Distributed Message Service for RocketMQ

Developer Guide

Issue 01

Date 2023-03-22

HUAWEI

HUAWEI TECHNOLOGIES CO., LTD.

Trademarks and Permissions


and other Huawei trademarks are trademarks of Huawei Technologies Co., Ltd.
All other trademarks and trade names mentioned in this document are the property of their respective holders.

Notice

The purchased products, services and features are stipulated by the contract made between Huawei and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

Contents

1 Overview

Chapter 2 describes how to obtain the connection information of a RocketMQ instance.

Chapter 3 to Chapter 5 describe the sample code (Table 1-1) for accessing DMS for RocketMQ from a Java, Go, or Python client.

Table 1-1 Sample code

Language	Sample Code
Java	• Sending and Receiving Normal Messages • Sending and Receiving Ordered Messages • Sending and Receiving Transactional Messages • Delivering Scheduled Messages • Controlling Access with ACL • Controlling Traffic on Consumers
Go	• Sending and Receiving Normal Messages • Sending and Receiving Ordered Messages • Sending and Receiving Transactional Messages • Delivering Scheduled Messages • Controlling Access with ACL
Python	• Sending and Receiving Normal Messages • Sending and Receiving Ordered Messages • Sending and Receiving Transactional Messages • Delivering Scheduled Messages • Controlling Access with ACL

2 Collecting Connection Information

Obtaining Instance Connection Information

- Instance metadata address and port

  After an instance is created, you can obtain the IP address from the **Basic Information** tab page of the instance on the RocketMQ console. You can configure all IP addresses on the client.

  Figure 2-1 Viewing metadata address and port of an instance (intra-VPC access)

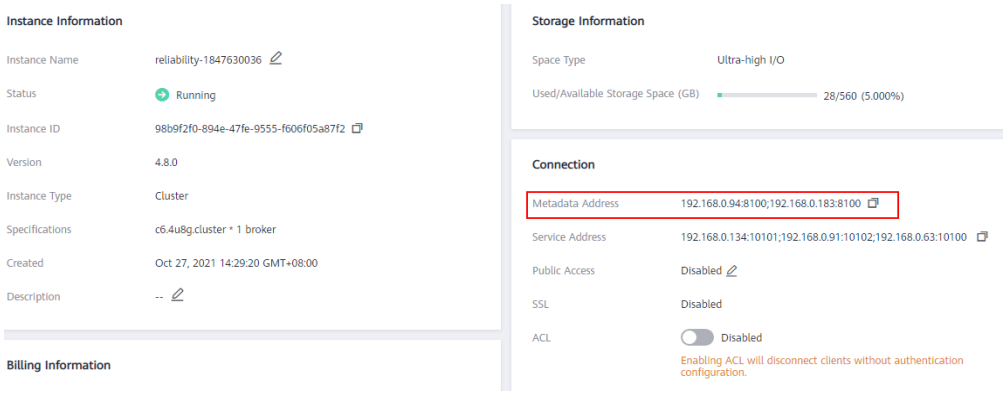
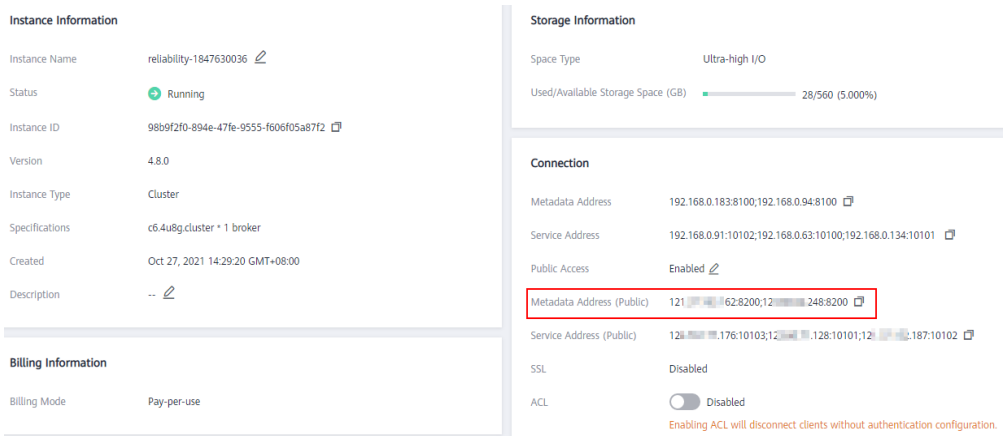


  Figure 2-2 Viewing metadata address and port of an instance (public access)



- Topic name

  Obtain the topic name from the **Topics** page of the instance on the RocketMQ console.

- Consumer group name

  Obtain the consumer group name from the **Consumer Groups** page of the instance on the RocketMQ console.

- Username and secret key

  Obtain the username from the **Users** tab page of the instance and obtain the secret key from the user details page on the RocketMQ console.

3 Java

3.1 Sending and Receiving Normal Messages

This section describes how to send and receive normal messages and provides sample code. Normal messages can be sent in the synchronous or asynchronous mode.

- Synchronous transmission: After sending a message, the sender waits for the server to receive and process the message, and does not send the next message until it receives a response from the server.
- Asynchronous transmission: After sending a message, the sender sends the next message without waiting for a response from the server.

Before sending and receiving normal messages, collect RocketMQ connection information by referring to **Collecting Connection Information**.

Preparing the Environment

You can connect open-source Java clients to DMS for RocketMQ. The recommended Java client version is 4.9.0.

Use either of the following methods to import a dependency:

- Using Maven
```
<dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-client</artifactId>
    <version>4.9.0</version>
</dependency>
```
- Downloading **the dependency**

Synchronous Transmission

After sending a message, the sender waits for the server to receive and process the message, and does not send the next message until it receives a response from the server.

Refer to the following sample code or obtain more sample code from **Producer.java**.


```
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.remoting.common.RemotingHelper;

public class Main {
    public static void main(String[] args) {
        DefaultMQProducer producer = new DefaultMQProducer("ProducerGroupName");
        // Enter the metadata address.
        producer.setNamesrvAddr("192.168.0.1:8100");
        //producer.setUseTLS(true); // Add this line if SSL has been enabled during instance creation.
        try {
            producer.start();
            Message msg = new Message("TopicTest",
                "TagA",
                "OrderID188",
                "Hello world".getBytes(RemotingHelper.DEFAULT_CHARSET));
            SendResult sendResult = producer.send(msg);
            System.out.printf("%s%n", sendResult);

        } catch (Exception e) {
            e.printStackTrace();

        }
        producer.shutdown();
    }
}
```

Asynchronous Transmission

After sending a message, the sender sends the next message without waiting for a response from the server.

Asynchronous transmission requires the SendCallback method to be supported on the client. After sending a message, the sender sends the next message without waiting for a server response. The sender calls the SendCallback method to receive the server's response and then processes the response.

Refer to the following sample code or obtain more sample code from **AsyncProducer.java**.

```
import org.apache.rocketmq.client.producer.DefaultMQProducer;
import org.apache.rocketmq.client.producer.SendCallback;
import org.apache.rocketmq.client.producer.SendResult;
import org.apache.rocketmq.common.message.Message;
import org.apache.rocketmq.remoting.common.RemotingHelper;


public class Main {
    public static void main(String[] args) throws InterruptedException {
        DefaultMQProducer producer = new DefaultMQProducer("ProducerGroupName");
        // Enter the metadata address.
        producer.setNamesrvAddr("192.168.120.45:8100;192.168.123.150:8100");
        //producer.setUseTLS(true);    // Add this line if SSL has been enabled during instance creation.
        try {
            producer.start();
            Message msg = new Message("TopicTest",
                    "TagA",
                    "OrderID188",
                    "Hello world".getBytes(RemotingHelper.DEFAULT_CHARSET));
            producer.send(msg, new SendCallback() {
                @Override
                public void onSuccess(SendResult result) {
                    // Message sent.
                    System.out.println("send message success. msgId= " + result.getMsgId());
                }
```



```
            @Override
            public void onException(Throwable throwable) {
                // If the message fails to be sent, you can resend the message or persist the data for
compensation.
                System.out.println("send message failed.");
                throwable.printStackTrace();
            }
        });


    } catch (Exception e) {
        e.printStackTrace();
    }
    Thread.sleep(2000);
    producer.shutdown();
  }}
```

Subscribing to Normal Messages

Refer to the following sample code or obtain more sample code from **PushConsumer.java**.

```
import java.util.List;
import org.apache.rocketmq.client.consumer.DefaultMQPushConsumer;
import org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlyContext;
import org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlyStatus;
import org.apache.rocketmq.client.consumer.listener.MessageListenerConcurrently;
import org.apache.rocketmq.client.exception.MQClientException;
import org.apache.rocketmq.common.consumer.ConsumeFromWhere;
import org.apache.rocketmq.common.message.MessageExt;

public class PushConsumer {

    public static void main(String[] args) throws InterruptedException, MQClientException {
        DefaultMQPushConsumer consumer = new
DefaultMQPushConsumer("please_rename_unique_group_name");
        // Enter the metadata address.
        consumer.setNamesrvAddr("192.168.0.1:8100");
        //consumer.setUseTLS(true);    // Add this line if SSL has been enabled during instance creation.
        consumer.subscribe("TopicTest", "*");
        consumer.registerMessageListener(new MessageListenerConcurrently() {
            @Override
            public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs,
ConsumeConcurrentlyContext context) {
                System.out.printf("%s Receive New Messages: %s %n", Thread.currentThread().getName(),
msgs);
                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
            }
        });
        consumer.start();
        System.out.printf("Consumer Started.%n");
    }
}
```

3.2 Sending and Receiving Ordered Messages

In DMS for RocketMQ, ordered messages are retrieved in the exact order that they are created.

Ordered messages are ordered globally or on the partition level.

- Globally ordered messages: There is only one queue in a specific topic. All messages in the queue will be published and subscribed to in the first in, first out (FIFO) order.
- Partition-level ordered message: Messages within a queue in a specific topic are published and subscribed to in the FIFO order. The producer specifies a partition selection algorithm to ensure that the messages to be ordered are allocated to the same queue.

The only difference between globally ordered messages and partition-level ordered messages is the number of queues. The code is the same.

Before sending and receiving ordered messages, collect RocketMQ connection information by referring to **Collecting Connection Information**.

Preparing the Environment

You can connect open-source Java clients to DMS for RocketMQ. The recommended Java client version is 4.9.0.

Use either of the following methods to import a dependency:

- Using Maven

```
<dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-client</artifactId>
    <version>4.9.0</version>
</dependency>
```

- Downloading **the dependency**

Sending Ordered Messages

Refer to the following sample code or obtain more sample code from **Producer.java**.

```
import java.nio.charset.StandardCharsets;
import java.util.List;
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.MessageQueueSelector;
import org.apache.rocketmq.client.producer.SendResult;
import org.apache.rocketmq.common.message.Message;
import org.apache.rocketmq.common.message.MessageQueue;
import org.apache.rocketmq.remoting.exception.RemotingException;

public class Producer {

    public static void main(String[] args) {
        try {
            DefaultMQProducer producer = new DefaultMQProducer("please_rename_unique_group_name");
            // Enter the metadata address.
            producer.setNamesrvAddr("192.168.0.1:8100");
            //producer.setUseTLS(true);    // Add this line if SSL has been enabled during instance creation.
            producer.start();

            String[] tags = new String[] {"TagA", "TagB", "TagC", "TagD", "TagE"};
            for (int i = 0; i < 100; i++) {
                String orderId = "order" + (i % 10);
                Message msg = new Message("TopicTest", tags[i % tags.length], "KEY" + i,
                    ("Hello RocketMQ " + i).getBytes(StandardCharsets.UTF_8));
                SendResult sendResult = producer.send(msg, new MessageQueueSelector() {
                    @Override
                    public MessageQueue select(List<MessageQueue> mqs, Message msg, Object arg) {
```



```
                String orderId = (String) arg;
                int index = Math.abs(orderId.hashCode() % mqs.size());
                return mqs.get(index);
            }
        }, orderId);

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

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

In the preceding code, only the sequence of messages with the same **orderId** must be kept unchanged. Therefore, in the partition selection algorithm, specify the remainder of the value of **orderId** divided by the number of queues as the queue where messages are sent.

Subscribing to Ordered Messages

Refer to the following sample code or obtain more sample code from **Consumer.java**.

```
import java.util.List;
import java.util.concurrent.atomic.AtomicLong;

import org.apache.rocketmq.client.consumer.DefaultMQPushConsumer;
import org.apache.rocketmq.client.consumer.listener.ConsumeOrderlyContext;
import org.apache.rocketmq.client.consumer.listener.ConsumeOrderlyStatus;
import org.apache.rocketmq.client.consumer.listener.MessageListenerOrderly;
import org.apache.rocketmq.client.exception.MQClientException;
import org.apache.rocketmq.common.message.MessageExt;

public class Consumer {

    public static void main(String[] args) throws MQClientException {
        DefaultMQPushConsumer consumer = new
DefaultMQPushConsumer("please_rename_unique_group_name_3");
        // Enter the metadata address.
        consumer.setNamesrvAddr("192.168.0.1:8100");
        //consumer.setUseTLS(true);    // Add this line if SSL has been enabled during instance creation.

        consumer.subscribe("TopicTest", "*");

        consumer.registerMessageListener(new MessageListenerOrderly() {
            AtomicLong consumeTimes = new AtomicLong(0);

            @Override
            public ConsumeOrderlyStatus consumeMessage(List<MessageExt> msgs, ConsumeOrderlyContext
context) {
                context.setAutoCommit(true);
                System.out.printf("%s Receive New Messages: %s %n", Thread.currentThread().getName(),
msgs);
                this.consumeTimes.incrementAndGet();
                if ((this.consumeTimes.get() % 3) == 0) {
                    context.setSuspendCurrentQueueTimeMillis(3000);
                    return ConsumeOrderlyStatus.SUSPEND_CURRENT_QUEUE_A_MOMENT;
                }

                return ConsumeOrderlyStatus.SUCCESS;
            }
        });

        consumer.start();
        System.out.printf("Consumer Started.%n");
```

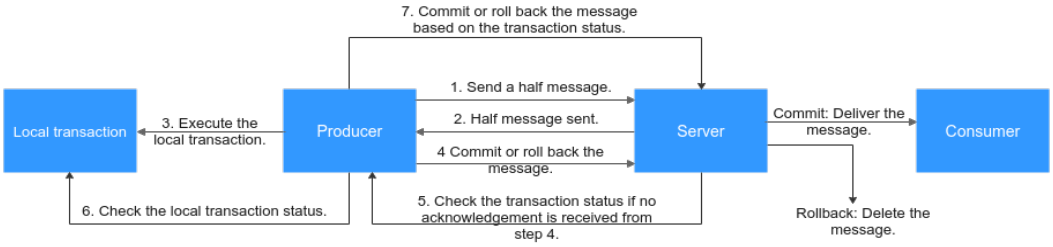


```
    }
}
```

3.3 Sending and Receiving Transactional Messages

DMS for RocketMQ ensures transaction consistency between the service logic and message transmission, and implements transaction support in two phases. **Figure 3-1** illustrates the interaction of transactional messages.

Figure 3-1 Transactional message interaction



The producer sends a half message and then executes the local transaction. If the execution is successful, the transaction is committed. If the execution fails, the transaction is rolled back. If the server does not receive any commit or rollback request after a period of time, it initiates a check. After receiving the check request, the producer resends a transaction commit or rollback request. The message is delivered to the consumer only after being committed. The consumer is unaware of the rollback.

Before sending and receiving transactional messages, collect RocketMQ connection information by referring to **Collecting Connection Information**.

Preparing the Environment

You can connect open-source Java clients to DMS for RocketMQ. The recommended Java client version is 4.9.0.

Use either of the following methods to import a dependency:

- Using Maven

```
<dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-client</artifactId>
    <version>4.9.0</version>
</dependency>
```

- Downloading **the dependency**

Sending Transactional Messages

Refer to the following sample code or obtain more sample code from **TransactionProducer.java**.

```
import org.apache.rocketmq.client.exception.MQClientException;
import org.apache.rocketmq.client.producer.LocalTransactionState;
import org.apache.rocketmq.client.producer.SendResult;
import org.apache.rocketmq.client.producer.TransactionListener;
import org.apache.rocketmq.client.producer.TransactionMQProducer;
```

```
import org.apache.rocketmq.common.message.Message;
import org.apache.rocketmq.common.message.MessageExt;
import org.apache.rocketmq.remoting.common.RemotingHelper;

import java.io.UnsupportedEncodingException;

public class Main {
    public static void main(String[] args) throws MQClientException, UnsupportedEncodingException {
        TransactionListener transactionListener = new TransactionListener() {
            @Override
            public LocalTransactionState executeLocalTransaction(Message message, Object o) {
                System.out.println("Start to execute the local transaction: "+ message);
                return LocalTransactionState.COMMIT_MESSAGE;
            }

            @Override
            public LocalTransactionState checkLocalTransaction(MessageExt messageExt) {
                System.out.println("Check request received. Check the transaction status: "+ messageExt);
                return LocalTransactionState.COMMIT_MESSAGE;
            }
        };

        TransactionMQProducer producer = new
TransactionMQProducer("please_rename_unique_group_name");
        // Enter the metadata address.
        producer.setNamesrvAddr("192.168.0.1:8100");
        //producer.setUseTLS(true);    // Add this line if SSL has been enabled during instance creation.
        producer.setTransactionListener(transactionListener);
        producer.start();

        Message msg =
            new Message("TopicTest", "TagA", "KEY",
                "Hello RocketMQ ".getBytes(RemotingHelper.DEFAULT_CHARSET));
        SendResult sendResult = producer.sendMessageInTransaction(msg, null);
        System.out.printf("%s%n", sendResult);

        producer.shutdown();
    }}
```

The producer needs to implement two callback functions. The executeLocalTransaction callback function is called after the half message is sent (see step 3 in the diagram). The checkLocalTransaction callback function is called after the check request is received (see step 6 in the diagram). The two callback functions can return three transaction states:

- **LocalTransactionState.COMMIT_MESSAGE**: Transaction committed. The consumer can retrieve the message.
- **LocalTransactionState.ROLLBACK_MESSAGE**: Transaction rolled back. The message will be discarded and cannot be retrieved.
- **LocalTransactionState.UNKNOW**: The status cannot be determined and the server is expected to check the message status from the producer again.

Subscribing to Transactional Messages

The code for subscribing to transactional messages is the same as that for **subscribing to normal messages**.

3.4 Delivering Scheduled Messages

In DMS for RocketMQ, you can schedule messages to be delivered at **any time**, with a maximum delay of one year. You can also cancel scheduled messages.

After being sent from producers to DMS for RocketMQ, scheduled messages are delivered to consumers only after a specified point in time.

Before delivering scheduled messages, collect RocketMQ connection information by referring to **Collecting Connection Information**.

NOTE

This function is supported only for instances created on or after March 30, 2022.

Application Scenarios

Scheduled messages can be used in the following scenarios:

- The service logic requires a time window. For example, an e-commerce order is closed if it is not paid within a period of time. When an order is created, a scheduled message is sent and will be delivered to the consumer five minutes later. After receiving the message, the consumer checks whether the order is paid. If the order is not paid, it is closed. If the order is paid, the message is ignored.
- A scheduled task is triggered by a message. For example, a reminder is sent to a user at a specific time.

Note

- The delivery time can be scheduled to up to one year later. If the delay time exceeds one year, the message cannot be delivered.
- If the delivery time is scheduled to a time point earlier than the current timestamp, the message is immediately sent to the consumer.
- Ideally, the difference between the scheduled delivery time and the actual delivery time is smaller than 0.1s. However, if the pressure of scheduled message delivery is too high, flow control will be triggered, and the precision will deteriorate.
- The message delivery order is not ensured for precision of 0.1s. That is, if the difference between the scheduled delivery time of two messages is smaller than 0.1s, they may not be delivered in the order that they were sent.
- Exactly-once delivery is not guaranteed. A scheduled message may be delivered repeatedly.
- The scheduled time is the time when the server starts to deliver a message to a consumer. If messages are stacked on the consumer, the scheduled message is delivered after the stacked messages, and cannot be delivered exactly at the configured time.
- Due to a potential time difference between the client and server, the actual delivery time may be different from the delivery time set by the client. The server time is used.
- Messages are retained for two days after the scheduled delivery time. For example, if a scheduled message is not retrieved in five days as scheduled, it is deleted on the seventh day.
- Scheduled messages occupy about three times the storage space of normal messages. If you use a large number of scheduled messages, pay attention to the storage space usage.

Preparing the Environment

You can connect open-source Java clients to DMS for RocketMQ. The recommended Java client version is 4.9.0.

Use either of the following methods to import a dependency:

- Using Maven

```
<dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-client</artifactId>
    <version>4.9.0</version>
</dependency>
```

- Downloading **the dependency**

Delivering Scheduled Messages

The code for delivering a scheduled message is as follows:

```
import java.nio.charset.StandardCharsets;
import java.time.Instant;
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.UtilAll;
import org.apache.rocketmq.common.message.Message;
import org.apache.rocketmq.remoting.exception.RemotingException;


public class ScheduledMessageProducer1 {
    public static final String TOPIC_NAME = "ScheduledTopic";


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


        DefaultMQProducer producer = new DefaultMQProducer("ProducerGroupName");
        // Enter the metadata address.
        producer.setNamesrvAddr("192.168.0.1:8100");
        //producer.setUseTLS(true);    // Add this line if SSL has been enabled during instance creation.
        producer.start();


        // Timestamp for scheduled delivery. The message will be delivered in 10 seconds.
        final long deliverTimestamp = Instant.now().plusSeconds(10).toEpochMilli();
        // Create a message object.
        Message msg = new Message(TOPIC_NAME,
            "TagA",
            "KEY",
            "scheduled message".getBytes(StandardCharsets.UTF_8));
        // Set the timestamp for scheduled message delivery.
        msg.putUserProperty("__STARTDELIVERTIME", String.valueOf(deliverTimestamp));
        // Send the message. The message will be delivered in 10 seconds.
        SendResult sendResult = producer.send(msg);
        // Print the delivery result and estimated delivery time.
        System.out.printf("%s %s%n", sendResult, UtilAll.timeMillisToHumanString2(deliverTimestamp));


        producer.shutdown();
    }
}
```


Canceling a Scheduled Message

The code for canceling a scheduled message is as follows:

```
import java.nio.charset.StandardCharsets;
import java.time.Instant;
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.UtilAll;
import org.apache.rocketmq.common.message.Message;
import org.apache.rocketmq.remoting.exception.RemotingException;

public class ScheduledMessageProducer1 {
    public static final String TOPIC_NAME = "ScheduledTopic";

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

        DefaultMQProducer producer = new DefaultMQProducer("ProducerGroupName");
        // Enter the metadata address.
        producer.setNamesrvAddr("192.168.0.1:8100");
        //producer.setUseTLS(true);    // Add this line if SSL has been enabled during instance creation.
        producer.start();

        // Timestamp for scheduled delivery. The message will be delivered in 10 seconds.
        final long deliverTimestamp = Instant.now().plusSeconds(10).toEpochMilli();
        // Create a message object.
        Message msg = new Message(TOPIC_NAME,
            "TagA",
            "KEY",
            "scheduled message".getBytes(StandardCharsets.UTF_8));
        // Set the timestamp for scheduled message delivery.
        msg.putUserProperty("__STARTDELIVERTIME", String.valueOf(deliverTimestamp));
        // Send the message. The message will be delivered in 10 seconds.
        SendResult sendResult = producer.send(msg);
        // Print the delivery result and estimated delivery time.
        System.out.printf("%s %s%n", sendResult, UtilAll.timeMillisToHumanString2(deliverTimestamp));

        // ====== Sending the cancellation logic ======

        // Create an object for the cancellation.
        Message cancelMsg = new Message(TOPIC_NAME,
            "",
            "",
            "cancel".getBytes(StandardCharsets.UTF_8));
        // Set the timestamp of the message to be canceled. This timestamp must be the same as that of the
scheduled delivery.
        cancelMsg.putUserProperty("__STARTDELIVERTIME", String.valueOf(deliverTimestamp));
        // Set the unique ID (UNIQUE_KEY) of the message to be canceled. The ID can be obtained from the
message sending result.
        cancelMsg.putUserProperty("__CANCEL_SCHEDULED_MSG", sendResult.getMsgId());
        // Send the cancellation message before the scheduled delivery time.
        SendResult cancelSendResult = producer.send(cancelMsg, sendResult.getMessageQueue());
        System.out.printf("cancel %s%n", cancelSendResult);

        producer.shutdown();    }}
```

3.5 Controlling Access with ACL

After ACL is enabled for an instance, user authentication information must be added to both the producer and consumer configurations.

Preparing the Environment

You can connect open-source Java clients to DMS for RocketMQ. The recommended Java client version is 4.9.0.

Use either of the following methods to import a dependency:

- Using Maven

```
<dependency>
   <groupId>org.apache.rocketmq</groupId>
   <artifactId>rocketmq-client</artifactId>
   <version>4.9.0</version>
</dependency>

<dependency>
   <groupId>org.apache.rocketmq</groupId>
   <artifactId>rocketmq-acl</artifactId>
   <version>4.9.0</version>
</dependency>
```

- Downloading **the dependency**

Adding User Authentication Information to the Producer

Step 1 Create configuration file **config.yaml** on the producer client and add the following authentication information to the file. If the configuration file already exists, directly add the authentication information to it.

```
ACL_ACCESS_KEY: "******"
ACL_SECRET_KEY: "******"
```

ACL_ACCESS_KEY indicates the username, and **ACL_SECRET_KEY** indicates the user secret key. For details about how to create a user, see **Creating a User**. Encrypt the username and key for security.

Step 2 Add the **rpcHook** parameter during producer initialization.

- For normal, ordered, and scheduled messages, add the following code:

```
RPCHook rpcHook = new AclClientRPCHook(new SessionCredentials(ACL_ACCESS_KEY,
ACL_SECRET_KEY));
DefaultMQProducer producer = new DefaultMQProducer("ProducerGroupName", rpcHook);
```

- For transactional messages, add the following code:

```
RPCHook rpcHook = new AclClientRPCHook(new SessionCredentials(ACL_ACCESS_KEY,
ACL_SECRET_KEY));
TransactionMQProducer producer = new TransactionMQProducer("ProducerGroupName", rpcHook);
```

----End

Adding User Authentication Information to the Consumer

Step 1 Create configuration file **config.yaml** on the consumer client and add the following authentication information to the file. If the configuration file already exists, directly add the authentication information to it.

```
ACL_ACCESS_KEY: "******"
ACL_SECRET_KEY: "******"
```

ACL_ACCESS_KEY indicates the username, and **ACL_SECRET_KEY** indicates the user secret key. For details about how to create a user, see **Creating a User**. Encrypt the username and key for security.

Step 2 Add the **rpcHook** parameter during consumer initialization. Add the following code for normal, ordered, scheduled, and transactional messages:


```
RPCHook rpcHook = new AclClientRPCHook(new SessionCredentials(ACL_ACCESS_KEY, ACL_SECRET_KEY));
DefaultMQPushConsumer consumer = new DefaultMQPushConsumer(null, "ConsumerGroupName",
rpcHook);
```

----End

3.6 Controlling Traffic on Consumers

If consumers consume messages too fast for downstream services to keep up, system stability will be affected. This section provides sample code for controlling consumer traffic to ensure system stability.

```
package org.apache.rocketmq.example.simple;

import java.util.List;
import java.util.concurrent.TimeUnit;

import com.google.common.util.concurrent.RateLimiter;
import org.apache.rocketmq.client.consumer.DefaultMQPushConsumer;
import org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlyContext;
import org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlyStatus;
import org.apache.rocketmq.client.consumer.listener.MessageListenerConcurrently;
import org.apache.rocketmq.client.exception.MQClientException;
import org.apache.rocketmq.common.consumer.ConsumeFromWhere;
import org.apache.rocketmq.common.message.MessageExt;

public class PushConsumer {

    public static void main(String[] args) throws InterruptedException, MQClientException {
        DefaultMQPushConsumer consumer = new
DefaultMQPushConsumer("please_rename_unique_group_name");
        consumer.subscribe("TopicTest", "*");
        consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_OFFSET);
        RateLimiter rateLimiter = RateLimiter.create(200);
        consumer.registerMessageListener(new MessageListenerConcurrently() {

            @Override
            public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs,
ConsumeConcurrentlyContext context) {
                if (!rateLimiter.tryAcquire(msgs.size(),3, TimeUnit.SECONDS)) {
                    return ConsumeConcurrentlyStatus.RECONSUME_LATER;
                }
                System.out.printf("%s Receive New Messages: %s %n", Thread.currentThread().getName(),
msgs);
                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
            }
        });
        consumer.start();
        System.out.printf("Consumer Started.%n");
    }
}
```


4 Go

4.1 Sending and Receiving Normal Messages

This section describes how to send and receive normal messages and provides sample code. Normal messages can be sent in the synchronous or asynchronous mode.

- Synchronous transmission: After sending a message, the sender waits for the server to receive and process the message, and does not send the next message until it receives a response from the server.
- Asynchronous transmission: After sending a message, the sender sends the next message without waiting for a response from the server.

Before sending and receiving normal messages, collect RocketMQ connection information by referring to **Collecting Connection Information**.

Preparing the Environment

1. Run the following command to check whether Go has been installed:
   ```
   go version
   ```
   If the following information is displayed, Go has been installed:
   ```
   go version go1.16.5 linux/amd64
   ```
   If Go is not installed, **download** and install it.
2. Add the following code to **go.mod** to add the dependency:
   ```
   module rocketmq-example-go

   go 1.13

   require (
       github.com/apache/rocketmq-client-go/v2 v2.1.1
   )
   ```

Synchronous Transmission

After sending a message, the sender waits for the server to receive and process the message, and does not send the next message until it receives a response from the server.

The following code is an example. Replace the information in bold with the actual values.

```
package main

import (
    "context"
    "fmt"
    "github.com/apache/rocketmq-client-go/v2"
    "github.com/apache/rocketmq-client-go/v2/primitive"
    "github.com/apache/rocketmq-client-go/v2/producer"
    "os"
)

// implements a simple producer to send message.
func main() {
    p, _ := rocketmq.NewProducer(
        producer.WithNsResolver(primitive.NewPassthroughResolver([]string{"192.168.0.1:8100"})),
        producer.WithRetry(2),
    )
    err := p.Start()
    if err != nil {
        fmt.Printf("start producer error: %s", err.Error())
        os.Exit(1)
    }
    msg := &primitive.Message{
        Topic: "topic1",
        Body:  []byte("Hello RocketMQ Go Client!"),
    }
    msg.WithTag("TagA")
    msg.WithKeys([]string{"KeyA"})
    res, err := p.SendSync(context.Background(), msg)

    if err != nil {
        fmt.Printf("send message error: %s\n", err)
    } else {
        fmt.Printf("send message success: result=%s\n", res.String())
    }
    err = p.Shutdown()
    if err != nil {
        fmt.Printf("shutdown producer error: %s", err.Error())
    }
}
```

The parameters in the example code are described as follows. For details about how to obtain the parameter values, see **Collecting Connection Information**.

- **192.168.0.1:8100**: instance metadata address and port.
- **topic1**: topic name.

Asynchronous Transmission

After sending a message, the sender sends the next message without waiting for a response from the server.

Asynchronous transmission requires the SendCallback method to be supported on the client. After sending a message, the sender sends the next message without waiting for a server response. The sender calls the SendCallback method to receive the server's response and then processes the response.

The following code is an example. Replace the information in bold with the actual values.

```
package main
```



```
import (
    "context"
    "fmt"
    "os"
    "sync"

    "github.com/apache/rocketmq-client-go/v2"
    "github.com/apache/rocketmq-client-go/v2/primitive"
    "github.com/apache/rocketmq-client-go/v2/producer"
)

// implements an async producer to send message.
func main() {
    p, _ := rocketmq.NewProducer(
        producer.WithNsResolver(primitive.NewPassthroughResolver([]string{"192.168.0.1:8100"})),
        producer.WithRetry(2))

    err := p.Start()
    if err != nil {
        fmt.Printf("start producer error: %s", err.Error())
        os.Exit(1)
    }
    var wg sync.WaitGroup
    wg.Add(1)

    callback := func(ctx context.Context, result *primitive.SendResult, e error) {
        if e != nil {
            fmt.Printf("receive message error: %s\n", err)
        } else {
            fmt.Printf("send message success: result=%s\n", result.String())
        }
        wg.Done()
    }
    message := primitive.NewMessage("test", []byte("Hello RocketMQ Go Client!"))
    err = p.SendAsync(context.Background(), callback, message)
    if err != nil {
        fmt.Printf("send message error: %s\n", err)
        wg.Done()
    }

    wg.Wait()
    err = p.Shutdown()
    if err != nil {
        fmt.Printf("shutdown producer error: %s", err.Error())
    }
}
```

The parameters in the example code are described as follows. For details about how to obtain the parameter values, see **Collecting Connection Information**.

- **192.168.0.1:8100**: instance metadata address and port.
- **test**: topic name.

Subscribing to Normal Messages

The following code is an example. Replace the information in bold with the actual values.

```
package main

import (
    "context"
    "fmt"
    "os"
    "time"

    "github.com/apache/rocketmq-client-go/v2"
```



```
    "github.com/apache/rocketmq-client-go/v2/consumer"
    "github.com/apache/rocketmq-client-go/v2/primitive"
)

func main() {
    c, _ := rocketmq.NewPushConsumer(
        consumer.WithGroupName("testGroup"),
        consumer.WithNsResolver(primitive.NewPassthroughResolver([]string{"192.168.0.1:8100"})),
    )
    err := c.Subscribe("test", consumer.MessageSelector{}, func(ctx context.Context,
        msgs ...*primitive.MessageExt) (consumer.ConsumeResult, error) {
        for i := range msgs {
            fmt.Printf("subscribe callback: %v \n", msgs[i])
        }

        return consumer.ConsumeSuccess, nil
    })
    if err != nil {
        fmt.Println(err.Error())
    }
    // Note: start after subscribe
    err = c.Start()
    if err != nil {
        fmt.Println(err.Error())
        os.Exit(-1)
    }
    time.Sleep(time.Hour)
    err = c.Shutdown()
    if err != nil {
        fmt.Printf("shutdown Consumer error: %s", err.Error())
    }
}
```

The parameters in the example code are described as follows. For details about how to obtain the parameter values, see **Collecting Connection Information**.

- **testGroup**: consumer group name.
- **192.168.0.1:8100**: instance metadata address and port.
- **test**: topic name.

4.2 Sending and Receiving Ordered Messages

In DMS for RocketMQ, ordered messages are retrieved in the exact order that they are created.

Ordered messages are ordered globally or on the partition level.

- Globally ordered messages: There is only one queue in a specific topic. All messages in the queue will be published and subscribed to in the first in, first out (FIFO) order.
- Partition-level ordered message: Messages within a queue in a specific topic are published and subscribed to in the FIFO order. The producer specifies a partition selection algorithm to ensure that the messages to be ordered are allocated to the same queue.

The only difference between globally ordered messages and partition-level ordered messages is the number of queues. The code is the same.

Before sending and receiving ordered messages, collect RocketMQ connection information by referring to **Collecting Connection Information**.

Preparing the Environment

1. Run the following command to check whether Go has been installed:
   ```
   go version
   ```
   If the following information is displayed, Go has been installed:
   ```
   go version go1.16.5 linux/amd64
   ```
   If Go is not installed, **download** and install it.
2. Add the following code to **go.mod** to add the dependency:
   ```
   module rocketmq-example-go

   go 1.13

   require (
       github.com/apache/rocketmq-client-go/v2 v2.1.1
   )
   ```

Sending Ordered Messages

The following code is an example. Replace the information in bold with the actual values.

```
package main

import (
    "context"
    "fmt"
    "os"
    "strconv"

    "github.com/apache/rocketmq-client-go/v2"
    "github.com/apache/rocketmq-client-go/v2/primitive"
    "github.com/apache/rocketmq-client-go/v2/producer"
)

// Package main implements a simple producer to send message.
func main() {
    p, _ := rocketmq.NewProducer(
        producer.WithNsResolver(primitive.NewPassthroughResolver([]string{"192.168.0.1:8100"})),
        producer.WithRetry(2),
    )
    err := p.Start()
    if err != nil {
        fmt.Printf("start producer error: %s", err.Error())
        os.Exit(1)
    }
    topic := "test"

    for i := 0; i < 100; i++ {
        msg := &primitive.Message{
            Topic: topic,
            Body:  []byte("Hello RocketMQ Go Client! " + strconv.Itoa(i)),
        }
        orderId := strconv.Itoa(i % 10)
        msg.WithShardingKey(orderId)
        res, err := p.SendSync(context.Background(), msg)

        if err != nil {
            fmt.Printf("send message error: %s\n", err)
        } else {
            fmt.Printf("send message success: result=%s\n", res.String())
        }
    }
    err = p.Shutdown()
    if err != nil {
        fmt.Printf("shutdown producer error: %s", err.Error())
```



```
    }
}
```

The parameters in the example code are described as follows. For details about how to obtain the parameter values, see **Collecting Connection Information**.

- **192.168.0.1:8100**: instance metadata address and port.
- **test**: topic name.

In the preceding code, to ensure the sequence of messages with the same **orderId**, **orderId** is used as the sharding key of the specific queue.

Subscribing to Ordered Messages

You only need to add **consumer.WithConsumerOrder(true)** to the code for subscribing to normal messages. The following code is an example. Replace the information in bold with the actual values.

```
package main

import (
    "context"
    "fmt"
    "os"
    "time"

    "github.com/apache/rocketmq-client-go/v2"
    "github.com/apache/rocketmq-client-go/v2/consumer"
    "github.com/apache/rocketmq-client-go/v2/primitive"
)

func main() {
    c, _ := rocketmq.NewPushConsumer(
        consumer.WithGroupName("testGroup"),
        consumer.WithNsResolver(primitive.NewPassthroughResolver([]string{"192.168.0.1:8100"})),
        consumer.WithConsumerModel(consumer.Clustering),
        consumer.WithConsumeFromWhere(consumer.ConsumeFromFirstOffset),
        consumer.WithConsumerOrder(true),
    )
    err := c.Subscribe("test", consumer.MessageSelector{}, func(ctx context.Context,
        msgs ...*primitive.MessageExt) (consumer.ConsumeResult, error) {
        orderlyCtx, _ := primitive.GetOrderlyCtx(ctx)
        fmt.Printf("orderly context: %v\n", orderlyCtx)
        fmt.Printf("subscribe orderly callback: %v \n", msgs)
        return consumer.ConsumeSuccess, nil
    })
    if err != nil {
        fmt.Println(err.Error())
    }
    // Note: start after subscribe
    err = c.Start()
    if err != nil {
        fmt.Println(err.Error())
        os.Exit(-1)
    }
    time.Sleep(time.Hour)
    err = c.Shutdown()
    if err != nil {
        fmt.Printf("Shutdown Consumer error: %s", err.Error())
    }
}
```

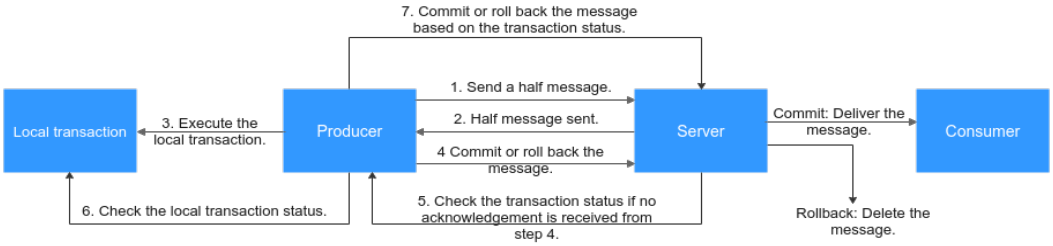
The parameters in the example code are described as follows. For details about how to obtain the parameter values, see **Collecting Connection Information**.

- **testGroup**: consumer group name.
- **192.168.0.1:8100**: instance metadata address and port.
- **test**: topic name.

4.3 Sending and Receiving Transactional Messages

DMS for RocketMQ ensures transaction consistency between the service logic and message transmission, and implements transaction support in two phases. **Figure 4-1** illustrates the interaction of transactional messages.

Figure 4-1 Transactional message interaction



The producer sends a half message and then executes the local transaction. If the execution is successful, the transaction is committed. If the execution fails, the transaction is rolled back. If the server does not receive any commit or rollback request after a period of time, it initiates a check. After receiving the check request, the producer resends a transaction commit or rollback request. The message is delivered to the consumer only after being committed. The consumer is unaware of the rollback.

Before sending and receiving transactional messages, collect RocketMQ connection information by referring to **Collecting Connection Information**.

Preparing the Environment

1. Run the following command to check whether Go has been installed:
   ```
   go version
   ```
   If the following information is displayed, Go has been installed:
   ```
   go version go1.16.5 linux/amd64
   ```
   If Go is not installed, **download** and install it.
2. Add the following code to **go.mod** to add the dependency:
   ```
   module rocketmq-example-go

   go 1.13

   require (
       github.com/apache/rocketmq-client-go/v2 v2.1.1
   )
   ```

Sending Transactional Messages

The following code is an example. Replace the information in bold with the actual values.


```
package main

import (
    "context"
    "fmt"
    "os"
    "strconv"
    "sync"
    "sync/atomic"
    "time"

    "github.com/apache/rocketmq-client-go/v2"
    "github.com/apache/rocketmq-client-go/v2/primitive"
    "github.com/apache/rocketmq-client-go/v2/producer"
)

type DemoListener struct {
    localTrans       *sync.Map
    transactionIndex int32
}

func NewDemoListener() *DemoListener {
    return &DemoListener{
        localTrans: new(sync.Map),
    }
}

func (dl *DemoListener) ExecuteLocalTransaction(msg *primitive.Message) primitive.LocalTransactionState {
    nextIndex := atomic.AddInt32(&dl.transactionIndex, 1)
    fmt.Printf("nextIndex: %v for transactionID: %v\n", nextIndex, msg.TransactionId)
    status := nextIndex % 3
    dl.localTrans.Store(msg.TransactionId, primitive.LocalTransactionState(status+1))

    fmt.Printf("dl")
    return primitive.UnknowState
}

func (dl *DemoListener) CheckLocalTransaction(msg *primitive.MessageExt) primitive.LocalTransactionState
{
    fmt.Printf("%v msg transactionID : %v\n", time.Now(), msg.TransactionId)
    v, existed := dl.localTrans.Load(msg.TransactionId)
    if !existed {
        fmt.Printf("unknow msg: %v, return Commit", msg)
        return primitive.CommitMessageState
    }
    state := v.(primitive.LocalTransactionState)
    switch state {
    case 1:
        fmt.Printf("checkLocalTransaction COMMIT_MESSAGE: %v\n", msg)
        return primitive.CommitMessageState
    case 2:
        fmt.Printf("checkLocalTransaction ROLLBACK_MESSAGE: %v\n", msg)
        return primitive.RollbackMessageState
    case 3:
        fmt.Printf("checkLocalTransaction unknow: %v\n", msg)
        return primitive.UnknowState
    default:
        fmt.Printf("checkLocalTransaction default COMMIT_MESSAGE: %v\n", msg)
        return primitive.CommitMessageState
    }
}

func main() {
    p, _ := rocketmq.NewTransactionProducer(
        NewDemoListener(),
        producer.WithNsResolver(primitive.NewPassthroughResolver([]string{"192.168.0.1:8100"})),
        producer.WithRetry(1),
    )
    err := p.Start()
```



```
    if err != nil {
        fmt.Printf("start producer error: %s\n", err.Error())
        os.Exit(1)
    }

    for i := 0; i < 10; i++ {
        res, err := p.SendMessageInTransaction(context.Background(),
            primitive.NewMessage("topic1", []byte("Hello RocketMQ again "+strconv.Itoa(i))))

        if err != nil {
            fmt.Printf("send message error: %s\n", err)
        } else {
            fmt.Printf("send message success: result=%s\n", res.String())
        }
    }
    time.Sleep(5 * time.Minute)
    err = p.Shutdown()
    if err != nil {
        fmt.Printf("shutdown producer error: %s", err.Error())
    }
}
```

The parameters in the example code are described as follows. For details about how to obtain the parameter values, see **Collecting Connection Information**.

- **192.168.0.1:8100**: instance metadata address and port.
- **topic1**: topic name.

The producer needs to implement two callback functions. The ExecuteLocalTransaction callback function is called after the half message is sent (see step 3 in the diagram). The CheckLocalTransaction callback function is called after the check request is received (see step 6 in the diagram). The two callback functions can return three transaction states:

- **primitive.CommitMessageState**: Transaction committed. The consumer can retrieve the message.
- **primitive.RollbackMessageState**: Transaction rolled back. The message will be discarded and cannot be retrieved.
- **primitive.UnknowState**: The status cannot be determined and the server is expected to check the message status from the producer again.

Subscribing to Transactional Messages

The code for subscribing to transactional messages is the same as that for **subscribing to normal messages**.

4.4 Delivering Scheduled Messages

In DMS for RocketMQ, you can schedule messages to be delivered at **any time**, with a maximum delay of one year.

After being sent from producers to DMS for RocketMQ, scheduled messages are delivered to consumers only after a specified point in time.

Before delivering scheduled messages, collect RocketMQ connection information by referring to **Collecting Connection Information**.

NOTE

This function is supported only for instances created on or after March 30, 2022.

Application Scenarios

Scheduled messages can be used in the following scenarios:

- The service logic requires a time window. For example, an e-commerce order is closed if it is not paid within a period of time. When an order is created, a scheduled message is sent and will be delivered to the consumer five minutes later. After receiving the message, the consumer checks whether the order is paid. If the order is not paid, it is closed. If the order is paid, the message is ignored.
- A scheduled task is triggered by a message. For example, a reminder is sent to a user at a specific time.

Note

- The delivery time can be scheduled to up to one year later. If the delay time exceeds one year, the message cannot be delivered.
- If the delivery time is scheduled to a time point earlier than the current timestamp, the message is immediately sent to the consumer.
- Ideally, the difference between the scheduled delivery time and the actual delivery time is smaller than 0.1s. However, if the pressure of scheduled message delivery is too high, flow control will be triggered, and the precision will deteriorate.
- The message delivery order is not ensured for precision of 0.1s. That is, if the difference between the scheduled delivery time of two messages is smaller than 0.1s, they may not be delivered in the order that they were sent.
- Exactly-once delivery is not guaranteed. A scheduled message may be delivered repeatedly.
- The scheduled time is the time when the server starts to deliver a message to a consumer. If messages are stacked on the consumer, the scheduled message is delivered after the stacked messages, and cannot be delivered exactly at the configured time.
- Due to a potential time difference between the client and server, the actual delivery time may be different from the delivery time set by the client. The server time is used.
- Messages are retained for two days after the scheduled delivery time. For example, if a scheduled message is not retrieved in five days as scheduled, it is deleted on the seventh day.
- Scheduled messages occupy about three times the storage space of normal messages. If you use a large number of scheduled messages, pay attention to the storage space usage.

Preparing the Environment

1. Run the following command to check whether Go has been installed:
   ```
   go version
   ```
   If the following information is displayed, Go has been installed:


```
go version go1.16.5 linux/amd64
```

If Go is not installed, **download** and install it.

2. Add the following code to **go.mod** to add the dependency:

```
module rocketmq-example-go

go 1.13

require (
    github.com/apache/rocketmq-client-go/v2 v2.1.1
)
```

Delivering Scheduled Messages

The following code is an example. Replace the information in bold with the actual values.

```
package main

import (
    "context"
    "fmt"
    "github.com/apache/rocketmq-client-go/v2"
    "github.com/apache/rocketmq-client-go/v2/primitive"
    "github.com/apache/rocketmq-client-go/v2/producer"
    "os"
)

func main() {
    p, _ := rocketmq.NewProducer(
        producer.WithNsResolver(primitive.NewPassthroughResolver([]string{"192.168.0.1:8100"})),
        producer.WithRetry(2),
    )
    err := p.Start()
    if err != nil {
        fmt.Printf("start producer error: %s", err.Error())
        os.Exit(1)
    }
    msg := primitive.NewMessage("test", []byte("Hello RocketMQ Go Client!"))
    msg.WithProperty("__STARTDELIVERTIME", strconv.FormatInt(time.Now().UnixMilli()+3000, 10))
    res, err := p.SendSync(context.Background(), msg)

    if err != nil {
        fmt.Printf("send message error: %s\n", err)
    } else {
        fmt.Printf("send message success: result=%s\n", res.String())
    }
    err = p.Shutdown()
    if err != nil {
        fmt.Printf("shutdown producer error: %s", err.Error())
    }
}
```

The parameters in the example code are described as follows. For details about how to obtain the parameter values, see **Collecting Connection Information**.

- **192.168.0.1:8100**: instance metadata address and port.
- **test**: topic name.

4.5 Controlling Access with ACL

After ACL is enabled for an instance, user authentication information must be added to both the producer and consumer configurations.

Adding User Authentication Information to the Producer

- For normal, ordered, and scheduled messages, add the following code. Replace the information in bold with the actual values.

```
import (
    "context"
    "fmt"
    "os"

    "github.com/apache/rocketmq-client-go/v2"
    "github.com/apache/rocketmq-client-go/v2/primitive"
    "github.com/apache/rocketmq-client-go/v2/producer"
)

func main() {
    p, err := rocketmq.NewProducer(
        producer.WithNsResolver(primitive.NewPassthroughResolver([]string{"192.168.0.1:8100"})),
        producer.WithRetry(2),
        producer.WithCredentials(primitive.Credentials{
            AccessKey: "RocketMQ",
            SecretKey: "12345678",
        }),
    )

    if err != nil {
        fmt.Println("init producer error: " + err.Error())
        os.Exit(0)
    }

    err = p.Start()
    if err != nil {
        fmt.Printf("start producer error: %s", err.Error())
        os.Exit(1)
    }
    res, err := p.SendSync(context.Background(), primitive.NewMessage("test",
        []byte("Hello RocketMQ Go Client!")))

    if err != nil {
        fmt.Printf("send message error: %s\n", err)
    } else {
        fmt.Printf("send message success: result=%s\n", res.String())
    }
    err = p.Shutdown()
    if err != nil {
        fmt.Printf("shutdown producer error: %s", err.Error())
    }
}
```

  The parameters in the example code are described as follows. For details about how to obtain the parameter values, see **Collecting Connection Information**.

  - **192.168.0.1:8100**: instance metadata address and port.
  - **AccessKey**: username. For details about how to create a user, see **Creating a User**.
  - **SecretKey**: secret key of the user.
  - **test**: topic name.

- For transactional messages, add the following code. Replace the information in bold with the actual values.

```
package main

import (
    "context"
    "fmt"
    "os"
    "strconv"
```



```
    "sync"
    "sync/atomic"
    "time"

    "github.com/apache/rocketmq-client-go/v2"
    "github.com/apache/rocketmq-client-go/v2/primitive"
    "github.com/apache/rocketmq-client-go/v2/producer"
)

type DemoListener struct {
    localTrans       *sync.Map
    transactionIndex int32
}

func NewDemoListener() *DemoListener {
    return &DemoListener{
        localTrans: new(sync.Map),
    }
}

func (dl *DemoListener) ExecuteLocalTransaction(msg *primitive.Message)
primitive.LocalTransactionState {
    nextIndex := atomic.AddInt32(&dl.transactionIndex, 1)
    fmt.Printf("nextIndex: %v for transactionID: %v\n", nextIndex, msg.TransactionId)
    status := nextIndex % 3
    dl.localTrans.Store(msg.TransactionId, primitive.LocalTransactionState(status+1))

    fmt.Printf("dl")
    return primitive.UnknowState
}

func (dl *DemoListener) CheckLocalTransaction(msg *primitive.MessageExt)
primitive.LocalTransactionState {
    fmt.Printf("%v msg transactionID : %v\n", time.Now(), msg.TransactionId)
    v, existed := dl.localTrans.Load(msg.TransactionId)
    if !existed {
        fmt.Printf("unknow msg: %v, return Commit", msg)
        return primitive.CommitMessageState
    }
    state := v.(primitive.LocalTransactionState)
    switch state {
    case 1:
        fmt.Printf("checkLocalTransaction COMMIT_MESSAGE: %v\n", msg)
        return primitive.CommitMessageState
    case 2:
        fmt.Printf("checkLocalTransaction ROLLBACK_MESSAGE: %v\n", msg)
        return primitive.RollbackMessageState
    case 3:
        fmt.Printf("checkLocalTransaction unknow: %v\n", msg)
        return primitive.UnknowState
    default:
        fmt.Printf("checkLocalTransaction default COMMIT_MESSAGE: %v\n", msg)
        return primitive.CommitMessageState
    }
}

func main() {
    p, _ := rocketmq.NewTransactionProducer(
        NewDemoListener(),
        producer.WithNsResolver(primitive.NewPassthroughResolver([]string{"192.168.0.1:8100"})),
        producer.WithRetry(2),
        producer.WithCredentials(primitive.Credentials{
            AccessKey: "RocketMQ",
            SecretKey: "12345678",
        }),
    )
    err := p.Start()
    if err != nil {
        fmt.Printf("start producer error: %s\n", err.Error())
```



```
        os.Exit(1)
    }

    for i := 0; i < 10; i++ {
        res, err := p.SendMessageInTransaction(context.Background(),
            primitive.NewMessage("topic1", []byte("Hello RocketMQ again "+strconv.Itoa(i))))

        if err != nil {
            fmt.Printf("send message error: %s\n", err)
        } else {
            fmt.Printf("send message success: result=%s\n", res.String())
        }
    }
    time.Sleep(5 * time.Minute)
    err = p.Shutdown()
    if err != nil {
        fmt.Printf("shutdown producer error: %s", err.Error())
    }
}
```

The parameters in the example code are described as follows. For details about how to obtain the parameter values, see **Collecting Connection Information**.

- **192.168.0.1:8100**: instance metadata address and port.
- **AccessKey**: username. For details about how to create a user, see **Creating a User**.
- **SecretKey**: secret key of the user.
- **topic1**: topic name.

Adding User Authentication Information to the Consumer

Add the following code for normal, ordered, scheduled, and transactional messages. Replace the information in bold with the actual values.

```
package main

import (
    "context"
    "fmt"
    "os"
    "time"

    "github.com/apache/rocketmq-client-go/v2"
    "github.com/apache/rocketmq-client-go/v2/consumer"
    "github.com/apache/rocketmq-client-go/v2/primitive"
)

func main() {
    c, err := rocketmq.NewPushConsumer(
        consumer.WithGroupName("testGroup"),
        consumer.WithNsResolver(primitive.NewPassthroughResolver([]string{"192.168.0.1:8100"})),
        consumer.WithCredentials(primitive.Credentials{
            AccessKey: "RocketMQ",
            SecretKey: "12345678",
        }),
    )
    if err != nil {
        fmt.Println("init consumer error: " + err.Error())
        os.Exit(0)
    }

    err = c.Subscribe("test", consumer.MessageSelector{}, func(ctx context.Context,
        msgs ...*primitive.MessageExt) (consumer.ConsumeResult, error) {
        fmt.Printf("subscribe callback: %v \n", msgs)
        return consumer.ConsumeSuccess, nil
```



```
    })
    if err != nil {
        fmt.Println(err.Error())
    }
    // Note: start after subscribe
    err = c.Start()
    if err != nil {
        fmt.Println(err.Error())
        os.Exit(-1)
    }
    time.Sleep(time.Hour)
    err = c.Shutdown()
    if err != nil {
        fmt.Printf("Shutdown Consumer error: %s", err.Error())
    }
}
```

The parameters in the example code are described as follows. For details about how to obtain the parameter values, see **Collecting Connection Information**.

- **testGroup**: consumer group name.
- **192.168.0.1:8100**: instance metadata address and port.
- **AccessKey**: username. For details about how to create a user, see **Creating a User**.
- **SecretKey**: secret key of the user.
- **test**: topic name.

5 Python

5.1 Sending and Receiving Normal Messages

This section describes how to send and receive normal messages and provides sample code. Normal messages can be sent in the synchronous or asynchronous mode.

- Synchronous transmission: After sending a message, the sender waits for the server to receive and process the message, and does not send the next message until it receives a response from the server.
- Asynchronous transmission: After sending a message, the sender sends the next message without waiting for a response from the server.

The following examples describe only the sample code of synchronous transmission.

Before sending and receiving normal messages, collect RocketMQ connection information by referring to **Collecting Connection Information**.

Preparing the Environment

1. Run the **python** command to check whether Python has been installed. If the following information is displayed, Python has been installed:
   ```
   Python 3.7.1 (default, Jul  5 2020, 14:37:24)
   [GCC 4.8.5 20150623 (Red Hat 4.8.5-39)] on linux
   Type "help", "copyright", "credits" or "license" for more information.
   >>>
   ```
   If Python is not installed, run the following command:
   ```
   yum install python
   ```
2. Install the librocketmq library and rocketmq-client-python. For details, see **rocketmq-client-python**.

   NOTE

   Download **rocketmq-client-cpp-2.2.0** to obtain the librocketmq library.
3. Add **librocketmq.so** to the system's dynamic library search path.
   a. Find the path of **librocketmq.so**.
   ```
   find / -name librocketmq.so
   ```


b. Add **librocketmq.so** to the system's dynamic library search path.

```
ln -s /librocketmq.so_path/librocketmq.so /usr/lib
sudo ldconfig
```

Synchronous Transmission

After sending a message, the sender waits for the server to receive and process the message, and does not send the next message until it receives a response from the server.

The following code is an example. Replace the information in bold with the actual values.

```
from rocketmq.client import Producer, Message

topic = '**TopicTest**'
gid = '**test**'
name_srv = '**192.168.0.1:8100**'


def create_message():
    msg = Message(topic)
    msg.set_keys('XXX')
    msg.set_tags('XXX')
    msg.set_property('property', 'test')
    msg.set_body('message body')
    return msg


def send_message_sync():
    producer = Producer(gid)
    producer.set_name_server_address(name_srv)
    producer.start()
    msg = create_message()
    ret = producer.send_sync(msg)
    print('send message status: ' + str(ret.status) + ' msgId: ' + ret.msg_id)
    producer.shutdown()


if __name__ == '__main__':
    send_message_sync()
```

The parameters in the example code are described as follows. For details about how to obtain the parameter values, see **Collecting Connection Information**.

- **topic**: topic name.
- **gid**: consumer group name.
- **name_srv**: instance metadata address and port.

Subscribing to Normal Messages

The following code is an example. Replace the information in bold with the actual values.

```
import time

from rocketmq.client import PushConsumer, ConsumeStatus


def callback(msg):
    print(msg.id, msg.body, msg.get_property('property'))
    return ConsumeStatus.CONSUME_SUCCESS
```



```
def start_consume_message():
    consumer = PushConsumer('consumer_group')
    consumer.set_name_server_address('192.168.0.1:8100')
    consumer.subscribe('TopicTest', callback)
    print('start consume message')
    consumer.start()

    while True:
        time.sleep(3600)


if __name__ == '__main__':
    start_consume_message()
```

The parameters in the example code are described as follows. For details about how to obtain the parameter values, see **Collecting Connection Information**.

- **consumer_group**: consumer group name.
- **192.168.0.1:8100**: instance metadata address and port.
- **TopicTest**: topic name.

5.2 Sending and Receiving Ordered Messages

In DMS for RocketMQ, ordered messages are retrieved in the exact order that they are created.

Ordered messages are ordered globally or on the partition level.

- Globally ordered messages: There is only one queue in a specific topic. All messages in the queue will be published and subscribed to in the first in, first out (FIFO) order.
- Partition-level ordered message: Messages within a queue in a specific topic are published and subscribed to in the FIFO order. The producer specifies a partition selection algorithm to ensure that the messages to be ordered are allocated to the same queue.

The only difference between globally ordered messages and partition-level ordered messages is the number of queues. The code is the same.

Before sending and receiving ordered messages, collect RocketMQ connection information by referring to **Collecting Connection Information**.

Sending Ordered Messages

The following code is an example. Replace the information in bold with the actual values.

```
from rocketmq.client import Producer, Message

topic = 'TopicTest'
gid = 'test'
name_srv = '192.168.0.1:8100'


def create_message():
    msg = Message(topic)
    msg.set_keys('XXX')
    msg.set_tags('XXX')
    msg.set_property('property', 'test')
```



```
    msg.set_body('message body')
    return msg


def send_orderly_with_sharding_key():
    producer = Producer(gid, True)
    producer.set_name_server_address(name_srv)
    producer.start()
    msg = create_message()
    ret = producer.send_orderly_with_sharding_key(msg, 'orderId')
    print('send message status: ' + str(ret.status) + ' msgId: ' + ret.msg_id)
    producer.shutdown()


if __name__ == '__main__':
    send_orderly_with_sharding_key()
```

The parameters in the example code are described as follows. For details about how to obtain the parameter values, see **Collecting Connection Information**.

- **topic**: topic name.
- **gid**: consumer group name.
- **name_srv**: instance metadata address and port.

In the preceding code, to ensure the sequence of messages with the same **orderId**, **orderId** is used as the sharding key of the specific queue.

Subscribing to Ordered Messages

You only need to add **orderly=True** to the code for subscribing to normal messages. The following code is an example. Replace the information in bold with the actual values.

```
import time

from rocketmq.client import PushConsumer, ConsumeStatus


def callback(msg):
    print(msg.id, msg.body, msg.get_property('property'))
    return ConsumeStatus.CONSUME_SUCCESS


def start_consume_message():
    consumer = PushConsumer('consumer_group', orderly=True)
    consumer.set_name_server_address('192.168.0.1:8100')
    consumer.subscribe('TopicTest', callback)
    print('start consume message')
    consumer.start()

    while True:
        time.sleep(3600)


if __name__ == '__main__':
    start_consume_message()
```

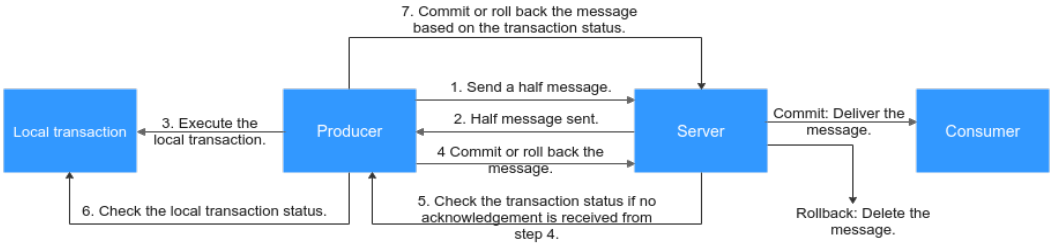
The parameters in the example code are described as follows. For details about how to obtain the parameter values, see **Collecting Connection Information**.

- **consumer_group**: consumer group name.
- **192.168.0.1:8100**: instance metadata address and port.
- **TopicTest**: topic name.

5.3 Sending and Receiving Transactional Messages

DMS for RocketMQ ensures transaction consistency between the service logic and message transmission, and implements transaction support in two phases. **Figure 5-1** illustrates the interaction of transactional messages.

Figure 5-1 Transactional message interaction



The producer sends a half message and then executes the local transaction. If the execution is successful, the transaction is committed. If the execution fails, the transaction is rolled back. If the server does not receive any commit or rollback request after a period of time, it initiates a check. After receiving the check request, the producer resends a transaction commit or rollback request. The message is delivered to the consumer only after being committed. The consumer is unaware of the rollback.

Before sending and receiving transactional messages, collect RocketMQ connection information by referring to **Collecting Connection Information**.

Sending Transactional Messages

The following code is an example. Replace the information in bold with the actual values.

```
import time

from rocketmq.client import Message, TransactionMQProducer, TransactionStatus

topic = 'TopicTest'
gid = 'test'
name_srv = '192.168.0.1:8100'


def create_message():
    msg = Message(topic)
    msg.set_keys('XXX')
    msg.set_tags('XXX')
    msg.set_property('property', 'test')
    msg.set_body('message body')
    return msg


def check_callback(msg):
    print('check: ' + msg.body.decode('utf-8'))
    return TransactionStatus.COMMIT


def local_execute(msg, user_args):
    print('local:   ' + msg.body.decode('utf-8'))
```



```
    return TransactionStatus.UNKNOWN


def send_transaction_message(count):
    producer = TransactionMQProducer(gid, check_callback)
    producer.set_name_server_address(name_srv)
    producer.start()
    for n in range(count):
        msg = create_message()
        ret = producer.send_message_in_transaction(msg, local_execute, None)
        print('send message status: ' + str(ret.status) + ' msgId: ' + ret.msg_id)
    print('send transaction message done')

    while True:
        time.sleep(3600)


if __name__ == '__main__':
    send_transaction_message(10)
```

The parameters in the example code are described as follows. For details about how to obtain the parameter values, see **Collecting Connection Information**.

- **topic**: topic name.
- **gid**: consumer group name.
- **name_srv**: instance metadata address and port.

The producer needs to implement two callback functions. The local_execute callback function is called after the half message is sent (see step 3 in the diagram). The check_callback callback function is called after the check request is received (see step 6 in the diagram). The two callback functions can return three transaction states:

- **TransactionStatus.COMMIT**: Transaction committed. The consumer can retrieve the message.
- **TransactionStatus.ROLLBACK**: Transaction rolled back. The message will be discarded and cannot be retrieved.
- **TransactionStatus.UNKNOWN**: The status cannot be determined and the server is expected to check the message status from the producer again.

Subscribing to Transactional Messages

The code for subscribing to transactional messages is the same as that for **subscribing to normal messages**.

5.4 Delivering Scheduled Messages

In DMS for RocketMQ, you can schedule messages to be delivered at **any time**, with a maximum delay of one year.

After being sent from producers to DMS for RocketMQ, scheduled messages are delivered to consumers only after a specified point in time.

Before delivering scheduled messages, collect RocketMQ connection information by referring to **Collecting Connection Information**.

NOTE

This function is supported only for instances created on or after March 30, 2022.

Application Scenarios

Scheduled messages can be used in the following scenarios:

- The service logic requires a time window. For example, an e-commerce order is closed if it is not paid within a period of time. When an order is created, a scheduled message is sent and will be delivered to the consumer five minutes later. After receiving the message, the consumer checks whether the order is paid. If the order is not paid, it is closed. If the order is paid, the message is ignored.
- A scheduled task is triggered by a message. For example, a reminder is sent to a user at a specific time.

Note

- The delivery time can be scheduled to up to one year later. If the delay time exceeds one year, the message cannot be delivered.
- If the delivery time is scheduled to a time point earlier than the current timestamp, the message is immediately sent to the consumer.
- Ideally, the difference between the scheduled delivery time and the actual delivery time is smaller than 0.1s. However, if the pressure of scheduled message delivery is too high, flow control will be triggered, and the precision will deteriorate.
- The message delivery order is not ensured for precision of 0.1s. That is, if the difference between the scheduled delivery time of two messages is smaller than 0.1s, they may not be delivered in the order that they were sent.
- Exactly-once delivery is not guaranteed. A scheduled message may be delivered repeatedly.
- The scheduled time is the time when the server starts to deliver a message to a consumer. If messages are stacked on the consumer, the scheduled message is delivered after the stacked messages, and cannot be delivered exactly at the configured time.
- Due to a potential time difference between the client and server, the actual delivery time may be different from the delivery time set by the client. The server time is used.
- Messages are retained for two days after the scheduled delivery time. For example, if a scheduled message is not retrieved in five days as scheduled, it is deleted on the seventh day.
- Scheduled messages occupy about three times the storage space of normal messages. If you use a large number of scheduled messages, pay attention to the storage space usage.

Preparing the Environment

1. Run the **python** command to check whether Python has been installed. If the following information is displayed, Python has been installed:

   ```
   Python 3.7.1 (default, Jul  5 2020, 14:37:24)
   [GCC 4.8.5 20150623 (Red Hat 4.8.5-39)] on linux
   Type "help", "copyright", "credits" or "license" for more information.
   >>>
   ```

   If Python is not installed, run the following command:

   ```
   yum install python
   ```


2. Install the librocketmq library and rocketmq-client-python. For details, see **rocketmq-client-python**.

   NOTE

   Download **rocketmq-client-cpp-2.2.0** to obtain the librocketmq library.

3. Add **librocketmq.so** to the system's dynamic library search path.

   a. Find the path of **librocketmq.so**.

   ```
   find / -name librocketmq.so
   ```

   b. Add **librocketmq.so** to the system's dynamic library search path.

   ```
   ln -s /librocketmq.so_path/librocketmq.so /usr/lib
   sudo ldconfig
   ```

Delivering Scheduled Messages

The following code is an example. Replace the information in bold with the actual values.

```
import time

from rocketmq.client import Producer, Message

topic = '**TopicTest**'
gid = '**test**'
name_srv = '**192.168.0.1:8100**'


def create_message():
    msg = Message(topic)
    msg.set_keys('XXX')
    msg.set_tags('XXX')
    msg.set_property('property', 'test')
    msg.set_body('message body')
    return msg


def send_delay_message():
    producer = Producer(gid)
    producer.set_name_server_address(name_srv)
    producer.start()
    msg = create_message()
    msg.set_property('__STARTDELIVERTIME', str(int(round((time.time() + 3) * 1000))))
    ret = producer.send_sync(msg)
    print('send message status: ' + str(ret.status) + ' msgId: ' + ret.msg_id)
    producer.shutdown()


if __name__ == '__main__':
    send_delay_message()
```

The parameters in the example code are described as follows. For details about how to obtain the parameter values, see **Collecting Connection Information**.

- **topic**: topic name.
- **gid**: consumer group name.
- **name_srv**: instance metadata address and port.

5.5 Controlling Access with ACL

After ACL is enabled for an instance, user authentication information must be added to both the producer and consumer configurations.

Adding User Authentication Information to the Producer

- For normal, ordered, and scheduled messages, add the following code. Replace the information in bold with the actual values.

```
from rocketmq.client import Producer, Message

topic = 'TopicTest'
gid = 'test'
name_srv = '192.168.0.1:8100'


def create_message():
    msg = Message(topic)
    msg.set_keys('XXX')
    msg.set_tags('XXX')
    msg.set_property('property', 'test')
    msg.set_body('message body')
    return msg


def send_message_sync():
    producer = Producer(gid)
    producer.set_name_server_address(name_srv)
    producer.set_session_credentials("access_key", "secret_key", "")
    producer.start()
    msg = create_message()
    ret = producer.send_sync(msg)
    print('send message status: ' + str(ret.status) + ' msgId: ' + ret.msg_id)
    producer.shutdown()


if __name__ == '__main__':
    send_message_sync()
```

  The parameters in the example code are described as follows. For details about how to obtain the parameter values, see **Collecting Connection Information**.

  - **topic**: topic name.
  - **gid**: consumer group name.
  - **name_srv**: instance metadata address and port.
  - **access_key**: username. For details about how to create a user, see **Creating a User**.
  - **secret_key**: secret key of the user.

- For transactional messages, add the following code. Replace the information in bold with the actual values.

```
import time

from rocketmq.client import Message, TransactionMQProducer, TransactionStatus

topic = 'TopicTest'
gid = 'test'
name_srv = '192.168.0.1:8100'


def create_message():
```



```
    msg = Message(topic)
    msg.set_keys('XXX')
    msg.set_tags('XXX')
    msg.set_property('property', 'test')
    msg.set_body('message body')
    return msg


def check_callback(msg):
    print('check: ' + msg.body.decode('utf-8'))
    return TransactionStatus.COMMIT


def local_execute(msg, user_args):
    print('local:   ' + msg.body.decode('utf-8'))
    return TransactionStatus.UNKNOWN


def send_transaction_message(count):
    producer = TransactionMQProducer(gid, check_callback)
    producer.set_name_server_address(name_srv)
    producer.set_session_credentials("access_key", "secret_key", "")
    producer.start()
    for n in range(count):
        msg = create_message()
        ret = producer.send_message_in_transaction(msg, local_execute, None)
        print('send message status: ' + str(ret.status) + ' msgId: ' + ret.msg_id)
    print('send transaction message done')

    while True:
        time.sleep(3600)


if __name__ == '__main__':
    send_transaction_message(10)
```

The parameters in the example code are described as follows. For details about how to obtain the parameter values, see **Collecting Connection Information**.

- **topic**: topic name.
- **gid**: consumer group name.
- **name_srv**: instance metadata address and port.
- **access_key**: username. For details about how to create a user, see **Creating a User**.
- **secret_key**: secret key of the user.

Adding User Authentication Information to the Consumer

Add the following code for normal, ordered, scheduled, and transactional messages. Replace the information in bold with the actual values.

```
import time

from rocketmq.client import PushConsumer, ConsumeStatus


def callback(msg):
    print(msg.id, msg.body, msg.get_property('property'))
    return ConsumeStatus.CONSUME_SUCCESS


def start_consume_message():
    consumer = PushConsumer('consumer_group')
    consumer.set_name_server_address('192.168.0.1:8100')
```



```
    consumer.set_session_credentials("access_key", "secret_key", "")
    consumer.subscribe('TopicTest', callback)
    print('start consume message')
    consumer.start()

    while True:
        time.sleep(3600)


if __name__ == '__main__':
    start_consume_message()
```

The parameters in the example code are described as follows. For details about how to obtain the parameter values, see **Collecting Connection Information**.

- **consumer_group**: consumer group name.
- **192.168.0.1:8100**: instance metadata address and port.
- **access_key**: username. For details about how to create a user, see **Creating a User**.
- **secret_key**: secret key of the user.
- **TopicTest**: topic name.

