMessageException) {
                    printf("Ack Error, RequestId:%s\n", $e-
>getRequestId());
                    foreach ($e->getAckMessageErrorItems() as
$errorItem) {
                        printf("\tReceiptHandle:%s, ErrorCode:%s,
ErrorMsg:%s\n", $errorItem->getReceiptHandle(), $errorItem-
>getErrorCode(), $errorItem->getErrorCode());
                    }
                }
            }
            print "ack finish\n";
        }
    }
```

Parameter	Description
batchSize	The message count pulled once supports up to 16.
waitSeconds	The polling wait time pulled once supports up to 30 seconds.

Client Risk Explanation and Update Guide

Last updated：2025-07-15 18:35:42

1. Java PushConsumer Sequential Messaging Cannot Guarantee Sequence in Specific Scenarios

Client List

Version 5.0.6 and lower versions may encounter this issue in some special scenarios. In cases of network exceptions, timeouts, etc., message order cannot be guaranteed. Related issue: [Bug] PushConsumer reentrant pop orderly doesn't work as expected.

Upgrading Client Operation Guide

Recommend upgrading to 5.0.7. Just direct upgrade the dependency version.

2. Golang Version Too Low, Route Table May Be Emptied

Client List

v5.0.1 and below may clear the route table when unable to obtain it due to network timeout in some scenarios. Related issue: golang: optimize the logic of obtaining routing tasks.

Upgrading Client Operation Guide

Recommend upgrading to v5.1.1-rc1. Just upgrade the dependency version directly.

3. Java Message Trace Possibly Incomplete

Client List

4.9.4 and below may encounter in some special scenarios.

Upgrading Client Operation Guide

Recommend upgrading to 4.9.5. Just direct upgrade the dependency version.

4. Go SDK First Launch May Display Error: fetch offset of mq from broker error

Client List

master branch and below may encounter, such as v2.1.2 and below, startup error `fetch offset of mq from broker error`.

related issue: fetch offset of mq from broker error

Upgrading Client Operation Guide

We recommend upgrading to the latest master branch as the Commit ID version.

5. Java SDK Error Prompts during Consumption Process: the queue's messages, span too long, so do flow control

Client List

4.7.1 and below may encounter the above logs printed in logs during the consumption process, and the consumption will be rate-limited in some special scenarios.
Related issue: [Bug] the queue's messages span too long, so do flow control.

Upgrading Client Operation Guide

Recommend upgrading to 4.9.5. Just direct upgrade the dependency version.

Migration Guide for TencentCloud API Related to RocketMQ 4.X Cluster Role

Last updated：2025-07-15 18:35:42

Background Note

Due to historical reasons, subproducts under the TDMQ product family share certain APIs. For example, Pulsar and RocketMQ have existing clusters that share Role-related TencentCloud APIs. To achieve the smallest granularity for permission isolation, RocketMQ subsequently introduced a separate Role-based management API. For customers using these shared TencentCloud APIs, TDMQ RocketMQ will notify them individually.
This document guides customers to better replace TencentCloud APIs. If you have not used the following shared APIs, you can ignore this document.

Relevant APIs

Historic Shared API Name	Description	Replace API
CreateRole	Creating a Role	CreateRocketMQRole
DeleteRoles	Deleting the Role	DeleteRocketMQRoles
DescribeRoles	Get role list	DescribeRocketMQRoles
ModifyRole	Role modification	ModifyRocketMQRole
ModifyEnvironmentRole	Modify environment role authorization	ModifyRocketMQEnvironmentRole
CreateEnvironmentRole	Create environment role authorization	CreateRocketMQEnvironmentRole
DescribeEnvironmentRoles	Namespace role list	DescribeRocketMQEnvironmentRoles
DeleteEnvironmentRoles	Delete environment role authorization	DeleteRocketMQEnvironmentRoles
DescribeMsgTrace	Querying the Message Trace	DescribeRocketMQMsgTrace
ModifyVipInstanceState	Modify dedicated instance status	ModifyRocketMQVipInstanceState
DescribeAppIdStats	Retrieve user	DescribeRocketMQAppIdStats

	information	

Migrating Example Code

During migration, only the instantiated request object needs to be altered to a RocketMQ-related object. Here, we take the API migration from CreateRole to CreateRocketMQRole as an example, using SDK integration with Java as the development language.

Old API Integration

```
package com.tencent;

import com.tencentcloudapi.common.Credential;
import com.tencentcloudapi.common.profile.ClientProfile;
import com.tencentcloudapi.common.profile.HttpProfile;
import com.tencentcloudapi.common.exception.TencentCloudSDKException;
import com.tencentcloudapi.tdmq.v20200217.TdmqClient;
import com.tencentcloudapi.tdmq.v20200217.models.*;

public class Sample
{
    public static void main(String [] args) {
        try{
            // Instantiate an authentication object. Input parameters
require passing in secretId and secretKey of the cloud account. Keep the
confidentiality of the key pair.
            // The key can be obtained at
https://console.intl.cloud.tencent.com/cam/capi.
            Credential cred = new Credential("", "");
            // Instantiate an HTTP option, optional, which can be
skipped if there is no special requirement.
            HttpProfile httpProfile = new HttpProfile();
            // Recommended for use Polaris. Relevant guidelines are
accessible at the following link
            // https://git.woa.com/tencentcloud-internal/tencentcloud-
sdk-java#%E5%8C%97%E6%9E%81%E6%98%9F
            httpProfile.setEndpoint("tdmq.ap-
guangzhou.tencentcloudapi.woa.com");
            // Instantiate a client option, optional, which can be
skipped if there is no special requirement.
```

```
            ClientProfile clientProfile = new ClientProfile();
            clientProfile.setHttpProfile(httpProfile);
            // Instantiate the client object to request the product, and
clientProfile is optional.
            TdmqClient client = new TdmqClient(cred, "", clientProfile);
            // Instantiate a request object. Each interface will
correspond to a request object.
            CreateRoleRequest req = new CreateRoleRequest();

            // The returned resp is an instance of CreateRoleResponse,
corresponding to the request object
            CreateRoleResponse resp = client.CreateRole(req);
            // Output string in json format.
            System.out.println(resp);
        } catch (TencentCloudSDKException e) {
            System.out.println(e.toString());
        }
    }
}
```

New API Integration

```
package com.tencent;


import com.tencentcloudapi.common.Credential;
import com.tencentcloudapi.common.profile.ClientProfile;
import com.tencentcloudapi.common.profile.HttpProfile;
import com.tencentcloudapi.common.exception.TencentCloudSDKException;
import com.tencentcloudapi.tdmq.v20200217.TdmqClient;
import com.tencentcloudapi.tdmq.v20200217.models.*;

public class Sample
{
    public static void main(String [] args) {
        try{
            // Instantiate an authentication object. Input parameters
require passing in secretId and secretKey of the cloud account. Keep the
confidentiality of the key pair.
```

```
            // The key can be obtained at
https://console.intl.cloud.tencent.com/cam/capi.
            Credential cred = new Credential("", "");
            // Instantiate an HTTP option, optional, which can be
skipped if there is no special requirement.
            HttpProfile httpProfile = new HttpProfile();
            // Recommended for use Polaris. Relevant guidelines are
accessible at the following link
            // https://git.woa.com/tencentcloud-internal/tencentcloud-
sdk-java#%E5%8C%97%E6%9E%81%E6%98%9F
            httpProfile.setEndpoint("tdmq.ap-
guangzhou.tencentcloudapi.woa.com");
            // Instantiate a client option, optional, which can be
skipped if there is no special requirement.
            ClientProfile clientProfile = new ClientProfile();
            clientProfile.setHttpProfile(httpProfile);
            // Instantiate the client object to request the product, and
clientProfile is optional.
            TdmqClient client = new TdmqClient(cred, "", clientProfile);
            // Instantiate a request object. Each interface will
correspond to a request object.
            CreateRocketMQRoleRequest req = new
CreateRocketMQRoleRequest();

            // The returned resp is an instance of
CreateRocketMQRoleResponse, corresponding to the request object
            CreateRocketMQRoleResponse resp =
client.CreateRocketMQRole(req);
            // Output string in json format.
            System.out.println(resp);
        } catch (TencentCloudSDKException e) {
            System.out.println(e.toString());
        }
    }
}
```