

Alibaba Cloud

Apsara Stack Enterprise

Alibaba Cloud Message Queue

User Guide

Product Version: v3.16.2

Document Version: 20220816

Legal disclaimer

Alibaba Cloud reminds you to carefully read and fully understand the terms and conditions of this legal disclaimer before you read or use this document. If you have read or used this document, it shall be deemed as your total acceptance of this legal disclaimer.

1. You shall download and obtain this document from the Alibaba Cloud website or other Alibaba Cloud-authorized channels, and use this document for your own legal business activities only. The content of this document is considered confidential information of Alibaba Cloud. You shall strictly abide by the confidentiality obligations. No part of this document shall be disclosed or provided to any third party for use without the prior written consent of Alibaba Cloud.
2. No part of this document shall be excerpted, translated, reproduced, transmitted, or disseminated by any organization, company or individual in any form or by any means without the prior written consent of Alibaba Cloud.
3. The content of this document may be changed because of product version upgrade, adjustment, or other reasons. Alibaba Cloud reserves the right to modify the content of this document without notice and an updated version of this document will be released through Alibaba Cloud-authorized channels from time to time. You should pay attention to the version changes of this document as they occur and download and obtain the most up-to-date version of this document from Alibaba Cloud-authorized channels.
4. This document serves only as a reference guide for your use of Alibaba Cloud products and services. Alibaba Cloud provides this document based on the "status quo", "being defective", and "existing functions" of its products and services. Alibaba Cloud makes every effort to provide relevant operational guidance based on existing technologies. However, Alibaba Cloud hereby makes a clear statement that it in no way guarantees the accuracy, integrity, applicability, and reliability of the content of this document, either explicitly or implicitly. Alibaba Cloud shall not take legal responsibility for any errors or lost profits incurred by any organization, company, or individual arising from download, use, or trust in this document. Alibaba Cloud shall not, under any circumstances, take responsibility for any indirect, consequential, punitive, contingent, special, or punitive damages, including lost profits arising from the use or trust in this document (even if Alibaba Cloud has been notified of the possibility of such a loss).
5. By law, all the contents in Alibaba Cloud documents, including but not limited to pictures, architecture design, page layout, and text description, are intellectual property of Alibaba Cloud and/or its affiliates. This intellectual property includes, but is not limited to, trademark rights, patent rights, copyrights, and trade secrets. No part of this document shall be used, modified, reproduced, publicly transmitted, changed, disseminated, distributed, or published without the prior written consent of Alibaba Cloud and/or its affiliates. The names owned by Alibaba Cloud shall not be used, published, or reproduced for marketing, advertising, promotion, or other purposes without the prior written consent of Alibaba Cloud. The names owned by Alibaba Cloud include, but are not limited to, "Alibaba Cloud", "Aliyun", "HiChina", and other brands of Alibaba Cloud and/or its affiliates, which appear separately or in combination, as well as the auxiliary signs and patterns of the preceding brands, or anything similar to the company names, trade names, trademarks, product or service names, domain names, patterns, logos, marks, signs, or special descriptions that third parties identify as Alibaba Cloud and/or its affiliates.
6. Please directly contact Alibaba Cloud for any errors of this document.

Document conventions

Style	Description	Example
 Danger	A danger notice indicates a situation that will cause major system changes, faults, physical injuries, and other adverse results.	 Danger: Resetting will result in the loss of user configuration data.
 Warning	A warning notice indicates a situation that may cause major system changes, faults, physical injuries, and other adverse results.	 Warning: Restarting will cause business interruption. About 10 minutes are required to restart an instance.
 Notice	A caution notice indicates warning information, supplementary instructions, and other content that the user must understand.	 Notice: If the weight is set to 0, the server no longer receives new requests.
 Note	A note indicates supplemental instructions, best practices, tips, and other content.	 Note: You can use Ctrl + A to select all files.
>	Closing angle brackets are used to indicate a multi-level menu cascade.	Click Settings > Network > Set network type .
Bold	Bold formatting is used for buttons, menus, page names, and other UI elements.	Click OK .
<code>Courier font</code>	Courier font is used for commands	Run the <code>cd /d C:/window</code> command to enter the Windows system folder.
<i>Italic</i>	Italic formatting is used for parameters and variables.	<code>bae log list --instanceid</code> <i>Instance_ID</i>
[] or [a b]	This format is used for an optional value, where only one item can be selected.	<code>ipconfig [-all -t]</code>
{ } or {a b}	This format is used for a required value, where only one item can be selected.	<code>switch {active stand}</code>

Table of Contents

1.What is Message Queue for Apache RocketMQ?	10
2.Updates	11
3.Quick start	13
3.1. Overview	13
3.2. Log on to the Message Queue for Apache RocketMQ cons... ..	15
3.3. Create resources	16
3.4. Send messages	18
3.4.1. Use the TCP client SDK for Java to send and subscribe... ..	18
3.4.2. Use the HTTP client SDK for Java to send and subscri... ..	21
3.4.3. Check whether messages are sent	25
3.5. Subscribe to messages	25
4.Message types	27
4.1. Normal messages	27
4.2. Scheduled messages and delayed messages	27
4.3. Transactional messages	29
4.4. Ordered messages	31
5.Console guide	35
5.1. Resource management	35
5.1.1. Resource management overview	35
5.1.2. Manage instances	35
5.1.3. Manage topics	36
5.1.4. Manage groups	38
5.2. Message query	40
5.2.1. Overview	40
5.2.2. Query messages	41
5.2.3. Query results	42

5.3. Message tracing	43
5.3.1. Overview	43
5.3.2. Query message traces	45
5.3.3. Status in message traces	47
5.4. View the consumer status	48
5.5. Reset consumer offsets	50
5.6. Dead-letter queues	51
5.7. Resource statistics	54
5.7.1. Overview	54
5.7.2. Query the statistics of produced messages	54
5.7.3. Query the statistics of consumed messages	55
5.8. Account authorization management	56
5.9. Switch between different access modes	58
5.10. Bind a VPC to a Message Queue for Apache RocketMQ in... ..	58
5.11. Route messages from a cluster to another cluster	61
6.SDK user guide	69
6.1. Overview	69
6.2. SDK user guide	69
6.2.1. Demo projects	70
6.2.1.1. Overview	70
6.2.1.2. Prepare the environment	70
6.2.1.3. Configure a demo project	70
6.2.1.4. Run the demo project	72
6.2.2. Client parameters	73
6.2.3. Client error codes	80
6.2.4. SDK for Java	84
6.2.4.1. Usage notes	85
6.2.4.2. Prepare the environment	87

6.2.4.3. Configure logging	87
6.2.4.4. Spring integration	90
6.2.4.4.1. Overview	90
6.2.4.4.2. Integrate a producer with Spring	90
6.2.4.4.3. Integrate a transactional message producer with... ..	92
6.2.4.4.4. Integrate a consumer with Spring	94
6.2.4.5. Three modes for sending messages	96
6.2.4.5.1. Overview	96
6.2.4.5.2. Reliable synchronous transmission	97
6.2.4.5.3. Reliable asynchronous transmission	99
6.2.4.5.4. One-way transmission	101
6.2.4.6. Send messages by using multiple threads	103
6.2.4.7. Send and subscribe to ordered messages	105
6.2.4.8. Send and subscribe to transactional messages	108
6.2.4.9. Send and subscribe to delayed messages	112
6.2.4.10. Send and subscribe to scheduled messages	114
6.2.4.11. Subscribe to messages	115
6.2.5. SDK for C or C++	118
6.2.5.1. Prepare the SDK for C or C++ environment	118
6.2.5.1.1. Overview	118
6.2.5.1.2. Download SDK for C++	118
6.2.5.1.3. Use SDK for C++ in Linux	119
6.2.5.2. Send and subscribe to normal messages	119
6.2.5.3. Send and subscribe to ordered messages	119
6.2.5.4. Send and subscribe to scheduled messages	122
6.2.5.5. Send and subscribe to transactional messages	124
6.2.5.6. Subscribe to messages	126
6.2.6. SDK for .NET	128

6.2.6.1. .Prepare the SDK for .NET environment	128
6.2.6.1.1. Overview	128
6.2.6.1.2. Download SDK for .NET	128
6.2.6.1.3. .Configure SDK for .NET	129
6.2.6.2. Send and subscribe to normal messages	135
6.2.6.3. Send and subscribe to ordered messages	136
6.2.6.4. Send and subscribe to scheduled messages	139
6.2.6.5. Send and subscribe to transactional messages	140
6.2.6.6. Subscribe to messages	145
6.3. HTTP client SDK reference	147
6.3.1. Protocol description	147
6.3.1.1. Common parameters	147
6.3.1.2. Request signatures	148
6.3.1.3. Operation for sending messages	149
6.3.1.4. Operation for consuming messages	151
6.3.1.5. Operation for acknowledging messages	156
6.3.2. Java SDK	159
6.3.2.1. Prepare the environment	159
6.3.2.2. Send and consume normal messages	160
6.3.2.3. Send and consume ordered messages	164
6.3.2.4. Send and consume scheduled messages and delaye... ..	168
6.3.2.5. Send and consume transactional messages	172
6.3.3. Go SDK	178
6.3.3.1. Prepare the environment	178
6.3.3.2. Send and consume normal messages	179
6.3.3.3. Send and consume ordered messages	183
6.3.3.4. Send and consume scheduled messages and delaye... ..	187
6.3.3.5. Send and consume transactional messages	191

6.3.4. Python SDK	197
6.3.4.1. Prepare the environment	197
6.3.4.2. Send and consume normal messages	198
6.3.4.3. Send and consume ordered messages	201
6.3.4.4. Send and consume scheduled messages and delaye... ..	205
6.3.4.5. Send and consume transactional messages	208
6.3.5. Node.js SDK	213
6.3.5.1. Prepare the environment	214
6.3.5.2. Send and consume normal messages	214
6.3.5.3. Send and consume ordered messages	217
6.3.5.4. Send and consume scheduled messages and delaye... ..	221
6.3.5.5. Send and consume transactional messages	225
6.3.6. PHP SDK	230
6.3.6.1. Prepare the environment	230
6.3.6.2. Send and consume normal messages	231
6.3.6.3. Send and consume ordered messages	235
6.3.6.4. Send and consume scheduled messages and delaye... ..	239
6.3.6.5. Send and consume transactional messages	243
6.3.7. C# SDK	248
6.3.7.1. Prepare the environment	248
6.3.7.2. Send and consume normal messages	249
6.3.7.3. Send and consume ordered messages	253
6.3.7.4. Send and consume scheduled messages and delaye... ..	257
6.3.7.5. Send and consume transactional messages	261
6.3.8. C++ SDK	268
6.3.8.1. Prepare the environment	268
6.3.8.2. Send and consume normal messages	271
6.3.8.3. Send and consume ordered messages	275

6.3.8.4. Send and consume scheduled messages and delaye...	279
6.3.8.5. Send and consume transactional messages	283
7. Best practices	291
7.1. Clustering consumption and broadcasting consumption	291
7.2. Message filtering	293
7.3. Subscription consistency	295
7.4. Consumption idempotence	298
7.5. Active geo-redundancy	299
7.6. Message routing	301
8. Service usage FAQ	304
8.1. FAQ	304
8.1.1. Quick start	304
8.1.2. Configurations	305
8.1.3. Message tracing	306
8.1.4. Alert handling	307
8.1.5. Ordered messages	307
8.2. Exceptions	308
8.2.1. Usage-related exceptions	308
8.2.2. Nonexistent resources	310
8.2.3. Inconsistent status	311
8.3. Troubleshooting	314
8.3.1. Unexpected consumer connections	314
8.3.2. Inconsistent subscriptions	315
8.3.3. Message accumulation	317
8.3.4. Message accumulation in Java processes	318
8.3.5. Application OOM due to message caching on the client	319
8.3.6. AuthenticationException reported due to failure in sen...	320

1. What is Message Queue for Apache RocketMQ?

Message Queue for Apache RocketMQ is a distributed messaging middleware that is developed based on Apache RocketMQ. Message Queue for Apache RocketMQ features low latency, high concurrency, high availability, and high reliability.

Message Queue for Apache RocketMQ provides a complete set of cloud messaging services based on the technologies that are used for building highly available and distributed clusters. The messaging services include message subscription and publishing, message tracing, scheduled and delayed messages, and resource statistics. Message Queue for Apache RocketMQ is used as a core service in an enterprise-grade Internet architecture. Message Queue for Apache RocketMQ provides asynchronous decoupling and peak-load shifting capabilities for distributed application systems. It also supports various features for Internet applications, including accumulation of large numbers of messages, high throughput, and reliable message consumption retries. Message Queue for Apache RocketMQ is one of the core cloud services that are used to support the Double 11 Shopping Festival.

Message Queue for Apache RocketMQ supports connections over TCP and HTTP and supports multiple programming languages such as Java, C++, and .NET. This allows you to connect applications that are developed in different programming languages to Message Queue for Apache RocketMQ.

2. Updates

This topic describes the updates of Message Queue for Apache RocketMQ from V3.8.0 to V3.8.1 to help you get started with the updated version.

Optimization of resource isolation by instance

Message Queue for Apache RocketMQ provides instances for multi-tenancy isolation. Each user can purchase multiple instances and logically isolate them from each other.

To ensure the compatibility with the existing resources of existing users, Message Queue for Apache RocketMQ provides the following types of instances and namespaces:

- Default instances, which are compatible with the existing resources of existing users
 - This type of instance has no separate namespace. Resource names must be globally unique within and across all instances.
 - By default, an instance without a namespace is automatically generated for the existing resources of each existing user. If no existing resources are available, you can create at most one instance without a namespace.
 - You can configure the endpoint, which can be obtained from the **Instances** page in the Message Queue for Apache RocketMQ console.

```
// Recommended configuration:
properties.put(PropertyKeyConst.NAMESRV_ADDR, "xxxx");
// Compatible configuration, which is not recommended. We recommend that you update this configuration to the recommended configuration:
properties.put(PropertyKeyConst.ONSAddr, "xxxx");
```

- New instances
 - A new instance has a separate namespace. Resource names must be unique within an instance but can be the same across different instances.
 - You can configure the endpoint, which can be obtained from the **Instances** page in the Message Queue for Apache RocketMQ console.

```
// Recommended configuration:
properties.put(PropertyKeyConst.NAMESRV_ADDR, "xxx");
```

- A RocketMQ client must be updated to the following latest versions for different programming languages:
 - Java: **V1.8.7.1.Final**
 - C and C++: **V2.0.0**
 - .NET: **V1.1.3**

Optimization of resource application

Previously, Message Queue for Apache RocketMQ resources consisted of topics, producer IDs, and consumer IDs. Each two of the resources have a many-to-many relationship, which was difficult to comprehend. Each time you created a topic, you must associate the topic with a producer ID and a consumer ID. This process was too complex for medium- and large-sized enterprise customers.

To optimize user experience and help new users get started, the resource application process has been simplified.

The resource application process has been optimized in the following aspects:

- Topic management, which is unchanged
 - You need to apply for a topic. A topic is used to classify messages. It is the primary classifier.
- Group management
 - You do not need to apply for a producer ID. Producer IDs and consumer IDs are integrated into group IDs. In the Message Queue for Apache RocketMQ console, the Producers module has been removed. The Producers and Consumers modules have been integrated into the Groups module.
 - You do not need to associate a producer ID or consumer ID with a topic. Instead, you need only to apply for a group ID and associate it with a topic in the code.
 - Compatibility:
 - The list of producer IDs is no longer displayed. This does not affect the current services.
 - The consumer IDs that start with CID- or CID_ and that you have applied for can still be used and can be set in the `PropertyKeyConst.ConsumerId` or `PropertyKeyConst.GROUP_ID` parameter of the code.
- Sample code

 **Note**

- We recommend that you update a RocketMQ client to the following latest versions for different programming languages:
 - Java: **V1.8.7.1.Final**
 - C and C++: **V2.0.0**
 - .NET: **V1.1.3**
- Existing producer IDs or consumer IDs can still be used and do not affect the current services. However, we recommend that you update your instance configuration to the recommended configuration.

- Recommended configuration: Integrate producer IDs and consumer IDs into group IDs.

```
// Set the PropertyKeyConst.GROUP_ID parameter. The original PropertyKeyConst.ProducerId and PropertyKeyConst.ConsumerId parameters are deprecated.
properties.put(PropertyKeyConst.GROUP_ID, "The original CID-XXX or the GID-XXX");
```

- Compatible configuration: Use a producer ID to identify a producer and a consumer ID to identify a consumer.

```
// When you create a producer, you must set the PropertyKeyConst.ProducerId parameter.
properties.put(PropertyKeyConst.ProducerId, "The original PID-XXX or the GID-XXX");
// When you create a consumer, you must set the PropertyKeyConst.ConsumerId parameter.
properties.put(PropertyKeyConst.ConsumerId, "The original CID-XXX or the GID-XXX");
```

3. Quick start

3.1. Overview

Message Queue for Apache RocketMQ provides TCP client SDKs and HTTP client SDKs for multiple programming languages. You can use the SDKs to send and subscribe to different types of messages. This topic describes how to use TCP client SDKs and HTTP client SDKs for multiple programming languages to send and subscribe to normal messages and the relevant usage notes.

Background information

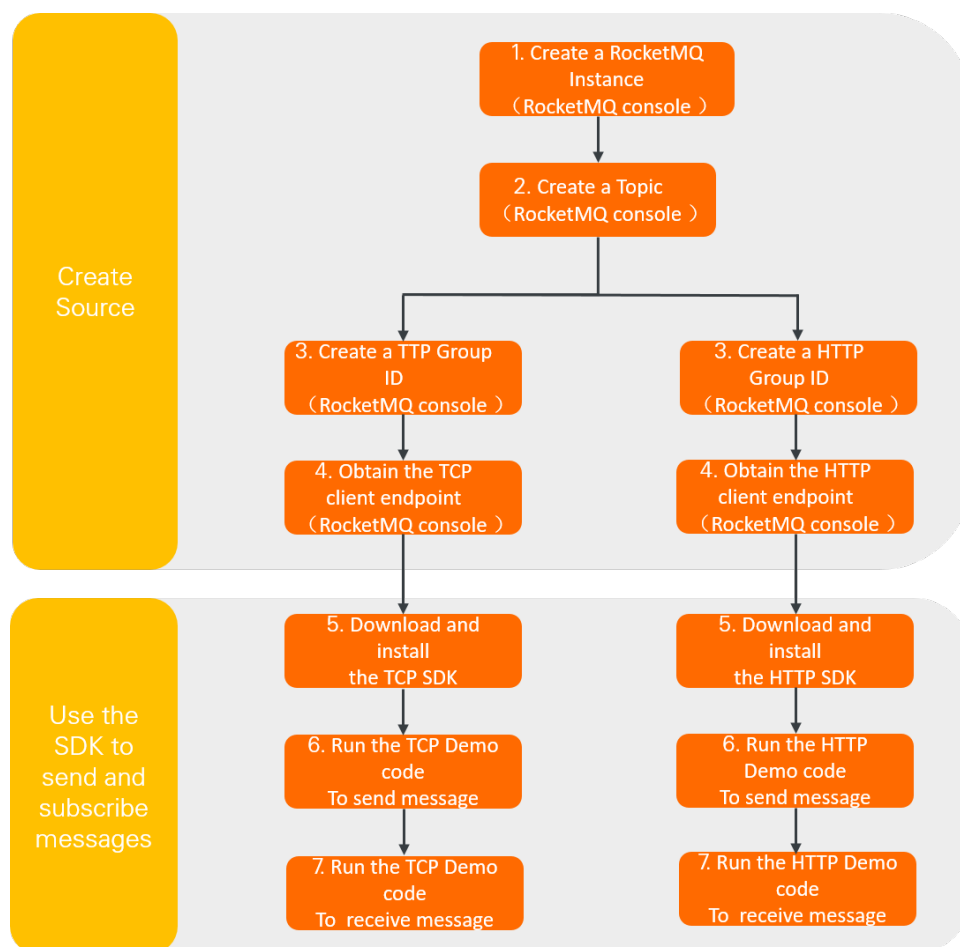
If your application that is deployed on a server uses Message Queue for Apache RocketMQ, we recommend that you use an SDK to access Message Queue for Apache RocketMQ. This method is easy-to-use and provides high availability.

This topic provides examples to show how to use the SDK for Java to connect to Message Queue for Apache RocketMQ and send and subscribe to messages over TCP or HTTP.

Message Queue for Apache RocketMQ supports four types of messages. For more information, see [Message types](#). In the following examples, normal messages are used. The topics that you create by using the procedure that is described in the following section cannot be used to send or subscribe to other types of messages. Each topic can be used to send and subscribe to messages only of a specific type.

Process

You can follow the process illustrated in the following figure based on the protocol that you select.



1. **Create resources.** You must create a Message Queue for Apache RocketMQ instance, a topic, and a group, and obtain the endpoint information of the instance.
2. Use the corresponding SDK to send and subscribe to messages based on the protocol that you select.
 - [Call SDKs to send messages](#)
 - [Use the HTTP client SDK for Java to send and subscribe to normal messages](#)

Usage notes

- Message Queue for Apache RocketMQ provides TCP client SDKs and HTTP client SDKs for you to send and consume messages. You cannot specify the same group ID in the code of a TCP client SDK and the code of an HTTP client SDK at the same time. If you want to use a TCP client SDK to send and consume messages, you must create a group for the TCP protocol. You cannot specify a group that is created for the HTTP protocol in the code of the TCP client SDK.
- A Message Queue for Apache RocketMQ instance provides a TCP endpoint and an HTTP endpoint. An endpoint for a specific protocol must be used together with an SDK for the same protocol. For example, if you want to use a TCP client SDK to send and consume messages, you must obtain the TCP endpoint of your Message Queue for Apache RocketMQ instance. You cannot use the HTTP endpoint to connect to the instance.

References

For information about how to use TCP client SDKs and HTTP client SDKs for other programming languages to send and subscribe to messages, see the following topics:

- TCP
 - C and C++: [Send and receive normal messages](#)
 - .NET: [Send and subscribe to normal messages](#)
- HTTP
 - Go: [Send and consume normal messages](#)
 - Python: [Send and consume normal messages](#)
 - Node.js: [Send and consume normal messages](#)
 - PHP: [Send and consume normal messages](#)
 - C#: [Send and consume normal messages](#)
 - C++: [Send and consume normal messages](#)

3.2. Log on to the Message Queue for Apache RocketMQ console

This topic describes how to log on to the Message Queue for Apache RocketMQ console.


Prerequisites

- The URL of the Apsara Uni-manager Management Console is obtained from the deployment personnel before you log on to the Apsara Uni-manager Management Console.
- A browser is available. We recommend that you use the Google Chrome browser.

Procedure

1. In the address bar, enter the URL of the Apsara Uni-manager Management Console. Press the Enter key.
2. Enter your username and password.

Obtain the username and password that you can use to log on to the console from the operations administrator.

 **Note** When you log on to the Apsara Uni-manager Management Console for the first time, you must change the password of your username. Your password must meet complexity requirements. The password must be 8 to 20 characters in length and must contain at least two of the following character types:


- Uppercase or lowercase letters
- Digits
- Special characters, which include ! @ # \$ %

3. Click **Login**.
4. If your account has multi-factor authentication (MFA) enabled, perform corresponding operations in the following scenarios:
 - It is the first time that you log on to the console after MFA is forcibly enabled by the

administrator.

- a. On the Bind Virtual MFA Device page, bind an MFA device.
 - b. Enter the account and password again as in Step 2 and click **Log On**.
 - c. Enter a six-digit MFA verification code and click **Authenticate**.
- o You have enabled MFA and bound an MFA device.

Enter a six-digit MFA authentication code and click **Authenticate**.

 **Note** For more information, see the *Bind a virtual MFA device to enable MFA* topic in *Alpsara Uni-manager Operations Console User Guide*.

5. In the top navigation bar, choose **Products > Middleware > Message Queue**.


3.3. Create resources

Before you use a client SDK to send and subscribe to messages, create the required resources and obtain the resource information in the Message Queue for Apache RocketMQ console. When you use the SDK, you must configure the resource parameters based on the resource information.


Context

If you want to connect a new application to Message Queue for Apache RocketMQ, you must create the following resources for the application:

- **Instance:** a virtual machine that provides the Message Queue for Apache RocketMQ service. An instance stores topics and group IDs.
- **Topic:** a topic of messages. In Message Queue for Apache RocketMQ, a producer sends a message to a specified topic, and a consumer subscribes to the topic to consume the message.
- **Group ID:** a group ID that is used to identify a group of producers or consumers.

 **Notice** A TCP client cannot share a group ID with an HTTP client. You must create a group for each of them. For example, if you want to use the TCP client SDK to send and subscribe to messages, you must use the group that is created only for TCP clients.

- **Endpoint:** an endpoint of the Message Queue for Apache RocketMQ broker. You can use an endpoint to connect producer or consumer clients to a specified Message Queue for Apache RocketMQ instance.

 **Notice** A Message Queue for Apache RocketMQ instance has a TCP endpoint and an HTTP endpoint. Each endpoint can be used only for clients over the same protocol. For example, if you want to use the TCP client SDK to send and subscribe to messages, you must specify the TCP endpoint in the SDK. You cannot use the HTTP endpoint in the SDK.

- **AccessKey ID and AccessKey secret:** the user credentials that are used to verify the identity of the user. For more information, see the *Obtain an AccessKey pair* topic of *Message Queue for Apache RocketMQ Developer Guide*.

Create an instance

1. Log on to the Message Queue for Apache RocketMQ console and click **Instances** in the left-side

navigation pane. For information about how to log on to the Message Queue for Apache RocketMQ console, see [Log on to the Message Queue for Apache RocketMQ console](#).

2. On the **Instances** page, click **Create Instance**.
3. On the **Create Instance** page, configure the parameters and click **Submit**.
4. In the message that appears, click **Back to Console**.
On the **Instances** page, you can view the basic information about the instance that is created.

Obtain an endpoint

1. In the left-side navigation pane of the Message Queue for Apache RocketMQ console, click **Instances**.
2. In the upper part of the **Instances** page, select the name of the instance that you want to view.
3. Click the **Network Management** tab. On this tab, view the endpoint information of the instance.
 - **TCP Endpoint**: If you want to use the TCP client SDK to send and subscribe to messages, specify the TCP endpoint in the code.
 - **HTTP Endpoint**: If you want to use the HTTP client SDK to send and subscribe to messages, specify the HTTP endpoint in the code.

Create a topic

1. [Log on to the Message Queue for Apache RocketMQ console](#). In the left-side navigation pane, click **Topics**.
2. In the upper part of the **Topics** page, select the instance that you want to manage.
3. Click **Create Topic**.
4. In the **Create Topic** dialog box, enter a name for the topic in the **Topic** field.
5. From the **Message Type** drop-down list, select a message type. Your topic is used to send and subscribe to messages of this type.

In this example, **Normal Message** is selected to create a topic. For more information about other message types, see [Message types](#).

6. In the **Description** field, enter a description about the topic. Then, click **OK**.
The topic that you created appears in the topic list.

Create a group

1. Log on to the [Message Queue for Apache RocketMQ console](#) and click **Groups** in the left-side navigation pane.
2. On the **Groups** page, click the name of the Message Queue for Apache RocketMQ instance in which you want to create a group.
3. Select a protocol for the group that you want to create.
 - If you want to use TCP-based SDKs to publish and consume messages, you must click the **TCP Protocol** tab to create a group.
 - If you want to use HTTP-based SDKs to publish and consume messages, you must click the **HTTP Protocol** tab to create a group.
 - Click **Create Group ID**.
 - In the **Create Group ID** dialog box, configure the **Group ID** parameter and **Description** parameter. Then, click **OK**.

The group that is created appears in the group list.

3.4. Send messages

3.4.1. Use the TCP client SDK for Java to send and subscribe to normal messages

After you create the required resources in the Message Queue for Apache RocketMQ console, you can use Message Queue for Apache RocketMQ TCP client SDK for Java to send and subscribe to normal messages.

Before you begin

Create resources

Install Message Queue for Apache RocketMQ SDK for Java

You can use one of the following methods to install Message Queue for Apache RocketMQ SDK for Java:

- Introduce a dependency by using Maven:

```
<dependency>
  <groupId>com.aliyun.openservices</groupId>
  <artifactId>ons-client</artifactId>
  <!-- Set the value to the version of Message Queue for Apache RocketMQ SDK for Java. -
->
  <version>"XXX"</version>
</dependency>
```

- Download a JAR file that contains a dependency

For more information about the download link, see [Overview](#).

Use the TCP client SDK for Java to send normal messages

The following sample code provides an example on how to use the TCP client SDK for Java to send normal messages. Before you run the code, we recommend that you specify the information about the required resources that are created in advance based on the comments included in the code.

```
import com.aliyun.openservices.ons.api.Message;
import com.aliyun.openservices.ons.api.Producer;
import com.aliyun.openservices.ons.api.SendResult;
import com.aliyun.openservices.ons.api.ONSTFactory;
import com.aliyun.openservices.ons.api.PropertyKeyConst;
import java.util.Properties;

public class ProducerTest {
    public static void main(String[] args) {
        Properties properties = new Properties();
        // Specify the ID of the group that you created for TCP clients in the Message Queue for Apache RocketMQ console.
        properties.put(PropertyKeyConst.GROUP_ID, "XXX");
        // Specify the AccessKey ID for identity verification.
        properties.put(PropertyKeyConst.AccessKey, "XXX");
        // Specify the AccessKey secret for identity verification.
```

```

    // Specify the AccessKey secret for identity verification.
    properties.put(PropertyKeyConst.SecretKey, "XXX");
    // Specify the TCP endpoint of your instance. To view the TCP endpoint, log on to
    the Message Queue for Apache RocketMQ console and go to the Network Management tab of the I
    nstances page.
    properties.put(PropertyKeyConst.NAMESRV_ADDR, "XXX");
    Producer producer = ONSFactory.createProducer(properties);
    // Before you send a message, call the start() method only once to start the produ
    cer.

    producer.start();
    // Cyclically send messages.
    while(true){
        Message msg = new Message(
            // Specify the topic that you created in the Message Queue for Apache Rock
            etMQ console. The value is the name of the topic to which you want to send messages.
            "TopicTestMQ",
            // Message Tag,
            // Specify the message tag, which is similar to a Gmail tag. The message t
            ag is used to sort messages and filter messages for the consumer on the Message Queue for A
            pache RocketMQ broker based on specified conditions.
            "TagA",
            // Message Body
            // Specify the message body in the binary format. Message Queue for Apache
            RocketMQ does not process the message body.
            // The producer and consumer must agree on the message serialization and d
            eserialization methods.
            "Hello MQ".getBytes());
        // Specify the message key. The message key is the business-specific attribute
        of the message and must be globally unique. A unique key helps you query and resend a messa
        ge in the console if the message fails to be consumed.
        // Note: You can send and subscribe to messages even if you do not specify mes
        sage keys.
        msg.setKey("ORDERID_100");
        // Send the message. If no exception is thrown, the message is sent.
        // Print the message ID. The message ID can be used to query the sending statu
        s of the message.
        SendResult sendResult = producer.send(msg);
        System.out.println("Send Message success. Message ID is: " + sendResult.getMes
        sageId());
    }
    // Before you exit the application, shut down the producer.
    // Note: This step is optional.
    producer.shutdown();
}
}

```

You can also send messages by performing the following operations in the Message Queue for Apache RocketMQ console: [Log on to the Message Queue for Apache RocketMQ console](#). In the left-side navigation pane, click **Topics**. On the **Topics** page, find the topic that you created, and click **Send Message** in the **Actions** column.

Use the TCP client SDK for Java to subscribe to normal messages

The following sample code provides an example on how to use the TCP client SDK for Java to subscribe to normal messages. Before you run the code, we recommend that you specify the information about the required resources that are created in advance based on the comments included in the code.

```
import com.aliyun.openservices.ons.api.Action;
import com.aliyun.openservices.ons.api.ConsumeContext;
import com.aliyun.openservices.ons.api.Consumer;
import com.aliyun.openservices.ons.api.Message;
import com.aliyun.openservices.ons.api.MessageListener;
import com.aliyun.openservices.ons.api.ONSFactory;
import com.aliyun.openservices.ons.api.PropertyKeyConst;
import java.util.Properties;
public class ConsumerTest {
    public static void main(String[] args) {
        Properties properties = new Properties();
        // Specify the ID of the group that you created for TCP clients in the Message Queue
        // for Apache RocketMQ console.
        properties.put(PropertyKeyConst.GROUP_ID, "XXX");
        // Specify the AccessKey ID for identity verification.
        properties.put(PropertyKeyConst.AccessKey, "XXX");
        // Specify the AccessKey secret for identity verification.
        properties.put(PropertyKeyConst.SecretKey, "XXX");
        // Specify the TCP endpoint of your instance. To view the TCP endpoint, log on to
        // the Message Queue for Apache RocketMQ console and go to the Network Management tab of the I
        // nstances page.
        properties.put(PropertyKeyConst.NAMESRV_ADDR, "XXX");
        Consumer consumer = ONSFactory.createConsumer(properties);
        consumer.subscribe("TopicTestMQ", "*", new MessageListener() {
            public Action consume(Message message, ConsumeContext context) {
                System.out.println("Receive: " + message);
                return Action.CommitMessage;
            }
        });
        consumer.start();
        System.out.println("Consumer Started");
    }
}
```

Check whether your message subscription is successful

1. [Log on to the Message Queue for Apache RocketMQ console](#). In the left-side navigation pane, click **Groups**.
2. In the upper part of the **Groups** page, select the instance that you want to manage.
3. On the **Groups** page, find the group ID for the consumer of which you want to view the subscription, and click **Subscription** in the **Actions** column.

If the value of **Online** is **Yes**, the consumer has been started and the subscription is successful. Otherwise, the subscription fails.

What's next

- [Query messages](#)
- [Query message traces](#)

3.4.2. Use the HTTP client SDK for Java to send and subscribe to normal messages

After you create the required resources in the Message Queue for Apache RocketMQ console, you can use Message Queue for Apache RocketMQ HTTP client SDK for Java to send and subscribe to normal messages.

Before you begin

Create resources

Install Message Queue for Apache RocketMQ SDK for Java

You can use one of the following methods to install Message Queue for Apache RocketMQ SDK for Java:

- Introduce a dependency by using Maven:

```
<dependency>
  <groupId>com.aliyun.mq</groupId>
  <artifactId>mq-http-sdk</artifactId>
  <!-- Set the value to the version of Message Queue for Apache RocketMQ SDK for Java.
  -->
  <version>X.X.X</version>
  <classifier>jar-with-dependencies</classifier>
</dependency>
```

- Download a JAR file that contains a dependency

For more information about the download link, see [Overview](#).

Use the HTTP client SDK for Java to send normal messages

The following sample code provides an example on how to use the HTTP client SDK for Java to send normal messages. Before you run the code, we recommend that you specify the information about the required resources that are created in advance based on the comments included in the code.

```
import com.aliyun.mq.http.MQClient;
import com.aliyun.mq.http.MQProducer;
import com.aliyun.mq.http.model.TopicMessage;
import java.util.Date;

public class Producer {
    public static void main(String[] args) {
        MQClient mqClient = new MQClient(
            // Specify the HTTP endpoint of your instance. To view the HTTP endpoint, log on to the Message Queue for Apache RocketMQ console and go to the Network Management tab of the Instances page.
            "${HTTP_ENDPOINT}",
            // Specify the AccessKey ID for identity verification.
            "${ACCESS_KEY}",
            // Specify the AccessKey secret for identity verification.
            "${SECRET_KEY}"
        );
        // Specify the topic that you created in the Message Queue for Apache RocketMQ console
```

```

    // The value is the name of the topic to which you want to send messages.
    final String topic = "${TOPIC}";
    // Specify the ID of the instance on which the topic is created.
    // If the instance has a namespace, specify the ID of the instance. If the instance
    // does not have a namespace, set the instance ID to null or an empty string. You can check wh
    // ether your instance has a namespace on the Instances page in the Message Queue for Apache R
    // ocketMQ console.
    final String instanceId = "${INSTANCE_ID}";
    // Obtain the producer that sends messages to the topic.
    MQProducer producer;
    if (instanceId != null && instanceId != "") {
        producer = mqClient.getProducer(instanceId, topic);
    } else {
        producer = mqClient.getProducer(topic);
    }
    try {
        // Cyclically send four messages.
        for (int i = 0; i < 4; i++) {
            TopicMessage pubMsg;          // Specify the normal message.
            pubMsg = new TopicMessage(
                // Specify the content of the message.
                "hello mq!".getBytes(),
                // Specify the message tag.
                "A"
            );
            // Specify the custom attributes of the message.
            pubMsg.getProperties().put("a", String.valueOf(i));
            // Specify the key of the message.
            pubMsg.setMessageKey("MessageKey");
            // Send the message in synchronous mode. If no exception is thrown, the message
            // is sent.
            TopicMessage pubResultMsg = producer.publishMessage(pubMsg);
            // Send the message in synchronous mode. If no exception is thrown, the message
            // is sent.
            System.out.println(new Date() + " Send mq message success. Topic is:" + topic +
                ", msgId is: " + pubResultMsg.getMessageId()
                + ", bodyMD5 is: " + pubResultMsg.getMessageBodyMD5());
        }
    } catch (Throwable e) {
        // Specify the logic that you want to use to resend or persist the message if t
        // he message fails to be sent and needs to be sent again.
        System.out.println(new Date() + " Send mq message failed. Topic is:" + topic);
        e.printStackTrace();
    }
    mqClient.close();
}
}

```

You can also send messages by performing the following operations in the Message Queue for Apache RocketMQ console: [Log on to the Message Queue for Apache RocketMQ console](#). In the left-side navigation pane, click **Topics**. On the **Topics** page, find the topic that you created, and click **Send Message** in the **Actions** column.

Use the HTTP client SDK for Java to subscribe to normal messages

The following sample code provides an example on how to use the HTTP client SDK for Java to subscribe to normal messages. Before you run the code, we recommend that you specify the information about the required resources that are created in advance based on the comments included in the code.

```
import com.aliyun.mq.http.MQClient;
import com.aliyun.mq.http.MQConsumer;
import com.aliyun.mq.http.common.AckMessageException;
import com.aliyun.mq.http.model.Message;
import java.util.ArrayList;
import java.util.List;
public class Consumer {
    public static void main(String[] args) {
        MQClient mqClient = new MQClient(
            // Specify the HTTP endpoint of your instance. To view the HTTP endpoint, log on to the Message Queue for Apache RocketMQ console and go to the Network Management tab of the Instances page.
            "${HTTP_ENDPOINT}",
            // Specify the AccessKey ID for identity verification.
            "${ACCESS_KEY}",
            // Specify the AccessKey secret for identity verification.
            "${SECRET_KEY}"
        );
        // Specify the topic that you created in the Message Queue for Apache RocketMQ console. The value is the name of the topic from which you want to consume messages.
        final String topic = "${TOPIC}";
        // Specify the ID of the group that you created for HTTP clients in the Message Queue for Apache RocketMQ console.
        final String groupId = "${GROUP_ID}";
        // Specify the ID of the instance on which the topic is created.
        // If the instance has a namespace, specify the ID of the instance. If the instance does not have a namespace, set the instance ID to null or an empty string. You can check whether your instance has a namespace on the Instances page in the RocketMQ console.
        final String instanceId = "${INSTANCE_ID}";
        final MQConsumer consumer;
        if (instanceId != null && instanceId != "") {
            consumer = mqClient.getConsumer(instanceId, topic, groupId, null);
        } else {
            consumer = mqClient.getConsumer(topic, groupId);
        }
        // Cyclically consume messages in the current thread. We recommend that you use multiple threads to concurrently consume messages.
        do {
            List<Message> messages = null;
            try {
                // Consume messages in long polling mode.
                // In long polling mode, if no message in the topic is available for consumption, the request is suspended on the broker for a specified period of time. If a message becomes available for consumption within this period, the broker immediately sends a response to the consumer. In this example, the period is set to 3 seconds.
                messages = consumer.consumeMessage(
                    3, // Specify the maximum number of messages that can be consumed at
```

a time. In this example, the value is set to 3. The maximum value that you can specify is 16.

3// Specify the duration of a long polling cycle. Unit: seconds. In this example, the value is set to 3. The maximum value that you can specify is 30.

```

    );
    } catch (Throwable e) {
        e.printStackTrace();
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e1) {
            e1.printStackTrace();
        }
    }
    // No messages in the topic are available for consumption.
    if (messages == null || messages.isEmpty()) {
        System.out.println(Thread.currentThread().getName() + ": no new message, continue!");
        continue;
    }
    // Specify the message consumption logic.
    for (Message message : messages) {
        System.out.println("Receive message: " + message);
    }
    // If the broker does not receive an acknowledgment (ACK) for a message from the consumer before the delivery retry interval elapses, the broker sends the message for consumption again.
    // A unique timestamp is specified for the handle of a message each time the message is consumed.
    {
        List<String> handles = new ArrayList<String>();
        for (Message message : messages) {
            handles.add(message.getReceiptHandle());
        }
        try {
            consumer.ackMessage(handles);
        } catch (Throwable e) {
            // If the handle of a message times out, the broker cannot receive an ACK for the message from the consumer.
            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() + ", ErrorMessage:" + errors.getErrorMessages().get(errorHandle).getErrorMessage());
                    }
                }
                continue;
            }
            e.printStackTrace();
        }
    }
}

```

```
    } while (true);  
  }  
}
```

Check whether your message subscription is successful

1. [Log on to the Message Queue for Apache RocketMQ console](#). In the left-side navigation pane, click **Groups**.
2. In the upper part of the **Groups** page, select the instance that you want to manage.
3. On the Groups page, find the group ID for the consumer of which you want to view the subscription, and click **Subscription** in the **Actions** column.

If the value of **Online** is **Yes**, the consumer has been started and the subscription is successful. Otherwise, the subscription fails.

What's next

- [Query messages](#)
- [Query message traces](#)


3.4.3. Check whether messages are sent

After you send a message, you can check the status of the message in the console.

Procedure

1. [Log on to the Message Queue for Apache RocketMQ console](#). In the left-side navigation pane, click **Message Query**.
2. On the **Message Query** page, click the **By Message ID** tab.
3. In the search box, enter the topic name that corresponds to the message and the message ID returned after the message is sent, and click **Search** to query the sending status of the message.

Storage Time indicates the time when the Message Queue for Apache RocketMQ broker stores the message. If the message appears in the search results, the message has been sent to the Message Queue for Apache RocketMQ broker.

 **Notice** This step demonstrates the situation where Message Queue for Apache RocketMQ is used for the first time and the consumer has never been started. Therefore, no consumption data appears in the message status information.

What's next

You can start the consumer and subscribe to messages. For more information, see [Subscribe to messages](#). For more information about the message status, see [Query messages](#) and [Message tracing status](#).

3.5. Subscribe to messages

After a message is sent, the consumer can subscribe to the message. You need to use the SDK for the corresponding protocol and programming language to subscribe to the message. This topic describes how to subscribe to messages by using TCP client SDK for Java.

Procedure

1. Run the following sample code to test the message subscription feature. Set parameters based on the descriptions before you run the code.

```
import com.aliyun.openservices.ons.api.Action;
import com.aliyun.openservices.ons.api.ConsumeContext;
import com.aliyun.openservices.ons.api.Consumer;
import com.aliyun.openservices.ons.api.Message;
import com.aliyun.openservices.ons.api.MessageListener;
import com.aliyun.openservices.ons.api.ONSFactory;
import com.aliyun.openservices.ons.api.PropertyKeyConst;
import java.util.Properties;

public class ConsumerTest {
    public static void main(String[] args) {
        Properties properties = new Properties();
        // The group ID that you created in the console.
        properties.put(PropertyKeyConst.GROUP_ID, "XXX");
        // The AccessKey ID used for identity authentication.
        properties.put(PropertyKeyConst.AccessKey, "XXX");
        // The AccessKey secret used for identity authentication.
        properties.put(PropertyKeyConst.SecretKey, "XXX");
        // The TCP endpoint. To obtain the endpoint, log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click Instance Details. In the upper part of the Instance Details page, select your instance. On the Instance Information tab, view the endpoint in the Obtain Endpoint Information section.
        properties.put(PropertyKeyConst.NAMESRV_ADDR, "XXX");
        Consumer consumer = ONSFactory.createConsumer(properties);
        consumer.subscribe("TopicTestMQ", "*", new MessageListener() {
            public Action consume(Message message, ConsumeContext context) {
                System.out.println("Receive: " + message);
                return Action.CommitMessage;
            }
        });
        consumer.start();
        System.out.println("Consumer Started");
    }
}
```

After you run the code, you can check whether the consumer is started in the Message Queue for Apache RocketMQ console. This operation checks whether the message subscription is successful.

2. Log on to the Message Queue for Apache RocketMQ console.
3. In the left-side navigation pane, click **Groups**.
4. Find the group ID of the consumer whose subscription you want to view, and click **Subscription** in the **Actions** column.

If the value of **Online** is **Yes**, the consumer has been started and the subscription is successful. Otherwise, the subscription fails.

4.Message types

4.1. Normal messages

This topic describes the definition of normal messages and provides the sample code for sending and subscribing to normal messages.

In Message Queue for Apache RocketMQ, normal messages do not have special features. They are different from the following featured messages: scheduled messages, delayed messages, ordered messages, and transactional messages. For more information about these featured messages, see [Scheduled messages and delayed messages](#), [Ordered messages](#), and [Transactional messages](#). The following sample code provide examples on how to use TCP client SDKs and HTTP client SDKs for different programming languages to send and subscribe to normal messages:

Sample code for using TCP client SDKs

For information about the sample code for using TCP client SDKs to send and subscribe to normal messages, see the following topics:

- The SDK for Java:
 - [Overview](#)
 - [Send messages in multiple threads](#)
 - [Subscribe to messages](#)
- The SDK for C and the SDK for C++: [Send and receive normal messages](#)
- The SDK for .NET: [Send and subscribe to normal messages](#)

Sample code for using HTTP client SDKs

For information about the sample code for using HTTP client SDKs to send and subscribe to normal messages, see the following topics:

- The SDK for Java: [Send and subscribe to normal messages](#)
- The SDK for Go: [Send and subscribe to normal messages](#)
- The SDK for Python: [Send and subscribe to normal messages](#)
- The SDK for Node.js: [Send and subscribe to normal messages](#)
- The SDK for PHP: [Send and subscribe to normal messages](#)
- The SDK for C#: [Send and subscribe to normal messages](#)
- The SDK for C++: [Send and subscribe to normal messages](#)

4.2. Scheduled messages and delayed messages

This topic introduces the concepts related to scheduled messages and delayed messages and describes the scenarios, usage, and usage notes of these messages in Message Queue for Apache RocketMQ.

Concepts

- **Scheduled message:** A scheduled message is not immediately sent to consumers after the Message Queue for Apache RocketMQ broker receives the message from the producer. The broker is configured to send the message to consumers at a specific point in time later than the current time.
- **Delayed message:** A delayed message is not immediately sent to consumers after the Message Queue for Apache RocketMQ broker receives the message. The broker is configured to send the message to consumers after a specific period of time elapses.

Scheduled messages and delayed messages are slightly different in coding, but are consistent in the effect. Scheduled messages and delayed messages are not immediately sent to consumers after the Message Queue for Apache RocketMQ broker receives the messages. These messages are sent to consumers after being delayed for a specific period of time, which is specified in the attributes of the messages.

Scenarios

Scheduled messages and delayed messages can be used in the following scenarios:

- A time window between message production and message consumption is required. For example, when a transaction order is created on an e-commerce platform, a producer sends a delayed message to the Message Queue for Apache RocketMQ broker. A delay of 30 minutes is specified for the message to be sent to a consumer. The message is used to remind the consumer to check whether the order is paid. If the order is not paid, the related system closes the order. If the order is paid, the consumer ignores the message.
- Scheduled messages are sent to trigger scheduled tasks. For example, a notification message is sent to a user at a specified point in time.

Usage

Scheduled messages and delayed messages are slightly different in coding.

- To send a scheduled message, specify a point in time that is later than the point in time when the message is sent by a producer. The broker sends the message to consumers at the specified point in time.
- To send a delayed message, specify a period of time that starts from the point in time when the message is sent by a producer. The broker sends the message to consumers after the specified period of time elapses.

Usage notes

- The `msg.setStartDeliverTime` parameter for a scheduled message or a delayed message must be set to a specific point in time after the current timestamp. Unit: milliseconds. If the scheduled time is earlier than the current time, the message is immediately sent to the consumer.
- The `msg.setStartDeliverTime` parameter for a scheduled message or a delayed message can be set to a specific point in time within 40 days after the scheduled message or the delayed message is generated. Unit: milliseconds. If the specified point in time is not within the 40 days, the message cannot be sent.
- The `StartDeliverTime` parameter specifies the time when the Message Queue for Apache RocketMQ broker starts to send the message to the consumer. If messages have been accumulated for the consumer, the scheduled message or the delayed message are queued after the accumulated messages, and are not sent to the consumer at the specified time.
- Due to the potential time difference between the producer and the broker, the actual delivery time may be different from the delivery time specified in the producer.
- Scheduled messages and delayed messages can be retained for at most three days on the Message

Queue for Apache RocketMQ broker. For example, a message is scheduled to be consumed in 5 days. If the message is not consumed after 5 days, the message will be deleted on the eighth day.

Sample code for using TCP client SDKs

For information about the sample code for using TCP client SDKs to send and subscribe to scheduled messages or delayed messages, see the following topics:

- Java SDK
 - [Send and receive scheduled messages](#)
 - [Send and receive delayed messages](#)
- The SDK for C++: [Send and receive scheduled messages](#)
- The SDK for .NET: [Send and subscribe to scheduled messages](#)

Sample code for using HTTP client SDKs

For information about the sample code for using HTTP client SDKs to send and subscribe to scheduled messages and delayed messages, see the following topics:

- The SDK for Java: [Send and subscribe to scheduled messages and delayed messages](#)
- The SDK for Go: [Send and consume scheduled messages and delayed messages](#)
- The SDK for Python: [Send and consume scheduled messages and delayed messages](#)
- The SDK for Node.js: [Send and consume scheduled messages and delayed messages](#)
- The SDK for PHP: [Send and consume scheduled messages and delayed messages](#)
- The SDK for C#: [Send and consume scheduled messages and delayed messages](#)
- The SDK for C++: [Send and consume scheduled messages and delayed messages](#)

4.3. Transactional messages

This topic introduces the terms that are related to transactional messages in Message Queue for Apache RocketMQ. This topic also describes the scenarios, methods, and usage notes of using transactional messages.

Terms

- Transactional message: Message Queue for Apache RocketMQ provides a distributed transaction processing feature that is similar to X/Open XA to ensure transaction consistency by using transactional messages.
- Half transactional message: A half transactional message is a message that cannot be sent to consumers by the Message Queue for Apache RocketMQ broker. If a Message Queue for Apache RocketMQ broker receives a message from a producer and does not receive the second acknowledgment (ACK) from the producer, the message is temporarily undeliverable. A message in this state is called a half transactional message.
- Message status check: In scenarios such as a transient connection occurs in the network or the producer application is restarted, the Message Queue for Apache RocketMQ broker does not receive the second ACK for a transactional message. When the Message Queue for Apache RocketMQ broker finds that a message remains as a half transactional message for an excessive long period of time, the broker sends a request to the producer to check whether the final status of the message is Commit or Rollback.

Common scenarios

Message Queue for Apache RocketMQ allows you to use transactional messages in the following scenarios:

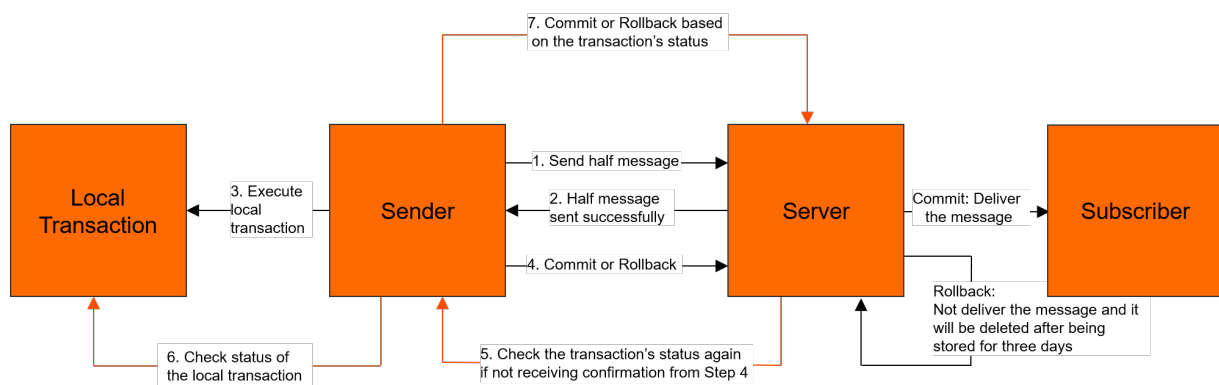
The distributed transaction processing feature provided by Message Queue for Apache RocketMQ can be used to ensure transaction consistency based on transactional messages. The distributed transaction processing feature is similar to X/Open XA.

For example, an e-commerce platform provides a shopping cart system and a transaction system. When customers use the shopping cart to place orders, the customer entry point is the shopping cart system, and the entry point for order placement is the transaction system. Data in the two systems must be eventually consistent. In this case, transactional messages can be used. After an order is placed, the transaction system sends a transactional message about the order to Message Queue for Apache RocketMQ. The shopping cart system subscribes to the transactional message about the order from Message Queue for Apache RocketMQ, performs the required operations, and then updates data.

How to use transactional messages

Interaction process

The following figure shows the process of using transactional messages in Message Queue for Apache RocketMQ.



The procedure to send a transactional message includes the following steps:

1. A producer sends a message to the Message Queue for Apache RocketMQ broker.
2. The Message Queue for Apache RocketMQ broker persists the message and sends an ACK to the producer. At this stage, the message is a half transactional message.
3. The producer executes a local transaction.
4. The producer sends an ACK to the Message Queue for Apache RocketMQ broker to submit the execution result of the local transaction. In the execution result, the status of the transaction may be Commit or Rollback. If the status of the transaction is Commit, the broker marks the half transactional message as deliverable. Then, consumers can consume the message. If the status of the transaction is Rollback, the broker deletes the message. In this case, consumers cannot consume the message.

The following list describes how to check the status of a transactional message:

- If the Message Queue for Apache RocketMQ broker does not receive an ACK from the producer because the network is disconnected or the message producer application is restarted, the Message Queue for Apache RocketMQ broker sends a request to query the status of the message after a specific period of time.

- After the producer receives the request, the producer checks the final status of the local transaction that corresponds to the message.
- The producer sends a new ACK to the Message Queue for Apache RocketMQ broker based on the final status of the local transaction. The broker processes the half transactional message based on the content of the ACK. For more information, see Step 4.

Usage notes

1. Producers of transactional messages cannot be in the same group to which producers of messages of other types belong. Message Queue for Apache RocketMQ brokers perform status check operations for transactional messages based on the IDs of the groups to which the producers belong.
2. You must specify the implementation class of the `LocalTransactionChecker` method when you call `ONSFactory.createTransactionProducer` to create a transactional message producer. This way, the broker can check the status of transactional messages after exceptions occur.
3. After the local transaction that corresponds to a transactional message is executed, the `execute` method returns one of the following results:
 - `TransactionStatus.CommitTransaction`: The transaction is committed. The message can be consumed by consumers.
 - `TransactionStatus.RollbackTransaction`: The transaction is rolled back. The message is discarded and cannot be consumed.
 - `TransactionStatus.Unknown`: The transaction is in an unknown state. The Message Queue for Apache RocketMQ broker is expected to send a request to the producer to check the status of the transaction again after a specific period of time.

Sample code for using TCP SDKs

For information about sample code, see the following references:

- SDK for Java: [Send and subscribe to transactional messages](#)
- SDK for C and C++: [Send and receive transactional messages](#)
- SDK for .NET: [Send and receive transactional messages](#)

Sample code for using HTTP-based SDKs

For information about sample code, see the following references:

- SDK for Java: [Send and consume transactional messages](#)
- SDK for Go: [Send and consume transactional messages](#)
- SDK for Python: [Send and consume transactional messages](#)
- SDK for Node.js: [Send and consume transactional messages](#)
- SDK for PHP: [Send and consume transactional messages](#)
- SDK for C#: [Send and consume transactional messages](#)
- SDK for C++: [Send and consume transactional messages](#)

4.4. Ordered messages

This topic introduces the terms that are related to ordered messages in Message Queue for Apache RocketMQ. This topic also describes the scenarios and usage notes of using ordered messages.

Terms

Ordered messages in Message Queue for Apache RocketMQ are messages that are published and consumed strictly based on specific orders. Ordered messages are also known as first-in-first-out (FIFO) messages. Ordered messages are published and consumed in FIFO order.

An ordered message involves ordered publishing and ordered consumption.

- **Ordered publishing:** Each producer sends messages to a specified topic in FIFO order.
- **Ordered consumption:** Each consumer consumes messages in a specified topic in FIFO order. A message that is first sent is first consumed by consumers.

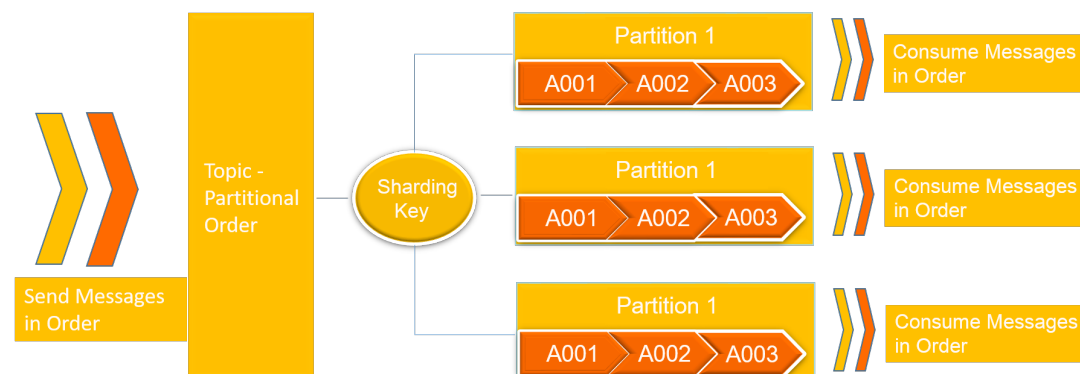
Ordered messages are classified into globally ordered messages and partitionally ordered messages.

- **Globally ordered messages:** All messages in a topic are published and consumed in FIFO order.
- **Partitionally ordered messages:** Messages in a specified topic are partitioned based on a partition key. Messages in each partition are published and consumed in FIFO order. The partition key for ordered messages in a topic is used to distinguish message partitions. Partition keys are used in a manner that is different from the manner in which message keys of normal messages are used.

Globally ordered messages



Partitionally ordered messages



Common scenarios

- **Globally ordered messages**

Your business does not require high performance and requires that all messages must be published and consumed in FIFO order.

- **Partitionally ordered messages**

Your business requires high performance and requires that messages in each partition must be published and consumed in FIFO order. In this case, you can specify a partition key field to partition messages to multiple partitions.

Examples:

- Example 1

When a user signs up with your application, a verification code is used to verify the identity of the user. In this case, you can use the field that stores user IDs as the partition key. This way, messages that are sent by the same user are published and consumed in FIFO order.

- Example 2

To partition messages that are generated when orders are made on an e-commerce platform, you can use the field that stores order IDs as the partition key. This way, order creation messages, payment messages, refund messages, and logistics messages of the same order are published and consumed in FIFO order.

All internal e-commerce systems of Alibaba Group use partitionally ordered messages. This ensures that messages in each partition are in FIFO order and the systems provide high performance.

Comparison between globally ordered messages and partitionally ordered messages

You are required to create different types of topics for different types of messages in the Message Queue for Apache RocketMQ console. The following table describes the comparison among types of topics.

Message types

Topic type	Support transactional messages	Support scheduled messages	Performance
Unordered messages, including normal messages, transactional messages, scheduled messages, and delayed messages	Yes	Yes	Highest
Partitionally ordered messages	No	No	High
Globally ordered messages	No	No	Medium

Methods of sending messages

Message type	Support reliable synchronous transmission	Support reliable asynchronous transmission	Support one-way transmission
Unordered messages, including normal messages, transactional messages, scheduled messages, and delayed messages	Yes	Yes	Yes
Partitionally ordered messages	Yes	No	No

Message type	Support reliable synchronous transmission	Support reliable asynchronous transmission	Support one-way transmission
Globally ordered messages	Yes	No	No

Usage notes

- Ordered messages cannot be published in a broadcasting manner.
- A producer or consumer can publish messages to or consume messages from only one type of topic. A producer or consumer cannot be used to publish or consume both ordered messages and unordered messages.
- Ordered messages cannot be sent in asynchronous mode. If ordered messages are sent in asynchronous mode, messages may be disordered.
- If you want to use globally ordered messages, we recommend that you create at least two Message Queue for Apache RocketMQ instances. The multi-instance structure is used to prevent your business from being interrupted when the primary instance unexpectedly fails. When the primary instance fails, another instance immediately takes over the workloads. This helps ensure that your business is not interrupted. In the multi-instance structure, only one instance works at a time.

Sample code for using TCP SDKs

For information about sample code, see the following references:

- SDK for Java: [Send and receive ordered messages](#)
- SDK for C and C++: [Send and receive ordered messages](#)
- SDK for .NET: [Send and subscribe to ordered messages](#)

Sample code for using HTTP-based SDKs

For information about sample code, see the following references:

- SDK for Java: [Send and consume ordered messages](#)
- SDK for Go: [Send and consume ordered messages](#)
- SDK for Python: [Send and consume ordered messages](#)
- SDK for Node.js: [Send and consume ordered messages](#)
- SDK for PHP: [Send and consume ordered messages](#)
- SDK for C#: [Send and consume ordered messages](#)
- SDK for C++: [Send and consume ordered messages](#)

5. Console guide

5.1. Resource management

5.1.1. Resource management overview

This topic describes how to manage resources in Message Queue for Apache RocketMQ.

If a new application needs to access Message Queue for Apache RocketMQ, you must create the following Message Queue for Apache RocketMQ resources for the application:

- **Instance:** As a virtual machine (VM) resource of Message Queue for Apache RocketMQ, an instance stores the topics and group IDs of messages.
- **Topic:** In Message Queue for Apache RocketMQ, a producer sends a message to a specified topic, and a consumer subscribes to the topic to obtain and consume the message.
- **Group ID:** A group ID is used to identify a group of message consumers or producers.

You can add, delete, modify, and query these resources by using the Message Queue for Apache RocketMQ console or by calling the Message Queue for Apache RocketMQ API.

When you use SDKs to send and subscribe to messages, you must specify the topic and group ID that you created in the Message Queue for Apache RocketMQ console. You must also enter the AccessKey ID and AccessKey secret that you created in the Apsara Uni-manager Management Console for identity authentication.

If you have not obtained the AccessKey ID and AccessKey secret, you can obtain them in the Apsara Uni-manager Management Console. For more information, see [Obtain the AccessKey ID and AccessKey secret](#).

5.1.2. Manage instances

In Message Queue for Apache RocketMQ, topics and groups are included in instances. This topic describes how to create, update, view, and delete an instance in the Message Queue for Apache RocketMQ console.

Create an instance

1. Log on to the Message Queue for Apache RocketMQ console and click **Instances** in the left-side navigation pane. For information about how to log on to the Message Queue for Apache RocketMQ console, see [Log on to the Message Queue for Apache RocketMQ console](#).
2. On the **Instances** page, click **Create Instance**.
3. On the **Create Instance** page, configure the parameters and click **Submit**.
4. In the message that appears, click **Back to Console**.
On the **Instances** page, you can view the basic information about the instance that is created.


Modify the configuration of an instance

You can upgrade or downgrade the specification of an instance.

1. In the left-side navigation pane of the Message Queue for Apache RocketMQ console, click **Instances**.
2. On the **Instances** page, click the name of the instance that you want to modify. Then, click

Update Specifications.

3. On the **Update Specifications** page, configure the **Maximum Topics**, **Outbound Message TPS**, **Inbound Message TPS**, and **Description** parameters.

 **Note** The value that you specify for each parameter must be in the range that is displayed.

4. Click **Submit**.
5. In the message that appears, click **Back to Console**.
On the **Instances** page, you can view the new configuration of the instance.

View the details of an instance

1. In the left-side navigation pane of the Message Queue for Apache RocketMQ console, click **Instances**.
2. On the **Instances** page, click the name of the instance that you want to view. You can view information about the instance on the details page of the instance.

Delete an instance

Prerequisites

- All resources in the instance are deleted, including topics and groups.
 - No Message Queue for MQTT instances are bound to the instance.
1. In the left-side navigation pane of the Message Queue for Apache RocketMQ console, click **Instances**.
 2. On the **Instances** page, click the name of the instance that you want to delete. Then, click **Delete Instance**.
 3. In the message that appears, read the message and click **OK**.
After the instance is deleted, a message whose content is **The instance is deleted.** is displayed

References

If you want to call API operations to perform the operations that are described in this topic, see *Message Queue for Apache RocketMQ Developer Guide*.

5.1.3. Manage topics

Topic is the first-level identifier that classifies messages in Message Queue for Apache RocketMQ. For example, you can create a topic named Topic_Trade to identify transaction-specific messages. This topic describes how to create, update, view, and delete topics in the Message Queue for Apache RocketMQ console.

Prerequisites

[Create an instance](#)

Create a topic

1. [Log on to the Message Queue for Apache RocketMQ console](#). In the left-side navigation pane, click **Topics**.

2. In the upper part of the **Topics** page, select your instance.
3. Click **Create Topic**.
4. In the **Create Topic** dialog box, enter a name for the topic in the **Topic** field.



Notice The topic name must be unique in the instance where you create the topic and must comply with the following rules:


- The topic name cannot start with CID or GID, because CID and GID are reserved fields for group IDs.
- The topic name can contain only letters, digits, hyphens (-), and underscores (_).
- The topic name must be 3 to 64 characters in length.

5. From the **Message Type** drop-down list, select a value. This value defines the type of message that this topic sends and receives.

We recommend that you create different topics to send different types of messages. For example, create Topic A for normal messages, Topic B for transactional messages, and Topic C for scheduled messages or delayed messages. For more information about message types, see [Message types](#).

6. In the **Description** field, enter a description about the topic and click **OK**.
The created topic appears in the topic list.

Modify the description of a topic

1. Log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click **Topics**.
2. In the upper part of the **Topics** page, select your instance.
3. In the topic list, find the topic whose description you want to modify and click the  icon in the **Description** column.
4. In the **Edit Topic** dialog box, enter the new description and click **OK**.
The message **The operation is successful.** appears.

View topic information


1. Log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click **Topics**.
2. In the upper part of the **Topics** page, click the name of your instance. You can view all topics in the instance and the details about a specific topic. The details include the subscription, permissions, and message type.

Delete a topic



Note After a topic is deleted, producers that send messages to the topic and consumers that subscribe to the topic immediately stop services and all resources are deleted within 10 minutes. Proceed with caution.

1. Log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click **Topics**.
2. In the upper part of the **Topics** page, select your instance.

3. Find the topic that you want to delete, click the  icon, and then select Delete.
4. In the **Caution** message, read the prompt carefully. If you are sure to delete the topic, click **OK**.
The topic no longer appears in the topic list in the instance.

References

If you need to call the Message Queue for Apache RocketMQ API to perform relevant operations, follow the instructions provided in *Message Queue for Apache RocketMQ Developer Guide*.

5.1.4. Manage groups

After you create an instance and a topic, you need to create a group for message consumers or producers. This topic describes how to create, view, and delete a group in the Message Queue for Apache RocketMQ console.

Prerequisites

[Create an instance](#)

Context

Producers or consumers in a group produce or consume messages of the same type based on the same logic. To use Message Queue for Apache RocketMQ to produce or consume messages, you must create a group to identify producer instances or consumer instances of the same type.

A consumer can subscribe to multiple topics, and a topic can be subscribed to by multiple consumers in a group. A producer can send messages to multiple topics, and a topic can subscribe to multiple producers in a group to receive messages.

Usage notes

- A group cannot be used across instances. For example, a group created in Instance A is unavailable in Instance B.
- All clients in a group communicate with Message Queue for Apache RocketMQ brokers over the same protocol. Message Queue for Apache RocketMQ allows you to use HTTP-based SDKs and TCP-based SDKs to produce and consume messages. If you specify TCP as the protocol when you create a group, clients in the group can use only TCP-based SDKs to send and receive messages.
- If a consumer group or an existing consumer that was created in an earlier version of Message Queue for Apache RocketMQ was created by using the credential of a Resource Access Management (RAM) user, the RAM user and the Apsara Stack tenant account to which the RAM user belongs can use the consumer group or consumer.
- If a consumer group or an existing consumer that was created in an earlier version of Message Queue for Apache RocketMQ was created by using the credential of an Apsara Stack tenant account, only the Apsara Stack tenant account can use the consumer group or consumer. RAM users of this Apsara Stack tenant account cannot use the consumer group or consumer.
- For information about how to modify existing configurations of clients to change consumer IDs and producer IDs to group IDs, see [Updates](#).

Rules for naming group IDs

- The ID of a group must start with CID or GID, and contain 7 to 64 characters in length. A group ID can consist of only letters, digits, hyphens (-), and underscores (_).

- If a group belongs to an instance with a namespace, the ID of the group must be unique within the instance. Group IDs in different instances can be the same. For example, the ID of a group in Instance A can be the same as ID of a group in Instance B.
- If a group belongs to an instance without a namespace, the group ID must be globally unique across instances and regions.
- The ID of a group cannot be modified after the group is created.


Create a group


1. Log on to the [Message Queue for Apache RocketMQ console](#) and click **Groups** in the left-side navigation pane.
2. On the **Groups** page, click the name of the Message Queue for Apache RocketMQ instance in which you want to create a group.
3. Select a protocol for the group that you want to create.
 - If you want to use TCP-based SDKs to publish and consume messages, you must click the **TCP Protocol** tab to create a group.
 - If you want to use HTTP-based SDKs to publish and consume messages, you must click the **HTTP Protocol** tab to create a group.
 - Click **Create Group ID**.
 - In the **Create Group ID** dialog box, configure the **Group ID** parameter and **Description** parameter. Then, click **OK**.
The group that is created appears in the group list.

View information about a group

1. In the left-side navigation pane of the Message Queue for Apache RocketMQ console, click **Groups**.
2. On the **Groups** page, click the name of a Message Queue for Apache RocketMQ instance. Then, you can view all groups in the instance and the details of a specific group, including the subscription relationships, permissions, and status of consumers in the group.

Delete a group

 **Notice** After a group is deleted, producers and consumers in the group fail authentication when they attempt to connect to the Message Queue for Apache RocketMQ instance. The producers and consumers that are connected to the Message Queue for Apache RocketMQ instance are not affected.

1. In the left-side navigation pane of the Message Queue for Apache RocketMQ console, click **Groups**.
2. On the **Groups** page, click the name of the Message Queue for Apache RocketMQ instance that contains the group you want to delete.
3. Find the group that you want to delete and click the  icon in the **Actions** column.
4. In the message that appears, read the message. If you are sure you want to delete the group, click **OK**.
After the group is deleted, the group is not displayed in the group list of the instance.

References

If you want to call API operations to perform the operations that are described in this topic, see [Message Queue for Apache RocketMQ Developer Guide](#).

5.2. Message query

5.2.1. Overview

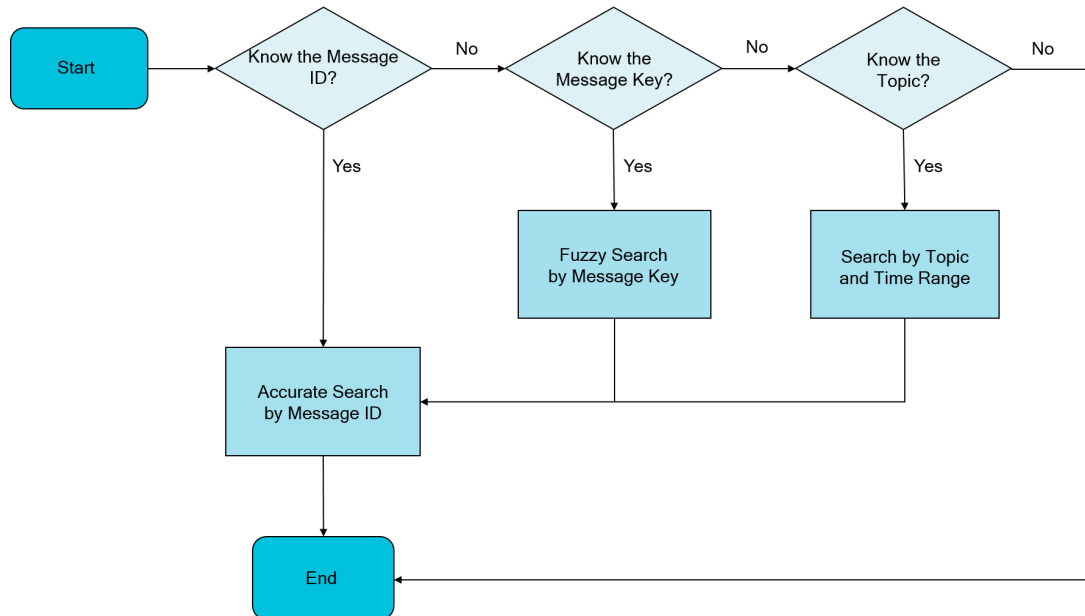
If a message is not consumed as expected, you can query the message content to troubleshoot problems. Message Queue for Apache RocketMQ allows you to query messages by message ID, by message key, and by topic.

Comparison of query methods

Method	Condition	Type	Description
By message ID	Topic+Message ID	Exact match	You can specify a topic and a message ID to obtain a message and its attributes.
By message key	Topic+Message Key	Fuzzy match	You can specify a topic and a message key to query the 64 messages that are most recently sent and contain the specified message key. We recommend that you specify a unique key for each message in producers whenever possible to ensure that the number of messages with the same key does not exceed 64. Otherwise, some messages cannot be queried.
By topic	Topic and time range	Range match	You can specify a topic and a time range to query all messages that meet the specified conditions. This type of query returns a large number of messages. It is difficult to find a specific message that you want to query.

We recommend that you query messages by using the following process.

Message query process



5.2.2. Query messages

This topic describes how to query messages in the Message Queue for Apache RocketMQ console by using three different methods.

1. [Log on to the Message Queue for Apache RocketMQ console.](#)
2. In the left-side navigation pane, click **Message Query**.
3. On the **Message Query** page, click a tab. On the tab that appears, enter the information and click **Search** to query messages.

- **By message ID**

If you query messages by message ID, exact match is used. You can specify a topic and a message ID to query a message by using exact match. Therefore, we recommend that you print the message ID to the log to facilitate troubleshooting after the message is sent.

In the following sample code, SDK for Java is used to obtain a message ID:

```
SendResult sendResult = producer.send(msg);  
String msgId = sendResult.getMessageId();
```

To obtain the sample code for other programming languages, click **Groups** in the left-side navigation pane. On the Groups page, find the group ID of the message and click **Sample Code** in the Actions column.

- **By message key**

Message Queue for Apache RocketMQ creates an index for messages based on the message keys that you specify. When you enter a topic name and a message key to query messages, Message Queue for Apache RocketMQ returns the matched messages based on the index.

 Notice

If you query messages by message key, take note of the following points:

- The query condition is the specified message key.
- Only the 64 messages that are most recently sent and contain the specified message key are returned. Therefore, we recommend that you specify a unique and business-distinctive key for each message.

The following sample code provides an example on how to specify a message key:

```
Message msg = new Message("Topic", "", "Hello MQ".getBytes());  
/**  
 * Specify the key to be indexed for each message. The key value is the key attribute  
 * of the message. We recommend that you specify a unique key for each message.  
 * If you do not receive a message as expected, you can query the message in the Message Queue for Apache RocketMQ console. Messages can be sent and received even if this  
 * attribute is not specified.  
 */  
msg.setKey("TestKey"+System.currentTimeMillis());
```

○ **By topic**

If you cannot query messages by message ID or message key, query messages by topic. You can specify a topic and time range for message sending, retrieve messages in batches, and then find the data that you need.

 Notice

If you query messages by topic, take note of the following points:


- If you specify a topic and time range to query messages, range match is used to retrieve all messages that meet the time condition within the topic. The number of retrieved messages is large. Therefore, we recommend that you narrow down the time range.
- If you query messages by topic, a large number of messages are returned on multiple pages.

5.2.3. Query results

This topic describes the results returned when you query messages.

You can view the queried messages on the **Message Query** page of the Message Queue for Apache RocketMQ console. The displayed information includes the message ID, tag, key, and storage time. In addition, click the corresponding buttons in the Actions column of a message to download the message content, [Query the message trace](#), and view the message details.


The delivery status is calculated by Message Queue for Apache RocketMQ based on the consumption progress of each group ID. For more information about the delivery status, see .

 **Note** The delivery status is estimated based on the consumption progress. Use the message tracing feature to query the consumption details. The message tracing feature allows you to query the complete trace of a message. For more information, see [Query the message trace](#).

Message delivery status

Delivery status	Possible cause
The message has been subscribed to and consumed at least once.	The group ID has properly consumed the message.
The message has been subscribed to but is filtered out by the filter expression. Check the tag of the message.	The tag of the message does not comply with the subscription of the consumer and the message is filtered out. To query the subscription of the consumer, log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click Groups . On the Groups page, find the group ID of the consumer whose subscription you want to view and click Consumer Status in the Actions column.
The message has been subscribed to but is not consumed.	A consumer identified by the group ID has subscribed to the message, but the message has not been consumed possibly because the consumption is slow or is blocked due to an exception.
The message has been subscribed to but the consumer that subscribes to the message and is identified by the group ID is not online. Use the message tracing feature to query the details about the message in an exact match.	A consumer identified by the group ID has subscribed to the message but the consumer is not online. Check the reason why the consumer is not online.
An unknown exception occurred.	Contact the customer service.

Message Queue for Apache RocketMQ provides the consumption verification feature. You can push a specified message to a specified online consumer to check whether the consumer can consume the message based on the correct logic as expected.

 **Notice** The consumption verification feature is used only to verify whether consumers can consume messages based on the correct logic as expected. This feature does not affect the normal process of receiving messages. Therefore, information such as the consumption status of a message does not change after the consumption is verified.

5.3. Message tracing

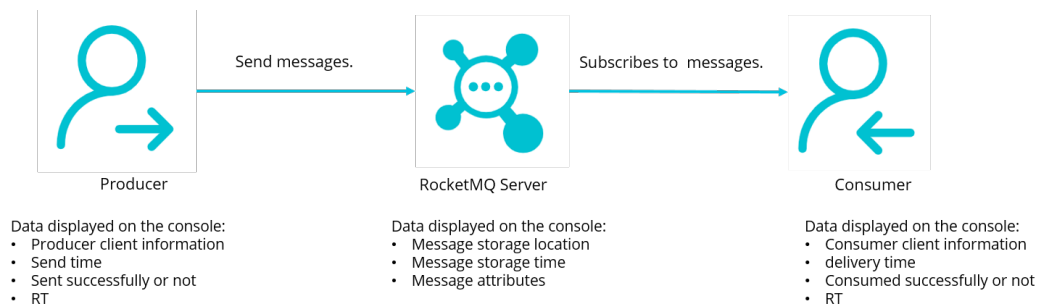
5.3.1. Overview

A message trace is the complete trace of a message that is sent from a producer to the Message Queue for Apache RocketMQ broker and then consumed by a consumer. The message trace includes the time, status, and other information of each node in the preceding process. The message trace provides robust data support for troubleshooting in production environments. This topic describes the scenarios, query procedure, and parameters of query results for message tracing.

Message trace data

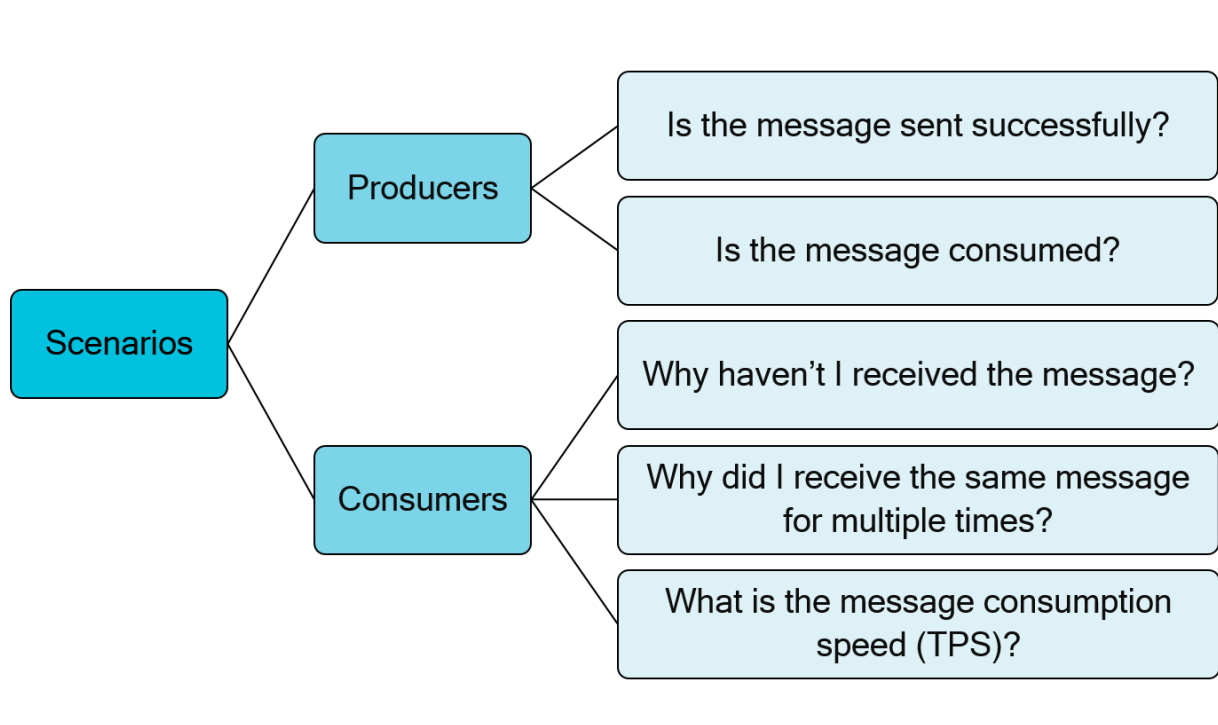
In Message Queue for Apache RocketMQ, the complete trace of a message involves three roles: producer, broker, and consumer. Each role adds relevant information to the trace when the role processes the message. The information is aggregated to indicate the status of the message. The following figure shows the relevant data.

Message trace data



Scenarios

You can use the message tracing feature to troubleshoot problems if a message is not sent or received as expected in your production environment. You can query the message trace by message ID, message key, or topic to check whether the message is sent and received as expected within the specified time range.



Usage notes

No extra fees are incurred when you use the message tracing feature. After a message is sent, you can query the trace of the message based on the ID or key of the message in the Message Queue for Apache RocketMQ console. You must take note of the following points.

Rules for querying message traces

Message type	Query description
Normal messages	A sending trace is generated after the message is sent. If the message is not consumed, Not Consumed appears. After the message is consumed, the delivery and consumption information appears.
Ordered messages	A sending trace is generated after the message is sent. If the message is not consumed, Not Consumed appears. After the message is consumed, the delivery and consumption information appears.
Scheduled messages and delayed messages	If the current system time does not reach the specified message consumption time, the trace can be queried but the message cannot be queried.
Transactional messages	Before the transaction is committed, the trace can be queried but the message cannot be queried.

Examples

If you find that a message is not received as expected based on the log information, perform the following steps to troubleshoot the problem by using the message trace:

1. Collect the information about the message that is suspected to be abnormal. The information includes the message ID, message key, topic, and approximate sending time.
2. Log on to the Message Queue for Apache RocketMQ console, and create a query task to query the message trace based on the available information.
3. Check the query results and analyze the cause.
 - If **Not Consumed** appears in the trace, go to the **Groups** page to [View the consumer status](#). Then, you can determine whether message accumulation is the reason why the message is not consumed.
 - If the message is consumed, find the corresponding consumer and the time when the message is consumed in the consumption information. Then, log on to the consumer to view the relevant log.

5.3.2. Query message traces

No extra fees are incurred when you use the message tracing feature. To use this feature, you must make sure that the version of your client SDK supports this feature. After a message is sent or received, you can query the trace of the message based on the message attributes in the Message Queue for Apache RocketMQ console.


Prerequisites

Make sure that the version of your SDK supports the message tracing feature. You can use the following versions of SDKs:

- SDK for Java: V1.2.7 and later
- SDK for C and C++: V1.1.2 and later
- .SDK for .NET: V1.1.2 and later

Procedure

1. [Log on to the Message Queue for Apache RocketMQ console](#). In the left-side navigation pane, click **Message Tracing**. On the page that appears, click **Create Query Task** in the upper-right corner.
2. In the **Create Query Task** dialog box, click the **By Message ID**, **By Message Key**, or **By Topic** tab and enter the information as prompted. Then, click **OK**.

 **Notice** Specify a time range as accurate as possible to narrow the query scope and speed up the query.

Message tracing supports the following three query methods. Select a query method and specify the query criteria.

- **By message ID:** You must specify the unique message ID, topic, and approximate sending time of a message. We recommend that you use this method because this method uses exact match and allows you to query message traces fast.
- **By message key:** You must specify the message key, topic, and approximate sending time of a message. This method uses fuzzy match. A maximum of 1,000 traces can be displayed for a query based on the specified message key. This method applies only to scenarios where the message ID is not recorded but a business-distinctive message key is specified.
- **By topic:** You must specify the topic and approximate sending time for batch query. This method uses range match and applies to scenarios where both the message ID and the message key are unavailable and the message volume is small. We do not recommend this query method, because a large volume of messages exist in a topic within the specified time range and you cannot find the message you want among these messages in this time range.

After you create a query task, you can view the query task on the **Message Tracing** page. If the value of **Task Status** is **Querying**, you cannot view the message trace.

3. In the upper-right corner, click **Refresh** until the value of **Task Status** becomes **Query Completed**. You can click the **+** icon to view the brief trace information, including the message attributes and consumption status.
 - If no data is found, verify whether the query information you entered is valid.
 - If the trace is queried out, brief trace information appears, including the message attributes and consumption status.
4. Click **View Traces** to check the complete trace.

The message trace consists of three parts:

- Producer information
- Topic information
- Consumer information

You can move the pointer over a field to view the details about the field.

If you query traces by message key or topic, multiple traces may be displayed. You can page up and down to view and compare the traces.

For more information about the query results of message traces, see [Message tracing status](#).

5.3.3. Status in message traces

This topic describes the terms and status information displayed on the Message Trace page.

Terms for message tracing

Role	Field	Description
Producer	Sending Time	The time when the message was sent from the producer. The time follows the ISO 8601 standard in the yyyy-MM-ddTHh:mm:ssZ format. The time is displayed in UTC.
	Time Consumed	The period of time that the producer took to send a message by calling the Send method. Unit: milliseconds.
Topic	Region	The region where the message is stored or the region where the consumer is located.
Consumer	Time Consumed	The period of time that the consumer took to execute the consumeMessage method after the message is pushed to the consumer.
	Delivery Time	The time when the consumer executed the consumeMessage method to start consuming the message. The time follows the ISO 8601 standard in the yyyy-MM-ddTHh:mm:ssZ format. The time is displayed in UTC.

Sending status and consumption status

Sending status and consumption status	Field	Description
	Sent	The message is sent and stored in the Message Queue for Apache RocketMQ broker.

Sending status and consumption status	Field	Description
Sending status	Sending Failed	The message fails to be sent and is not stored in the Message Queue for Apache RocketMQ broker. In this case, the broker tries to redeliver the message.
	Message Standing By	The message is a scheduled or delayed message and it is not the time to deliver the message.
	Transaction Uncommitted	The message is a transactional message and has not been committed.
	Message Rolled Back	The message is a transactional message and has been rolled back.
Consumption status	All Succeeded	The message has been consumed by all the consumers to which it is delivered.
	Partially Succeeded	The message fails to be consumed in specific deliveries, or the message is consumed after it is redelivered.
	All Failed	The message still fails to be consumed after all delivery retries.
	Not Consumed	The message is not delivered to consumers.
	Consumption Result Unreturned	No results are returned for the message consumption method or the method is interrupted. Therefore, the consumption status is not returned to the Message Queue for Apache RocketMQ broker.
	Consumed	The message is consumed.
	Consumption Failed	A failure result is returned for the message consumption method, or the method threw an exception.

5.4. View the consumer status

The Message Queue for Apache RocketMQ console allows you to check the consumer status to troubleshoot exceptions that occur during message consumption. This feature allows you to view the information about a group ID or a consumer identified by the group ID. The information includes the connection status, subscription, consumption TPS, number of accumulated messages, and thread stacks. This topic describes how to view the information.

Context

The cause of an exception that occurs during message consumption is complicated. In most cases, the consumer status information in the console alone is insufficient to troubleshoot a problem. You must perform further troubleshooting by analyzing logs and business scenarios.

Scenarios

You can query the consumer status for troubleshooting in the following scenarios:

- Subscription inconsistency
 - Symptom: In the **Consumer Status** panel, the value of **Consistent Subscription** column is **No** for the group ID.
 - Solution: For more information about how to handle subscription inconsistency, see [Subscription inconsistency](#).
- Message accumulation alerts
 - Symptom: In the **Consumer Status** panel, the value in the **Accumulated Messages** column is large for the group ID.
 - Solution: For more information about how to handle message accumulation alerts, see [Message accumulation](#).

View the information about a group ID

1. [Log on to the Message Queue for Apache RocketMQ console](#).
2. In the left-side navigation pane, click **Groups**.
3. On the **Groups** page, find the group ID that you want to view and click **Consumer Status** in the **Actions** column.

The following table describes the fields in the **Consumer Status** panel.

Description of fields in the Consumer Status panel

Field	Description
Online status icon	The value is Yes if one consumer instance identified by the group ID is online. In this case, you can view information about all online consumer instances in the Connection Information section. If none of the consumer instances identified by the group ID is online, the value is Offline and no information is displayed in the Connection Information section.

Field	Description
Consistent Subscription	Indicates whether the subscriptions of all consumer instances identified by the group ID are consistent. For more information about subscription consistency, see Subscription consistency .
Real-time Consumption Speed	The total TPS at which messages are received by the consumer instances identified by the group ID. Unit: messages/s.
Real-time Accumulated Messages	The total number of messages that are not consumed by the consumer instances identified by the group ID.
Last Consumed At	The time when the consumer instances identified by the group ID last consumed a message.
Message Delay Time	The difference between the production time of the earliest unconsumed message and the current time.

View information about a single consumer instance identified by a specific group ID

1. If the online status of the group ID is **Yes**, you can view information about each online consumer instance identified by the group ID in the **Connection Information** section. The information includes the client ID, host or public IP address, current process ID, and number of accumulated messages.
2. If you want to view more information about a specific consumer instance, click **Detailed Information** in the **Detailed Description** column. The information includes the number of consumer threads, consumption start time, subscription, and message consumption statistics.
3. If you want to view the stack information of the current process for a specific consumer instance, find the consumer instance and click **Stack Information** in the **Stack Information** column.

5.5. Reset consumer offsets

You can reset consumer offsets to skip the accumulated or undesired messages instead of consuming them, or to consume messages sent after a point in time regardless of whether the messages sent before this point in time are consumed.

Context

When you reset consumer offsets, take note of the following points:

- You cannot reset consumer offsets in broadcasting consumption mode.
- You cannot reset consumer offsets by specifying a message ID, message key, or tag.

Procedure

1. [Log on to the Message Queue for Apache RocketMQ console](#). In the left-side navigation pane, click **Groups**.

2. Find the group ID whose consumer offset you want to reset, click the More icon in the **Actions** column, and then select **Reset Consumer Offset**.
3. In the **Reset Consumer Offset** dialog box, enter the corresponding topic in the **Topic** field, and then select one of the following options as needed:
 - **Consumption from Latest Offset (All Accumulated Messages Cleared)**: If this option is selected, consumers identified by the group ID skip all accumulated (unconsumed) messages within the topic and restart consumption from the latest offset.

If "reconsumeLater" is returned, the messages in the delivery retry process cannot be skipped.
 - **Consumption from a Specific Point in Time**: If this option is selected, a time picker appears. Select a point in time. Only the messages that are sent after the selected point in time will be consumed.

The period allowed for the time picker ranges from the production time of the earliest message stored in the topic to the production time of the latest message stored in the topic. You can select a point in time only within the allowed time range.
4. Click **OK** to reset the consumer offset.

5.6. Dead-letter queues

Dead-letter queues are used to process messages that cannot be consumed as expected. This topic describes how to query, export, and resend dead-letter messages in dead-letter queues. This helps you manage dead-letter messages as needed and prevent missing messages.

Background information

When a message fails to be consumed for the first time, the Message Queue for Apache RocketMQ broker automatically redelivers the message. If the message still cannot be consumed after the broker redelivers the message for a maximum of allowed times, the message cannot be properly consumed. Instead of immediately discarding the message, Message Queue for Apache RocketMQ sends it to a particular queue of the corresponding consumer.

In Message Queue for Apache RocketMQ, a message that cannot be properly consumed is called a *dead-letter message*, which is stored in a particular queue named *dead-letter queue*.

Features

Dead-letter messages have the following features:

- They can no longer be consumed by consumers as expected.
- They have a valid period of three days, which is the same as that of normal messages. After the three days, dead-letter messages are automatically deleted. Therefore, process dead-letter messages within three days after they are generated.

Dead-letter queues have the following features:

- A dead-letter queue corresponds to a group ID instead of a consumer instance.
- If no dead-letter message is generated for a group ID, Message Queue for Apache RocketMQ does not create a dead-letter queue for the group ID.
- A dead-letter queue contains all the dead-letter messages of the corresponding group ID regardless of the message topic.

In the Message Queue for Apache RocketMQ console, you can query, export, and resend dead-letter messages.

Methods of querying dead-letter messages

Message Queue for Apache RocketMQ provides the following methods for you to query dead-letter messages.

Method	Condition	Type	Description
By group ID	Group ID and time range	Range match	You can specify a group ID and a time range to query all messages that meet the specified conditions. This type of query returns a large number of messages. It is difficult to find a specific message that you want to query.
By message ID	Group ID+Message ID	Exact match	You can specify a group ID and a message ID to query a message by using exact match.

By group ID

You can batch query all the dead-letter messages of a group ID within a time range by specifying the group ID and time range.



Notice The production time of a dead-letter message refers to the time when a message is sent to the dead-letter queue after the maximum number of redelivery retries for this message is reached.

1. [Log on to the Message Queue for Apache RocketMQ console.](#)
2. In the left-side navigation pane, click **Dead-letter Queues**.
3. On the **Dead-letter Queues** page, click the **By Group ID** tab.
4. From the drop-down list of group IDs, select the group ID whose dead-letter messages you want to view.
5. Click the icon for the time picker and specify the start time and end time.
6. Click **Search**. All dead-letter messages that meet the preceding conditions appear.
7. Find the dead-letter message that you want to view and click **View Details** in the **Actions** column to view the details about the message. The details include the basic attributes, download URL of the message body, message trace, and delivery status.

By message ID

If you query messages by message ID, exact match is used. You can precisely locate a message by specifying its group ID and message ID.

1. [Log on to the Message Queue for Apache RocketMQ console.](#)
2. In the left-side navigation pane, click **Dead-letter Queues**.
3. On the **Dead-letter Queues** page, select your instance and click the **By Message ID** tab.

4. From the drop-down list of group IDs, select the group ID whose dead-letter messages you want to view.
5. In the search box of message IDs, enter the ID of the message that you want to query.
6. Click **Search**. All dead-letter messages that meet the preceding conditions appear.
7. Find the dead-letter message that you want to view and click **View Details** in the **Actions** column to view the details about the message. The details include the basic attributes, download URL of the message body, message trace, and delivery status.

Export dead-letter messages

If you cannot process dead-letter messages within the validity period, export the messages in the Message Queue for Apache RocketMQ console.

The Message Queue for Apache RocketMQ console allows you to export a single dead-letter message or export dead-letter messages in batches. The exported file is in the CSV format.

The following table describes the fields of an exported message.

Field	Definition
topic	The topic to which the message belongs.
msgId	The ID of the message.
bornHost	The URL of the producer that produced the message.
bornTimestamp	The time when the message was produced.
storeTimestamp	The time when the message turned into a dead-letter message.
reconsumeTimes	The number of times that the message failed to be consumed.
properties	The message attributes in the JSON format.
body	The Base64-encoded message body.
bodyCRC	The cyclic redundancy check (CRC) of the message body.

- Export a single dead-letter message


In the Message Queue for Apache RocketMQ console, find the dead-letter message that you want to export and click **Export** in the **Actions** column.

- Export dead-letter messages in batches

In the Message Queue for Apache RocketMQ console, enter the group ID to query the dead-letter messages, select the dead-letter messages that you want to export, and then click **Batch Export**.

Resend dead-letter messages

If a message enters a dead-letter queue, the message cannot be consumed as expected for specific reasons. Therefore, you must process the message in a special way. After you troubleshoot the problems, you can resend the message to the corresponding consumer in the Message Queue for Apache RocketMQ console.

 **Notice** After a dead-letter message is resent to the consumer, the message will still be stored in the dead-letter queue for three days. The system automatically deletes the message after the three days.

- Resend a single dead-letter message

In the Message Queue for Apache RocketMQ console, query one dead-letter message by message ID or query dead-letter messages by group ID. Find the dead-letter message that you want to resend and click **Resend** in the **Actions** column.

- Resend dead-letter messages in batches

In the Message Queue for Apache RocketMQ console, query dead-letter messages by group ID, select the dead-letter messages that you want to resend, and then click **Batch Resend**.

5.7. Resource statistics

5.7.1. Overview

This topic describes how to use the resource statistics feature to query the statistics of produced messages and consumed messages.

The resource statistics feature provides the statistics of produced messages and consumed messages. This feature allows you to query the following data:

- **Statistics of produced messages**
 - Queries by topic: You can query the total number of messages that are received by a topic or the average number of messages that are received by a topic per second in a specified period of time.
 - Queries by instance: You can query the total number of messages that are received by the topics in a specified instance or the average number of messages that are received by the topics in a specified instance per second in a specified period of time.
- **Statistics of consumed messages**
 - Queries by group ID: You can query the total number of messages that are sent from a topic to consumers identified by a group ID or the average number of messages that are sent from a topic to consumers identified by a group ID per second in a specified period of time.
 - Queries by instance: You can query the total number of messages that are sent to all groups in a specified instance or the average number of messages that are sent to all groups in a specified instance per second in a specified period of time.

5.7.2. Query the statistics of produced messages

This topic describes how to query the statistics of produced messages. You can query the total number of messages that are received by a topic or all topics across the brokers in a specified instance in a specified period of time. You can also query the average number of messages that are received by a topic or all topics across the brokers in a specified instance per second in a specified period of time.

Procedure

1. [Log on to the Message Queue for Apache RocketMQ console.](#)
2. In the left-side navigation pane, click **Resource Statistics**.
3. On the **Resource Statistics** page, click the **Message Production** tab.
4. From the **Resource Type** drop-down list, select a resource type for which you want to query the statistics of produced messages. Configure the related fields. Then, click **Search**.

The following information describes the related fields:

- **Resource Type**: The value can be Instance or Topic. Select Instance to query the total number of messages that are received by the topics in a specified instance or the average number of messages that are received by the topics in a specified instance per second in a specified period of time. Select Topic to query the total number of messages that are received by a specified topic or the average number of messages that are received by a specified topic per second in a specified period of time.
- **Current Instance**: This parameter is displayed if **Resource Type** is set to Instance. This parameter is automatically set to the name and ID of the current instance.
- **Topic**: This parameter is displayed if **Resource Type** is set to Topic. Select a topic to query the statistics of the produced messages that are sent to a specified topic in the current instance.
- **Collection Type**: The value can be Total or TPS. Select Total to query the total number of messages that are received by the topic in each collection cycle. Select TPS to query the average number of messages that are received by the topic per second in each collection cycle.
- **Collection Interval**: The value can be 1 Minute, 10 Minutes, 30 Minutes, or 1 Hour. This parameter specifies the interval at which data is collected. A smaller value indicates a higher data collection frequency and more detailed data.
- **Time Range**: Message Queue for Apache RocketMQ allows you to query messages that are produced in the last three days.

Query results are displayed in charts.

5.7.3. Query the statistics of consumed messages

This topic describes how to query the statistics of consumed messages. You can query the total number of messages that are sent from a topic to consumers identified by a group ID in a specified period of time. You can also query the average number of messages that are sent from a topic to consumers identified by a group ID per second in a specified period of time.

Procedure

1. [Log on to the Message Queue for Apache RocketMQ console.](#)
2. In the left-side navigation pane, click **Resource Statistics**.
3. On the **Resource Statistics** page, click the **Message Consumption** tab.
4. From the **Resource Type** drop-down list, select a resource type for which you want to query the statistics of consumed messages. Configure the related fields. Then, click **Search**.

The following information describes the related fields:

- **Resource Type**: The value can be Instance or Group ID. Select Instance to query the total number of messages that are sent to the groups in a specified instance or the average number of messages that are sent to the groups in a specified instance per second in a specified period

of time. Select **Group ID** to query the total number of messages that are sent from a topic to consumers identified by a group ID or the average number of messages that are sent from a topic to consumers identified by a group ID per second in a specified period of time.

- **Current Instance**: This parameter is displayed if **Resource Type** is set to Instance. This parameter is automatically set to the name and ID of the current instance.
- **Group ID**: This parameter is displayed if **Resource Type** is set to Group ID. You must select the group ID for which you want to query data.
- **Topic**: This parameter is displayed if **Resource Type** is set to Group ID. You must select a topic from which the messages that you want to query are sent.
- **Collection Type**: The value can be Total or TPS. Select Total to query the total number of messages that are sent to consumers identified by the group ID in each collection cycle. Select TPS to query the average number of messages that are sent to consumers identified by the group ID per second in each collection cycle.
- **Collection Interval**: The value can be 1 Minute, 10 Minutes, 30 Minutes, or 1 Hour. This parameter specifies the interval at which data is collected. A smaller value indicates a higher data collection frequency and more detailed data.
- **Time Range**: Message Queue for Apache RocketMQ allows you to query messages that are consumed in the last three days.


Query results are displayed in charts.

5.8. Account authorization management

Message Queue for Apache RocketMQ allows you to use an Apsara Stack tenant account to grant permissions to publish and subscribe to a topic to another Apsara Stack tenant account or a Resource Access Management (RAM) user. An Apsara Stack tenant account is a level-1 department account. A RAM user is a personal account that is used to access the Apsara Stack resources.


Grant permissions to another Apsara Stack tenant account


You can use an Apsara Stack tenant account to grant permissions to another Apsara Stack tenant account. To grant permissions to publish and subscribe to a topic, perform the following steps:

1. Log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click **Topics**.
2. On the **Topics** page, find the topic that you want to authorize another account to manage, click  in the **Actions** column, and then select **Authorize** from the drop-down list.
3. In the **Authorize** dialog box, set **Account Type** to **Apsara Stack Account**.
4. In the **Apsara Stack Account ID** field, enter the ID of the Apsara Stack tenant account to which you want to grant permissions.
5. From the **Authorization Type** drop-down list, select the permissions that you want to grant to the Apsara Stack tenant account. Then, click **OK**.

Grant permissions to a RAM user


You can use an Apsara Stack tenant account to grant permissions to a RAM user that belongs to the Apsara Stack tenant account. To grant permissions to publish and subscribe to a topic, perform the following steps:


1. Log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click **Topics**.
2. On the **Topics** page, find the topic that you want to authorize a RAM user to manage, click  in the **Actions** column, and then select **Authorize** from the drop-down list.
3. In the **Authorize** dialog box, set **Account Type** to **RAM User**.
4. In the **RAM User Name** field, enter the name of the RAM user to which you want to grant permissions.
5. From the **Authorization Type** drop-down list, select the permissions that you want to grant to the RAM user. Then, click **OK**.

 **Note** The RAM user to which you want to grant permissions must be an account that is used to access the Apsara Stack resources and is owned by the department to which the Apsara Stack tenant account belongs.

View authorization information

You can view the authorization records and the details of each topic in the Message Queue for Apache RocketMQ console. To view authorization information, perform the following steps:

1. Log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click **Topics**.
2. On the **Topics** page, find the topic that you want to view, click  in the **Actions** column, and then select **View Authorization** from the drop-down list to view the authorization records and information of the topic.

-  **Note**
- You do not need to authorize your account to manage the topics that you create.
 - After you log on to the Message Queue for Apache RocketMQ console by using an authorized account, you can view the topic that the account is authorized to manage. Then, you must create a group ID. You cannot use the group ID of the Apsara Stack tenant account that is used to grant permissions to your account.
 - A topic that an account is authorized to manage belongs to the Apsara Stack tenant account that is used to grant permissions. Therefore, you cannot use the authorized account to delete the topic.
 - If you grant permissions to a RAM user, you cannot use the authorized RAM user to create topics.
 - If you grant permissions to another Apsara Stack tenant account, you can use the authorized Apsara Stack tenant account to create topics. However, the created topics are not associated to the Apsara Stack tenant account that is used to grant permissions.

5.9. Switch between different access modes

Message Queue for Apache RocketMQ supports instance-specific management. By default, one instance can be deployed at a time. Message Queue for Apache RocketMQ supports advanced access control by using virtual private clouds (VPC) for each instance.

Context

By default, a Message Queue for Apache RocketMQ instance supports the Any Tunnel access mode. This means that the Message Queue for Apache RocketMQ instance can be accessed in each VPC environment. You can switch the access mode in the console at any time. If the access mode of a Message Queue for Apache RocketMQ instance is switched to Single Tunnel, the instance can be accessed only in a specified VPC environment.

Procedure

1. [Log on to the RocketMQ console](#). In the left-side navigation pane, click **Cluster Management**.
2. [Log on to the Message Queue for Apache RocketMQ console](#). In the left-side navigation pane, click **Cluster Management**.
3. Find the instance whose access mode you want to switch and click **Switch Access Method** in the **Actions** column.
4. Select an access mode. You can select one of the following options:
 - **Single Tunnel**: If this option is selected, the instance can be accessed only in a specified VPC environment. The page displays the **vSwitch ID** field. You must specify the vSwitch ID of the VPC that you use.
 - **Any Tunnel**: If this option is selected, the instance can be accessed in each VPC environment.
5. Select an option for **Forced Switch** to indicate whether to forcibly switch the access mode. The switching between access modes may cause the transient interruption of services. When forcible switching is disabled, one access mode can be switched to another access mode only when the instance traffic is light. This means that the transactions per second (TPS) must be no more than 10. When forcible switching is enabled, one access mode can be switched to another access mode regardless of the service traffic.
6. Click **OK**.

5.10. Bind a VPC to a Message Queue for Apache RocketMQ instance

Message Queue for Apache RocketMQ allows you to bind multiple virtual private clouds (VPCs) to an instance. Message Queue for Apache RocketMQ provides an endpoint for each Single Tunnel VPC. You can assign different endpoints of the same instance to different business services to perform fine-grained access control. This topic describes how to bind a VPC to an instance and unbind a VPC from an instance in the Message Queue for Apache RocketMQ console.

Context

By default, Message Queue for Apache RocketMQ provides an HTTP-based Any Tunnel endpoint and a TCP-based Any Tunnel endpoint for each instance. You can connect to the Message Queue for Apache RocketMQ instance by using an Any Tunnel endpoint over a VPC. You can bind multiple VPCs of the Single Tunnel type to provide multiple networks for an instance. After a Single Tunnel VPC is bound to an instance, the system automatically allocates an endpoint for the VPC. The endpoint can be used to connect to the Message Queue for Apache RocketMQ instance over only the VPC. You can bind multiple Single Tunnel VPCs to your Message Queue for Apache RocketMQ instance. This way, you can assign isolated networks to your business services to perform fine-grained network access control.

Prerequisites

- [Create an instance](#)
- A VPC is created.

For more information, see the *Create a VPC* topic in the *VPC User Guide*.

- A vSwitch is created.

For more information, see the *Create a vSwitch* topic in the *VPC User Guide*.

Precautions

- VPC connections are established based on Any Tunnel network connections. VPCs that are bound to a Message Queue for Apache RocketMQ instance do not affect the existing Any Tunnel networks of the instance. The Any Tunnel networks remain available after VPCs are bound to the instance.
- We recommend that you use the credential of the administrator of the Message Queue for Apache RocketMQ instance to bind VPCs and manage VPCs that are bound to the instance. If you use the credentials of non-administrator users to bind and manage VPCs for a Message Queue for Apache RocketMQ instance, your business services may fail to connect to the instance.
- Only TCP-based clients can use the endpoint for a VPC to connect to the instance.
- After a VPC is bound to a Message Queue for Apache RocketMQ instance, the system ensures only that Message Queue for Apache RocketMQ brokers can connect to the instance over the VPC. To connect your client to the Message Queue for Apache RocketMQ instance over the VPC, make sure that your client is connected to the VPC.

Bind a VPC to an instance

1. Log on to the Message Queue for Apache RocketMQ console. For information about how to log on to the Message Queue for Apache RocketMQ console, see [Log on to the Message Queue for Apache RocketMQ console](#).
2. In the left-side navigation pane, click **Instances**.
3. On the **Instances** page, click the name of the instance to which you want to bind a VPC. Then, click the **Network Management** tab.
4. On the **Network Management** tab, click **Binding VPC**.
5. In the **Binding VPC** dialog box, configure the parameters and click **OK**.

The following table describes the parameters that you need to configure to bind a VPC.

Parameter	Description
-----------	-------------

Parameter	Description
Cluster Name	The name of the cluster to which the Message Queue for Apache RocketMQ instance belongs. The system automatically obtains the value of this parameter. You do not need to specify a value.
Instance	The ID of the Message Queue for Apache RocketMQ instance. The system automatically obtains the value of this parameter. You do not need to specify a value.
VPC ID	Select the VPC that you want to bind to the Message Queue for Apache RocketMQ instance from the drop-down list.
Switch ID	Select the vSwitch that you want to connect to the instance from the drop-down list.
VPC NAME	Specify a name for the VPC that you want to bind to the instance. We recommend that you specify a name that can help you identify the business service that connects to the instance over the VPC.

After the VPC is bound, you can view information about the VPC in the VPC list on the **Network Management** tab, including the endpoint that is assigned for the VPC, the ID of the VPC, and the vSwitch to which the instance is connected. You can also click **VPC Instances** in the **Actions** column to view the Message Queue for Apache RocketMQ instances to which the VPC is bound.

Unbind a VPC from an instance

Notice

- After a VPC is unbound from an RocketMQ instance, the endpoint for the VPC becomes invalid and cannot be used to access the instance. Proceed with caution when you unbind a VPC from an instance.
- If a VPC is bound to multiple instances in a cluster and you want to prevent access requests that are sent over the VPC, you must unbind the VPC from all instances in the cluster.

- Log on to the Message Queue for Apache RocketMQ console. For information about how to log on to the Message Queue for Apache RocketMQ console, see [Log on to the Message Queue for Apache RocketMQ console](#).
- In the left-side navigation pane, click **Instances**.
- On the **Instances** page, click the name of the instance from which you want to unbind a VPC. Then, click the **Network Management** tab.
- In the VPC list, find the VPC that you want to unbind and click **Unbind VPCs** in the **Actions** column.

5. In the **Unbind** dialog box, check the information about the VPC and click **OK**.

5.11. Route messages from a cluster to another cluster

The message routing feature provided by Message Queue for Apache RocketMQ allows you to synchronize messages across clusters. This topic describes how to configure a message routing task.

Context

The message routing feature of Message Queue for Apache RocketMQ is used to synchronize messages across clusters. You can configure routing rules to dynamically plan the synchronization path of messages so that messages can be synchronized from the source node to the destination node based on filter conditions. This implements remote message synchronization and allows you to synchronize messages across clusters within milliseconds. This way, data consistency and integrity across clusters are ensured.

The following figure shows how the message routing feature works in Message Queue for Apache RocketMQ. In the figure, one-way synchronization is performed based on topics to synchronize messages from a specified source topic in the source instance to a specified destination topic in the destination instance.

For more information about the message routing feature, see [Message routing](#).

Precautions

- Only instances that have a namespace support the message routing feature. If you want to enable the message routing feature for an instance, specify a namespace for the instance when you create the instance.
- The message routing feature does not support chain routing. For example, you cannot route messages from Cluster A to Cluster B and then from Cluster B to Cluster C. You must create a task to route messages from Cluster A to Cluster C.
- The message type of the source topic must be the same as the message type of destination topic. For example, if the message type of the source topic is normal message, the message type of the destination message must also be normal message.
- A routing task must be created in the production environment in which the destination cluster is deployed. If you want to route messages from Cluster A to Cluster B, you must create a message routing task in the cloud environment of Cluster B. In the message routing task configuration, specify Cluster A as the source cluster and Cluster B as the destination cluster.
- Message routing requires the CPU resources and memory resources of the source cluster and the destination cluster, and the storage resources of the destination cluster. You must evaluate the amount of resources that are required before you create a message routing task.

Prerequisites

- [Create an instance](#)
- [Create a topic](#)

Procedure

You can perform the following steps to create a Message Queue for Apache RocketMQ message routing task:

- **Step 1: Create a destination cloud**

Before you create a routing task, you must specify information to create a cloud where your Message Queue for Apache RocketMQ cluster is deployed. The information includes the endpoint of your Message Queue for Apache RocketMQ instance and the AccessKey ID and AccessKey secret of the account to which the cloud belongs. Message Queue for Apache RocketMQ obtains the permissions that are required to access Message Queue for Apache RocketMQ resources across clouds based on the cloud information that you specified.

- **Step 2: Create a routing task**

Specify the message source and the message destination, and configure relevant information. For example, specify filter conditions and set the start offset of message synchronization.

Step 1: Create a destination cloud

1. Log on to the Message Queue for Apache RocketMQ console. For information about how to log on to the Message Queue for Apache RocketMQ console, see [Log on to the Message Queue for Apache RocketMQ console](#).
2. In the left-side navigation pane, click **Message Route**.
3. On the **Message Route** page, click the **Cloud Information** tab. In the upper-right corner of the Cloud Information tab, click **Add Message Synchronization Cloud**.
4. In the **Create Message Synchronization Cloud** dialog box that appears, configure the parameters. Then, click **OK**.

The following table describes the parameters.

Parameter	Description	Example
Cloud Name	The name of the cloud to which you want to route messages. The cloud name must be unique. The system identifies the message routing destination based on only the name of the cloud and the endpoint of the Message Queue for Apache RocketMQ instance .	Center

Parameter	Description	Example
AccessKey	<p>The AccessKey ID of the account that is used to log on to the cloud.</p> <p>Make sure that the role of the account is the administrator of Message Queue for Apache RocketMQ or a user that is granted the permissions to write data to and read data from instances and topics.</p> <p>For information about how to obtain the AccessKey ID, see the <i>Obtain an AccessKey pair</i> topic in the <i>Message Queue for Apache RocketMQ Developer Guide</i>.</p>	j8geROUEAW1k****
Secret	<p>The AccessKey Secret of the account that is used to log on to the cloud.</p> <p>For information about how to obtain the AccessKey Secret, see <i>Obtain an AccessKey pair</i> topic in the <i>Message Queue for Apache RocketMQ Developer Guide</i>.</p>	AMx4ainrLWht8jYUPHkdI4zY7t**
Region	<p>The ID of the region where the Message Queue for Apache RocketMQ cluster is deployed.</p> <p>To obtain the ID of the region, log on to the Message Queue for Apache RocketMQ console and view the ID of the region in the top navigation bar.</p>	cn-qingdao-env33-d01
Message Queue for Apache RocketMQ Instance Endpoint	<p>The endpoint of the Message Queue for Apache RocketMQ instance.</p> <p>For information about how to obtain the endpoint of the Message Queue for Apache RocketMQ instance, see the References section in this topic.</p>	http://mq.server.xxxx.com:8080/rocketmq/nsaddr4client-internal


Parameter	Description	Example
MQ API	<p>The interface of the Message Queue for Apache RocketMQ console in the cloud.</p> <p>The value of this parameter must be in the following format: <code>http:{console.domain}/json</code>.</p> <p>For information about how to obtain the value of {console.domain}, see the References section in this topic.</p>	<code>http:mq.console.xxxx.com/json</code>
Description	<p>The description of the cloud. You can specify a description that can help you identify the cloud.</p>	Core node

Step 2: Create a message routing task

1. Log on to the Message Queue for Apache RocketMQ console. For information about how to log on to the Message Queue for Apache RocketMQ console, see [Log on to the Message Queue for Apache RocketMQ console](#).
2. In the left-side navigation pane, click **Message Route**.
3. On the **Message Route** page, click the **Task** tab. Then, click **Create Task** in the upper-right corner.
4. In the **Create Task** dialog box that appears, configure the parameters. Then, click **OK**.

The following table describes the parameters.

Parameter	Description
Message Source	<ul style="list-style-type: none">◦ Message Synchronization Cloud: the name of the source cloud from which you want to route messages.◦ Source Region: the region where the source Message Queue for Apache RocketMQ cluster is deployed. After you configure the Message Synchronization Cloud parameter, the system automatically obtains the ID of the region where the source cloud is deployed. You do not need to specify a value for this parameter.◦ Source Instance: the instance to which the source topic belongs.◦ Source Topic: the name of the topic from which you want to route messages.

Parameter	Description
Message Destination	<ul style="list-style-type: none">◦ Message Synchronization Cloud: the name of the destination cloud to which you want to route messages.◦ Destination Region: the region where the destination Message Queue for Apache RocketMQ cluster is deployed. After you configure the Message Synchronization Cloud parameter, the system automatically obtains the ID of the region where the destination cloud is deployed. You do not need to specify a value for this parameter.◦ Destination Instance: the instance to which the destination topic belongs.◦ Destination Topic: the name of the topic to which you want route messages.
Offset to Start	<p>The consumer offset from which you want to route messages from the source topic to the destination topic.</p> <p>Default value: Maximum Offset. If the value of this parameter is set to Maximum Offset, the message routing task routes messages to the destination topic from the most recent message after the task is started.</p> <div> Notice If the value of this parameter is set to Maximum Offset, messages that are sent to the source topic before the message routing task is started are not routed to the destination topic.</div>

Parameter	Description
Filtering Rule	<ul style="list-style-type: none"> If no filtering rules are specified, the message routing task routes all messages from the source topic to the destination topic. If a filtering rule is specified, the message routing task routes messages that match the specified conditions from the source topic to the destination topic. <p>You can specify tags as conditions.</p> <p>If the name of a tag of the messages that you want to route to the destination topic is <code>CartService</code>, you can enter <code>CartService</code> in the Filtering Rule field. If you want to filter messages based on multiple tags, you can specify multiple tags and separate the tags with two vertical bars (<code> </code>). For example, you can enter <code>CartService Inventory Payment</code>. For more information about how messages are filtered based on tags, see Message filtering.</p>
Description	Enter a description for the message routing task to help you identify the task.



Modify a cloud

1. On the **Message Route** page, click the **Cloud Information** tab.
2. In the cloud list, find the destination cloud that you want to modify and click **Modify** in the **Actions** column.
3. In the **Modify Message Synchronization Cloud** dialog box that appears, modify the configuration of the cloud and click **OK**.
4. In the message that appears, read the message and click **OK**.


Delete a cloud


1. On the **Message Route** page, click the **Cloud Information** tab.
2. In the cloud list, find the cloud that you want to delete and click **Delete** in the **Actions** column.
3. In the message that appears, read the message and click **OK**.

Start or stop a message routing task


1. On the **Message Route** page, click the **Task** tab.
2. In the message routing task list, find the message routing task that you want to manage and click the  icon in the **Actions** column to start the task or click the  icon to stop the task.

Modify a message routing task

 **Notice** After you modify the **filtering rule** of a task, you must stop the task and then restart the task to make the modification take effect.

1. On the **Message Route** page, click the **Task** tab.
2. In the message routing task list, find the task that you want to modify and click the  icon in the **Actions** column.
3. In the **Modify Task** dialog box that appears, modify the configuration of the task and click **OK**.


View details of a message routing task


1. On the **Message Route** page, click the **Task** tab.
2. In the message routing task list, find the task that you want to view and click the  icon in the **Actions** column. The details of the task are displayed.

The following table describes the parameters that are included in the details of a message routing task.

Parameter	Description
Task Status	<ul style="list-style-type: none">◦ Created◦ Running◦ Suspended
Message Synchronization TPS	Transactions per second (TPS) indicates the number of messages that are routed from the source topic to the destination topic per second. The system collects the average TPS value at an interval of 1 minute.
Message Delay	The time difference between the consumer offset of the most recent message that was routed and the consumer offset of the latest message.
Accumulation Amount	The number of messages that are not routed to the destination topic.
Latest Synchronization Time	The point in time when the last message was routed.

Delete a message routing task


 **Notice** After a message routing task is deleted, the task stops running.

1. On the **Message Route** page, click the **Task** tab.
2. In the message routing task list, find the task that you want to delete and click the  icon in the **Actions** column.
3. In the message that appears, read the message and click **OK**.

References

Obtain information about the endpoint of a Message Queue for Apache RocketMQ instance and the Message Queue for Apache RocketMQ API

When you create a destination cloud for message routing, configure the **Message Queue for Apache RocketMQ Instance Endpoint** parameter and the **MQ API** parameter that is used specify the interface of the RocketMQ service in the destination cloud. You can perform the following steps to obtain the values for the parameters.

1. Log on to the Apsara Infrastructure Management Console.
 - i. Go to the Apsara Uni-manager Operations Console.
 - ii. In the top navigation bar, choose **Products > Platforms > Apsara Infrastructure Management Framework**.
2. In the left-side navigation pane, click **Reports**.
3. On the **All Reports** page, search for *Registration Vars of Services*. In the report list that appears, click the name of the report that you want to view.
4. On the **Registration Vars of Services** page, click the  icon next to **Service**. Then, search for *mq*.
5. Find the **mq-cai** service, right-click the **Service Registration** column. Then, select **Show More** from the shortcut menu.

On the **Details** page, the value that is displayed for the **client.onsAddr** parameter is the value of the **Message Queue for Apache RocketMQ Instance Endpoint** parameter.
6. Find the **mq-console** service, right-click the **Service Registration** column. Then, select **Show More** from the shortcut menu.

On the **Details** page, the value that is displayed for the **console.domain** parameter is the value of {console.domain} in the **MQ API** parameter.

6.SDK user guide

6.1. Overview

This topic lists the protocols supported by Message Queue for Apache RocketMQ and the related SDKs for multiple programming languages.

SDKs for different protocols and programming languages

The following table lists the protocols and SDKs for multiple programming languages that are supported by Message Queue for Apache RocketMQ.

Protocol	Programming language	SDK download link	Sample code
TCP	Java	mq-tcp-java-sdk	mq-tcp-samples-java
	C/C++	mq-tcp-csharp-sdk	mq-tcp-samples-csharp
	.NET	mq-tcp-.net-sdk	mq-tcp-samples-.net
HTTP	Java	mq-http-java-sdk	mq-http-samples-java
	PHP	mq-http-php-sdk	mq-http-samples-php
	Go	mq-http-go-sdk	mq-http-samples-go
	Python	mq-http-python-sdk	mq-http-samples-python
	Node.js	mq-http-nodejs-sdk	mq-http-samples-node.js
	C#	mq-http-cpp-sdk	mq-http-samples-cpp
	C++	mq-http-csharp-sdk	mq-http-samples-csharp

Usage notes

- Message Queue for Apache RocketMQ provides TCP client SDKs and HTTP client SDKs for you to send and consume messages. You cannot specify the same group ID in the code of a TCP client SDK and the code of an HTTP client SDK at the same time. If you want to use a TCP client SDK to send and consume messages, you must create a group for the TCP protocol. You cannot specify a group that is created for the HTTP protocol in the code of the TCP client SDK.
- A Message Queue for Apache RocketMQ instance provides a TCP endpoint and an HTTP endpoint. An endpoint for a specific protocol must be used together with an SDK for the same protocol. For example, if you want to use a TCP client SDK to send and consume messages, you must obtain the TCP endpoint of your Message Queue for Apache RocketMQ instance. You cannot use the HTTP endpoint to connect to the instance.

6.2. SDK user guide

6.2.1. Demo projects

6.2.1.1. Overview

This topic helps engineers who are new to Message Queue for Apache RocketMQ to build a Message Queue for Apache RocketMQ test project. The demo project is a Java project. It contains test code for normal messages, transactional messages, and scheduled messages. The demo project also contains Spring configurations.

6.2.1.2. Prepare the environment

This topic describes how to prepare an environment for a Message Queue for Apache RocketMQ demo project.

Procedure

- Install an integrated development environment (IDE).

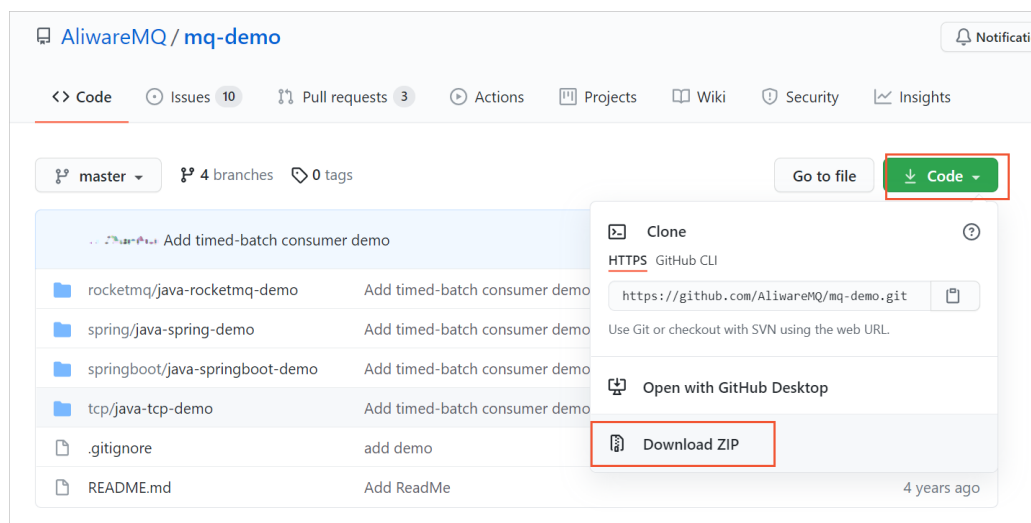
You can use IntelliJ IDEA or Eclipse as the IDE. IntelliJ IDEA is used in this example.

Download IntelliJ IDEA Ultimate Edition from [IntelliJ IDEA](#). Then, follow the installation instructions to install IntelliJ IDEA Ultimate Edition.

- Download a demo project.

Download a demo project from [GitHub](#) to your on-premises machine.

Download a demo project



After the downloaded package is decompressed, a folder named *mq-demo-master* appears on your on-premises machine.

>

6.2.1.3. Configure a demo project

This topic describes how to configure a demo project.

Prerequisites

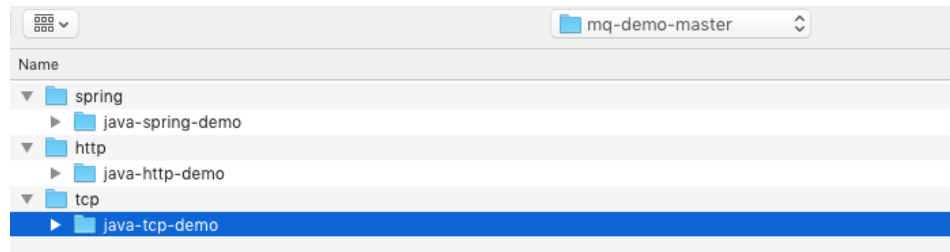
- You have prepared the environment for the demo project.
- You have installed the JDK on your on-premises machine. For more information, visit [Java SE Downloads](#). We recommend that you use JDK 8.

Procedure

1. Import the demo project to IntelliJ IDEA.

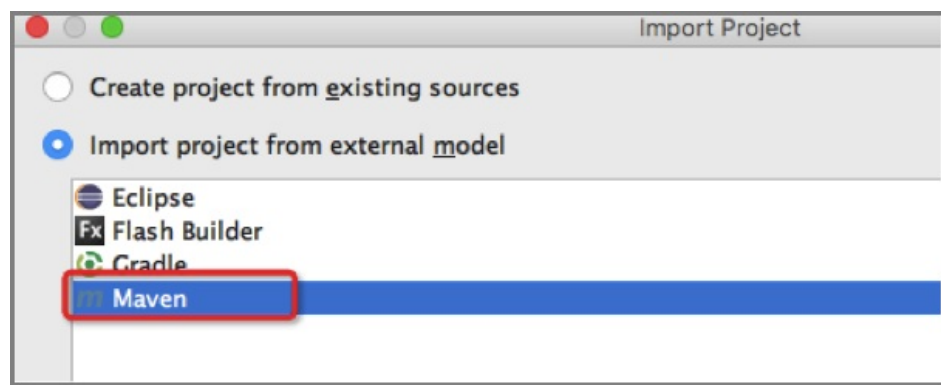
- i. On the IntelliJ IDEA page, click **Import Project** and select the *mq-demo-master* folder.

Select the mq-demo-master folder



- ii. Select **Import project from external model**.

Select Import project from external model



- iii. Click **Next** until the project is imported. The JAR dependency needs to be loaded to the demo project. Therefore, it takes two to three minutes to import the project.

2. Create resources.

Create the required resources, such as topics and group IDs in the Message Queue for Apache RocketMQ console and obtain the AccessKey pair in the Apsara Uni-manager Management Console for identity authentication.

- i. For more information about how to create topics and group IDs, see [Create resources](#).
- ii. Perform the following operations to obtain the AccessKey ID and AccessKey secret:


In the Apsara Uni-manager Management Console, move your pointer over the profile picture and select **User Information**. On the page that appears, view the AccessKey ID and AccessKey secret in the **Apsara Stack AccessKey Pair** section.

3. Configure the demo.

Configure the MqConfig class and the *common.xml* file.

- i. The following sample code provides an example on how to configure the MqConfig class:

```
public static final String TOPIC = "The topic that you created in the Message Queue  
for Apache RocketMQ console."  
public static final String GROUP_ID = "The group ID that you created in the Message  
Queue for Apache RocketMQ console."  
public static final String ACCESS_KEY = "The AccessKey ID that you created in the A  
psara Uni-manager Management Console for identity authentication."  
public static final String SECRET_KEY = "The AccessKey secret that you created in t  
he Apsara Uni-manager Management Console for identity authentication."  
public static final String NAMESRV_ADDR = "The TCP endpoint of your Message Queue f  
or Apache RocketMQ instance. You can obtain the endpoint in the Message Queue for A  
pache RocketMQ console."
```

 **Note** You must use the AccessKey ID and AccessKey secret of the account that you use to create the topic.

- ii. Configure the *common.xml* file.

```
<props>  
<prop key="AccessKey">XXX</prop> <!-- Modify the values based on your resources --  
>  
<prop key="SecretKey">XXX</prop>  
<prop key="GROUP_ID">XXX</prop>  
<prop key="Topic">XXX</prop>  
<prop key="NAMESRV_ADDR">XXX</prop>  
</props>
```

6.2.1.4. Run the demo project

After you configure the demo project, you can start the corresponding classes to send and receive messages of different types.

Call the main method to send and receive messages

1. Run the SimpleMQProducer class to send messages.
2. [Log on to the Message Queue for Apache RocketMQ console](#). In the left-side navigation pane, click **Message Query**. On the Message Query page, click the **By Topic** tab. On the By Topic tab, select the topic of the message that you sent. The query result shows that the message is sent to the topic.
3. Run the SimpleMQConsumer class to receive messages. A log is printed. The log indicates that the message is received. The class needs to be initialized. This takes several seconds. Initialization seldom occurs in the production environment.

Log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click Instance Details. On the Instance Details page, select your instance. In the left-side navigation pane, click **Groups**. On the Groups page, find the group ID that you want to view and click **Consumer Status** in the Actions column. In the **Consumer Status** panel, the information shows that the started consumers are online and the subscriptions of the consumers are consistent.

Use Spring to send and receive messages

1. Run the `ProducerClient` class to send messages.
2. Run the `ConsumerClient` class to receive messages.

Log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click **Instance Details**. On the Instance Details page, select your instance. In the left-side navigation pane, click **Groups**. On the Groups page, find the group ID that you want to view and click **Consumer Status** in the Actions column. In the **Consumer Status** panel, the information shows that the started consumers are online and the subscriptions of the consumers are consistent.

Send transactional messages

Run the `SimpleTransactionProducer` class to send messages.


Note

The `LocalTransactionCheckerImpl` class is used to check the status of local transactions. This class is used to check whether a local transaction is committed. For more information, see [Send and subscribe to transactional messages](#).

Send and receive ordered messages


Run the `SimpleOrderConsumer` class to receive messages.

Run the `SimpleOrderProducer` class to send messages.

 **Note** Ordered messages are sent and consumed in first-in-first-out (FIFO) order. For more information, see [Send and receive ordered messages](#).

Send scheduled or delayed messages

Run the `MQTimerProducer` class to send messages. These messages are delivered after a delay of 3 seconds.

 **Note** You can specify an exact delay, which is up to 40 days. For more information, see [Send and receive scheduled messages](#).

6.2.2. Client parameters

This topic describes the parameters that are configured for Message Queue for Apache RocketMQ clients.

Client parameters

Parameter	Client	Default value	Recommended value	Description	Client version
AccessKey	Producer or consumer	Configured by the user	Configured by the user	The AccessKey ID that is used to authenticate the user.	$\geq 1.2.7$.Final

Parameter	Client	Default value	Recommended value	Description	Client version
SecretKey	Producer or consumer	Configured by the user	Configured by the user	The AccessKey secret that is used to authenticate the user.	$\geq 1.2.7$.Final
NAMESRV_ADDR	Producer or consumer	Generated after deployment	Generated after deployment	The endpoint that is used to connect to Message Queue for Apache RocketMQ.	$\geq 1.2.7$.Final
MsgTraceSwitch	Producer or consumer	true	true	Specifies whether to enable the message tracing feature of Message Queue for Apache RocketMQ.	$\geq 1.7.0$.Final
GROUP_ID	Producer or consumer	Created in the console	Created in the console	The ID of the group to which the producer or consumer client belongs. Group IDs are compatible with producer IDs (PIDs) or consumer IDs (CIDs) in earlier versions.	$\geq 1.7.8$.Final

Parameter	Client	Default value	Recommended value	Description	Client version
ProducerId	Producer	Created in the console	Created in the console	The ID of the group to which the producer client belongs. This parameter takes effect only on transactional messages. If a producer client fails, the Message Queue for Apache RocketMQ broker initiates requests to check the status of transactional messages on other producer clients in the same group.	>=1.2.7.Final
SendMessageTimeoutMillis	Producer	5000	Default	The timeout period for sending a message. If the Message Queue for Apache RocketMQ broker does not return an acknowledgment to the producer client within the specified period of time, the producer client determines that the message failed to send.	>=1.2.7.Final

Parameter	Client	Default value	Recommended value	Description	Client version
ConsumerId	Consumer	Created in the console	Created in the console	The ID of the group to which the consumer client belongs.	$\geq 1.2.7.Final$
MessageModel	Consumer	CLUSTERING	Default	The consumption mode. Valid values: CLUSTERING and BROADCASTING. CLUSTERING specifies that each subscribed message is received only by one consumer client. BROADCASTING specifies that each subscribed message is received by all consumer clients.	$\geq 1.2.7.Final$
ConsumeThreadNums	Consumer	Dynamically adjusted from 20 to 64	Adjusted based on business requirements	The number of consumer threads. In most cases, this parameter is set to a larger value if a longer time is required to consume a single message.	$\geq 1.2.7.Final$

Parameter	Client	Default value	Recommended value	Description	Client version
MaxReconsumeTimes	Consumer	16	Default	The maximum number of delivery retries that can be performed when a message fails to be consumed.	>=1.2.7.Final
ConsumeTimeout	Consumer	15	Default	The timeout period for consumption of each message. If the time to consume a message exceeds the specified timeout period, the message fails to be consumed and is redelivered after a retry interval. Configure an appropriate value for each type of application. Unit: minute.	>=1.2.7.Final

Parameter	Client	Default value	Recommended value	Description	Client version
PostSubscriptionWhenPull	Consumer	false	Adjusted based on the consumption mode	Specifies whether to carry the latest subscription together with each request. If MessageModel is set to BROADCASTING, this parameter must be set to true to prevent messages from failing to be received due to subscription inconsistency. If MessageModel is set to CLUSTERING, this parameter must be set to false because subscription consistency is required for clustering consumption.	>=1.2.7.Final

Parameter	Client	Default value	Recommended value	Description	Client version
ConsumeMessageBatchMaxSize	Consumer	1	Adjusted based on business requirements	The maximum number of messages that can be consumed in each batch. The actual number of messages that are consumed in a batch can be smaller than the value of this parameter. The value must be an integer from 1 to 32. The default value is 1.	>=1.6.0.Final
MaxCachedMessageAmount	Consumer	5000	Adjusted based on the memory of consumer clients	The maximum number of messages that a consumer client can cache. A large value can cause an out of memory (OOM) issue on the client. The value must be an integer from 100 to 50000. The default value is 5000.	>=1.7.0.Final

Parameter	Client	Default value	Recommended value	Description	Client version
MaxCachedMessageSizeInMiB	Consumer	512	Adjusted based on the memory of consumer clients	The maximum size of messages that a consumer client can cache. A large value can cause an out of memory (OOM) issue on the client. The value ranges from 16 to 2048. The default value is 512. Unit: MB.	>=1.7.0.Final

6.2.3. Client error codes

This topic describes error codes related to sending and subscribing to messages and their references.

Error codes related to sending and subscribing to messages

HTTP status code	Status flag	Description	Cause and recommended solution	Broker version
13	MESSAGE_ILLEGAL	This error is returned when the message verification fails.	<p>Check whether the message body is empty.</p> <p>Check whether the length of the message property exceeds 32,767 bytes.</p> <p>Check whether the total size of the message exceeds 4 MB.</p>	>=4.0.1

HTTP status code	Status flag	Description	Cause and recommended solution	Broker version
17	TOPIC_NOT_EXIST	This error is returned when the specified message topic does not exist.	<ol style="list-style-type: none">1. Create a topic in the Message Queue for Apache RocketMQ console.2.Restart your application. <p>For more information, see the "Nonexistent topic" section in the Nonexistent resources topic.</p>	>=4.0.1
26	SUBSCRIPTION_GROUP_NOT_EXIST	This error is returned if the specified group ID does not exist.	<ol style="list-style-type: none">1.Create a group ID in the Message Queue for Apache RocketMQ console.2.Restart your application. <p>For more information, see the "Nonexistent group ID" section in the Nonexistent resources topic.</p>	>=4.0.1
24	SUBSCRIPTION_NOT_EXIST	This error is returned when the subscription does not exist.	<ol style="list-style-type: none">1.Check whether the consumers identified by the group ID have been started.2.Check whether subscription inconsistency occurs between consumers identified by the group ID.	>=4.0.1

HTTP status code	Status flag	Description	Cause and recommended solution	Broker version
23	SUBSCRIPTION_PARSE_FAILED	This error is returned when the system failed to parse the subscription expression.	Check the corresponding topic subscription expression and tag.	>=4.0.1
25	SUBSCRIPTION_NOT_LATEST	This error is returned if subscription inconsistency occurs.	If this status continues for a moment, it is automatically restored. For more information, see Subscription inconsistency .	>=4.0.1
14	SERVICE_NOT_AVAILABLE	This error is returned when messages cannot be sent.	The requested Message Queue for Apache RocketMQ broker is discontinued, the broker is abnormal and does not support write operations, or the broker is a standby broker. For more information, see the "The message failed to be sent." section in the Usage-related exceptions topic.	>=4.0.1

HTTP status code	Status flag	Description	Cause and recommended solution	Broker version
16	NO_PERMISSION (message sending)	This error is returned when the request is invalid.	<p>The requested Message Queue for Apache RocketMQ broker disallows write operations.</p> <p>The topic on the requested Message Queue for Apache RocketMQ broker disallows write operations.</p> <p>The requested Message Queue for Apache RocketMQ broker disallows transactional messages.</p>	>=4.0.1
16	NO_PERMISSION (message subscription)	This error is returned when the request is invalid.	<p>The requested Message Queue for Apache RocketMQ broker disallows read operations.</p> <p>The current consumer group does not have the read permissions.</p> <p>The pulled topic disallows read operations.</p> <p>The current consumer group disallows message broadcasting.</p>	>=4.0.1

HTTP status code	Status flag	Description	Cause and recommended solution	Broker version
1	SYSTEM_ERROR	This error is returned when a system exception occurs.	<p>This is a temporary timeout that results from the restart of the Message Queue for Apache RocketMQ broker or heavy load on the broker.</p> <p>For more information, see the "The message failed to be sent." section in the Usage-related exceptions topic.</p>	>=4.0.1
1	SYSTEM_ERROR (permission verification)	This error is returned when the permission verification fails.	Check whether the user is granted the permissions to publish messages to and subscribe messages from the topic.	>=4.0.1
2	SYSTEM_BUSY	This error is returned when the system is busy and the request is denied.	<p>This is a temporary timeout that results from the restart of the Message Queue for Apache RocketMQ broker or heavy load on the broker.</p> <p>For more information, see the "The message failed to be sent." section in the Usage-related exceptions topic.</p>	>=4.0.1

6.2.4. SDK for Java

6.2.4.1. Usage notes

Message Queue for Apache RocketMQ provides SDK for Java for you to send and subscribe to messages. This topic describes the parameters of Java methods and how to call these methods.

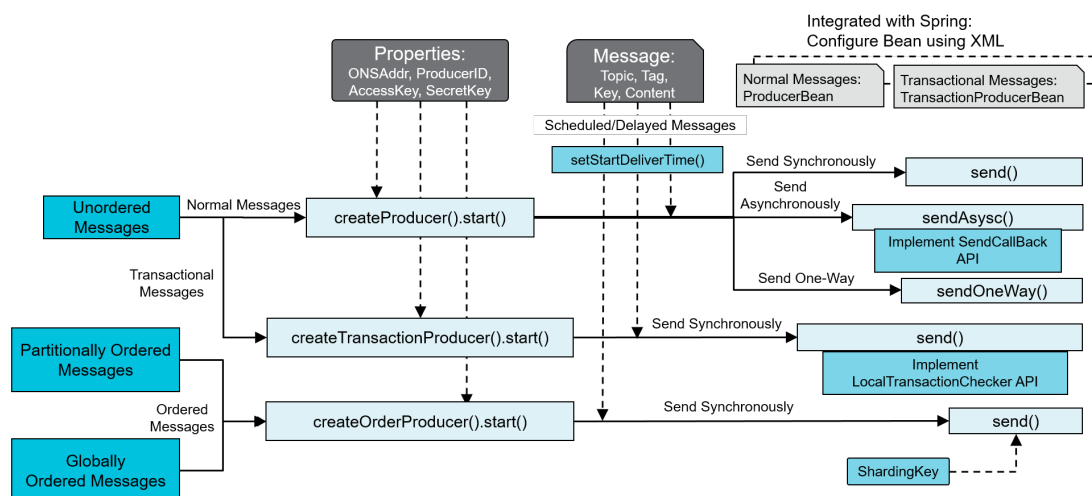
Common parameters

Parameter	Description
NAMESRV_ADDR	The TCP endpoint. You can obtain the endpoint on the Instance Details page in the Message Queue for Apache RocketMQ console.
AccessKey	The AccessKey ID that you created in the Apsara Uni-manager Management Console for identity authentication.
SecretKey	The AccessKey secret that you created in the Apsara Uni-manager Management Console for identity authentication.
OnsChannel	The source of the user. Default value: ALIYUN.

Parameters for sending messages

Parameter	Description
SendMsgTimeoutMillis	The timeout period for sending messages. Unit: milliseconds. Default value: 3000.
CheckImmunityTimeInSeconds (for transactional messages)	The shortest time interval before the first back-check for the status of local transaction. Unit: seconds.
shardingKey (for ordered messages)	The partition key that is used to determine the partitions to which ordered messages are distributed.

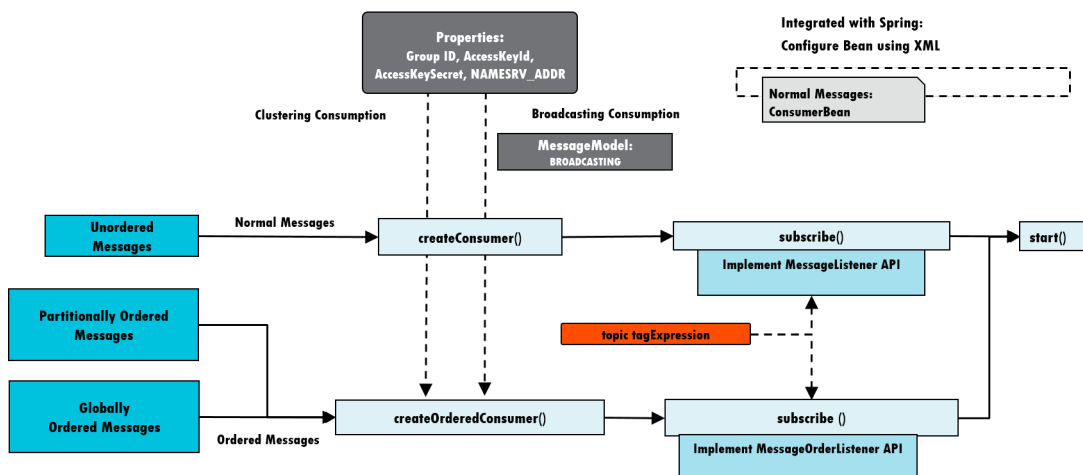
Methods and parameters for using SDK for Java to send messages



Parameters for subscribing to messages

Parameter	Description
GROUP_ID	The group ID that you created in the Message Queue for Apache RocketMQ console.
MessageModel	The mode in which a consumer instance consumes messages. Valid values: CLUSTERING and BROADCASTING. Default value: CLUSTERING.
ConsumeThreadNums	The number of consumer threads for a consumer instance. Default value: 64.
MaxReconsumeTimes	The maximum number of delivery retries for a message that fails to be consumed. Default value: 16.
ConsumeTimeout	The maximum timeout period for consuming a message. If a message fails to be consumed within this period, the consumption fails and the message can be redelivered. A proper value must be set for each type of business. Unit: minutes. Default value: 15.
suspendTimeMillis (for ordered messages)	The interval between delivery retries for an ordered message that fails to be consumed.
maxCachedMessageAmount	The maximum number of messages cached on the on-premises client. Default value: 1000.
maxCachedMessageSizeInMiB	The maximum size of messages cached on the on-premises client. Valid values: 16 MB to 2 GB. Default value: 512 MB.

Methods and parameters for using SDK for Java to subscribe to messages



Sample code for sending and subscribing to messages

- [Send and subscribe to normal messages](#)
- [Send and receive ordered messages](#)
- [Send and receive scheduled messages](#)
- [Send and receive delayed messages](#)
- [Send and subscribe to transactional messages](#)

6.2.4.2. Prepare the environment

Before you run the Java code provided in this topic, prepare the environment based on the following instructions:

Procedure

1. Introduce the dependency by using one of the following methods:

- Introduce the dependency by using Maven:

```
<dependency>
  <groupId>com.aliyun.openservices</groupId>
  <artifactId>ons-client</artifactId>
  <version>1.8.4.Final</version>
</dependency>
```

- Download the [JAR dependency](#).
2. Go to the console to create the topics and group IDs involved in the code.
You can customize message tags in your application. For more information about how to create a message tag, see [Create resources](#).
 3. For applications that use the TCP client SDK to access Message Queue for Apache RocketMQ, make sure that the applications are deployed on Elastic Compute Service (ECS) instances in the same region.

6.2.4.3. Configure logging

Client logs record exceptions that occur when the Message Queue for Apache RocketMQ clients are running. Client logs help you locate and handle these exceptions in a quick manner. This topic describes how to print the logs of a Message Queue for Apache RocketMQ client and provides the default and custom configurations.

Print client logs

TCP client SDK for Java of Message Queue for Apache RocketMQ is programmed by using the Simple Logging Facade for Java (SLF4J).

- **Message Queue for Apache RocketMQ SDK for Java 1.7.8.Final or later**

Message Queue for Apache RocketMQ SDK for Java 1.7.8.Final has a built-in framework for logging. You do not need to add a dependency on the corresponding logging framework for an application on the client before you print the logs of a Message Queue for Apache RocketMQ client.

For information about the default logging configuration for a Message Queue for Apache RocketMQ client and how to modify this configuration, see [Configure client logs](#).

- **Message Queue for Apache RocketMQ SDK for Java versions earlier than 1.7.8.Final**

Message Queue for Apache RocketMQ SDK for Java versions earlier than 1.7.8.Final support only Log4j and Logback. These versions do not support Log4j2. For these versions, you must add a dependency on the corresponding logging framework to the *pom.xml* file or the *.lib* file before you print the logs of a Message Queue for Apache RocketMQ client.

The following sample code provides examples on how to add dependencies on Log4j and Logback:

◦ Method 1: Use Log4j as the logging framework

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>jcl-over-slf4j</artifactId>
  <version>1.7.7</version>
</dependency>
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>1.7.7</version>
</dependency>
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.17</version>
</dependency>
```

◦ Method 2: Use Logback as the logging framework

```
<dependency>
  <groupId>ch.qos.logback</groupId>
  <artifactId>logback-core</artifactId>
  <version>1.1.2</version>
</dependency>
<dependency>
  <groupId>ch.qos.logback</groupId>
  <artifactId>logback-classic</artifactId>
  <version>1.1.2</version>
</dependency>
```

Note

If an application uses both Log4j and Logback as logging frameworks, client logs cannot be properly printed due to logging conflicts. To properly print the logs of a Message Queue for Apache RocketMQ client, make sure that you add only one dependency on one logging framework for the application. We recommend that you run the **mvn clean dependency:tree | grep log** command to check whether your application uses only one of the logging frameworks.

Configure logging for a Message Queue for Apache RocketMQ client

You can customize the following settings for a Message Queue for Apache RocketMQ client: the path for storing log files, log level, and maximum number of historical log files retained. To facilitate log transmission and viewing, the maximum size of a single log file retains the default value of 64 MB. This value cannot be changed.

The following table describes these parameters that you can configure.

Parameter	Description
-----------	-------------

Parameter	Description
The path to store log files	Make sure that the application has the write permissions for this path. Otherwise, logs cannot be printed.
The maximum number of historical log files that are retained	You can set this parameter to a value between 1 and 100. If you enter a value that is not within this range or a value that is in an invalid format, the system retains 10 historical log files by default.
The log level	You can set this parameter to one of the following values: ERROR, WARN, INFO, and DEBUG. If this parameter is set to an invalid value, the system uses the default value INFO.

• Default configuration

After you start a Message Queue for Apache RocketMQ client, the client generates log files based on the following default configuration:

- The path to store log files: `{user.home}/logs/ons.log`, where `{user.home}` is the root directory of the account that runs the current Java process.
- The maximum number of historical log files that are retained: 10
- Log level: INFO
- The maximum size of a single log file: 64 MB

• Custom configuration

Note

To customize the logging configuration of a Message Queue for Apache RocketMQ client, update the SDK for Java to V1.2.5 or later.

To customize the logging configuration of a Message Queue for Apache RocketMQ client in the SDK for Java, configure the following system parameters:

- `ons.client.logRoot`: the path to store log files
- `ons.client.logFileMaxIndex`: the maximum number of historical log files that are retained
- `ons.client.logLevel`: the log level

Examples

Add the following system parameters to the startup script or integrated development environment (IDE) virtual machine (VM) options:

◦ Linux

```
-Dons.client.logRoot=/home/admin/logs -Dons.client.logLevel=WARN -Dons.client.logFileMaxIndex=20
```

- **Windows**

```
-Dons.client.logRoot=D:\logs -Dons.client.logLevel=WARN -Dons.client.logFileMaxIndex=20
```

`/home/admin/` and `D:\` are only examples. Replace them with your system directories.

6.2.4.4. Spring integration

6.2.4.4.1. Overview

This topic describes how to send and subscribe to messages by using Message Queue for Apache RocketMQ in the Spring framework. This topic includes three parts: the integration of a normal message producer and Spring, the integration of a transactional message producer and Spring, and the integration of a message consumer and Spring.

The subscriptions of all consumer instances identified by the same group ID must be consistent. For more information, see [Subscription consistency](#).

The configuration parameters supported in the Spring framework are the same as those used in TCP client SDK for Java. For more information, see [How to use the Java SDK](#).

6.2.4.4.2. Integrate a producer with Spring

This topic describes how to integrate a producer with Spring.

Procedure

1. Define information such as the producer bean in *producer.xml*.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="producer" class="com.aliyun.openservices.ons.api.bean.ProducerBean" init-method="start" destroy-method="shutdown">
        <!-- The Spring framework supports all the configuration items that SDK for Java supports. -->
        <property name="properties" > <!-- Configurations of the producer -->
            <props>
                <prop key="AccessKey">XXX</prop>
                <prop key="SecretKey">XXX</prop>
                <!-- The ons-client version is 1.8.4.Final, which must be configured.
                You can obtain the TCP endpoint on the Instance Details page in the Message Queue for Apache RocketMQ console.
                <prop key="NAMESRV_ADDR">XXX</prop>
            -->
            </props>
        </property>
    </bean>
</beans>
```

2. Produce messages by using the producer that is integrated with Spring.

```

package demo;
import com.aliyun.openservices.ons.api.Message;
import com.aliyun.openservices.ons.api.Producer;
import com.aliyun.openservices.ons.api.SendResult;
import com.aliyun.openservices.ons.api.exception.ONSClientException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class ProduceWithSpring {
    public static void main(String[] args) {
        /**
         * The producer bean is configured in producer.xml. You can call the Applicati
         onContext class to obtain the bean or inject the bean to other classes, such as a speci
         fic controller.
         */
        ApplicationContext context = new ClassPathXmlApplicationContext("producer.xml"
);
        Producer producer = (Producer) context.getBean("producer");
        // Cyclically send messages.
        for (int i = 0; i < 100; i++) {
            Message msg = new Message( //
                // The topic of the message.
                "TopicTestMQ",
                // The message tag, which is similar to a Gmail tag. The message t
                ag is used to sort messages and helps the consumer filter messages on the Message Queue
                for Apache RocketMQ broker based on specified conditions.
                "TagA",
                // The message body in the binary format. Message Queue for Apache
                RocketMQ does not process the message body.
                // The producer and consumer must agree on the serialization and d
                eserialization methods.
                "Hello MQ".getBytes());
            // The key of the message. The key is the business-specific attribute of t
            he message and must be globally unique whenever possible.
            // A unique key helps you query and resend a message in the Message Queue
            for Apache RocketMQ console if the message fails to be received.
            // Note: Messages can be sent and received even if you do not set this par
            ameter.
            msg.setKey("ORDERID_100");
            // Send the message. If no error occurs, the message is sent.
            try {
                SendResult sendResult = producer.send(msg);
                assert sendResult != null;
                System.out.println("send success: " + sendResult.getMessageId());
            } catch (ONSClientException e) {
                System.out.println("failed to send the message");
            }
        }
    }
}

```

6.2.4.4.3. Integrate a transactional message producer with Spring

This topic describes how to integrate a producer that produces transactional messages with Spring.

Context

For more information about transactional messages, see [Send and subscribe to transactional messages](#).

Procedure

1. Implement the `LocalTransactionChecker` class. A producer can have only one `LocalTransactionChecker` class.

```
package demo;
import com.aliyun.openservices.ons.api.Message;
import com.aliyun.openservices.ons.api.transaction.LocalTransactionChecker;
import com.aliyun.openservices.ons.api.transaction.TransactionStatus;
public class DemoLocalTransactionChecker implements LocalTransactionChecker {
    public TransactionStatus check(Message msg) {
        System.out.println("Start to back-check the status of local transaction.");
        return TransactionStatus.CommitTransaction; // Returns different values for TransactionStatus based on the status check result of the local transaction.
    }
}
```

2. Define information such as the producer bean in *transactionProducer.xml*.

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="localTransactionChecker" class="demo.DemoLocalTransactionChecker"></bean>
    <bean id="transactionProducer" class="com.aliyun.openservices.ons.api.bean.TransactionProducerBean" init-method="start" destroy-method="shutdown">
        <property name="properties" > <!-- Configurations of the transactional message producer -->
            <props>
                <prop key="AccessKey">AKDEMO</prop>
                <prop key="SecretKey">SKDEMO</prop>
                <prop key="GROUP_ID">GID_DEMO</prop>
                <!-- The ons-client version is 1.8.4.Final, which must be configured.
                You can obtain the TCP endpoint on the Instance Details page in the Message Queue for Apache RocketMQ console.
                <prop key="NAMESRV_ADDR">XXX</prop>
                -->
            </props>
        </property>
        <property name="localTransactionChecker" ref="localTransactionChecker"></property>
    </bean>
</beans>

```

3. Produce transactional messages by using the producer that is integrated with Spring.

```
package demo;
import com.aliyun.openservices.ons.api.Message;
import com.aliyun.openservices.ons.api.SendResult;
import com.aliyun.openservices.ons.api.transaction.LocalTransactionExecuter;
import com.aliyun.openservices.ons.api.transaction.TransactionProducer;
import com.aliyun.openservices.ons.api.transaction.TransactionStatus;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class ProduceTransMsgWithSpring {
    public static void main(String[] args) {
        /**
         * The bean of the transactional message producer is configured in transaction
         * Producer.xml. You can call the ApplicationContext class to obtain the bean or inject th
         * e bean to other classes, such as a specific controller.
         * Send transactional messages.
         */
        ApplicationContext context = new ClassPathXmlApplicationContext("transactionPr
        oducer.xml");
        TransactionProducer transactionProducer = (TransactionProducer) context.getBea
        n("transactionProducer");
        Message msg = new Message("XXX", "TagA", "Hello MQ transaction===".getBytes())
        ;
        SendResult sendResult = transactionProducer.send(msg, new LocalTransactionExec
        uter() {
            @Override
            public TransactionStatus execute(Message msg, Object arg) {
                System.out.println("A local transaction is executed.");
                return TransactionStatus.CommitTransaction; // Returns different value
                s for TransactionStatus based on the execution result of the local transaction.
            }
        }, null);
    }
}
```

6.2.4.4.4. Integrate a consumer with Spring

This topic describes how to integrate a consumer with Spring.

Procedure

1. Create a message listener. The following sample code provides an example:

```

package demo;
import com.aliyun.openservices.ons.api.Action;
import com.aliyun.openservices.ons.api.ConsumeContext;
import com.aliyun.openservices.ons.api.Message;
import com.aliyun.openservices.ons.api.MessageListener;
public class DemoMessageListener implements MessageListener {
    public Action consume(Message message, ConsumeContext context) {
        System.out.println("Receive: " + message.getMsgID());
        try {
            //do something..
            return Action.CommitMessage;
        } catch (Exception e) {
            // The message failed to be consumed.
            return Action.ReconsumeLater;
        }
    }
}

```

2. Define information such as the consumer bean in *consumer.xml*.

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="msgListener" class="demo.DemoMessageListener"></bean> <!--Configurations
of the message listener-->
    <!-- When multiple consumers identified by the same group ID subscribe to the same top
ic, you can create multiple consumer beans. -->
    <bean id="consumer" class="com.aliyun.openservices.ons.api.bean.ConsumerBean" init
-method="start" destroy-method="shutdown">
        <property name="properties" > <!-- Configurations of the consumer -->
            <props>
                <prop key="GROUP_ID">GID_DEMO</prop> <!-- Replace the value with the g
roup ID that you created in the console. -->
                <prop key="AccessKey">AKDEMO</prop>
                <prop key="SecretKey">SKDEMO</prop>
                <!-- The ons-client version is 1.8.4.Final, which must be configured.
You can obtain the TCP endpoint on the Instance Details page in the Message Queue for A
pache RocketMQ console.
                <prop key="NAMESRV_ADDR">XXX</prop>
                -->
                <!-- Set the number of consumer threads to 50.
                <prop key="ConsumeThreadNums">50</prop>
                -->
            </props>
        </property>
        <property name="subscriptionTable">
            <map>
                <entry value-ref="msgListener">
                    <key>
                        <bean class="com.aliyun.openservices.ons.api.bean.Subscription
">
                            <property name="topic" value="TopicTestMQ"/>

```

```

        <property name="expression" value="*" /><!--The expression
is the tag. You can set the value to a specific tag or *. For example, a specific tag c
an be taga||tagb||tagc. * indicates that all tags are subscribed to. Wildcards are not
supported. -->
    </bean>
</key>
</entry>
<!-- Add entry nodes to subscribe to more tags. -->
<entry value-ref="msgListener">
    <key>
        <bean class="com.aliyun.openservices.ons.api.bean.Subscription
">
            <property name="topic" value="TopicTestMQ-Other" /> <!--Sub
scribe to another topic. -->
            <property name="expression" value="taga||tagb" /> <!-- Subs
cribe to multiple tags. -->
        </bean>
    </key>
</entry>
</map>
</property>
</bean>
</beans>

```

3. Run the consumer that is integrated with Spring.

```


package demo;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class ConsumeWithSpring {
    public static void main(String[] args) {
        /**
         * The consumer bean is configured in consumer.xml. You can call the Applicatio
nContext class to obtain the bean or inject the bean to other classes, such as a specif
ic controller.
         */
        ApplicationContext context = new ClassPathXmlApplicationContext("consumer.xml")
;
        System.out.println("Consumer Started");
    }
}

```

6.2.4.5. Three modes for sending messages

6.2.4.5.1. Overview

In Message Queue for Apache RocketMQ, messages can be sent in reliable synchronous mode, reliable asynchronous mode, and one-way mode. This topic describes the principles, scenarios, and differences of these modes, and provides sample code for your reference.

 **Note** Ordered messages can be sent only in reliable synchronous mode.

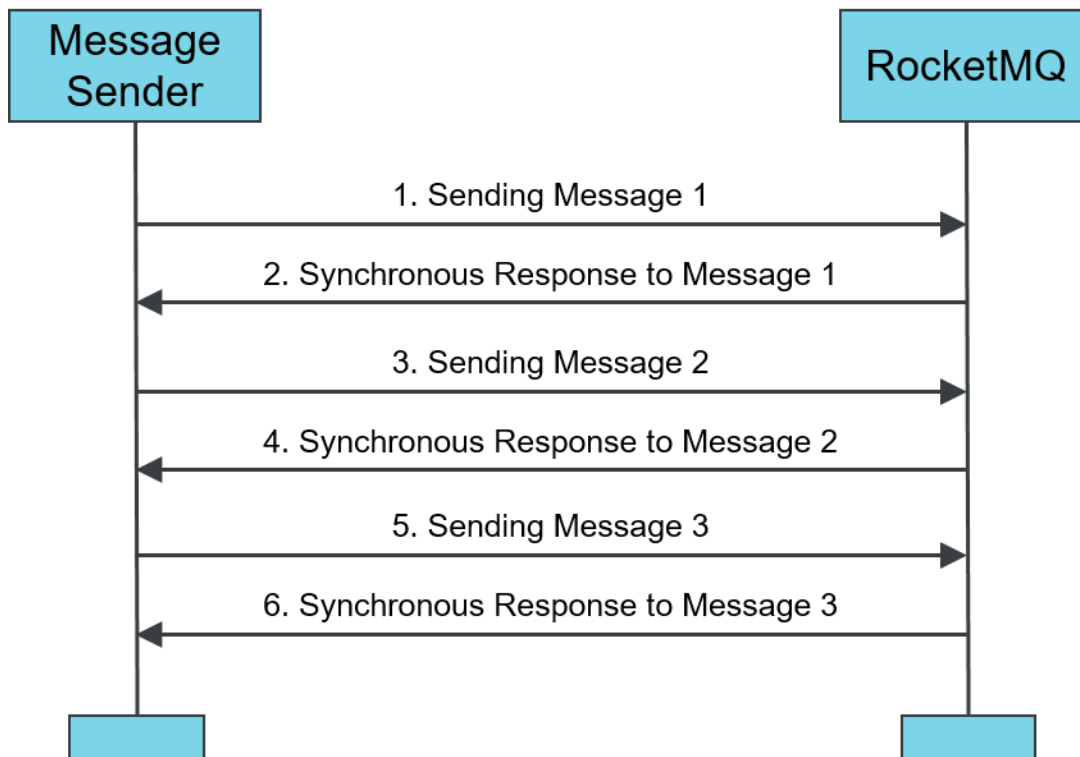
6.2.4.5.2. Reliable synchronous transmission

This topic describes the principle and scenarios of the reliable synchronous transmission mode.

How it works

Synchronous transmission means that the message producer sends the next message only after it receives a response to the previous message from the broker.

Synchronous transmission



Scenarios

This mode is applicable to various scenarios, such as important notification emails, short message service (SMS) notifications for registration results, and SMS marketing systems.

Sample code

```

import com.aliyun.openservices.ons.api.Message;
import com.aliyun.openservices.ons.api.Producer;
import com.aliyun.openservices.ons.api.SendResult;
import com.aliyun.openservices.ons.api.ONSFactory;
import com.aliyun.openservices.ons.api.PropertyKeyConst;
import java.util.Properties;

public class ProducerTest {
    public static void main(String[] args) {
        Properties properties = new Properties();
        // The AccessKey ID that you created in the Apsara Uni-manager Management Console for identity authentication.
        properties.put(PropertyKeyConst.AccessKey, "XXX");
        // The AccessKey secret that you created in the Apsara Uni-manager Management Console for identity authentication.
    }
}
  
```

```

// For identity authentication.
properties.put(PropertyKeyConst.SecretKey, "XXX");
// The timeout interval for sending a message, in milliseconds.
properties.setProperty(PropertyKeyConst.SendMsgTimeoutMillis, "3000");
// The TCP endpoint. To obtain the endpoint, log on to the Message Queue for Apache
RocketMQ console. In the left-side navigation pane, click Instance Details. On the Instance
Details page, select your instance. On the Instance Information tab, view the endpoint in t
he Obtain Endpoint Information section.
properties.put(PropertyKeyConst.NAMESRV_ADDR,
    "XXX");
Producer producer = ONSFactory.createProducer(properties);
// Before you use the producer to send a message, call the start() method once to s
tart the producer.
producer.start();
// Cyclically send messages.
for (int i = 0; i < 10; i++){
    Message msg = new Message( //
        // The topic of the message.
        "TopicTestMQ",
        // The message tag, which is similar to a Gmail tag. The message tag is use
d to sort messages and helps the consumer filter messages on the Message Queue for Apache R
ocketMQ broker based on specified conditions.
        "TagA",
        // The message body in the binary format. Message Queue for Apache RocketMQ
does not process the message body.
        // The producer and consumer must agree on the serialization and deserializ
ation methods.
        "Hello MQ".getBytes());
    // The key of the message. The key is the business-specific attribute of the me
ssage and must be globally unique whenever possible.
    // A unique key helps you query and resend a message in the Message Queue for A
pache RocketMQ console if the message fails to be received.
    // Note: Messages can be sent and received even if you do not specify the messa
ge key.
    msg.setKey("ORDERID_" + i);
    try {
        SendResult sendResult = producer.send(msg);
        // Send the message in synchronous mode. If no error occurs, the message is
sent.
        if (sendResult != null) {
            System.out.println(new Date() + " Send mq message success. Topic is:" +
msg.getTopic() + " msgId is: " + sendResult.getMessageId());
        }
    }
    catch (Exception e) {
        // Specify the logic to resend or persist the message if the message fails
to be sent.
        System.out.println(new Date() + " Send mq message failed. Topic is:" + msg.
getTopic());
        e.printStackTrace();
    }
}
// Before you exit the application, shut down the producer object.
// Note: You can choose not to shut down the producer object.
producer.shutdown();

```

```

    }
}

```

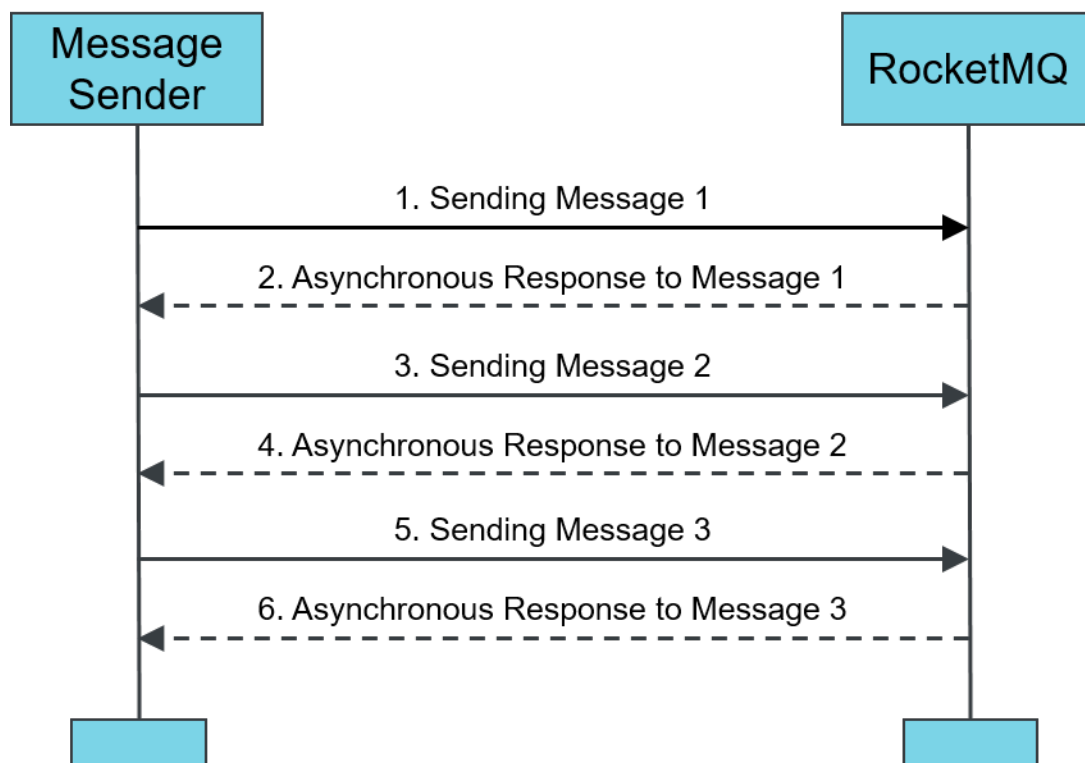
6.2.4.5.3. Reliable asynchronous transmission

This topic describes the principle and scenarios of the reliable asynchronous transmission mode.

How it works

In reliable asynchronous transmission mode, a producer sends the next message without waiting for a response to the previous message from the Message Queue for Apache RocketMQ broker. This mode uses the `SendCallback` method to fire a callback after a message is sent. An application sends the next message before it receives a response to the previous message from the Message Queue for Apache RocketMQ broker. After the `SendCallback` method is called, the application receives the response to the previous message from the Message Queue for Apache RocketMQ broker and processes the response.

Asynchronous transmission



Scenarios

This mode is used for time-consuming processes in business scenarios that are sensitive to the response time. For example, after you upload a video, a callback is fired to enable transcoding. After the video is transcoded, a callback is fired to push transcoding results.

Sample code

```

import com.aliyun.openservices.ons.api.Message;
import com.aliyun.openservices.ons.api.OnExceptionContext;
import com.aliyun.openservices.ons.api.Producer;
import com.aliyun.openservices.ons.api.SendCallback;

```

```

import com.aliyun.openservices.ons.api.SendResult;
import com.aliyun.openservices.ons.api.ONSFactory;
import com.aliyun.openservices.ons.api.PropertyKeyConst;
import java.util.Properties;

public static void main(String[] args) {
    Properties properties = new Properties();
    // The AccessKey ID that you created in the Apsara Uni-manager Management Console for identity authentication.
    properties.put(PropertyKeyConst.AccessKey, "XXX");
    // The AccessKey secret that you created in the Apsara Uni-manager Management Console for identity authentication.
    properties.put(PropertyKeyConst.SecretKey, "XXX");
    // The timeout interval for sending a message, in milliseconds.
    properties.setProperty(PropertyKeyConst.SendMsgTimeoutMillis, "3000");
    // The TCP endpoint. To obtain the endpoint, log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click Instance Details. On the Instance Details page, select your instance. On the Instance Information tab, view the endpoint in the Obtain Endpoint Information section.
    properties.put(PropertyKeyConst.NAMESRV_ADDR,
        "XXX");
    Producer producer = ONSFactory.createProducer(properties);
    // Before you use the producer to send a message, call the start() method once to start the producer.
    producer.start();
    Message msg = new Message(
        // The topic of the message.
        "TopicTestMQ",
        // The message tag, which is similar to a Gmail tag. The message tag is used to sort messages and helps the consumer filter messages on the Message Queue for Apache RocketMQ broker based on specified conditions.
        "TagA",
        // The message body in the binary format. Message Queue for Apache RocketMQ does not process the message body. The producer and consumer must agree on the serialization and deserialization methods.
        "Hello MQ".getBytes());
    // The key of the message. The key is the business-specific attribute of the message and must be globally unique whenever possible. // A unique key helps you query and resend a message in the Message Queue for Apache RocketMQ console if the message fails to be received.
    // Note: Messages can be sent and received even if you do not set this parameter.
    msg.setKey("ORDERID_100");
    // Send the message in asynchronous mode. The result is returned to the producer after the producer calls the callback function.
    producer.sendAsync(msg, new SendCallback() {
        @Override
        public void onSuccess(final SendResult sendResult) {
            // The message is sent to the consumer.
            System.out.println("send message success. topic=" + sendResult.getTopic() + ", msgId=" + sendResult.getMessageId());
        }
        @Override
        public void onException(OnExceptionContext context) {
            // Specify the logic to resend or persist the message if the message fails to be sent.
            System.out.println("send message failed. topic=" + context.getTopic() + "

```

```
        System.out.println("send message failed. topic=" + context.getTopic() + ", " +
        msgId=" + context.getMessageId());
    }
    });
    // The message ID can be obtained before the callback function returns the result.
    System.out.println("send message async. topic=" + msg.getTopic() + ", msgId=" + msg
    .getMsgID());
    // Before you exit the application, shut down the producer object. Note: You can c
    hoose not to shut down the producer object.
    producer.shutdown();
}
```

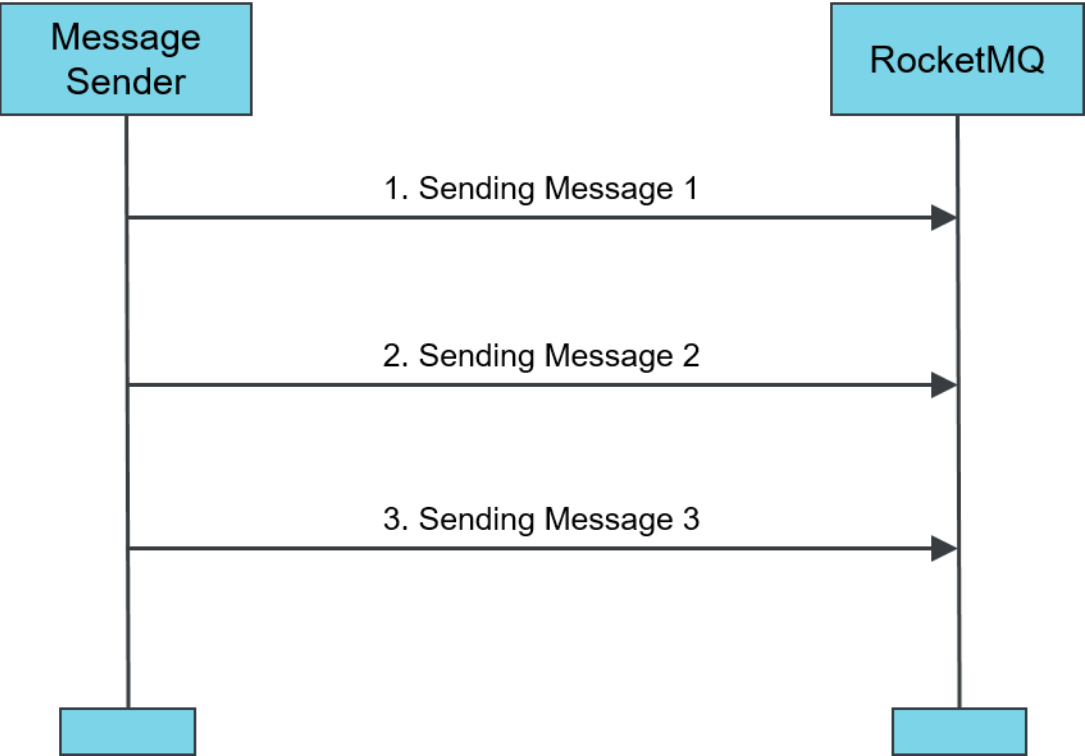
6.2.4.5.4. One-way transmission

This topic describes the principle and scenarios of the one-way transmission mode, and provides sample code.

How it works

In one-way transmission mode, a producer only sends messages and does not wait for a response from the Message Queue for Apache RocketMQ broker. In addition, no callback function is triggered. In this mode, a message can be sent within microseconds.

One-way transmission



Scenarios

This mode is applicable to scenarios where message transmission takes a short time and has no demanding reliability requirements. For example, this mode can be used for log collection.

The following table summarizes the features and major differences among the three modes.

Transmission mode	Transactions per second (TPS)	Response	Reliability
Synchronous transmission	High	Supported	No message loss
Asynchronous transmission	High	Supported	No message loss
One-way transmission	Highest	None	Possible message loss

Sample code

```
import com.aliyun.openservices.ons.api.Message;
import com.aliyun.openservices.ons.api.Producer;
import com.aliyun.openservices.ons.api.ONSFactory;
import com.aliyun.openservices.ons.api.PropertyKeyConst;
import java.util.Properties;

public static void main(String[] args) {
    Properties properties = new Properties();
    // The AccessKey ID that you created in the Apsara Uni-manager Management Console for identity authentication.
    properties.put(PropertyKeyConst.AccessKey, "XXX");
    // The AccessKey secret that you created in the Apsara Uni-manager Management Console for identity authentication.
    properties.put(PropertyKeyConst.SecretKey, "XXX");
    // The timeout interval for sending a message, in milliseconds.
    properties.setProperty(PropertyKeyConst.SendMsgTimeoutMillis, "3000");
    // The TCP endpoint. To obtain the endpoint, log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click Instance Details. On the Instance Details page, select your instance. On the Instance Information tab, view the endpoint in the Obtain Endpoint Information section.
    properties.put(PropertyKeyConst.NAMESRV_ADDR, "XXX");
    Producer producer = ONSFactory.createProducer(properties);
    // Before you use the producer to send a message, call the start() method once to start the producer.
    producer.start();
    // Cyclically send messages.
    for (int i = 0; i < 10; i++){
        Message msg = new Message(
            // The topic of the message.
            "TopicTestMQ",
            // Message Tag,
            // The message tag, which is similar to a Gmail tag. The message tag is used to sort messages and helps the consumer filter messages on the Message Queue for Apache RocketMQ broker based on specified conditions.
            "TagA",
            // Message Body
            // The message body in the binary format. Message Queue for Apache RocketMQ does not process the message body. The producer and consumer must agree on the serialization and deserialization methods.
            "Hello MQ".getBytes());
        // The key of the message. The key is the business-specific attribute of the message.
```

```

// The key of the message. The key is the business-specific attribute of the me
ssage and must be globally unique whenever possible.
// A unique key helps you query and resend a message in the Message Queue for A
pache RocketMQ console if the message fails to be received.
// Note: Messages can be sent and received even if you do not specify the messa
ge key.
msg.setKey("ORDERID_" + i);
// In one-way transmission mode, the producer does not wait for the response fr
om the Message Queue for Apache RocketMQ broker. Therefore, data loss occurs if messages th
at fail to be delivered are not redelivered. If data loss is not acceptable, we recommend t
hat you use the reliable synchronous or asynchronous transmission mode.
producer.sendOneway(msg);
}
// Before you exit the application, shut down the producer object.
// Note: You can choose not to shut down the producer object.
producer.shutdown();
}

```

6.2.4.6. Send messages by using multiple threads

This topic describes how to send messages by using multiple threads and provides sample code.

The consumer and producer objects of Message Queue for Apache RocketMQ are thread-secure and can be shared among threads.

You can deploy multiple producer and consumer instances on one or more cloud servers. A producer or consumer instance can also run multiple threads to send or receive messages. This improves the transactions per second (TPS) for sending or receiving messages. Do not create a producer instance or consumer instance for every thread.

The following sample code provides an example on how to share a producer among threads:

```

import com.aliyun.openservices.ons.api.Message;
import com.aliyun.openservices.ons.api.Producer;
import com.aliyun.openservices.ons.api.ONSFactory;
import com.aliyun.openservices.ons.api.PropertyKeyConst;
import com.aliyun.openservices.ons.api.SendResult;
import java.util.Properties;
public class SharedProducer {
    public static void main(String[] args) {
        // Initialize the configuration of the producer instance.
        Properties properties = new Properties();
        // The group ID that you created in the Message Queue for Apache RocketMQ console.
        properties.put(PropertyKeyConst.GROUP_ID, "XXX");
        // The AccessKey ID that you created in the Apsara Uni-manager Management Console f
or identity authentication.
        properties.put(PropertyKeyConst.AccessKey, "XXX");
        // The AccessKey secret that you created in the Apsara Uni-manager Management Conso
le for identity authentication.
        properties.put(PropertyKeyConst.SecretKey, "XXX");
        // The timeout interval for sending a message, in milliseconds.
        properties.setProperty(PropertyKeyConst.SendMsgTimeoutMillis, "3000");
        // The TCP endpoint. To obtain the endpoint, log on to the Message Queue for Apache
RocketMQ console. In the left-side navigation pane, click Instance Details. On the Instance
Details page, select your instance. On the Instance Information tab, view the endpoint in t

```

Details page, select your instance. On the instance information tab, view the endpoint in the Obtain Endpoint Information section.

```
properties.put(PropertyKeyConst.NAMESRV_ADDR,
    "XXX");
final Producer producer = ONSFactory.createProducer(properties);
// Before you use the producer to send a message, call the start() method once to s
tart the producer.
producer.start();
// The created producer and consumer objects are thread-secure and can be shared am
ong threads. Do not create a producer instance or consumer instance for every thread.
// Two threads share the producer object and concurrently send messages to Message
Queue for Apache RocketMQ.
Thread thread = new Thread(new Runnable() {
    @Override
    public void run() {
        try {
            Message msg = new Message( //
                // The topic of the message.
                "TopicTestMQ",
                // The message tag, which is similar to a Gmail tag. The message tag is
used to sort messages and helps the consumer filter messages on the Message Queue for Apach
e RocketMQ broker based on specified conditions.
                "TagA",
                // The message body in the binary format. Message Queue for Apache Rock
etMQ does not process the message body.
                // The producer and consumer must agree on the serialization and deseri
alization methods.
                "Hello MQ".getBytes());
            SendResult sendResult = producer.send(msg);
            // Send the message in synchronous mode. If no error occurs, the messag
e is sent.

            if (sendResult != null) {
                System.out.println(new Date() + " Send mq message success. Topic is
:" + MqConfig.TOPIC + " msgId is: " + sendResult.getMessageId());
            }
        } catch (Exception e) {
            // Specify the logic to resend or persist the message if the message fa
ils to be sent.
            System.out.println(new Date() + " Send mq message failed. Topic is:" +
MqConfig.TOPIC);
            e.printStackTrace();
        }
    }
});
thread.start();
Thread anotherThread = new Thread(new Runnable() {
    @Override
    public void run() {
        try {
            Message msg = new Message("TopicTestMQ", "TagA", "Hello MQ".getBytes())
;

            SendResult sendResult = producer.send(msg);
            // Send the message in synchronous mode. If no error occurs, the messag
e is sent.

            if (sendResult != null) {
```

```

        System.out.println(new Date() + " Send mq message success. Topic is
:" + MqConfig.TOPIC + " msgId is: " + sendResult.getMessageId());
    }
    } catch (Exception e) {
        // Specify the logic to resend or persist the message if the message fa
ils to be sent.
        System.out.println(new Date() + " Send mq message failed. Topic is:" +
MqConfig.TOPIC);
        e.printStackTrace();
    }
    }
    });
    anotherThread.start();
    // If the producer instance is no longer used, shut it down to release resources.
    // producer.shutdown();
}
}

```

6.2.4.7. Send and subscribe to ordered messages

This topic describes how to send and subscribe to ordered messages and provides sample code.

Ordered messages, also known as first-in-first-out (FIFO) messages, are a type of message provided by Message Queue for Apache RocketMQ. Such messages are published and consumed in a strict order. This topic provides the sample code for using TCP client SDK for Java to send and subscribe to ordered messages. For more information, see [Ordered messages](#).

Use SDK for Java 1.2.7 or later to send and subscribe to ordered messages.

The methods of sending and subscribing to globally ordered messages and partitionally ordered messages are the same. The following code provides examples on how to send and subscribe to ordered messages:

Sample code for sending ordered messages

```

package com.aliyun.openservices.ons.example.order;
import com.aliyun.openservices.ons.api.Message;
import com.aliyun.openservices.ons.api.ONSTFactory;
import com.aliyun.openservices.ons.api.PropertyKeyConst;
import com.aliyun.openservices.ons.api.SendResult;
import com.aliyun.openservices.ons.api.order.OrderProducer;
import java.util.Properties;
public class ProducerClient {
    public static void main(String[] args) {
        Properties properties = new Properties();
        // The group ID that you created in the Message Queue for Apache RocketMQ console.
        properties.put(PropertyKeyConst.GROUP_ID, "XXX");
        // The AccessKey ID that you created in the Apsara Uni-manager Management Console f
or identity authentication.
        properties.put(PropertyKeyConst.AccessKey, "XXX");
        // The AccessKey secret that you created in the Apsara Uni-manager Management Conso
le for identity authentication.
        properties.put(PropertyKeyConst.SecretKey, "XXX");
        // The TCP endpoint. To obtain the endpoint, log on to the Message Queue for Apache

```

RocketMQ console. In the left-side navigation pane, click Instance Details. On the Instance Details page, select your instance. On the Instance Information tab, view the endpoint in the Obtain Endpoint Information section.

```
properties.put(PropertyKeyConst.NAMESRV_ADDR,
    "XXX");
OrderProducer producer = ONSFactory.createOrderProducer(properties);
// Before you use the producer to send a message, call the start() method once to start the producer.
producer.start();
for (int i = 0; i < 10; i++) {
    String orderId = "biz_" + i % 10;
    Message msg = new Message(//
        // The topic of the message.
        "Order_global_topic",
        // The message tag, which is similar to a Gmail tag. The message tag is used to sort messages and helps the consumer filter messages on the Message Queue for Apache RocketMQ broker based on specified conditions.
        "TagA",
        // The message body in the binary format. Message Queue for Apache RocketMQ does not process the message body. The producer and consumer must agree on the serialization and deserialization methods.
        "send order global msg".getBytes()
    );
    // The key of the message. The key is the business-specific attribute of the message and must be globally unique whenever possible.
    // A unique key helps you query and resend a message in the Message Queue for Apache RocketMQ console if the message fails to be received.
    // Note: Messages can be sent and received even if you do not specify the message key.
    msg.setKey(orderId);
    // The key field that is used in ordered messages to distinguish among different partitions. A partition key is different from the key of a normal message.
    // This field can be set to a non-empty string for globally ordered messages.
    String shardingKey = String.valueOf(orderId);
    try {
        SendResult sendResult = producer.send(msg, shardingKey);
        // Send the message. If no error occurs, the message is sent.
        if (sendResult != null) {
            System.out.println(new Date() + " Send mq message success. Topic is:" + msg.getTopic() + " msgId is: " + sendResult.getMessageId());
        }
    }
    catch (Exception e) {
        // Specify the logic to resend or persist the message if the message fails to be sent.
        System.out.println(new Date() + " Send mq message failed. Topic is:" + msg.getTopic());
        e.printStackTrace();
    }
}
// Before you exit the application, shut down the producer object.
// Note: You can choose not to shut down the producer object.
producer.shutdown();
}
```

```
}
```

Sample code for subscribing to ordered messages

```
package com.aliyun.openservices.ons.example.order;
import com.aliyun.openservices.ons.api.Message;
import com.aliyun.openservices.ons.api.ONSFactory;
import com.aliyun.openservices.ons.api.PropertyKeyConst;
import com.aliyun.openservices.ons.api.order.ConsumeOrderContext;
import com.aliyun.openservices.ons.api.order.MessageOrderListener;
import com.aliyun.openservices.ons.api.order.OrderAction;
import com.aliyun.openservices.ons.api.order.OrderConsumer;
import java.util.Properties;
public class ConsumerClient {
    public static void main(String[] args) {
        Properties properties = new Properties();
        // The group ID that you created in the Message Queue for Apache RocketMQ console.
        properties.put(PropertyKeyConst.GROUP_ID, "XXX");
        // The AccessKey ID that you created in the Apsara Uni-manager Management Console for identity authentication.
        properties.put(PropertyKeyConst.AccessKey, "XXX");
        // The AccessKey secret that you created in the Apsara Uni-manager Management Console for identity authentication.
        properties.put(PropertyKeyConst.SecretKey, "XXX");
        // The TCP endpoint. To obtain the endpoint, log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click Instance Details. On the Instance Details page, select your instance. On the Instance Information tab, view the endpoint in the Obtain Endpoint Information section.
        properties.put(PropertyKeyConst.NAMESRV_ADDR,
            "XXX");
        // The time to wait to redeliver the ordered message when the message fails to be consumed. Valid values: 10 to 1800. Unit: milliseconds.
        properties.put(PropertyKeyConst.SuspendTimeMillis, "100");
        // The maximum number of delivery retries when the message fails to be consumed.
        properties.put(PropertyKeyConst.MaxReconsumeTimes, "20");
        // Before you use the consumer to subscribe to a message, call the start() method once to start the consumer.
        OrderConsumer consumer = ONSFactory.createOrderedConsumer(properties);
        consumer.subscribe(
            // The topic of the message.
            "Jodie_Order_Topic",
            // Subscribe to messages with specified tags in the specified topic.
            // 1. * indicates that the consumer subscribes to all messages in the specified topic.
            // 2. TagA || TagB || TagC indicates that the consumer subscribes to messages with TagA, TagB, or TagC.
            "**",
            new MessageOrderListener() {
                /**
                 * 1. OrderAction.Suspend is returned if a message fails to be consumed or an exception occurs during message processing.<br>
                 * 2. OrderAction.Success is returned if a message is processed.
                 */
            }
        );
    }
}
```

```

@Override
public OrderAction consume(Message message, ConsumeOrderContext context
) {
    System.out.println(message);
    return OrderAction.Success;
}
});
consumer.start();
}
}

```

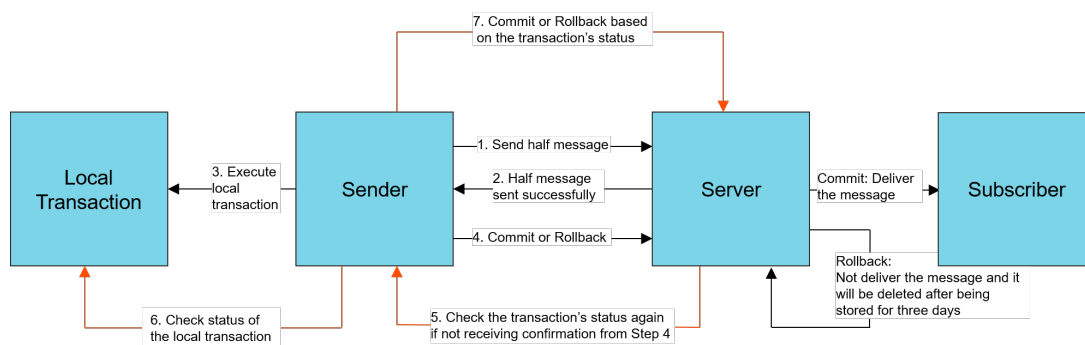
6.2.4.8. Send and subscribe to transactional messages

This topic describes the interaction process and the back-check mechanism of transactional messages. This topic also shows you how to send and subscribe to transactional messages, and provides sample code.

Interaction process

Interaction process of transactional messages shows the interaction process of transactional messages in Message Queue for Apache RocketMQ.

Interaction process of transactional messages



Send transactional messages

Perform the following steps to send a transactional message:

1. Send a half message and execute a local transaction. The following code provides an example:

```

package com.alibaba.webx.TryHsf.appl;
import com.aliyun.openservices.ons.api.Message;
import com.aliyun.openservices.ons.api.PropertyKeyConst;
import com.aliyun.openservices.ons.api.SendResult;
import com.aliyun.openservices.ons.api.transaction.LocalTransactionExecuter;
import com.aliyun.openservices.ons.api.transaction.TransactionProducer;
import com.aliyun.openservices.ons.api.transaction.TransactionStatus;
import java.util.Properties;
import java.util.concurrent.TimeUnit;
public class TransactionProducerClient {
    private final static Logger log = ClientLogger.getLog(); // Configure logging to facilitate troubleshooting.
    public static void main(String[] args) throws InterruptedException {
        final BusinessService businessService = new BusinessService(); // Your on-premises

```

```

business.
    Properties properties = new Properties();
    // The group ID that you created in the Message Queue for Apache RocketMQ console.
    Note: Transactional messages cannot share group IDs with other types of messages.
    properties.put(PropertyKeyConst.GROUP_ID, "XXX");
    // The AccessKey ID that you created in the Apsara Uni-manager Management Console
    for identity authentication.
    properties.put(PropertyKeyConst.AccessKey, "XXX");
    // The AccessKey secret that you created in the Apsara Uni-manager Management Cons
    ole for identity authentication.
    properties.put(PropertyKeyConst.SecretKey, "XXX");
    // The TCP endpoint. To obtain the endpoint, log on to the Message Queue for Apach
    e RocketMQ console. In the left-side navigation pane, click Instance Details. On the In
    stance Details page, select your instance. On the Instance Information tab, view the en
    dpoint in the Obtain Endpoint Information section.
    properties.put(PropertyKeyConst.NAMESRV_ADDR,
        "XXX");
    TransactionProducer producer = ONSFactory.createTransactionProducer(properties,
        new LocalTransactionCheckerImpl());
    producer.start();
    Message msg = new Message("Topic", "TagA", "Hello MQ transaction===".getBytes());
    try {
        SendResult sendResult = producer.send(msg, new LocalTransactionExecutor()
    {
        @Override
        public TransactionStatus execute(Message msg, Object arg) {
            // The ID of the message. Two messages may have the same message b
            ody but cannot have the same ID. The current message ID cannot be queried in the consol
            e.

            String msgId = msg.getMsgId();
            // Calculate the message body by using CRC32 or other algorithms,
            such as MD5.

            long crc32Id = HashUtil.crc32Code(msg.getBody());
            // The message ID and CRC32 ID are used to prevent duplicate messa
            ges.

            // You do not need to specify the message ID or CRC32 ID if your b
            usiness itself achieves idempotence. Otherwise, specify the message ID or CRC32 ID to e
            nsure idempotence.

            // To prevent duplicate messages, calculate the message body by us
            ing the CRC32 or MD5 algorithm.
            Object businessServiceArgs = new Object();
            TransactionStatus transactionStatus = TransactionStatus.Unknow;
            try {
                boolean isCommit =
                    businessService.execbusinessService(businessServiceArgs);
                if (isCommit) {
                    // Commit the message if the local transaction succeeds.
                    transactionStatus = TransactionStatus.CommitTransaction;
                } else {
                    // Roll back the message if the local transaction fails.
                    transactionStatus = TransactionStatus.RollbackTransaction;
                }
            } catch (Exception e) {
                log.error("Message Id:{", msgId, e);
            }
        }
    });
    }
}

```

```

        }
        System.out.println(msg.getMsgID());
        log.warn("Message Id:{}transactionStatus:{}", msgId, transactionStatus.name());
        return transactionStatus;
    }
    }, null);
}
catch (Exception e) {
    // Specify the logic to resend or persist the message if the message fails
    to be sent.
    System.out.println(new Date() + " Send mq message failed. Topic is:" + msg
        .getTopic());
    e.printStackTrace();
}
// Use the demo example to prevent the process from exiting. This is not required
in actual use.
TimeUnit.MILLISECONDS.sleep(Integer.MAX_VALUE);
}
}

```

2. Commit the status of the transactional message.

After the local transaction is executed, the Message Queue for Apache RocketMQ broker must be notified of the transaction status of the current message no matter whether the execution is successful or fails. The Message Queue for Apache RocketMQ broker can be notified in one of the following ways:

- Commit the status after the local transaction is executed.
- Wait until the Message Queue for Apache RocketMQ broker sends a request to check the transaction status of the message.

A transaction can be in one of the following states:

- `TransactionStatus.CommitTransaction`: The transaction is committed. The consumer can consume the message.
- `TransactionStatus.RollbackTransaction`: The transaction is rolled back. The message is discarded and cannot be consumed.
- `TransactionStatus.Unknown`: The status of the transaction is unknown. The Message Queue for Apache RocketMQ broker is expected to send a request again to the producer to query the status of the local transaction that corresponds to the message.

```

public class LocalTransactionCheckerImpl implements LocalTransactionChecker {
    private final static Logger log = ClientLogger.getLog();
    final BusinessService businessService = new BusinessService();
    @Override
    public TransactionStatus check(Message msg) {
        // The ID of the message. Two messages may have the same message body but cannot
        // have the same ID. The current message is a half message. Therefore, its message ID cannot
        // be queried in the console.
        String msgId = msg.getMsgID();
        // Calculate the message body by using CRC32 or other algorithms, such as MD5.
        long crc32Id = HashUtil.crc32Code(msg.getBody());
        // The message ID and CRC32 ID are used to prevent duplicate messages.
        // You do not need to specify the message ID or CRC32 ID if your business itself
        // achieves idempotence. Otherwise, specify the message ID or CRC32 ID to ensure idempotence.
        // To prevent duplicate messages, calculate the message body by using the CRC32
        // or MD5 algorithm.
        // The parameter object of your business. Specify the object based on your business.
        Object businessServiceArgs = new Object();
        TransactionStatus transactionStatus = TransactionStatus.Unknown;
        try {
            boolean isCommit = businessService.checkbusinessService(businessServiceArgs);
            ;
            if (isCommit) {
                // Commit the message if the local transaction succeeds.
                transactionStatus = TransactionStatus.CommitTransaction;
            } else {
                // Roll back the message if the local transaction fails.
                transactionStatus = TransactionStatus.RollbackTransaction;
            }
        } catch (Exception e) {
            log.error("Message Id:{", msgId, e);
        }
        log.warn("Message Id:{}transactionStatus:{", msgId, transactionStatus.name());
        return transactionStatus;
    }
}

```

Utility class

```

import java.util.zip.CRC32;
public class HashUtil {
    public static long crc32Code(byte[] bytes) {
        CRC32 crc32 = new CRC32();
        crc32.update(bytes);
        return crc32.getValue();
    }
}

```

Back-check mechanism for transaction status

- Why must the back-check mechanism for transaction status be implemented when transactional

messages are sent?

If the half message is sent in Step 1 but `TransactionStatus.Unknow` is returned for the local transaction, or no status is committed for the local transaction because the application exits, the status of the half message is unknown to the Message Queue for Apache RocketMQ broker. Therefore, the Message Queue for Apache RocketMQ broker periodically requests the producer to check and report the status of the half message.

- What does the business logic do when the check method is called back?

The check method for transactional messages in Message Queue for Apache RocketMQ must contain the logic of transaction consistency check. After a transactional message is sent, Message Queue for Apache RocketMQ must call the `LocalTransactionChecker` method to respond to the request of the Message Queue for Apache RocketMQ broker for the status of the local transaction. Therefore, the check method for transactional messages must contain the following check items:

- Check the status of the local transaction that corresponds to the half message. The status is committed or rollback.
- Commit the status of the local transaction that corresponds to the half message to the Message Queue for Apache RocketMQ broker.

Subscribe to transactional messages

The method for subscribing to transactional messages is the same as that for subscribing to normal messages. For more information, see [Subscribe to messages](#).

6.2.4.9. Send and subscribe to delayed messages

This topic describes how to send and subscribe to delayed messages and provides sample code.

Delayed messages are delivered to a consumer after a specified period of time from when they are sent to the Message Queue for Apache RocketMQ broker. For example, the specified period of time can be 3 seconds. Delayed messages are used in scenarios where a time window between message production and consumption is required or tasks need to be triggered after a delay. Delayed messages are used in a similar way to delay queues.

For more information about the concepts and usage notes of delayed messages, see [Scheduled messages and delayed messages](#).

Send delayed messages

The following sample code provides an example on how to send delayed messages:

```
import com.aliyun.openservices.ons.api.Message;
import com.aliyun.openservices.ons.api.ONSFactory;
import com.aliyun.openservices.ons.api.Producer;
import com.aliyun.openservices.ons.api.PropertyKeyConst;
import com.aliyun.openservices.ons.api.SendResult;
import java.util.Properties;

public class ProducerDelayTest {
    public static void main(String[] args) {
        Properties properties = new Properties();
        // The AccessKey ID that you created in the Apsara Uni-manager Management Console for
        // identity authentication.
        properties.put(PropertyKeyConst.AccessKey, "XXX");
        // The AccessKey secret that you created in the Apsara Uni-manager Management Console
```

```

ie for identity authentication.
    properties.put(PropertyKeyConst.SecretKey, "XXX");
    // The TCP endpoint. To obtain the endpoint, log on to the Message Queue for Apache
    RocketMQ console. In the left-side navigation pane, click Instance Details. On the Instance
    Details page, select your instance. On the Instance Information tab, view the endpoint in t
    he Obtain Endpoint Information section.
    properties.put(PropertyKeyConst.NAMESRV_ADDR,
        "XXX");
    Producer producer = ONSFactory.createProducer(properties);
    // Before you use the producer to send a message, call the start() method once to s
    tart the producer.
    producer.start();
    Message msg = new Message( //
        // The topic that you created in the Message Queue for Apache RocketMQ cons
        ole.
        "Topic",
        // The message tag, which is similar to a Gmail tag. The message tag is use
        d to sort messages and helps the consumer filter messages on the Message Queue for Apache R
        ocketMQ broker based on specified conditions.
        "tag",
        // The message body in the binary format. Message Queue for Apache RocketMQ
        does not process the message body. The producer and consumer must agree on the serializatio
        n and deserialization methods.
        "Hello MQ".getBytes());
    // The key of the message. The key is the business-specific attribute of the messag
    e and must be globally unique whenever possible.
    // A unique key helps you query and resend a message in the Message Queue for Apach
    e RocketMQ console if the message fails to be received.
    // Note: Messages can be sent and received even if you do not specify the message k
    ey.
    msg.setKey("ORDERID_100");
    try {
        // The specified period of time, in milliseconds. After the specified period of
        time elapses, the Message Queue for Apache RocketMQ broker delivers the message to the cons
        umer. For example, you can set this parameter to 3 and the Message Queue for Apache RocketM
        Q broker delivers the message to the consumer after 3 seconds. The value must be later than
        the current time.
        long delayTime = System.currentTimeMillis() + 3000;
        // The time when the Message Queue for Apache RocketMQ broker starts to deliver
        the message.
        msg.setStartDeliverTime(delayTime);
        SendResult sendResult = producer.send(msg);
        // Send the message in synchronous mode. If no error occurs, the message is sen
        t.
        if (sendResult != null) {
            System.out.println(new Date() + " Send mq message success. Topic is:" + msg.getT
            opic() + " msgId is: " + sendResult.getMessageId());
        }
        } catch (Exception e) {
        // Specify the logic to resend or persist the message if the message fails to b
        e sent.
        System.out.println(new Date() + " Send mq message failed. Topic is:" + msg.getT
        opic());
        e.printStackTrace();
    }
}

```

```
    },  
    // Before you exit the application, shut down the producer object.<br>  
    // Note: You can choose not to shut down the producer object.  
    producer.shutdown();  
}  
}
```

Subscribe to delayed messages

The method for subscribing to delayed messages is the same as that for subscribing to normal messages. For more information, see [Subscribe to messages](#).

6.2.4.10. Send and subscribe to scheduled messages

This topic describes the scenarios for sending and subscribing to scheduled messages and provides sample code.

Scheduled messages are consumed after a specified timestamp. Such messages are used in scenarios where a time window between message production and consumption is required or tasks need to be triggered at a scheduled time.

For more information about the concepts and usage notes of scheduled messages, see [Scheduled messages and delayed messages](#).

Send scheduled messages

The following sample code provides an example on how to send scheduled messages:

```
import com.aliyun.openservices.ons.api.Message;  
import com.aliyun.openservices.ons.api.ONSTFactory;  
import com.aliyun.openservices.ons.api.Producer;  
import com.aliyun.openservices.ons.api.PropertyKeyConst;  
import com.aliyun.openservices.ons.api.SendResult;  
import java.text.ParseException;  
import java.text.SimpleDateFormat;  
import java.util.Properties;  
public class ProducerDelayTest {  
    public static void main(String[] args) {  
        Properties properties = new Properties();  
        // The AccessKey ID that you created in the Apsara Uni-manager Management Console for  
        // or identity authentication.  
        properties.put(PropertyKeyConst.AccessKey, "XXX");  
        // The AccessKey secret that you created in the Apsara Uni-manager Management Console  
        // for identity authentication.  
        properties.put(PropertyKeyConst.SecretKey, "XXX");  
        // The TCP endpoint. To obtain the endpoint, log on to the Message Queue for Apache  
        // RocketMQ console. In the left-side navigation pane, click Instance Details. On the Instance  
        // Details page, select your instance. On the Instance Information tab, view the endpoint in the  
        // Obtain Endpoint Information section.  
        properties.put(PropertyKeyConst.NAMESRV_ADDR,  
            "XXX");  
        Producer producer = ONSTFactory.createProducer(properties);  
        // Before you use the producer to send a message, call the start() method once to start  
        // the producer.  
        producer.start();  
    }  
}
```

```

        Message msg = new Message( //
            // The topic of the message.
            "Topic",
            // The message tag, which is similar to a Gmail tag. The message tag is used to sort messages and helps the consumer filter messages on the Message Queue for Apache RocketMQ broker based on specified conditions.
            "tag",
            // The message body in the binary format. Message Queue for Apache RocketMQ does not process the message body. The producer and consumer must agree on the serialization and deserialization methods.
            "Hello MQ".getBytes());

        // The key of the message. The key is the business-specific attribute of the message and must be globally unique whenever possible.
        // A unique key helps you query and resend a message in the Message Queue for Apache RocketMQ console if the message fails to be received.
        // Note: Messages can be sent and received even if you do not specify the message key.
        msg.setKey("ORDERID_100");
        try {
            // The time when the Message Queue for Apache RocketMQ broker delivers the message to the consumer, in milliseconds. For example, you can set this parameter to 2016-03-07 16:21:00 and the broker delivers the message at 16:21:00 on March 7, 2016. The value must be later than the current time. If the scheduled time is earlier than the current time, the message is immediately delivered to the consumer.
            long timeStamp = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").parse("2016-03-07 16:21:00").getTime();
            msg.setStartDeliverTime(timeStamp);
            // Send the message. If no error occurs, the message is sent.
            SendResult sendResult = producer.send(msg);
            System.out.println("Message Id:" + sendResult.getMessageId());
        }
        catch (Exception e) {
            // Specify the logic to resend or persist the message if the message fails to be sent.
            System.out.println(new Date() + " Send mq message failed. Topic is:" + msg.getTopic());
            e.printStackTrace();
        }
        // Before you exit the application, shut down the producer object.
        // Note: You can choose not to shut down the producer object.
        producer.shutdown();
    }
}


```

Subscribe to scheduled messages

The method for subscribing to scheduled messages is the same as that for subscribing to normal messages. For more information, see [Subscribe to messages](#).

6.2.4.11. Subscribe to messages

This topic describes message subscription modes and provides sample code.

 **Note** The subscriptions of all consumer instances identified by the same group ID must be consistent. For more information, see [Subscription consistency](#).

Subscription modes

Message Queue for Apache RocketMQ supports the following message subscription modes:

- **Clustering subscription:** In this mode, all the consumer instances identified by the same group ID evenly share messages. Assume that a topic contains nine messages and a group ID identifies three consumer instances. In clustering consumption mode, each instance consumes three messages.

```
// Configure clustering subscription, which is the default mode.
properties.put(PropertyKeyConst.MessageModel, PropertyValueConst.CLUSTERING);
```

- **Broadcasting subscription:** In this mode, each consumer instance identified by a group ID consumes each message once. Assume that a topic contains nine messages and a group ID identifies three consumer instances. In broadcasting consumption mode, each instance consumes nine messages.


```
// Configure broadcasting subscription.
properties.put(PropertyKeyConst.MessageModel, PropertyValueConst.BROADCASTING);
```

Sample code

```

import com.aliyun.openservices.ons.api.Action;
import com.aliyun.openservices.ons.api.ConsumeContext;
import com.aliyun.openservices.ons.api.Consumer;
import com.aliyun.openservices.ons.api.Message;
import com.aliyun.openservices.ons.api.MessageListener;
import com.aliyun.openservices.ons.api.ONSFactory;
import com.aliyun.openservices.ons.api.PropertyKeyConst;
import java.util.Properties;
public class ConsumerTest {
    public static void main(String[] args) {
        Properties properties = new Properties();
        // The group ID that you created in the Message Queue for Apache RocketMQ console.
        properties.put(PropertyKeyConst.GROUP_ID, "XXX");
        // The AccessKey ID that you created in the Apsara Uni-manager Management Console for identity authentication.
        properties.put(PropertyKeyConst.AccessKey, "XXX");
        // The AccessKey secret that you created in the Apsara Uni-manager Management Console for identity authentication.
        properties.put(PropertyKeyConst.SecretKey, "XXX");
        // The TCP endpoint. To obtain the endpoint, log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click Instance Details. On the Instance Details page, select your instance. On the Instance Information tab, view the endpoint in the Obtain Endpoint Information section.
        properties.put(PropertyKeyConst.NAMESRV_ADDR,
            "XXX");
        // Clustering subscription, which is the default mode.
        // properties.put(PropertyKeyConst.MessageModel, PropertyValueConst.CLUSTERING);
        // Broadcasting subscription.
        // properties.put(PropertyKeyConst.MessageModel, PropertyValueConst.BROADCASTING);
        ;
        Consumer consumer = ONSFactory.createConsumer(properties);
        consumer.subscribe("TopicTestMQ", "TagA||TagB", new MessageListener() { // Subscribe to multiple tags.
            public Action consume(Message message, ConsumeContext context) {
                System.out.println("Receive: " + message);
                return Action.CommitMessage;
            }
        });
        // Subscribe to another topic.
        consumer.subscribe("TopicTestMQ-Other", "*", new MessageListener() { // Subscribe to all tags.
            public Action consume(Message message, ConsumeContext context) {
                System.out.println("Receive: " + message);
                return Action.CommitMessage;
            }
        });
        consumer.start();
        System.out.println("Consumer Started");
    }
}

```


 **Note** In broadcasting consumption mode, you cannot query message accumulation information in the Message Queue for Apache RocketMQ console. You can create multiple group IDs to achieve the effect of broadcasting consumption. For more information, see [Clustering consumption and broadcasting consumption](#).

6.2.5. SDK for C or C++

6.2.5.1. Prepare the SDK for C or C++ environment

6.2.5.1.1. Overview

Make sure that the following prerequisites are met before you use SDK for C++ to access Message Queue for Apache RocketMQ:

-  **Note**
- You have created the topics and group IDs involved in the code in the Message Queue for Apache RocketMQ console. You can customize message tags in your application. For more information about how to create a message tag, see [Create resources](#).
 - Applications that use Message Queue for Apache RocketMQ are deployed on Elastic Compute Service (ECS) instances.

6.2.5.1.2. Download SDK for C++

This topic describes the preparations, instructions, and usage notes for using SDK for C++ to access Message Queue for Apache RocketMQ so that you can use SDK for C++ to send and subscribe to messages.

Procedure

1. Download [SDK for C++ used in Linux](#).
2. Decompress the downloaded package.

After the package is decompressed, the following directory structure appears:

- *demoss/*
Contains examples on how to send and consume normal messages and ordered messages and how to send messages in one-way transmission mode. This directory also contains the *CMakeList.txt* file that is used to compile and manage *demoss*.
- *include/*
Contains header files that are required by your own programs.
- *lib/*
Contains dynamic libraries based on x86_64. The libraries include the *libonsclient4cpp.so* interface library and the *librocketmq_client_core.so* core library.
- *changelog*
Contains bug fixes and new features in the new releases.

6.2.5.1.3. Use SDK for C++ in Linux

This topic describes how to use SDK for C++ in Linux.

Starting June 28, 2019, the new SDK version provides only dynamic library solutions. The library file of Message Queue for Apache RocketMQ is stored in the *lib/* directory. You must link *librocketmq_client_core.so* with *libonsclient4cpp.so* when you generate executable files. *demos* has introduced the features of C++ 11 and uses CMake for management. Therefore, you must install CMake 3.0 or later and g++ 4.8 or later in advance.

Dynamic solution

GCC 5.x or later has introduced [Dual ABI](#). Therefore, you must add the `-D_GLIBCXX_USE_CXX11_ABI=0` option when you compile the preceding links.

The following sample code provides an example on how to use *demos*:

```
cd aliyun-mq-linux-cpp-sdk // The path to which the downloaded SDK package is decompressed.
cd demos // Go to the demos directory and modify the demos file by entering information such as the topic and key that you created in the Message Queue for Apache RocketMQ console.
cmake . // Check the dependencies and generate a compilation script.
make // Compile the code.
cd bin // Run the program in the directory where the generated executable files are located
.
```

6.2.5.2. Send and subscribe to normal messages

This topic provides the sample code for sending and subscribing to normal messages.

Send normal messages

The following sample code provides an example on how to send normal messages:

Subscribe to normal messages

For more information about how to subscribe to normal messages and about relevant sample code, see [Subscribe to messages](#).

6.2.5.3. Send and subscribe to ordered messages

This topic provides the sample code for sending and subscribing to ordered messages.

Send ordered messages

The following sample code provides an example on how to send ordered messages:

```
#include "ONSTFactory.h"
#include "ONSClientException.h"
#include <iostream>
using namespace ons;
int main()
{
```

```

// Set the parameters that are required to create and use a producer.
ONSFactoryProperty factoryInfo;
    .factoryInfo.setFactoryProperty(ONSFactoryProperty::ProducerId, "XXX");// The ID of the
group that you created in the
Message Queue for Apache RocketMQ console.
    factoryInfo.setFactoryProperty(ONSFactoryProperty::NamesrvAddr, "XXX");// The TCP endp
oint of your instance. To obtain the TCP endpoint, log on to the
Message Queue for Apache RocketMQ console. In the left-side navigation pane, click Instance
s. On the Instances page, click the name of your instance. On the Instance Details page, sc
roll to the Basic Information section and view the TCP endpoint on the Endpoints tab.
    factoryInfo.setFactoryProperty(ONSFactoryProperty::PublishTopics, "XXX" );// The topic t
hat you created in the
Message Queue for Apache RocketMQ console.
    factoryInfo.setFactoryProperty(ONSFactoryProperty::MsgContent, "XXX");// The message co
ntent.
    factoryInfo.setFactoryProperty(ONSFactoryProperty::AccessKey, "XXX");// The AccessKey I
D that you created in the Alibaba Cloud Management Console for identity authentication.
    factoryInfo.setFactoryProperty(ONSFactoryProperty::SecretKey, "XXX" );// The AccessKey
secret that you created in the Alibaba Cloud Management Console for identity authentication
.
    // Create a producer.
    OrderProducer *pProducer = ONSFactory::getInstance()->createOrderProducer(factoryInfo);
    //Before you send a message, call the start() method to start the producer. You can cal
l the start() method only once.
    pProducer->start();
    Message msg(
        //Message Topic
        factoryInfo.getPublishTopics(),
        // The message tag, which is similar to a Gmail tag. The message tag is use
d to sort messages and filter messages for the consumer on the
Message Queue for Apache RocketMQ broker based on specified conditions.
        "TagA",
        // The message body in the binary format.
        Message Queue for Apache RocketMQ does not process the message body. The producer and the c
onsumer must agree on the serialization and deserialization methods.
        factoryInfo.getMessageContent()
    );
    // The key of the message. The key is the business-specific attribute of the message an
d must be globally unique.
    // A unique key helps you query and resend a message in the
Message Queue for Apache RocketMQ console if the message fails to be received.
    // Note: Messages can be sent and received even if you do not specify the message key.
    msg.setKey("ORDERID_100");
    // The key field that is used to identify partitions for partitionally ordered messages
.
    // This field can be set to a non-empty string for globally ordered messages.
    std::string shardingKey = "abc";
    // Messages that have the same Sharding Key are sent in order.
    try
    {
        // Send the message. If no exception is thrown, the message is sent.
        SendResultONS sendResult = pProducer->send(msg, shardingKey);
        std::cout << "send success" << std::endl;
    }
    catch (ONSClientException e)
    {
        // Handle the exception.
    }
}

```

```

        catch(ONSClientException & e)
        {
            // Add the exception handling operation.
        }
        // Before you exit your application, shut down the producer. If you do not shut down the
        // producer, issues such as memory leaks may occur.
        pProducer->shutdown();
        return 0;
    }

```

Subscribe to ordered messages

The following sample code provides an example on how to subscribe to ordered messages:

```

#include "ONSFactory.h"
using namespace std;
using namespace ons;
// Create a consumer instance.
//After pushConsumer pulls the message, pushConsumer calls the consumeMessage function of the
//instance.
class ONSCLIENT_API MyMsgListener : public MessageOrderListener
{
public:
    MyMsgListener()
    {
    }
    virtual ~MyMsgListener()
    {
    }
    virtual OrderAction consume(Message &message, ConsumeOrderContext &context)
    {
        // Consume messages based on business requirements.
        return Success; //CONSUME_SUCCESS;
    }
};

int main(int argc, char* argv[])
{
    // Set the parameters that are required to create and use orderConsumer.
    ONSFactoryProperty factoryInfo;
    factoryInfo.setFactoryProperty(ONSFactoryProperty::ConsumerId, "XXX");// The ID of the
    //group that you created in the
    //Message Queue for Apache RocketMQ console.
    factoryInfo.setFactoryProperty(ONSFactoryProperty::PublishTopics,"XXX" );// The topic that
    //you created in the
    //Message Queue for Apache RocketMQ console.
    factoryInfo.setFactoryProperty(ONSFactoryProperty::AccessKey, "XXX");// The AccessKey ID
    //that you created in the Alibaba Cloud Management Console for identity authentication.
    factoryInfo.setFactoryProperty(ONSFactoryProperty::SecretKey, "XXX");// The AccessKey
    //secret that you created in the Alibaba Cloud Management Console for identity authentication
    .
    factoryInfo.setFactoryProperty(ONSFactoryProperty::NAMESRV_ADDR, "XXX");// The TCP endpoint
    //of your instance. To obtain the TCP endpoint, log on to the
    //Message Queue for Apache RocketMQ console. In the left-side navigation pane, click Instance
    //s. On the Instances page, click the name of your instance. On the Instance Details page, sc

```

```

roll to the Basic Information section and view the TCP endpoint on the Endpoints tab.
// Create orderConsumer.
OrderConsumer* orderConsumer = ONSFactory::getInstance()->createOrderConsumer(factoryInfo);
MyMsgListener msglistener;
// Specify the message topic and tag to which orderConsumer subscribes.
orderConsumer->subscribe(factoryInfo.getPublishTopics(), "*", &msglistener);
// Register the instance to listen to messages. After orderConsumer pulls the messages,
orderConsumer calls the consumeMessage function of the message listening class.
//Start orderConsumer.
orderConsumer->start();
for(volatile int i = 0; i < 1000000000; ++i) {
    //wait
}
// Shut down orderConsumer. Before you exit the application, shut down orderConsumer. If
you do not shut down orderConsumer, issues such as memory leaks may occur.
orderConsumer->shutdown();
return 0;
}

```

6.2.5.4. Send and subscribe to scheduled messages

This topic provides the sample code for sending and subscribing to scheduled messages.

Scheduled messages are consumed by consumers after a specified period of time. Such messages are used in scenarios where a time window between message production and consumption is required or tasks need to be triggered at a scheduled time. Scheduled messages are used in a similar way to delay queues.

Send scheduled messages

The following sample code provides an example on how to send scheduled messages:

```

#include "ONSFactory.h"
#include "ONSClientException.h"
#include <windows.h>
using namespace ons;
int main()
{
    // Create a producer and set the parameters that are required to send messages.
    ONSFactoryProperty factoryInfo;
    factoryInfo.setFactoryProperty(ONSFactoryProperty::ProducerId, "XXX");// The ID of the
group that you created in the
Message Queue for Apache RocketMQ console.
    factoryInfo.setFactoryProperty(ONSFactoryProperty::NAME_SRV_ADDR, "XXX");// The TCP endpoint
of your instance. To obtain the TCP endpoint, log on to the
Message Queue for Apache RocketMQ console. In the left-side navigation pane, click Instance
s. On the Instances page, click the name of your instance. On the Instance Details page, view
the endpoint on the Endpoint Information tab.
    factoryInfo.setFactoryProperty(ONSFactoryProperty::PublishTopics, "XXX");// The topic that
you created in the
Message Queue for Apache RocketMQ console.
    factoryInfo.setFactoryProperty(ONSFactoryProperty::MsgContent, "xxx");// The content of
the message.
}

```

```

    factoryInfo.setFactoryProperty(ONSFactoryProperty::AccessKey, "xxx");// The AccessKey ID that you created in the Alibaba Cloud Management Console for identity authentication.
    factoryInfo.setFactoryProperty(ONSFactoryProperty::SecretKey, "xxx" );// The AccessKey secret that you created in the Alibaba Cloud Management Console for identity authentication
    .
    //create producer;
    Producer *pProducer = ONSFactory::getInstance()->createProducer(factoryInfo);
    // Before you send messages, call the start method once to start the producer.
    pProducer->start();
    Message msg(
        //Message Topic
        factoryInfo.getPublishTopics(),
        // The tag of the message, which is similar to a Gmail tag. Message tags are used to sort messages and filter messages for the consumer on the Message Queue for Apache RocketMQ broker based on specified conditions.
        "TagA",
        // The body of the message. This parameter is required.
        Message Queue for Apache RocketMQ does not process the message body. The producer and consumer must agree on the methods to serialize and deserialize the message body.
        factoryInfo.getMessageContent()
    );
    // The key of the message. The key is the business-specific attribute of the message and must be globally unique whenever possible.
    // The key helps you query and resend a message in the Message Queue for Apache RocketMQ console if the message fails to be received.
    // Note: Messages can be sent and received even if you do not specify message keys.
    msg.setKey("ORDERID_100");
    // The time when the Message Queue for Apache RocketMQ broker delivers the message to the consumer. Unit: milliseconds. The message can be consumed only after the specified time. In this example, the message can be consumed 3 seconds later.
    long deliverTime = GetTickCount64() + 3000;
    msg.setStartDeliverTime(deliverTime);
    // Send the message. If no exception occurs, the message is sent.
    try
    {
        SendResultONS sendResult = pProducer->send(msg);
    }
    catch(ONSClientException & e)
    {
        // Specify the logic to process the exception.
    }
    // Before you exit the application, shut down the producer. Otherwise, issues such as memory leaks occur.
    pProducer->shutdown();
    return 0;
}

```

Subscribe to scheduled messages

For more information about how to subscribe to scheduled messages and about relevant sample code, see [Subscribe to messages](#).

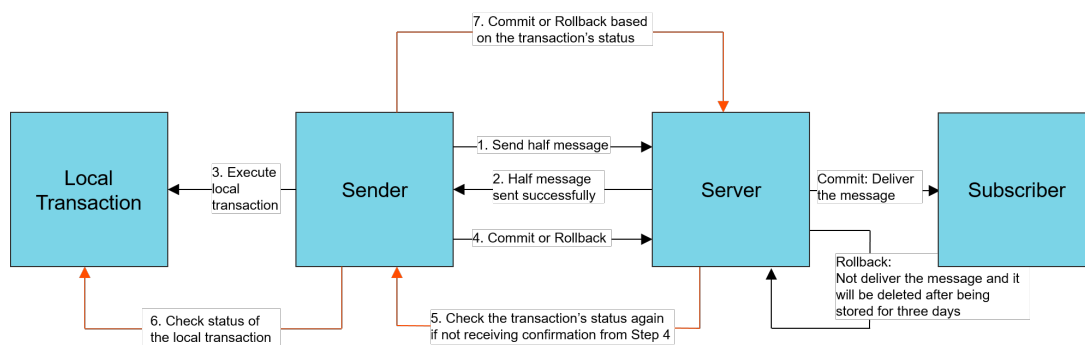
6.2.5.5. Send and subscribe to transactional messages

This topic describes the interaction process and the back-check mechanism of transactional messages. This topic also shows you how to send and subscribe to transactional messages, and provides sample code.

Interaction process

[Transactional message interaction flowchart](#) shows the interaction process of transactional messages in Message Queue for Apache RocketMQ.

Interaction process of transactional messages



Send transactional messages

Perform the following steps to send a transactional message:

1. Send a half message and execute a local transaction. The following code provides examples on how to send and subscribe to transactional messages:

2. Commit the status of the transactional message.

After the local transaction is executed, the Message Queue for Apache RocketMQ broker must be notified of the transaction status of the current message no matter whether the execution is successful or fails. The Message Queue for Apache RocketMQ broker can be notified in one of the following ways:

- Commit the status after the local transaction is executed.
- Wait until the Message Queue for Apache RocketMQ broker sends a request to check the transaction status of the message.

A transaction can be in one of the following states:

- `TransactionStatus.CommitTransaction`: The transaction is committed. The consumer can consume the message.
- `TransactionStatus.RollbackTransaction`: The transaction is rolled back. The message is discarded and cannot be consumed.
- `TransactionStatus.Unknown`: The status of the transaction is unknown. The Message Queue for Apache RocketMQ broker is expected to send a request again to the producer to query the status of the local transaction that corresponds to the message.

```

class MyLocalTransactionChecker : LocalTransactionChecker
{
    MyLocalTransactionChecker()
    {
    }
    ~MyLocalTransactionChecker()
    {
    }
    virtual TransactionStatus check(Message &value)
    {
        // The ID of the message. The current message ID cannot be queried in the console.
        // Two messages may have the same message body but cannot have the same ID.
        string msgId = value.getMsgID();
        // Calculate the message body by using CRC32 or other algorithms, such as MD5.
        // The message ID and CRC32 ID are used to prevent duplicate messages.
        // You do not need to specify the message ID or CRC32 ID if your business itself achieves idempotence. Otherwise, specify the message ID or CRC32 ID to ensure idempotence.
        // To prevent duplicate messages, calculate the message body by using the CRC32 or MD5 algorithm.
        TransactionStatus transactionStatus = Unknown;
        try {
            boolean isCommit = Execution result of the local transaction;
            if (isCommit) {
                // Commit the message if the local transaction succeeds.
                transactionStatus = CommitTransaction;
            } else {
                // Roll back the message if the local transaction fails.
                transactionStatus = RollbackTransaction;
            }
        } catch (...) {
            //exception error
        }
        return transactionStatus;
    }
}

```

Back-check mechanism for transaction status

- Why must the back-check mechanism for transaction status be implemented when transactional messages are sent?

If the half message is sent in Step 1 but `TransactionStatus.Unknown` is returned for the local transaction, or no status is committed for the local transaction because the application exits, the status of the half message is unknown to the Message Queue for Apache RocketMQ broker. Therefore, the Message Queue for Apache RocketMQ broker periodically requests the producer to check and report the status of the half message.

- What does the business logic do when the check method is called back?

The check method for transactional messages in Message Queue for Apache RocketMQ must contain the logic of transaction consistency check. After a transactional message is sent, Message Queue for Apache RocketMQ must call the `LocalTransactionChecker` method to respond to the request of the Message Queue for Apache RocketMQ broker for the status of the local transaction. Therefore, the check method for transactional messages must contain the following check items:

- i. Check the status of the local transaction that corresponds to the half message. The status is committed or rollback.
 - ii. Commit the status of the local transaction that corresponds to the half message to the Message Queue for Apache RocketMQ broker.
- How do different states of the local transaction affect the half message?
 - `TransactionStatus.CommitTransaction`: The transaction is committed. The consumer can consume the message.
 - `TransactionStatus.RollbackTransaction`: The transaction is rolled back. The message is discarded and cannot be consumed.
 - `TransactionStatus.Unknown`: The status of the transaction is unknown. The Message Queue for Apache RocketMQ broker is expected to send a request again to the producer to query the status of the local transaction that corresponds to the message.


For more information about the code, see the implementation of `MyLocalTransactionChecker`.

Subscribe to transactional messages

For more information about how to subscribe to transactional messages and about relevant sample code, see [Subscribe to messages](#).

6.2.5.6. Subscribe to messages

This topic describes how to subscribe to messages by using SDK for C or C++ provided by Message Queue for Apache RocketMQ.

 **Note** The subscriptions of all consumer instances identified by the same group ID must be consistent. For more information, see [Subscription consistency](#).

Subscription modes

Message Queue for Apache RocketMQ supports the following message subscription modes:

- **Clustering subscription:**

This mode is used to implement clustering consumption. In clustering consumption mode, all the consumer instances identified by the same group ID evenly share messages. Assume that a topic contains nine messages and a group ID identifies three consumer instances. In clustering consumption mode, each instance consumes three messages.

```
// Configure clustering subscription, which is the default mode.
factoryInfo.setFactoryProperty(ONSFactoryProperty:: MessageModel, ONSFactoryProperty::CLUSTERING);
```

- **Broadcasting subscription:**

This mode is used to implement broadcasting consumption. In broadcasting consumption mode, each consumer instance identified by a group ID consumes a message once. Assume that a topic contains nine messages and a group ID identifies three consumer instances. In broadcasting consumption mode, each instance consumes nine messages.

```
// Configure broadcasting subscription.
factoryInfo.setFactoryProperty(ONFactoryProperty:: MessageModel, ONFactoryProperty::BR
OADCASTING);
```

Sample code

```
#include "ONFactory.h"
#include <iostream>
#include <thread>
#include <mutex>
using namespace ons;
std::mutex console_mtx;
class ExampleMessageListener : public MessageListener {
public:
    Action consume(Message& message, ConsumeContext& context) {
        // The consumer receives the message and attempts to consume it. After the message
        is consumed, CommitMessage is returned.
        // If the consumer fails to consume the message or wants to consume the message aga
        in, ReconsumeLater is returned. Then, the message is delivered to the consumer again after
        a predefined period of time.
        std::lock_guard<std::mutex> lk(console_mtx);
        std::cout << "Received a message. Topic: " << message.getTopic() << ", MsgId: "
        << message.getMsgID() << std::endl;
        return CommitMessage;
    }
};
int main(int argc, char* argv[]) {
    std::cout << "=====Before consuming messages===== " << std::endl;
    ONFactoryProperty factoryInfo;
    // Specify the group ID that you created in the
    Message Queue for Apache RocketMQ console. Message Queue for Apache RocketMQ instances use
    the group ID instead of the producer ID and consumer ID. Specifying this value ensures comp
    atibility with earlier versions.
    factoryInfo.setFactoryProperty(ONFactoryProperty::ConsumerId, "GID_XXX");
    // Specify the AccessKey ID of your Alibaba Cloud account.
    factoryInfo.setFactoryProperty(ONFactoryProperty::AccessKey, "Your Access Key");
    // Specify the AccessKey secret of your Alibaba Cloud account.
    factoryInfo.setFactoryProperty(ONFactoryProperty::SecretKey, "Your Secret Key");
    // Specify the TCP endpoint of your Message Queue for Apache RocketMQ instance. You can
    view the endpoint in the
    Message Queue for Apache RocketMQ console.
    factoryInfo.setFactoryProperty(ONFactoryProperty::NAMESRV_ADDR,
        "http://xxxxxxxxxxxxxxxxx.aliyuncs.com:80");
    PushConsumer *consumer = ONFactory::getInstance()->createPushConsumer(factoryInfo);
    // Specify a topic that you created in the
    Message Queue for Apache RocketMQ console.
    const char* topic_1 = "topic-1";
    // Subscribe to the messages attached with tag-1 in topic-1.
    const char* tag_1 = "tag-1";
    const char* topic_2 = "topic-2";
    // Subscribe to all messages in topic-2.
    const char* tag_2 = "";
    // Use a custom listener function to process the received messages and return the resul
    to
```

```

cs.
    ExampleMessageListener * message_listener = new ExampleMessageListener();
    consumer->subscribe(topic_1, tag_1, message_listener);
    consumer->subscribe(topic_2, tag_2, message_listener);
    // The preparation is complete. You must invoke the startup function to start the consu
mer.
    consumer->start();
    // Keep the thread running and do not shut down the consumer.
    std::this_thread::sleep_for(std::chrono::milliseconds(60 * 1000));
    consumer->shutdown();
    delete message_listener;
    std::cout << "====After consuming messages====" << std::endl;
    return 0;
}

```

6.2.6. SDK for .NET

6.2.6.1. .Prepare the SDK for .NET environment

6.2.6.1.1. Overview

Before you use SDK for .NET to access Message Queue for Apache RocketMQ and send and subscribe to messages, make sure that the following prerequisites are met:

Note


- You have created the topics and group IDs involved in the code in the Message Queue for Apache RocketMQ console. You can customize message tags in your application. For more information about how to create a message tag, see [Create resources](#).
- Applications that use Message Queue for Apache RocketMQ are deployed on Elastic Compute Service (ECS) instances.

6.2.6.1.2. Download SDK for .NET

Message Queue for Apache RocketMQ SDK for .NET is a managed wrapper based on Apache RocketMQ Client CPP. Message Queue for Apache RocketMQ SDK for .NET is independent of Windows .NET public library. Multithreading and parallel processing in C++ are used to ensure the efficiency and stability of Message Queue for Apache RocketMQ SDK for .NET.

Context

If Visual Studio is used to develop .NET applications and class libraries, the default target platform is Any CPU. This means that x86 or x64 is automatically selected based on the CPU type at runtime. This capability is provided because the assembly compiled by using .NET is based on the intermediate language (IL). At runtime, the just-in-time compiler (JIT) in the common language runtime (CLR) of .NET converts the IL code into the x86 or x64 machine code. The DLL generated by the C or C++ compiler is the machine code. Therefore, a target platform is selected during compilation. The C or C++ project is compiled as an x64 64-bit DLL by configuring compilation options. Therefore, the 64-bit DLL in release mode compiled by using Visual Studio 2015 is provided. The 64-bit DLL in release mode is also available to other Visual Studio versions.

 **Note** C++ DLL files require the installation package of the Visual C++ 2015 runtime environment. If the Visual Studio 2015 runtime environment is not installed, run the vc_redist.x64.exe program provided in the SDK.

Procedure

1. Download the SDK package.

We recommend that both new users and existing users that are not concerned with upgrade costs download the latest SDK. [Download the latest version of SDK for .NET that are used in Windows](#)

2. Decompress the downloaded package.

After the package is decompressed, the following directory structure appears:

- *demo/*

Contains examples on how to send normal messages, send messages in one-way mode, send ordered messages, consume normal messages, and consume ordered messages.

- *lib/*

Contains files related to the underlying C++ DLL and the installation package of the Visual C++ 2015 runtime environment. If Visual Studio 2015 is not installed, copy and run the vc_redist.x64.exe program, as shown in the following information:

```
64/
  NSClient4CPP.lib
  ONSCClient4CPP.dll
  ONSCClient4CPP.pdb
  vc_redist.x64.exe
```

- *interface/*

Encapsulates P/Invoke code. The code must be included in the user project code.

- *SDK_GUIDE.pdf*

Contains the documentation and frequently asked questions (FAQ) about how to prepare the SDK environment.

- *changelog*

Contains bug fixes and new features in the new releases.

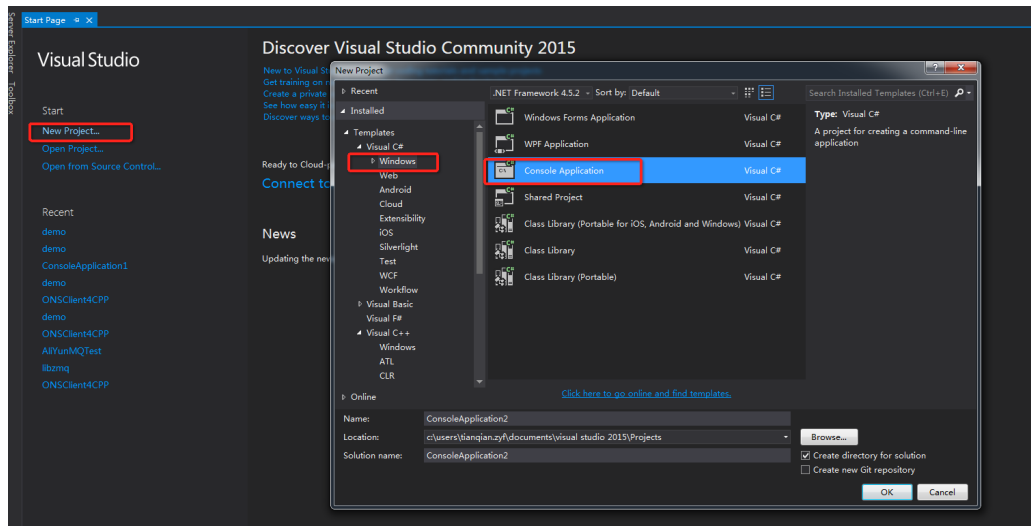
6.2.6.1.3. .Configure SDK for .NET

This topic shows you how to use SDK for .NET in Windows.

Procedure

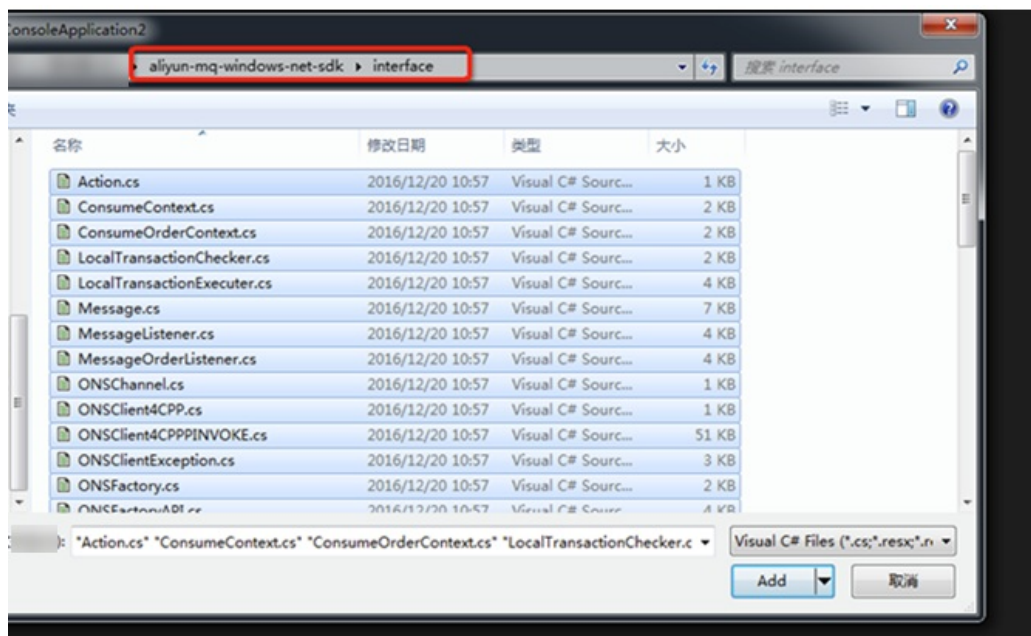
Use SDK for .NET in Visual Studio 2015.NET SDK

1. Use Visual Studio 2015 to create your project.
.NET SDK-1



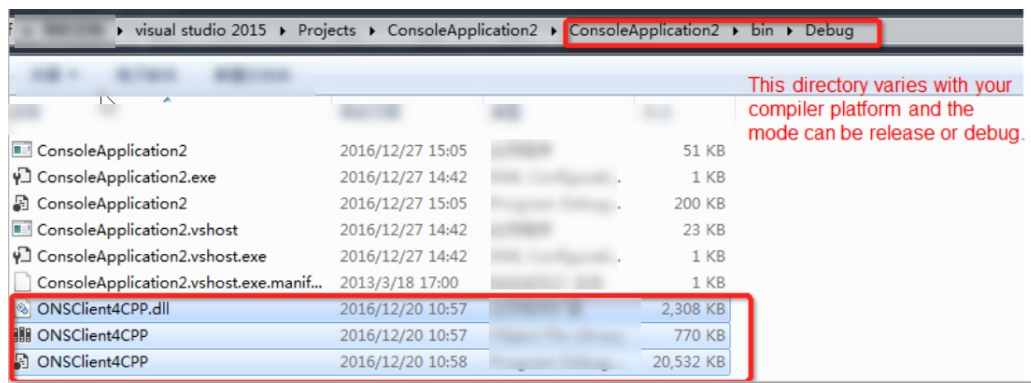
- Right-click the project and choose **Add > Add Existing Item** to add all files in the interface directory of the downloaded SDK package.

.NET SDK-2



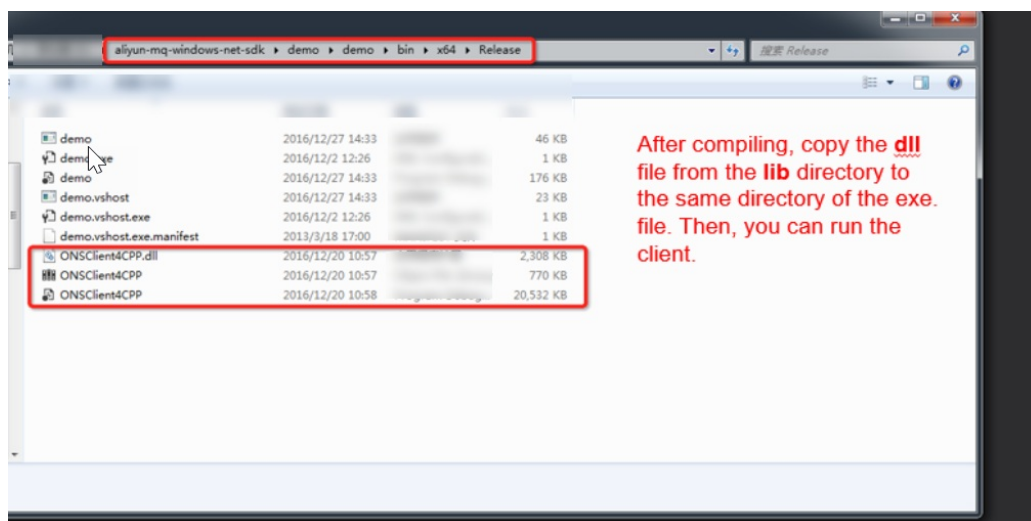
- Right-click the project and choose **Properties > Configuration Manager**. Set Active solution configuration to Release and set Active solution platform to x64.
- Write and compile the test program, save the DLL file of the SDK to the directory of the executable file or to the system directory, and then run the program.

.NET SDK-3



Note The SDK provides a preconfigured demo project. You can directly open the project and compile it. When you run the project, copy the related DLL file to the directory of the executable file, as shown in the following figure.

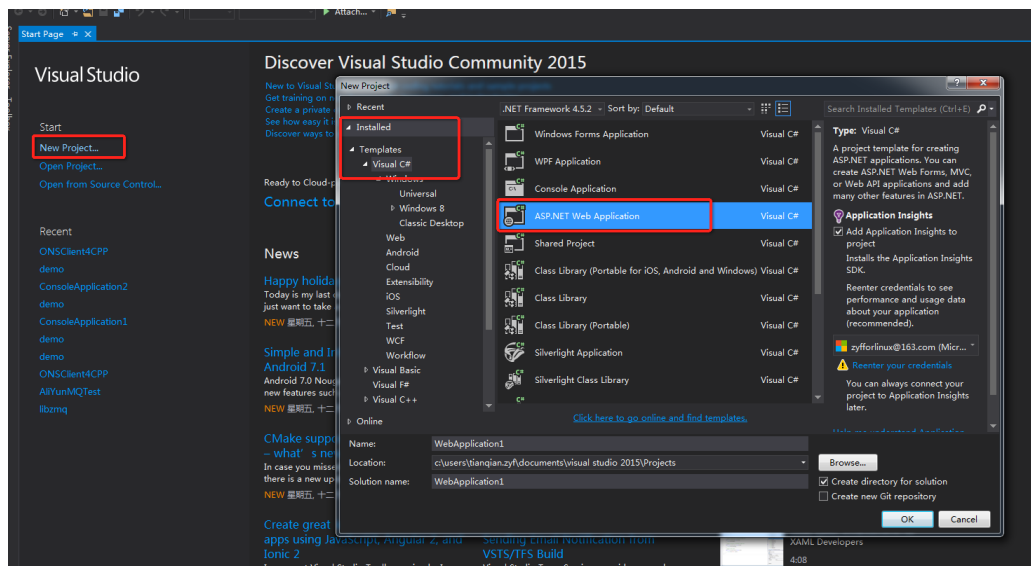
.NET SDK-4



Configure ASP.NET in Visual Studio 2015 to use Message Queue for Apache RocketMQ SDK

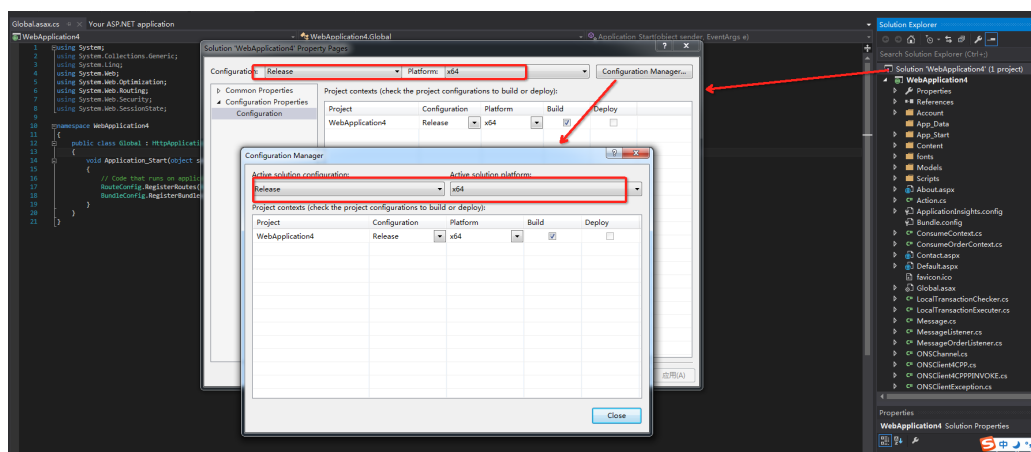
5. Create a Web Forms project for ASP.NET by using Visual Studio 2015.

.NET SDK-5



- Right-click the project and choose **Properties > Configuration Manager**. Set **Active solution configuration** to **Release** and set **Active solution platform** to **x64**.

.NET SDK-6



- Right-click the project and choose **Add > Add Existing Item** to add all files in the interface directory of the downloaded SDK package.

For more information about how to configure a common .NET project, see Step 2.

- Add the code for starting and stopping the SDK to the **Global.asax.cs** file.

Note We recommend that you encapsulate the SDK code as a singleton class so that the code cannot be recycled by the garbage collector due to scope problems. The example directory of the SDK contains the **Example.cs** file for implementing a simple singleton class. To use **Example.cs**, you must include it in your own project.

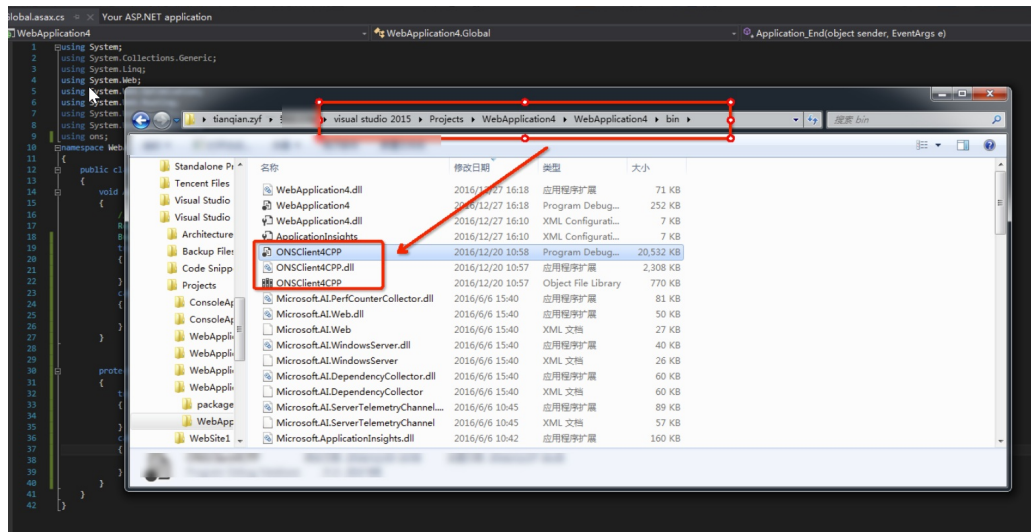
```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Optimization;
using System.Web.Routing;
using System.Web.Security;
using System.Web.SessionState;
using ons;    // The namespace where the SDK is located.
using test;   // The namespace where the class with the roughly encapsulated SDK is located. See the Example.cs file in the example directory of the SDK.
namespace WebApplication4
{
    public class Global : HttpApplication
    {
        void Application_Start(object sender, EventArgs e)
        {
            // Code that runs on application startup
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);
            try
            {
                // The code for starting the SDK. The following code is the code after
the SDK is roughly encapsulated.
                OnscSharp.CreateProducer();
                OnscSharp.StartProducer();
            }
            catch (Exception ex)
            {
                // Specify the logic for handling errors.
            }
        }
        protected void Application_End(object sender, EventArgs e)
        {
            try
            {
                // The code for stopping the SDK.
                OnscSharp.ShutdownProducer();
            }
            catch (Exception ex)
            {
                // Specify the logic for handling errors.
            }
        }
    }
}

```

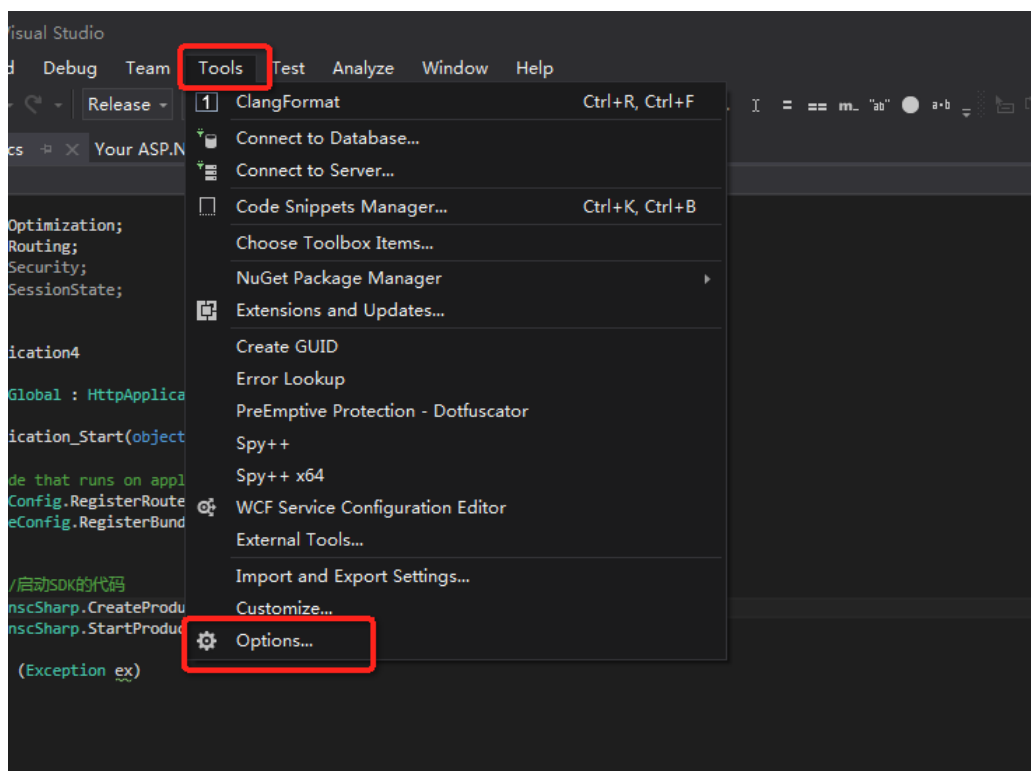
9. Write and compile the test program.
10. Save the DLL file of the SDK to the directory of the executable file or to the system directory and run the program.

.NET SDK-7

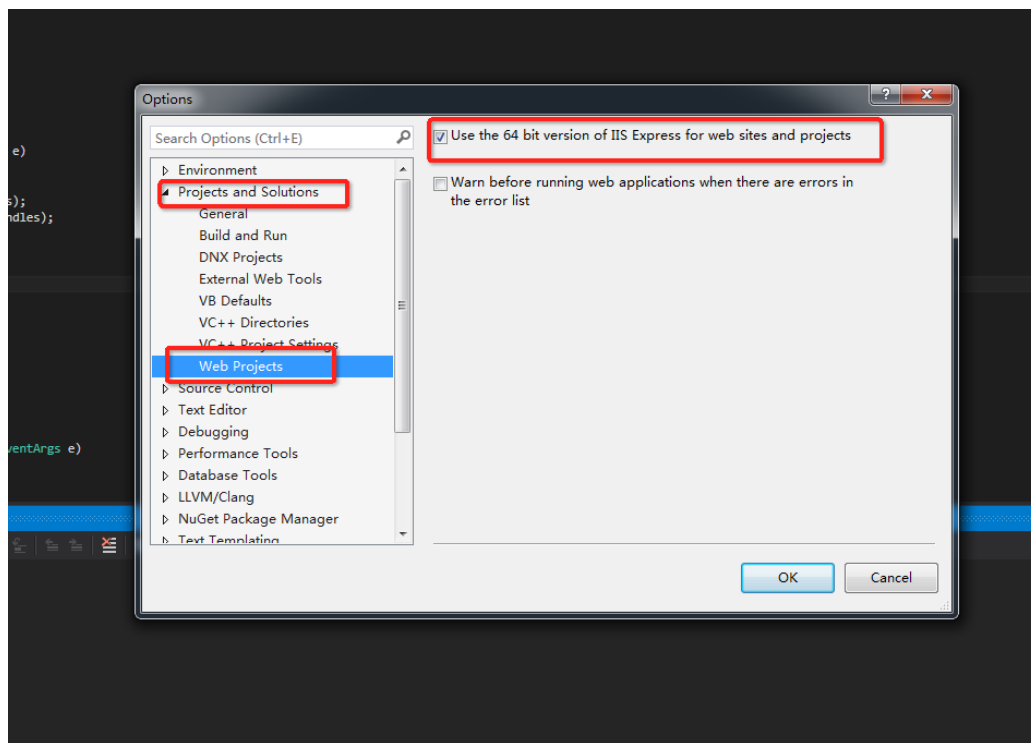


11. Choose **Tools > Options > Projects and Solutions > Web Projects**. Then, select the **Use the 64 bit version of IIS Express for websites and projects** check box.

.NET SDK-8



.NET SDK-9



6.2.6.2. Send and subscribe to normal messages

This topic provides the sample code for sending and subscribing to normal messages.

Send normal messages

The following sample code provides an example on how to send normal messages: Set related parameters based on the instructions.

```
using System;
using ons;
public class ProducerExampleForEx
{
    public ProducerExampleForEx()
    {
    }
    static void Main(string[] args) {
        // Configure your account based on the resources that you created in the console.
        ONSFactoryProperty factoryInfo = new ONSFactoryProperty();
        // The AccessKey ID that you created in the Apsara Uni-manager Management Console f
or identity authentication.
        factoryInfo.setFactoryProperty(ONSFactoryProperty.AccessKey, "Your access key");
        // The AccessKey secret that you created in the Apsara Uni-manager Management Conso
le for identity authentication.
        factoryInfo.setFactoryProperty(ONSFactoryProperty.SecretKey, "Your access secret");
        // The group ID that you created in the Message Queue for Apache RocketMQ console.
        factoryInfo.setFactoryProperty(ONSFactoryProperty.ProducerId, "GID_example");
        // The topic that you created in the Message Queue for Apache RocketMQ console.
        factoryInfo.setFactoryProperty(ONSFactoryProperty.PublishTopics, "T_example_topic_n
ame");
        // The TCP endpoint. To obtain the endpoint, log on to the Message Queue for Apache
```

RocketMQ console. In the left-side navigation pane, click Instance Details. On the Instance Details page, select your instance. On the Instance Information tab, view the endpoint in the Obtain Endpoint Information section.

```
factoryInfo.setFactoryProperty(ONFactoryProperty.NAME_SRV_ADDR, "NameSrv_Addr");
// Specify the log path.
factoryInfo.setFactoryProperty(ONFactoryProperty.LogPath, "C://log");
// Create a producer instance.
// Note: Producer instances are thread-secure and can be used to send messages of different topics. Each of your threads
// needs only one producer instance.
Producer producer = ONFactory.getInstance().createProducer(factoryInfo);
// Start the producer instance.
producer.start();
// Create a message object.
Message msg = new Message(factoryInfo.getPublishTopics(), "tagA", "Example message body");
msg.setKey(Guid.NewGuid().ToString());
for (int i = 0; i < 32; i++) {
    try
    {
        SendResultONS sendResult = producer.send(msg);
        Console.WriteLine("send success {0}", sendResult.getMessageId());
    }
    catch (Exception ex)
    {
        Console.WriteLine("send failure{0}", ex.ToString());
    }
}
// Shut down the producer instance when your thread is about to exit.
producer.shutdown();
}
```

Subscribe to normal messages

For more information about how to subscribe to normal messages and about relevant sample code, see [Subscribe to messages](#).

6.2.6.3. Send and subscribe to ordered messages

This topic describes how to send and subscribe to ordered messages and provides sample code.

Send ordered messages

The following sample code provides an example on how to send ordered messages:

```
using System;
using ons;
public class OrderProducerExampleForEx
{
    public OrderProducerExampleForEx()
    {
    }
    static void Main(string[] args) {
```

```

        // Configure your account based on the resources that you created in the Apsara Uni-
        -manager Management Console.
        ONSFactoryProperty factoryInfo = new ONSFactoryProperty();
        // The AccessKey ID that you created in the Apsara Uni-manager Management Console f
        or identity authentication.
        factoryInfo.setFactoryProperty(ONSFactoryProperty.AccessKey, "Your access key");
        // The AccessKey secret that you created in the Apsara Uni-manager Management Conso
        le for identity authentication.
        factoryInfo.setFactoryProperty(ONSFactoryProperty.SecretKey, "Your access secret");
        // The group ID that you created in the Message Queue for Apache RocketMQ console.
        factoryInfo.setFactoryProperty(ONSFactoryProperty.ProducerId, "GID_example");
        // The topic that you created in the Message Queue for Apache RocketMQ console.
        factoryInfo.setFactoryProperty(ONSFactoryProperty.PublishTopics, "T_example_topic_n
        ame");
        // The TCP endpoint. To obtain the endpoint, log on to the Message Queue for Apache
        RocketMQ console. In the left-side navigation pane, click Instance Details. On the Instance
        Details page, select your instance. On the Instance Information tab, view the endpoint in t
        he Obtain Endpoint Information section.
        factoryInfo.setFactoryProperty(ONSFactoryProperty.NAMESRV_ADDR, "NameSrv_Addr");
        // Specify the log path.
        factoryInfo.setFactoryProperty(ONSFactoryProperty.LogPath, "C://log");
        // Create a producer instance.
        // Note: Producer instances are thread-safe and can be used to send messages of dif
        ferent topics. Each thread
        // requires only one producer instance.
        OrderProducer producer = ONSFactory.getInstance().createOrderProducer(factoryInfo);
        // Start the producer instance.
        producer.start();
        // Create a message.
        Message msg = new Message(factoryInfo.getPublishTopics(), "tagA", "Example message
        body");
        string shardingKey = "App-Test";
        for (int i = 0; i < 32; i++) {
            try
            {
                SendResultONS sendResult = producer.send(msg, shardingKey);
                Console.WriteLine("send success {0}", sendResult.getMessageId());
            }
            catch (Exception ex)
            {
                Console.WriteLine("send failure{0}", ex.ToString());
            }
        }
        // Disable the producer instance when your thread is about to exit.
        producer.shutdown();
    }
}

```

Subscribe to ordered messages

The following sample code provides an example on how to subscribe to ordered messages:

```

using System;
using System.Text;

```

```

using System.Threading;
using ons;
namespace demo
{
    public class MyMsgOrderListener : MessageOrderListener
    {
        public MyMsgOrderListener()
        {
        }
        ~MyMsgOrderListener()
        {
        }
        public override ons.OrderAction consume(Message value, ConsumeOrderContext context)
        {
            Byte[] text = Encoding.Default.GetBytes(value.getBody());
            Console.WriteLine(Encoding.UTF8.GetString(text));
            return ons.OrderAction.Success;
        }
    }
    class OrderConsumerExampleForEx
    {
        static void Main(string[] args)
        {
            // Configure your account based on the resources that you created in the Apsara
            // Uni-manager Management Console.
            ONSFactoryProperty factoryInfo = new ONSFactoryProperty();
            // The AccessKey ID that you created in the Apsara Uni-manager Management Console
            // for identity authentication.
            factoryInfo.setFactoryProperty(ONSFactoryProperty.AccessKey, "Your access key");
            ;
            // The AccessKey secret that you created in the Apsara Uni-manager Management Console
            // for identity authentication.
            factoryInfo.setFactoryProperty(ONSFactoryProperty.SecretKey, "Your access secret");
            ;
            // The group ID that you created in the Message Queue for Apache RocketMQ console.
            factoryInfo.setFactoryProperty(ONSFactoryProperty.ConsumerId, "GID_example");
            // The topic that you created in the Message Queue for Apache RocketMQ console.
            factoryInfo.setFactoryProperty(ONSFactoryProperty.PublishTopics, "T_example_topic_name");
            ;
            // The TCP endpoint. To obtain the endpoint, log on to the Message Queue for Apache
            // RocketMQ console. In the left-side navigation pane, click Instance Details. On the Instance
            // Details page, select your instance. On the Instance Information tab, view the endpoint
            // in the Obtain Endpoint Information section.
            factoryInfo.setFactoryProperty(ONSFactoryProperty.NAMESRV_ADDR, "NameSrv_Addr");
            ;
            // Specify the log path.
            factoryInfo.setFactoryProperty(ONSFactoryProperty.LogPath, "C://log");
            // Create a consumer instance.
            OrderConsumer consumer = ONSFactory.getInstance().createOrderConsumer(factoryInfo);
            ;
            // Subscribe to topics.
            consumer.subscribe(factoryInfo.getPublishTopics(), "*", new MyMsgOrderListener());
        }
    }
}

```

```

        // Start the consumer instance.
        consumer.start();
        // Put the main thread to sleep for a period of time.
        Thread.Sleep(30000);
        // Disable the consumer instance when the instance is no longer used.
        consumer.shutdown();
    }
}
}

```

6.2.6.4. Send and subscribe to scheduled messages

Scheduled messages are consumed by consumers after a specified period of time. Such messages are used in scenarios where a time window between message production and consumption is required or tasks need to be triggered at a scheduled time. Scheduled messages are used in a similar way to delay queues.

Send scheduled messages

The following sample code provides an example on how to send scheduled messages:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.InteropServices;
using ons;
namespace ons
{
    class onscsharp
    {
        static void Main(string[] args)
        {
            // Set the parameters that are required to create a producer. These parameters
            ensure that you can use the producer.
            ONSFactoryProperty factoryInfo = new ONSFactoryProperty();
            factoryInfo.setFactoryProperty(factoryInfo.ProducerId, "XXX");// The group ID
            that you created in the Message Queue for Apache RocketMQ console.
            factoryInfo.setFactoryProperty(factoryInfo.NAMESRV_ADDR, "XXX");// The TCP end
            point. To obtain the endpoint, log on to the Message Queue for Apache RocketMQ console. In
            the left-side navigation pane, click Instance Details. On the Instance Details page, select
            your instance. On the Instance Information tab, view the endpoint in the Obtain Endpoint In
            formation section.
            factoryInfo.setFactoryProperty(factoryInfo.PublishTopics, "XXX");// The topic t
            hat you created in the Message Queue for Apache RocketMQ console.
            factoryInfo.setFactoryProperty(factoryInfo.MsgContent, "XXX");// The message co
            ntent.
            factoryInfo.setFactoryProperty(factoryInfo.AccessKey, "XXX");// The AccessKey I
            D that you created in the Apsara Uni-manager Management Console for identity authentication
            .
            factoryInfo.setFactoryProperty(factoryInfo.SecretKey, "XXX");// The AccessKey se
            cret that you created in the Apsara Uni-manager Management Console for identity authenticat
            ion.
        }
    }
}

```

```

        // Create a producer.
        ONSFactory onsfactory = new ONSFactory();
        Producer pProducer = onsfactory.getInstance().createProducer(factoryInfo);
        // Before you use the producer to send a message, call the start() method once
        to start the producer.
        pProducer.start();
        Message msg = new Message(
            //Message Topic
            factoryInfo.getPublishTopics(),
            //Message Tag
            "TagA",
            //Message Body
            factoryInfo.getMessageContent()
        );
        // The key of the message. The key is the business-specific attribute of the me
        ssage and must be globally unique whenever possible.
        // A unique key helps you query and resend a message in the Message Queue for A
        pache RocketMQ console if the message fails to be received.
        // Note: Messages can be sent and received even if you do not specify the messa
        ge key.
        msg.setKey("ORDERID_100");
        // The period of time after which the Message Queue for Apache RocketMQ broker
        delivers the message to the consumer. Unit: milliseconds. The message can be consumed only
        after the specified period of time. In this example, the message can be consumed 3 seconds
        later.
        long deliverTime = Current system time (ms) + 3000;
        msg.setStartDeliverTime(deliverTime);
        // Send the message. If no error occurs, the message is sent.
        try
        {
            SendResultONS sendResult = pProducer.send(msg);
        }
        catch(ONSClientException e)
        {
            // Specify the logic for handling failures.
        }
        // Before you exit the application, shut down the producer object. Otherwise, m
        emory leaks may occur.
        pProducer.shutdown();
    }
}

```

Subscribe to scheduled messages

For more information about how to subscribe to scheduled messages and about relevant sample code, see [Subscribe to messages](#).

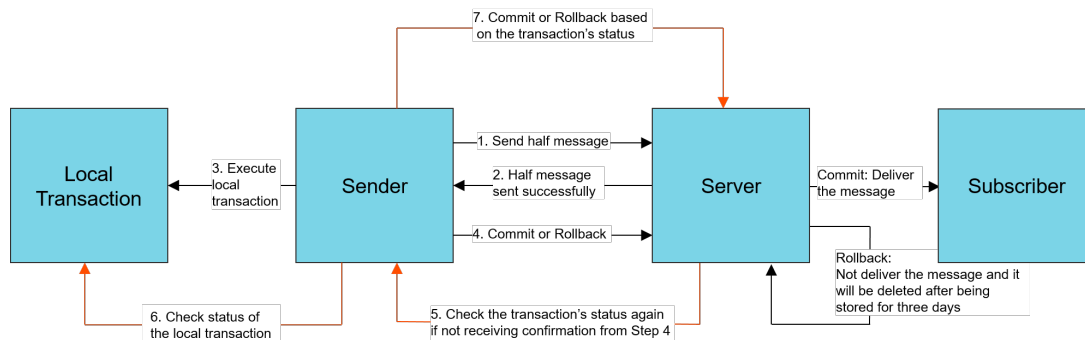
6.2.6.5. Send and subscribe to transactional messages

This topic describes the interaction process and the back-check mechanism of transactional messages. This topic also shows you how to send and subscribe to transactional messages, and provides sample code.

Interaction process

Transactional message interaction shows the interaction process of transactional messages in Message Queue for Apache RocketMQ.

Interaction process of transactional messages



Send transactional messages

Perform the following steps to send a transactional message:

1. Send a half message and execute a local transaction. The following code provides examples on how to send and subscribe to transactional messages:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.InteropServices;
using ons;
namespace ons
{
    public class MyLocalTransactionExecuter : LocalTransactionExecuter
    {
        public MyLocalTransactionExecuter()
        {
        }
        ~MyLocalTransactionExecuter()
        {
        }
        public override TransactionStatus execute(Message value)
        {
            Console.WriteLine("execute topic: {0}, tag:{1}, key:{2}, msgId:{3},msgbody :{4}, userProperty:{5}",
                value.getTopic(), value.getTag(), value.getKey(), value.getMsgID(), value.getBody(), value.getUserProperty("VincentNoUser"));
            // The ID of the message. Two messages may have the same message body but cannot have the same ID. The current message ID cannot be queried in the console.
            string msgId = value.getMsgID();
            // Calculate the message body by using CRC32 or other algorithms, such as MD5.

            // The message ID and CRC32 ID are used to prevent duplicate messages.
            // To prevent duplicate messages, calculate the message body by using the CRC32 or MD5 algorithm.
            TransactionStatus transactionStatus = TransactionStatus.Unknow;
        }
    }
}
  
```

```

        try {
            boolean isCommit = Execution result of the local transaction;
            if (isCommit) {
                // Commit the message if the local transaction succeeds.
                transactionStatus = TransactionStatus.CommitTransaction;
            } else {
                // Roll back the message if the local transaction fails.
                transactionStatus = TransactionStatus.RollbackTransaction;
            }
        } catch (Exception e) {
            //exception handle
        }
        return transactionStatus;
    }
}

class onscsharp
{
    static void Main(string[] args)
    {
        ONSFactoryProperty factoryInfo = new ONSFactoryProperty();
        factoryInfo.setFactoryProperty(factoryInfo.NAMESRV_ADDR, "XXX");//The TCP endpoint. To obtain the endpoint, log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click Instance Details. On the Instance Details page, select your instance. On the Instance Information tab, view the endpoint in the Obtain Endpoint Information section.
        factoryInfo.setFactoryProperty(factoryInfo.ProducerId, "");// The group ID that you created in the Message Queue for Apache RocketMQ console.
        factoryInfo.setFactoryProperty(factoryInfo.PublishTopics, "");// The topic that you created in the Message Queue for Apache RocketMQ console.
        factoryInfo.setFactoryProperty(factoryInfo.MsgContent, "");//message body
        factoryInfo.setFactoryProperty(factoryInfo.AccessKey, "");// The AccessKey ID that you created in the Apsara Uni-manager Management Console for identity authentication.
        factoryInfo.setFactoryProperty(factoryInfo.SecretKey, "");// The AccessKey secret that you created in the Apsara Uni-manager Management Console for identity authentication.

        //create transaction producer
        ONSFactory onsfactory = new ONSFactory();
        LocalTransactionChecker myChecker = new MyLocalTransactionChecker();
        TransactionProducer pProducer = onsfactory.getInstance().createTransactionProducer(factoryInfo, ref myChecker);
        // Before you use the producer to send a message, call the start() method once to start the producer. After the producer is started, messages can be concurrently sent in multiple threads.
        pProducer.start();
        Message msg = new Message(
            //Message Topic
            factoryInfo.getPublishTopics(),
            //Message Tag
            "TagA",
            //Message Body
            factoryInfo.getMessageContent()
        );
        // The key of the message. The key is the business-specific attribute of the message.
    }
}

```

```

message and must be globally unique whenever possible.
    // A unique key helps you query and resend a message in the Message Queue for
    Apache RocketMQ console if the message fails to be received.
    // Note: Messages can be sent and received even if you do not specify the mess
    age key.
    msg.setKey("ORDERID_100");
    // Send the message. If no error occurs, the message is sent.
    try
    {
        LocalTransactionExecuter myExecuter = new MyLocalTransactionExecuter();
        SendResultONS sendResult = pProducer.send(msg, ref myExecuter);
    }
    catch(ONSClientException e)
    {
        Console.WriteLine("\nexception of sendmsg:{0}",e.what() );
    }
    // Before you exit the application, shut down the producer object. Otherwise,
    memory leaks may occur.
    // The producer cannot be started again after the producer object is shut down
    .
    pProducer.shutdown();
}
}
}

```

2. Commit the status of the transactional message.

After the local transaction is executed, the Message Queue for Apache RocketMQ broker must be notified of the transaction status of the current message no matter whether the execution is successful or fails. The Message Queue for Apache RocketMQ broker can be notified in one of the following ways:

- Commit the status after the local transaction is executed.
- Wait until the Message Queue for Apache RocketMQ broker sends a request to check the transaction status of the message.

A transaction can be in one of the following states:

- `TransactionStatus.CommitTransaction`: The transaction is committed. The consumer can consume the message.
- `TransactionStatus.RollbackTransaction`: The transaction is rolled back. The message is discarded and cannot be consumed.
- `TransactionStatus.Unknown`: The status of the transaction is unknown. The Message Queue for Apache RocketMQ broker is expected to send a request again to the producer to query the status of the local transaction that corresponds to the message.

```

public class MyLocalTransactionChecker : LocalTransactionChecker
{
    public MyLocalTransactionChecker()
    {
    }
    ~MyLocalTransactionChecker()
    {
    }
    public override TransactionStatus check(Message value)
    {
        Console.WriteLine("check topic: {0}, tag:{1}, key:{2}, msgId:{3},msgbody:{4}, userProperty:{5}",
            value.getTopic(), value.getTag(), value.getKey(), value.getMsgID(), value.getBody(), value.getUserProperty("VincentNoUser"));
        // The ID of the message. Two messages may have the same message body but cannot have the same ID. The current message ID cannot be queried in the console.
        string msgId = value.getMsgID();
        // Calculate the message body by using CRC32 or other algorithms, such as MD5.
        // The message ID and CRC32 ID are used to prevent duplicate messages.
        // You do not need to specify the message ID or CRC32 ID if your business itself achieves idempotence. Otherwise, specify the message ID or CRC32 ID to ensure idempotence.
        // To prevent duplicate messages, calculate the message body by using the CRC32 or MD5 algorithm.
        TransactionStatus transactionStatus = TransactionStatus.Unknow;
        try {
            boolean isCommit = Execution result of the local transaction;
            if (isCommit) {
                // Commit the message if the local transaction succeeds.
                transactionStatus = TransactionStatus.CommitTransaction;
            } else {
                // Roll back the message if the local transaction fails.
                transactionStatus = TransactionStatus.RollbackTransaction;
            }
        } catch (Exception e) {
            //exception handle
        }
        return transactionStatus;
    }
}

```

Back-check mechanism for transaction status

- Why must the back-check mechanism for transaction status be implemented when transactional messages are sent?

If the half message is sent in Step 1 but `TransactionStatus.Unknow` is returned for the local transaction, or no status is committed for the local transaction because the application exits, the status of the half message is unknown to the Message Queue for Apache RocketMQ broker. Therefore, the Message Queue for Apache RocketMQ broker periodically requests the producer to check and report the status of the half message.

- What does the business logic do when the check method is called back?

The check method for transactional messages in Message Queue for Apache RocketMQ must contain the logic of transaction consistency check. After a transactional message is sent, Message Queue for Apache RocketMQ must call the `LocalTransactionChecker` method to respond to the request of the Message Queue for Apache RocketMQ broker for the status of the local transaction. Therefore, the check method for transactional messages must contain the following check items:

- i. Check the status of the local transaction that corresponds to the half message. The status is committed or rollback.
 - ii. Commit the status of the local transaction that corresponds to the half message to the Message Queue for Apache RocketMQ broker.
- How do different states of the local transaction affect the half message?
 - `TransactionStatus.CommitTransaction`: The transaction is committed. The consumer can consume the message.
 - `TransactionStatus.RollbackTransaction`: The transaction is rolled back. The message is discarded and cannot be consumed.
 - `TransactionStatus.Unknown`: The status of the transaction is unknown. The Message Queue for Apache RocketMQ broker is expected to send a request again to the producer to query the status of the local transaction that corresponds to the message.

For more information about the code, see the implementation of `MyLocalTransactionChecker`.

Subscribe to transactional messages

For more information about how to subscribe to transactional messages and about relevant sample code, see [Subscribe to messages](#).

6.2.6.6. Subscribe to messages

This topic describes how to use Message Queue for Apache RocketMQ SDK for .NET to subscribe to messages.

Note

- The subscriptions of all consumer instances identified by the same group ID must be consistent. For more information, see [Subscription consistency](#).

Subscription modes

Message Queue for Apache RocketMQ supports the following message subscription modes:

- **Clustering subscription**: In this mode, all the consumer instances identified by the same group ID evenly share messages. Assume that a topic contains nine messages and a group ID identifies three consumer instances. In clustering consumption mode, each instance consumes three messages.

```
// Configure clustering subscription, which is the default mode.
factoryInfo.setFactoryProperty(ONSFactoryProperty.MessageModel, ONSFactoryProperty.CLUSTERING);
```

- **Broadcasting subscription**: In this mode, each consumer instance identified by a group ID consumes a message once. Assume that a topic contains nine messages and a group ID identifies three consumer instances. In broadcasting consumption mode, each instance consumes nine messages.

```
// Configure broadcasting subscription.
factoryInfo.setFactoryProperty(ONSFactoryProperty.MessageModel, ONSFactoryProperty.BROADCASTING);
```

Sample code

```
using System;
using System.Threading;
using System.Text;
using ons;
// The callback function to be executed when a message is pulled from the Message Queue for
// Apache RocketMQ broker.
public class MyMsgListener : MessageListener
{
    public MyMsgListener()
    {
    }
    ~MyMsgListener()
    {
    }
    public override ons.Action consume(Message value, ConsumeContext context)
    {
        Byte[] text = Encoding.Default.GetBytes(value.getBody());
        Console.WriteLine(Encoding.UTF8.GetString(text));
        return ons.Action.CommitMessage;
    }
}
public class ConsumerExampleForEx
{
    public ConsumerExampleForEx()
    {
    }
    static void Main(string[] args) {
        // Configure your account based on the resources that you created in the console.
        ONSFactoryProperty factoryInfo = new ONSFactoryProperty();
        // The AccessKey ID that you created in the Apsara Uni-manager Management Console f
        // or identity authentication.
        factoryInfo.setFactoryProperty(ONSFactoryProperty.AccessKey, "Your access key");
        // The AccessKey secret that you created in the Apsara Uni-manager Management Conso
        // le for identity authentication.
        factoryInfo.setFactoryProperty(ONSFactoryProperty.SecretKey, "Your access secret");
        // The group ID that you created in the Message Queue for Apache RocketMQ console.
        factoryInfo.setFactoryProperty(ONSFactoryProperty.ConsumerId, "GID_example");
        // The topic that you created in the Message Queue for Apache RocketMQ console.
        factoryInfo.setFactoryProperty(ONSFactoryProperty.PublishTopics, "T_example_topic_n
        ame");
        // The TCP endpoint. To obtain the endpoint, log on to the Message Queue for Apache
        // RocketMQ console. In the left-side navigation pane, click Instance Details. On the Instance
        // Details page, select your instance. On the Instance Information tab, view the endpoint in t
        // he Obtain Endpoint Information section.
        factoryInfo.setFactoryProperty(ONSFactoryProperty.NAMESRV_ADDR, "NameSrv_Addr");
        // Specify the log path.
        factoryInfo.setFactoryProperty(ONSFactoryProperty.LogPath, "C://log");
        // The clustering consumption mode
```

```
// The clustering consumption mode.
// factoryInfo.setFactoryProperty(ONSFactoryProperty:: MessageModel, ONSFactoryProp
erty.CLUSTERING);
// The broadcasting consumption mode.
// factoryInfo.setFactoryProperty(ONSFactoryProperty:: MessageModel, ONSFactoryProp
erty.BROADCASTING);
// Create a consumer instance.
PushConsumer consumer = ONSFactory.getInstance().createPushConsumer(factoryInfo);
// Subscribe to topics.
consumer.subscribe(factoryInfo.getPublishTopics(), "*", new MyMsgListener());
// Start the consumer instance.
consumer.start();
// This value is an example in the demo. In your production environment, you must s
et a proper value to make sure that the process does not unexpectedly exit.
Thread.Sleep(300000);
// Shut down the consumer instance when the process is about to exit.
consumer.shutdown();
}
}
```

6.3. HTTP client SDK reference

6.3.1. Protocol description

6.3.1.1. Common parameters

This topic describes the common request parameters in an HTTP request header and the common response parameters in an HTTP response header for Message Queue for Apache RocketMQ.

Common request header

Parameter	Required	Description
Authorization	Yes	The authorization string. Specify the value in the following format: <code>MQ <AccessKey ID>: <Signature></code> . For more information, see Sign signatures .
Content-Length	Yes	The length of the HTTP request body.
Content-Type	Yes	The Multipurpose Internet Mail Extensions (MIME) type of the request body. Set the value to <code>text/xml; charset=utf-8</code> . This value sets the MIME type to XML and the character encoding method to UTF-8.

Parameter	Required	Description
Date	Yes	The time when the request is constructed. The time must be in UTC. If the interval between the time when the request is constructed and the time when the request is received exceeds 15 minutes, the Message Queue for Apache RocketMQ broker determines that the request is invalid.
Host	Yes	The HTTP endpoint that you obtain on the Instances page of the console.
x-mq-version	Yes	The version of the Message Queue for Apache RocketMQ API. Set the value to 2015-06-06.
Content-MD5	No	The MD5 hash value of the message body. For more information, see Content-MD5 Header Field .

Common response header

Parameter	Description
Content-Length	The length of the HTTP response body.
Connection	The status of the HTTP connection.
Date	The time when the response was returned. The time is displayed in UT C.
x-mq-request-id	The ID of the request.
x-mq-version	The version of the Message Queue for Apache RocketMQ API. The value is 2015-06-06.

6.3.1.2. Request signatures

Message Queue for Apache RocketMQ verifies each HTTP access request. Each HTTP request that is sent to Message Queue for Apache RocketMQ contains the Authorization parameter in the request header, and the Authorization parameter contains a signature. This topic describes how to generate a signature.

Background information

Apsara Stack issues an AccessKey pair that consists of an AccessKey ID and an AccessKey secret to each user. The user can apply for and manage AccessKey pairs in the Apsara Uni-manager Management Console.

- The AccessKey ID is used to verify the identity of the user.
- The AccessKey secret is used to encrypt and verify the signature string. You must keep your AccessKey secret strictly confidential.

The HTTP service provided by Message Queue for Apache RocketMQ uses an AccessKey pair to perform symmetric encryption to verify the identity of a request sender. If the calculated verification code is the same as that provided in the request, the HTTP service determines that the request is valid. Otherwise, the HTTP service rejects the request and returns HTTP 403.

You must add the Authorization parameter to the header of each HTTP request to provide the signature of the request. This way, the HTTP service can determine the validity of the request.

How to sign a request

The Authorization parameter is specified in the following format:

```
MQ <AccessKey ID>:<Signature>
```

The following code shows the parameters that are used to generate a signature:

```
Signature = base64(hmac-sha1(HTTP_METHOD + "\n"
    + "\n" + CONTENT-TYPE + "\n"
    + DATE + "\n"
    + "x-mq-version:" + MQVersion + "\n"
    + CanonicalizedResource))
```

- HTTP_METHOD: specifies an HTTP method in uppercase, such as PUT, GET, POST, or DELETE.
- CONTENT-TYPE: specifies the type of the request body. Set the value to text/xml; charset=utf-8.
- DATE: specifies the time when you want to perform the operation. This parameter cannot be left empty and must be specified in UTC. For example, you can set this parameter to Thu, 07 Mar 2012 18:49:58 GMT.
- MQVersion: specifies the version of the Message Queue for Apache RocketMQ API. Set the value to 2015-06-06.
- CanonicalizedResource: specifies the Uniform Resource Identifier (URI) of the resource requested by the HTTP request. For example, set the URI of a consumption request to /topics/abc/messages?consumer=GID_abc.

Note

- The string-to-sign must be in the UTF-8 format.
- The HMAC-SHA1 method defined in [RFC 2104](#) is used to calculate the signature. In this method, the AccessKey secret is used as an encryption key.

6.3.1.3. Operation for sending messages

You can call this operation to send messages from a producer to a Message Queue for Apache RocketMQ broker.

Request structure

- Request line

```
POST /topics/TopicName/messages?ns=INSTANCE_ID HTTP/1.1
```

The following table describes the parameters in the request line.

Parameter	Required	Description
TopicName	Yes	The name of the destination topic to which you want to send messages.
ns	No	<p>The ID of the instance. This parameter is required for new instances that have namespaces. You can check whether your instance has a namespace on the Instances page in the Message Queue for Apache RocketMQ console. Instances are classified into default instances and new instances based on whether they have namespaces.</p> <ul style="list-style-type: none">Default instance: A default instance does not have a namespace. The names of all resources in a default instance must be globally unique.New instance: A new instance has a namespace. The names of all resources in a new instance must be unique within the instance. <p>For more information about namespaces for Message Queue for Apache RocketMQ instances, see Use instances.</p>

- Request body (XML format)

The following table describes the parameters in the request body.

Parameter	Required	Description
MessageTag	No	The tag of the message.
MessageBody	Yes	The content of the message.
Properties	No	The properties of the message.

The following information describes the key-value pairs in the serialized properties of the message:

- Specify the key-value pairs in the following format: `key1:value1|key2:value2|key3:value3`.
- The following table describes the parameters that are used to specify the key-value pairs.

Parameter	Type	Description
KEYS	String	The key of the message.
__STARTDELIVERTIME	Long	The absolute scheduled time of a scheduled message. Set this parameter to a UNIX timestamp that represents the number of milliseconds.
__TransCheckT	Long	The relative time when you want to perform the first status check for a transactional message. Unit: seconds. Valid values: 10 to 300.

Response structure

- Status line

```
HTTP/1.1 201
```

- Response body

The following table describes the parameters in the response body.

Parameter	Type	Description
MessageId	String	The ID of the message.
MessageBodyMD5	String	The MD5 hash value of the message body.

Examples

- Sample requests

```
<?xml version="1.0" encoding="UTF-8"?>
<Message xmlns="http://mq.aliyuncs.com/doc/v1/">
  <MessageBody>a</MessageBody>
  <MessageTag>Tag</MessageTag>
  <Properties>KEYS:MessageKey|__STARTDELIVERTIME:1571388173000</Properties>
</Message>
```

- Sample responses

```
<Message xmlns="http://mq.aliyuncs.com/doc/v1/">
  <MessageId>1E057D566EAD42A579935B5CD874****</MessageId>
  <MessageBodyMD5>0CC175B9C0F1B6A831C399E26977****</MessageBodyMD5>
</Message>
```

6.3.1.4. Operation for consuming messages

You can call this operation to consume messages from a Message Queue for Apache RocketMQ broker.

Request structure

- Request line

```
GET /topics/TopicName/messages?ns=INSTANCE_ID&consumer=GID&tag=taga&numOfMessages=3&waitseconds=3 HTTP/1.1
```

The following table describes the parameters in the request line.

Parameter	Required	Description
TopicName	Yes	The name of the topic from which you want to consume messages.
ns	No	<p>The ID of the instance. This parameter is required for new instances that have namespaces.</p> <p>You can check whether your instance has a namespace on the Instances page in the Message Queue for Apache RocketMQ console. Instances are classified into default instances and new instances based on whether they have namespaces.</p> <ul style="list-style-type: none">Default instance: A default instance does not have a namespace. The names of all resources in a default instance must be globally unique.New instance: A new instance has a namespace. The names of all resources in a new instance must be unique within the instance. <p>For more information, see Use instances.</p>
consumer	Yes	The ID of the consumer group.
tag	No	The tag of the message. If you do not specify a tag, all messages are pulled. If you want to specify multiple tags, separate them with double vertical bars (). For example, you can set this parameter to TagA TagB.

Parameter	Required	Description
numOfMessages	Yes	The maximum number of messages that can be consumed at a time. Valid values: 1 to 16.
waitseconds	No	The long polling period. If you do not specify this parameter, short polling is used. Valid values: 1 to 30. Unit: seconds.

- Request body (XML format)

None

Response structure

- A message is available for consumption.


- Status line

```
HTTP/1.1 200
```

- Response body

The following table describes the parameters in the response body.

Parameter	Type	Description
MessageId	String	The ID of the message.
MessageBodyMD5	String	The MD5 hash value of the message body.
MessageBody	String	The content of the message.
ReceiptHandle	String	The receipt handle that is used to acknowledge that a message is consumed. The receipt handle can be used only once and must be used before the period of time specified by the NextConsumeTime parameter elapses. The receipt handles that are obtained each time the same message is retried and consumed are different.
PublishTime	String	The timestamp that indicates the time when the message was sent. Unit: milliseconds.

Parameter	Type	Description
FirstConsumeTime	String	The timestamp that indicates the time when the message was consumed for the first time. Unit: milliseconds.
NextConsumeTime	String	<p>The timestamp that indicates the absolute time when the message was retried. Unit: milliseconds.</p> <div><p> Note If a message that is sent over HTTP fails to be consumed, Message Queue for Apache RocketMQ retries to send the message based on the following mechanism: Unordered messages are retried every 5 minutes, and ordered messages are retried every 1 minute. A maximum of 288 retries are allowed for both ordered and unordered messages.</p></div>
ConsumedTimes	String	The number of retries after the message failed to be consumed.
MessageTag	String	The tag of the message.
Properties	String	The properties of the message.

The following information describes the key-value pairs in the serialized properties of the message:

- The key-value pairs are displayed in the following format: `key1:value1|key2:value2|key3:valu
e3`.

- The following table describes the parameters that are used to indicate the key-value pairs.

Parameter	Type	Description
KEYS	String	The key of the message.
__STARTDELIVERTIME	Long	The absolute scheduled time of a scheduled message. This value is a UNIX timestamp that represents the number of milliseconds.
__TransCheckT	Long	The relative time that indicates the time when the first status check for a transactional message is performed. Unit: seconds. Valid values: 10 to 300.

- No message is available for consumption.

- Status line

```
HTTP/1.1 404
```

- Response body

The following table describes the parameters in the response body.

Parameter	Type	Description
Code	String	The error code. <code>MessageNotExist</code> indicates that no message is available for consumption. If this error code is returned, the response is a normal response.
Message	String	The error message returned.
RequestId	String	The ID of the request.
HostId	String	The host that sent the request.

Sample responses

- A message is available for consumption.

```
<?xml version="1.0" ?>
<Messages xmlns="http://mq.aliyuncs.com/doc/v1">
  <Message>
    <MessageId>1E057D5E6EAD42A579937046FE17****</MessageId>
    <MessageBodyMD5>0CC175B9C0F1B6A831C399E26977****</MessageBodyMD5>
    <MessageBody>a</MessageBody>
    <ReceiptHandle>1E057D5E6EAD42A579937046FE17****-MTI5N****</ReceiptHandle>
    <PublishTime>1571742900759</PublishTime>
    <FirstConsumeTime>1571742902463</FirstConsumeTime>
    <NextConsumeTime>1571742922463</NextConsumeTime>
    <ConsumedTimes>1</ConsumedTimes>
    <MessageTag>Tag</MessageTag>
    <Properties>KEYS:MessageKey|__BORNHOST:30.5.**.*|</Properties>
  </Message>
  <Message>
    <MessageId>1E057D5E6EAD42A579937046FE17****</MessageId>
    <MessageBodyMD5>0CC175B9C0F1B6A831C399E26977****</MessageBodyMD5>
    <MessageBody>a</MessageBody>
    <ReceiptHandle>1E057D5E6EAD42A579937046FE17****-MTI5N****</ReceiptHandle>
    <PublishTime>1571742900759</PublishTime>
    <FirstConsumeTime>1571742902463</FirstConsumeTime>
    <NextConsumeTime>1571742922463</NextConsumeTime>
    <ConsumedTimes>1</ConsumedTimes>
    <MessageTag>Tag</MessageTag>
    <Properties>KEYS:MessageKey|__BORNHOST:30.5.**.*|</Properties>
  </Message>
</Messages>
```

- No message is available for consumption.

```
<?xml version="1.0" ?>
<Error xmlns="http://mq.aliyuncs.com/doc/v1">
  <Code>MessageNotExist</Code>
  <Message>Message not exist.</Message>
  <RequestId>5DAEE3FF463541AD6E0322EB</RequestId>
  <HostId>http://123.mqrest.cn-hangzhou.aliyuncs.com</HostId>
</Error>
```

6.3.1.5. Operation for acknowledging messages

You can call this operation to acknowledge the consumption status of messages.

Request structure

- Request line

```
DELETE /topics/TopicName/messages?ns=INSTANCE_ID&consumer=GID HTTP/1.1
```

The following table describes the parameters in the request line.

Parameter	Required	Description
-----------	----------	-------------

Parameter	Required	Description
TopicName	Yes	The name of the topic that contains the messages you want to acknowledge.
ns	No	<p>The ID of the instance. This parameter is required for new instances that have namespaces. You can check whether your instance has a namespace on the Instances page in the Message Queue for Apache RocketMQ console. Instances are classified into default instances and new instances based on whether they have namespaces.</p> <ul style="list-style-type: none">◦ Default instance: A default instance does not have a namespace. The names of all resources in a default instance must be globally unique.◦ New instance: A new instance has a namespace. The names of all resources in a new instance must be unique within the instance. <p>For more information about namespaces for Message Queue for Apache RocketMQ instances, see Use instances.</p>
consumer	Yes	The ID of the consumer group.

- Request body (XML format)

The following table describes the parameters in the request body.

Parameter	Required	Description
-----------	----------	-------------

Parameter	Required	Description
ReceiptHandle	Yes	The receipt handle that is used to acknowledge that a message is consumed. You can call the message consumption operation to obtain the receipt handle. For more information, see Operation for consuming messages . The receipt handle can be used only once and must be used before the period of time specified by the NextConsumeTime parameter elapses. The receipt handles that are obtained each time the same message is retried and consumed are different.

Response structure

- The request is successful.

- Status line

```
HTTP/1.1 204
```

- Response body

None

- The request failed.

- Status line

```
HTTP/1.1 404
```

- Response body

For more information, see [Sample responses](#).

Examples

- Sample requests

```
<?xml version="1.0" encoding="UTF-8"?>
<ReceiptHandles xmlns="http://mq.aliyuncs.com/doc/v1/">
  <ReceiptHandle>1E057D5E6EAD42A57993704EC383****-MTI5NT****</ReceiptHandle>
  <ReceiptHandle>1E057D5E6EAD42A57993704EC383****-MTI5NT****</ReceiptHandle>
  <ReceiptHandle>1E057D5E6EAD42A57993704EC383****-MTI5NT****</ReceiptHandle>
</ReceiptHandles>
```

- Sample responses

- The request body does not contain a handle.

```
<?xml version="1.0" ?>
<Error xmlns="http://mq.aliyuncs.com/doc/v1">
  <Code>MissingReceiptHandle</Code>
  <Message>ReceiptHandle is required.</Message>
  <RequestId>5DAEF2B9463541AD6E04490F</RequestId>
  <HostId>http://123.mqrest.cn-hangzhou.aliyuncs.com</HostId>
</Error>
```

- The handle of the request is incorrect. The handle is `adfadfadf` .

```
<?xml version="1.0" ?>
<Errors xmlns="http://mq.aliyuncs.com/doc/v1">
  <Error>
    <ErrorCode>ReceiptHandleError</ErrorCode>
    <ErrorMessage>The receipt handle you provide is not valid.</ErrorMessage>
    <ReceiptHandle>adfadfadf</ReceiptHandle>
  </Error>
</Errors>
```

- The handle of the request has expired. This indicates that the handle is not used before the period of time specified by `NextConsumeTime` elapses.

```
<?xml version="1.0" ?>
<Errors xmlns="http://mq.aliyuncs.com/doc/v1">
  <Error>
    <ErrorCode>MessageNotExist</ErrorCode>
    <ErrorMessage>The receipt handle you provided has expired.</ErrorMessage>
    <ReceiptHandle>1E057D5E6EAD42A57993704EC383****-MTI5NT****</ReceiptHandle>
  </Error>
</Errors>
```

6.3.2. Java SDK

6.3.2.1. Prepare the environment

This topic describes how to prepare the environment before you use the HTTP client SDK for Java to send and consume messages.


Environment requirements

- Java Development Kit (JDK) 1.6 or later is installed. For more information, see [Java Downloads](#).
- Maven is installed. For more information, see [Downloading Apache Maven 3.8.6](#).

Install the SDK for Java

Use Maven to import dependencies and add the following dependency to the `pom.xml` file:

```
<dependency>
  <groupId>com.aliyun.mq</groupId>
  <artifactId>mq-http-sdk</artifactId>
  <!-- Specify the version number of the SDK for Java. -->
  <version>X.X.X</version>
  <classifier>jar-with-dependencies</classifier>
</dependency>
```

 **Note** For more information about the versions of the SDK for Java, see [Overview](#).

6.3.2.2. Send and consume normal messages

Normal messages are messages that have no special features in Message Queue for Apache RocketMQ. They are different from featured messages, such as scheduled messages, delayed messages, ordered messages, and transactional messages. This topic provides sample code to show how to use the HTTP client SDK for Java to send and consume normal messages.

Prerequisites

The following operations are performed:

- Install the SDK for Java. For more information about the message routing feature, see [Prepare the environment](#).
- Create resources that you want to specify in the code. For example, you must create the instance, topic, and group that you want to specify in the code in the Message Queue for Apache RocketMQ console in advance. For more information about the message routing feature, see [Create resources](#).

Send normal messages

The following sample code provides an example on how to send normal messages:

```
import com.aliyun.mq.http.MQClient;
import com.aliyun.mq.http.MQProducer;
import com.aliyun.mq.http.model.TopicMessage;
import java.util.Date;

public class Producer {
    public static void main(String[] args) {
        MQClient mqClient = new MQClient(
            // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click Instances. On the Instances page, select the name of your instance. Then, view the HTTP endpoint on the Network Management tab.
            "${HTTP_ENDPOINT}",
            // The AccessKey ID that is used for identity verification. You can obtain the AccessKey ID in the Apsara Uni-manager Operations Console.
            "${ACCESS_KEY}",
            // The AccessKey secret that is used for identity verification. You can obtain the AccessKey secret in the Apsara Uni-manager Operations Console.
            "${SECRET_KEY}"
        );
        // The topic to which you want to send messages. The topic is created in the Message Queue for Apache RocketMQ console.
```

```

// Each topic can be used to send and consume messages of a specific type. For exam
ple, a topic that is used to send and consume normal messages cannot be used to send and co
nsume messages of other types.
    final String topic = "${TOPIC}";
    // The ID of the instance to which the topic belongs. The instance is created in th
e Message Queue for Apache RocketMQ console.
    // If the instance has a namespace, specify the ID of the instance. If the instance
does not have a namespace, set the instance ID to null or an empty string. You can check wh
ether your instance has a namespace on the Instances page in the RocketMQ console.
    final String instanceId = "${INSTANCE_ID}";
    // Obtain the producer that sends messages to the topic.
MQProducer producer;
    if (instanceId != null && instanceId != "") {
        producer = mqClient.getProducer(instanceId, topic);
    } else {
        producer = mqClient.getProducer(topic);
    } try {
        // Cyclically send four messages.
        for (int i = 0; i < 4; i++) {
            TopicMessage pubMsg; // The normal message.
            pubMsg = new TopicMessage(
                // The content of the message.
                "hello mq!".getBytes(),
                // The tag of the message.
                "A"
            );
            // The custom property of the message.
            pubMsg.getProperties().put("a", String.valueOf(i));
            // The key of the message.
            pubMsg.setMessageKey("MessageKey");
            // Send the message in synchronous mode. If no exception is thrown, the message
is sent.
            TopicMessage pubResultMsg = producer.publishMessage(pubMsg);
            // Send the message in synchronous mode. If no exception is thrown, the message
is sent.
            System.out.println(new Date() + " Send mq message success. Topic is:" + topic +
", msgId is: " + pubResultMsg.getMessageId()
                + ", bodyMD5 is: " + pubResultMsg.getMessageBodyMD5());
        }
    } catch (Throwable e) {
        // Specify the logic that you want to use to resend or persist the message if t
he message fails to be sent and needs to be sent again.
        System.out.println(new Date() + " Send mq message failed. Topic is:" + topic);
        e.printStackTrace();
    }
    mqClient.close();
}
}

```

Consume normal messages

The following sample code provides an example on how to consume normal messages:

```
import com.aliyun.mq.http.MQClient;
```

```

import com.aliyun.mq.http.MQConsumer;
import com.aliyun.mq.http.common.AckMessageException;
import com.aliyun.mq.http.model.Message;
import java.util.ArrayList;
import java.util.List;
public class Consumer {
    public static void main(String[] args) {
        MQClient mqClient = new MQClient(
            // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click Instances. On the Instances page, select the name of your instance. Then, view the HTTP endpoint on the Network Management tab.
            "${HTTP_ENDPOINT}",
            // The AccessKey ID that is used for identity verification. You can obtain the AccessKey ID in the Apsara Uni-manager Operations Console.
            "${ACCESS_KEY}",
            // The AccessKey secret that is used for identity verification. You can obtain the AccessKey secret in the Apsara Uni-manager Operations Console.
            "${SECRET_KEY}"
        );
        // The topic from which you want to consume messages. The topic is created in the Message Queue for Apache RocketMQ console.
        // Each topic can be used to send and consume messages of a specific type. For example, a topic that is used to send and consume normal messages cannot be used to send and consume messages of other types.
        final String topic = "${TOPIC}";
        // The ID of the group that you created in the Message Queue for Apache RocketMQ console.
        final String groupId = "${GROUP_ID}";
        // The ID of the instance to which the topic belongs. The instance is created in the Message Queue for Apache RocketMQ console.
        // If the instance has a namespace, specify the ID of the instance. If the instance does not have a namespace, set the instance ID to null or an empty string. You can check whether your instance has a namespace on the Instances page in the RocketMQ console.
        final String instanceId = "${INSTANCE_ID}";
        final MQConsumer consumer;
        if (instanceId != null && instanceId != "") {
            consumer = mqClient.getConsumer(instanceId, topic, groupId, null);
        } else {
            consumer = mqClient.getConsumer(topic, groupId);
        }
        // Cyclically consume messages in the current thread. We recommend that you use multiple threads to concurrently consume messages.
        do {
            List<Message> messages = null;
            try {
                // Consume messages in long polling mode.
                // In long polling mode, if no message in the topic is available for consumption, the request is suspended on the broker for a specified period of time. If a message becomes available for consumption within this period, the broker immediately sends a response to the consumer. In this example, the period is set to 3 seconds.
                messages = consumer.consumeMessage(
                    3, // The maximum number of messages that can be consumed at a time. In this example, the value is set to 3. The maximum value that you can specify is 16.

```

```

        3// The length of a long polling period. Unit: seconds. In this example, the value is set to 3. The maximum value that you can specify is 30.
    );
    } catch (Throwable e) {
        e.printStackTrace();
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e1) {
            e1.printStackTrace();
        }
    }
    // No messages in the topic are available for consumption.
    if (messages == null || messages.isEmpty()) {
        System.out.println(Thread.currentThread().getName() + ": no new message, continue!");
        continue;
    }
    // Specify the message consumption logic.
    for (Message message : messages) {
        System.out.println("Receive message: " + message);
    }
    // If the broker does not receive an acknowledgment (ACK) for a message from the consumer before the delivery retry interval elapses, the broker delivers the message for consumption again.
    // A unique timestamp is specified for the handle of a message each time the message is consumed.
    {
        List<String> handles = new ArrayList<String>();
        for (Message message : messages) {
            handles.add(message.getReceiptHandle());
        }
        try {
            consumer.ackMessage(handles);
        } catch (Throwable e) {
            // If the handle of a message times out, the broker cannot receive an ACK for the message from the consumer.
            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);

```

```
}
}
```

6.3.2.3. Send and consume ordered messages

Ordered messages are a type of message that is published and consumed in a strict order. Ordered messages in Message Queue for Apache RocketMQ are also known as first-in-first-out (FIFO) messages. This topic provides sample code to show how to use the HTTP client SDK for Java to send and consume ordered messages.

Background information

Ordered messages are classified into the following types:

- Globally ordered message: All messages in a specified topic are published and consumed in first-in-first-out (FIFO) order.
- Partititionally ordered message: All messages in a specified topic are distributed to different partitions by using shard keys. The messages in each partition are published and consumed in FIFO order. A Sharding Key is a key field that is used for ordered messages to identify different partitions. The Sharding Key is different from the key of a normal message.

For more information about the message routing feature, see [Ordered messages](#).

Prerequisites

The following operations are performed:

- Install the SDK for Java. For more information about the message routing feature, see [Prepare the environment](#).
- Create resources that you want to specify in the code. For example, you must create the instance, topic, and group that you want to specify in the code in the Message Queue for Apache RocketMQ console in advance. For more information about the message routing feature, see [Create resources](#).

Send ordered messages

The following sample code provides an example on how to send ordered messages:

```
import com.aliyun.mq.http.MQClient;
import com.aliyun.mq.http.MQProducer;
import com.aliyun.mq.http.model.TopicMessage;
import java.util.Date;
public class OrderProducer {
    public static void main(String[] args) {
        MQClient mqClient = new MQClient(
            // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click Instances. On the Instances page, select the name of your instance. Then, view the HTTP endpoint on the Network Management tab.
            "${HTTP_ENDPOINT}",
            // The AccessKey ID that is used for identity verification. You can obtain the AccessKey ID in the Apsara Uni-manager Operations Console.
            "${ACCESS_KEY}",
            // The AccessKey secret that is used for identity verification. You can obtain the AccessKey secret in the Apsara Uni-manager Operations Console.
            "${SECRET_KEY}"
        );
    }
}
```

```

    );
    // The topic to which you want to send messages. The topic is created in the Message Queue for Apache RocketMQ console.
    // Each topic can be used to send and consume messages of a specific type. For example, a topic that is used to send and consume normal messages cannot be used to send and consume messages of other types.
    final String topic = "${TOPIC}";
    // The ID of the instance to which the topic belongs. The instance is created in the Message Queue for Apache RocketMQ console.
    // If the instance has a namespace, specify the ID of the instance. If the instance does not have a namespace, set the instance ID to null or an empty string. You can check whether your instance has a namespace on the Instances page in the RocketMQ console.
    final String instanceId = "${INSTANCE_ID}";
    // Obtain the producer that sends messages to the topic.
    MQProducer producer;
    if (instanceId != null && instanceId != "") {
        producer = mqClient.getProducer(instanceId, topic);
    } else {
        producer = mqClient.getProducer(topic);
    }
    try {
        // Cyclically send eight messages.
        for (int i = 0; i < 8; i++) {
            TopicMessage pubMsg = new TopicMessage(
                // The content of the message.
                "hello mq!".getBytes(),
                // The tag of the message.
                "A"
            );
            // The shard key that is used to distribute ordered messages to a specific partition. Shard keys can be used to identify different partitions. A shard key is different from a message key.
            pubMsg.setShardingKey(String.valueOf(i % 2));
            pubMsg.getProperties().put("a", String.valueOf(i));
            // Send the message in synchronous mode. If no exception is thrown, the message is sent.
            TopicMessage pubResultMsg = producer.publishMessage(pubMsg);
            // Send the message in synchronous mode. If no exception is thrown, the message is sent.
            System.out.println(new Date() + " Send mq message success. Topic is:" + topic + ", msgId is: " + pubResultMsg.getMessageId()
                + ", bodyMD5 is: " + pubResultMsg.getMessageBodyMD5());
        }
    } catch (Throwable e) {
        // Specify the logic that you want to use to resend or persist the message if the message fails to be sent and needs to be sent again.
        System.out.println(new Date() + " Send mq message failed. Topic is:" + topic);
        e.printStackTrace();
    }
    mqClient.close();
}
}

```

Consume ordered messages

The following sample code provides an example on how to consume ordered messages:

```
import com.aliyun.mq.http.MQClient;
import com.aliyun.mq.http.MQConsumer;
import com.aliyun.mq.http.common.AckMessageException;
import com.aliyun.mq.http.model.Message;
import java.util.ArrayList;
import java.util.List;
public class OrderConsumer {
    public static void main(String[] args) {
        MQClient mqClient = new MQClient(
            // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click Instances. On the Instances page, select the name of your instance. Then, view the HTTP endpoint on the Network Management tab.
            "${HTTP_ENDPOINT}",
            // The AccessKey ID that is used for identity verification. You can obtain the AccessKey ID in the Apsara Uni-manager Operations Console.
            "${ACCESS_KEY}",
            // The AccessKey secret that is used for identity verification. You can obtain the AccessKey secret in the Apsara Uni-manager Operations Console.
            "${SECRET_KEY}"
        );
        // The topic from which you want to consume messages. The topic is created in the Message Queue for Apache RocketMQ console.
        // Each topic can be used to send and consume messages of a specific type. For example, a topic that is used to send and consume normal messages cannot be used to send and consume messages of other types.
        final String topic = "${TOPIC}";
        // The ID of the group that you created in the Message Queue for Apache RocketMQ console.
        final String groupId = "${GROUP_ID}";
        // The ID of the instance to which the topic belongs. The instance is created in the Message Queue for Apache RocketMQ console.
        // If the instance has a namespace, specify the ID of the instance. If the instance does not have a namespace, set the instance ID to null or an empty string. You can check whether your instance has a namespace on the Instances page in the RocketMQ console.
        final String instanceId = "${INSTANCE_ID}";
        final MQConsumer consumer;
        if (instanceId != null && instanceId != "") {
            consumer = mqClient.getConsumer(instanceId, topic, groupId, null);
        } else {
            consumer = mqClient.getConsumer(topic, groupId);
        }
        // Cyclically consume messages in the current thread. We recommend that you use multiple threads to concurrently consume messages.
        do {
            List<Message> messages = null;
            try {
                // Consume messages in long polling mode. The consumer may pull partitionally ordered messages from multiple partitions. The consumer consumes messages from the same partition in the order in which the messages are sent.
                // A consumer pulls partitionally ordered messages from a partition. If the
```

```

        // A consumer pulls partitionally ordered messages from a partition. If the
        broker does not receive an acknowledgment (ACK) for a message after the message is consumed
        , the consumer consumes the message again.

        // The consumer can consume the next batch of messages from a partition only
        y after all messages that are pulled from the partition in the previous batch are acknowle
        ged to be consumed.

        // In long polling mode, if no message in the topic is available for consum
        ption, the request is suspended on the broker for a specified period of time. If a message
        becomes available for consumption within this period, the broker immediately sends a respon
        se to the consumer. In this example, the period is set to 3 seconds.

        messages = consumer.consumeMessageOrderly(
            3, // The maximum number of messages that can be consumed at a tim
            e. In this example, the value is set to 3. The maximum value that you can specify is 16.
            3 // The length of a long polling period. Unit: seconds. In this
            example, the value is set to 3. The maximum value that you can specify is 30.
        );
    } catch (Throwable e) {
        e.printStackTrace();
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e1) {
            e1.printStackTrace();
        }
    }
    // No messages in the topic are available for consumption.
    if (messages == null || messages.isEmpty()) {
        System.out.println(Thread.currentThread().getName() + ": no new message, co
        ntinue!");
        continue;
    }
    // Specify the message consumption logic.
    System.out.println("Receive " + messages.size() + " messages:");
    for (Message message : messages) {
        System.out.println(message);
        System.out.println("ShardingKey: " + message.getShardingKey() + ", a:" + me
        ssage.getProperties().get("a"));
    }
    // If the broker does not receive an ACK for a message from the consumer before
    the timeout period for a message retry elapses, the broker delivers the message for consump
    tion again.

    // A unique timestamp is specified for the handle of a message each time the me
    ssage is consumed.
    {
        List<String> handles = new ArrayList<String>();
        for (Message message : messages) {
            handles.add(message.getReceiptHandle());
        }
        try {
            consumer.ackMessage(handles);
        } catch (Throwable e) {
            // If the handle of a message times out, the broker cannot receive an A
            CK for the message from the consumer.
            if (e instanceof AckMessageException) {
                AckMessageException errors = (AckMessageException) e;
                System.out.println("Ack message fail. requestId is:" + errors.getRe

```

```

questId() + ", 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(err
orHandle).getErrorMessage());
            }
        }
        continue;
    }
    e.printStackTrace();
}
}
} while (true);
}
}

```

6.3.2.4. Send and consume scheduled messages and delayed messages

This topic provides sample code to show how to use the HTTP client SDK for Java to send and consume scheduled messages and delayed messages.

Background information

- **Delayed message:** The producer sends the message to the Message Queue for Apache RocketMQ server, but does not expect the message to be delivered immediately. Instead, the message is delivered to the consumer for consumption after a certain period of time. This message is a delayed message.
- **Scheduled message:** A producer sends a message to a Message Queue for Apache RocketMQ broker and expects the message to be delivered to a consumer at a specified point in time. This type of message is called a scheduled message.

If an HTTP client SDK is used, the code configurations of scheduled messages are the same as the code configurations of delayed messages. Both types of messages are delivered to consumers after a specific period of time based on the attributes of the messages.

For more information about the message routing feature, see [Scheduled messages and delayed messages](#).

Prerequisites

The following operations are performed:

- Install the SDK for Java. For more information about the message routing feature, see [Prepare the environment](#).
- Create resources that you want to specify in the code. For example, you must create the instance, topic, and group that you want to specify in the code in the Message Queue for Apache RocketMQ console in advance. For more information about the message routing feature, see [Create resources](#).

Send scheduled messages or delayed messages

The following sample code provides an example on how to send scheduled messages or delayed messages:

```
import com.aliyun.mq.http.MQClient;
import com.aliyun.mq.http.MQProducer;
import com.aliyun.mq.http.model.TopicMessage;
import java.util.Date;
public class Producer {
    public static void main(String[] args) {
        MQClient mqClient = new MQClient(
            // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click Instances. On the Instances page, select the name of your instance. Then, view the HTTP endpoint on the Network Management tab.
            "${HTTP_ENDPOINT}",
            // The AccessKey ID that is used for identity verification. You can obtain the AccessKey ID in the Apsara Uni-manager Operations Console.
            "${ACCESS_KEY}",
            // The AccessKey secret that is used for identity verification. You can obtain the AccessKey secret in the Apsara Uni-manager Operations Console.
            "${SECRET_KEY}"
        );
        // The topic to which you want to send messages. The topic is created in the Message Queue for Apache RocketMQ console.
        // Each topic can be used to send and consume messages of a specific type. For example, a topic that is used to send and consume normal messages cannot be used to send and consume messages of other types.
        final String topic = "${TOPIC}";
        // The ID of the instance to which the topic belongs. The instance is created in the Message Queue for Apache RocketMQ console.
        // If the instance has a namespace, specify the ID of the instance. If the instance does not have a namespace, set the instance ID to null or an empty string. You can check whether your instance has a namespace on the Instances page in the RocketMQ console.
        final String instanceId = "${INSTANCE_ID}";
        // Obtain the producer that sends messages to the topic.
        MQProducer producer;
        if (instanceId != null && instanceId != "") {
            producer = mqClient.getProducer(instanceId, topic);
        } else {
            producer = mqClient.getProducer(topic);
        }
        try {
            // Cyclically send four messages.
            for (int i = 0; i < 4; i++) {
                TopicMessage pubMsg;
                pubMsg = new TopicMessage(
                    // The content of the message.
                    "hello mq!".getBytes(),
                    // The tag of the message.
                    "A"
                );
                // The custom property of the message.
                pubMsg.setProperties().put("key", "value");
            }
        }
    }
}
```

```

        pubMsg.getProperties().put("a", String.valueOf(1));
        // The key of the message.
        pubMsg.setMessageKey("MessageKey");
        // The period of time after which the broker delivers the message. In this
        // example, when the broker receives a message, the broker waits for 10 seconds before it deli
        // vers the message to the consumer. Set this parameter to a timestamp in milliseconds.
        // If the producer sends a scheduled message, set the parameter to the tim
        // e interval between the scheduled point in time and the current point in time.
        pubMsg.setStartDeliverTime(System.currentTimeMillis() + 10 * 1000);

        // Send the message in synchronous mode. If no exception is thrown, the messag
        // e is sent.
        TopicMessage pubResultMsg = producer.publishMessage(pubMsg);
        // Send the message in synchronous mode. If no exception is thrown, the messag
        // e is sent.
        System.out.println(new Date() + " Send mq message success. Topic is:" + topic
        + ", msgId is: " + pubResultMsg.getMessageId()
        + ", bodyMD5 is: " + pubResultMsg.getMessageBodyMD5());
    }
    } catch (Throwable e) {
        // Specify the logic that you want to use to resend or persist the message if t
        // he message fails to be sent and needs to be sent again.
        System.out.println(new Date() + " Send mq message failed. Topic is:" + topic);
        e.printStackTrace();
    }
    mqClient.close();
}
}

```

Consume scheduled messages or delayed messages

The following sample code provides an example on how to consume scheduled messages or delayed messages:

```

import com.aliyun.mq.http.MQClient;
import com.aliyun.mq.http.MQConsumer;
import com.aliyun.mq.http.common.AckMessageException;
import com.aliyun.mq.http.model.Message;
import java.util.ArrayList;
import java.util.List;
public class Consumer {
    public static void main(String[] args) {
        MQClient mqClient = new MQClient(
            // The HTTP endpoint to which you want to connect. To obtain the HTTP endpo
            // int, log on to the Message Queue for Apache RocketMQ console. In the left-side navigation p
            // ane, click Instances. On the Instances page, select the name of your instance. Then, view t
            // he HTTP endpoint on the Network Management tab.
            "${HTTP_ENDPOINT}",
            // The AccessKey ID that is used for identity verification. You can obtain
            // the AccessKey ID in the Apsara Uni-manager Operations Console.
            "${ACCESS_KEY}",
            // The AccessKey secret that is used for identity verification. You can obt
            // ain the AccessKey secret in the Apsara Uni-manager Operations Console.
            "${SECRET_KEY}"
        );
    }
}

```

```

    );
    // The topic from which you want to consume messages. The topic is created in the M
    essage Queue for Apache RocketMQ console.
    // Each topic can be used to send and consume messages of a specific type. For exam
    ple, a topic that is used to send and consume normal messages cannot be used to send and co
    nsume messages of other types.
    final String topic = "${TOPIC}";
    // The ID of the group that you created in the Message Queue for Apache RocketMQ co
    nsole.
    final String groupId = "${GROUP_ID}";
    // The ID of the instance to which the topic belongs. The instance is created in th
    e Message Queue for Apache RocketMQ console.
    // If the instance has a namespace, specify the ID of the instance. If the instance
    does not have a namespace, set the instance ID to null or an empty string. You can check wh
    ether your instance has a namespace on the Instances page in the RocketMQ console.
    final String instanceId = "${INSTANCE_ID}";
    final MQConsumer consumer;
    if (instanceId != null && instanceId != "") {
        consumer = mqClient.getConsumer(instanceId, topic, groupId, null);
    } else {
        consumer = mqClient.getConsumer(topic, groupId);
    }
    // Cyclically consume messages in the current thread. We recommend that you use mul
    tiple threads to concurrently consume messages.
    do {
        List<Message> messages = null;
        try {
            // Consume messages in long polling mode.
            // In long polling mode, if no message in the topic is available for consum
            ption, the request is suspended on the broker for a specified period of time. If a message
            becomes available for consumption within this period, the broker immediately sends a respon
            se to the consumer. In this example, the period is set to 3 seconds.
            messages = consumer.consumeMessage(
                3, // The maximum number of messages that can be consumed at a time.
                In this example, the value is set to 3. The maximum value that you can specify is 16.
                3, // The length of a long polling period. Unit: seconds. In this exa
                mple, the value is set to 3. The maximum value that you can specify is 30.
            );
        } catch (Throwable e) {
            e.printStackTrace();
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e1) {
                e1.printStackTrace();
            }
        }
        // No messages in the topic are available for consumption.
        if (messages == null || messages.isEmpty()) {
            System.out.println(Thread.currentThread().getName() + ": no new message, co
            ntinue!");
            continue;
        }
        // Specify the message consumption logic.
        for (Message message : messages) {
            System.out.println("Received message: " + message);
        }
    } while (true);

```

```

        System.out.println("Receive message: " + message);
    }
    // If the broker does not receive an acknowledgment (ACK) for a message from the
    // consumer before the delivery retry interval elapses, the broker delivers the message for
    // consumption again.
    // A unique timestamp is specified for the handle of a message each time the message
    // is consumed.
    {
        List<String> handles = new ArrayList<String>();
        for (Message message : messages) {
            handles.add(message.getReceiptHandle());
        }
        try {
            consumer.ackMessage(handles);
        } catch (Throwable e) {
            // If the handle of a message times out, the broker cannot receive an ACK
            // for the message from the consumer.
            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()
                            + ", ErrorMessage:" + errors.getErrorMessages().get(errorHandle).getErrorMessage());
                    }
                }
                continue;
            }
            e.printStackTrace();
        }
    }
} while (true);
}
}

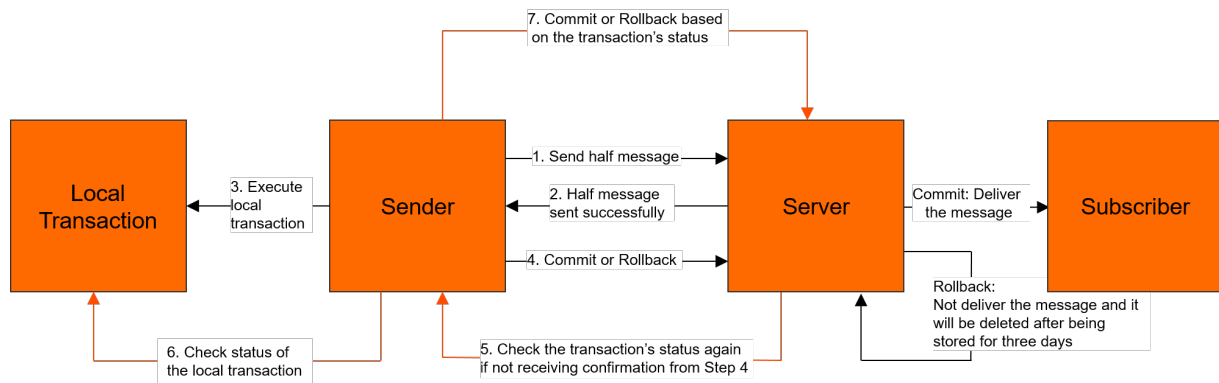
```

6.3.2.5. Send and consume transactional messages

Message Queue for Apache RocketMQ provides a distributed transaction processing feature that is similar to X/Open XA. Message Queue for Apache RocketMQ uses transactional messages to ensure transactional consistency. This topic provides sample code to show how to use the HTTP client SDK for Java to send and consume transactional messages.

Background information

The following figure shows the interaction process of transactional messages.



For more information about the message routing feature, see [Transactional messages](#).

Prerequisites

The following operations are performed:

- Install the SDK for Java. For more information about the message routing feature, see [Prepare the environment](#).
- Create resources that you want to specify in the code. For example, you must create the instance, topic, and group that you want to specify in the code in the Message Queue for Apache RocketMQ console in advance. For more information about the message routing feature, see [Create resources](#).

Send transactional messages

The following sample code provides an example on how to send transactional messages:

```

import com.aliyun.mq.http.MQClient;
import com.aliyun.mq.http.MQTransProducer;
import com.aliyun.mq.http.common.AckMessageException;
import com.aliyun.mq.http.model.Message;
import com.aliyun.mq.http.model.TopicMessage;
import java.util.List;
public class TransProducer {
    static void processCommitRollError(Throwable e) {
        if (e instanceof AckMessageException) {
            AckMessageException errors = (AckMessageException) e;
            System.out.println("Commit/Roll transaction error, 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());
                }
            }
        }
    }

    public static void main(String[] args) throws Throwable {
        MQClient mqClient = new MQClient(
            // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log on to the Message Queue for Apache RocketMQ console. In the left-side navigation p

```

and, log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click Instances. On the Instances page, select the name of your instance. Then, view the HTTP endpoint on the Network Management tab.

```

        "${HTTP_ENDPOINT}",
        // The AccessKey ID that is used for identity verification. You can obtain
the AccessKey ID in the Apsara Uni-manager Operations Console.
        "${ACCESS_KEY}",
        // The AccessKey secret that is used for identity verification. You can obt
ain the AccessKey secret in the Apsara Uni-manager Operations Console.
        "${SECRET_KEY}"
    );
    // The topic to which you want to send messages. The topic is created in the Messag
e Queue for Apache RocketMQ console.
    // Each topic can be used to send and consume messages of a specific type. For exam
ple, a topic that is used to send and consume normal messages cannot be used to send and co
nsume messages of other types.
    final String topic = "${TOPIC}";
    // The ID of the instance to which the topic belongs. The instance is created in th
e Message Queue for Apache RocketMQ console.
    // If the instance has a namespace, specify the ID of the instance. If the instance
does not have a namespace, set the instance ID to null or an empty string. You can check wh
ether your instance has a namespace on the Instances page in the RocketMQ console.
    final String instanceId = "${INSTANCE_ID}";
    // The ID of the group that you created in the Message Queue for Apache RocketMQ co
nsole.
    final String groupId = "${GROUP_ID}";
    final MQTransProducer mqTransProducer = mqClient.getTransProducer(instanceId, topic
, groupId);
    for (int i = 0; i < 4; i++) {
        TopicMessage topicMessage = new TopicMessage();
        topicMessage.setMessageBody("trans_msg");
        topicMessage.setMessageTag("a");
        topicMessage.setMessageKey(String.valueOf(System.currentTimeMillis()));
        // The time interval between the time when the transactional message is sent an
d the start time of the first transaction status check. Unit: seconds. Valid values: 10 to
300.
        // If the message is not committed or rolled back after the first transaction s
tatus check is performed, the broker initiates a request to check the status of the local t
ransaction at an interval of 10 seconds within the next 24 hours.
        topicMessage.setTransCheckImmunityTime(10);
        topicMessage.getProperties().put("a", String.valueOf(i));
        TopicMessage pubResultMsg = null;
        pubResultMsg = mqTransProducer.publishMessage(topicMessage);
        System.out.println("Send---->msgId is: " + pubResultMsg.getMessageId()
            + ", bodyMD5 is: " + pubResultMsg.getMessageBodyMD5()
            + ", Handle: " + pubResultMsg.getReceiptHandle()
        );
        if (pubResultMsg != null && pubResultMsg.getReceiptHandle() != null) {
            if (i == 0) {
                // After the producer sends the transactional message, the broker obtai
ns the handle of the half message that corresponds to the transactional message and commits
or rolls back the transactional message based on the status of the handle.
                try {
                    mqTransProducer.commit(pubResultMsg.getReceiptHandle());
                    System.out.println(String.format("MessageId:%s, commit", pubResultM

```

```

sg.getMessageId());
        } catch (Throwable e) {
            // If the transactional message is not committed or rolled back before the period of time specified by the TransCheckImmunityTime parameter for the handle of the transactional message elapses, the commit or rollback operation fails.
            if (e instanceof AckMessageException) {
                processCommitRollError(e);
                continue;
            }
        }
    }
}

// The client needs a thread or a process to process unacknowledged transactional messages.
// Start a thread to process unacknowledged transactional messages.
Thread t = new Thread(new Runnable() {
    public void run() {
        int count = 0;
        while(true) {
            try {
                if (count == 3) {
                    break;
                }
                List<Message> messages = mqTransProducer.consumeHalfMessage(3, 3);
                if (messages == null) {
                    System.out.println("No Half message!");
                    continue;
                }
                System.out.println(String.format("Half---->MessageId:%s,Properties:%s,Body:%s,Latency:%d",
                    messages.get(0).getMessageId(),
                    messages.get(0).getProperties(),
                    messages.get(0).getMessageBodyString(),
                    System.currentTimeMillis() - messages.get(0).getPublishTime
                ));
                for (Message message : messages) {
                    try {
                        if (Integer.valueOf(message.getProperties().get("a")) == 1) {
                            // Confirm to commit the transactional message.
                            mqTransProducer.commit(message.getReceiptHandle());
                            count++;
                            System.out.println(String.format("MessageId:%s, commit",
                                message.getMessageId()));
                        } else if (Integer.valueOf(message.getProperties().get("a")) == 2
                            && message.getConsumedTimes() > 1) {
                            // Confirm to commit the transactional message.
                            mqTransProducer.commit(message.getReceiptHandle());
                            count++;
                            System.out.println(String.format("MessageId:%s, commit",
                                message.getMessageId()));
                        } else if (Integer.valueOf(message.getProperties().get("a"))

```

```

) == 3) {
    // Confirm to roll back the transactional message.
    mqTransProducer.rollback(message.getReceiptHandle());
    count++;
    System.out.println(String.format("MessageId:%s, rollback", message.getMessageId()));
} else {
    // Do not perform operations. Check the status next time.
    System.out.println(String.format("MessageId:%s, unknown", message.getMessageId()));
}
} catch (Throwable e) {
    // If the transactional message is not committed or rolled back before the timeout period specified by the TransCheckImmunityTime parameter for the handle of the transactional message elapses or before the timeout period specified for the handle of consumeHalfMessage elapses, the commit or rollback operation fails. In this example, the timeout period for the handle of consumeHalfMessage is 10 seconds.
    processCommitRollError(e);
}
}
} catch (Throwable e) {
    System.out.println(e.getMessage());
}
}
}
});
t.start();
t.join();
mqClient.close();
}
}

```

Consume transactional messages

The following sample code provides an example on how to consume transactional messages:

```

import com.aliyun.mq.http.MQClient;
import com.aliyun.mq.http.MQConsumer;
import com.aliyun.mq.http.common.AckMessageException;
import com.aliyun.mq.http.model.Message;
import java.util.ArrayList;
import java.util.List;
public class Consumer {
    public static void main(String[] args) {
        MQClient mqClient = new MQClient(
            // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click Instances. On the Instances page, select the name of your instance. Then, view the HTTP endpoint on the Network Management tab.
            "${HTTP_ENDPOINT}",
            // The AccessKey ID that is used for identity verification. You can obtain the AccessKey ID in the Apsara Uni-manager Operations Console.
            "${ACCESS_KEY}",

```

```

        // The AccessKey secret that is used for identity verification. You can obtain the AccessKey secret in the Apsara Uni-manager Operations Console.
        "${SECRET_KEY}"

    );
    // The topic from which you want to consume messages. The topic is created in the Message Queue for Apache RocketMQ console.
    // Each topic can be used to send and consume messages of a specific type. For example, a topic that is used to send and consume normal messages cannot be used to send and consume messages of other types.
    final String topic = "${TOPIC}";
    // The ID of the group that you created in the Message Queue for Apache RocketMQ console.
    final String groupId = "${GROUP_ID}";
    // The ID of the instance to which the topic belongs. The instance is created in the Message Queue for Apache RocketMQ console.
    // If the instance has a namespace, specify the ID of the instance. If the instance does not have a namespace, set the instance ID to null or an empty string. You can check whether your instance has a namespace on the Instances page in the RocketMQ console.
    final String instanceId = "${INSTANCE_ID}";
    final MQConsumer consumer;
    if (instanceId != null && instanceId != "") {
        consumer = mqClient.getConsumer(instanceId, topic, groupId, null);
    } else {
        consumer = mqClient.getConsumer(topic, groupId);
    }
    // Cyclically consume messages in the current thread. We recommend that you use multiple threads to concurrently consume messages.
    do {
        List<Message> messages = null;
        try {
            // Consume messages in long polling mode.
            // In long polling mode, if no message in the topic is available for consumption, the request is suspended on the broker for a specified period of time. If a message becomes available for consumption within this period, the broker immediately sends a response to the consumer. In this example, the period is set to 3 seconds.
            messages = consumer.consumeMessage(
                3, // The maximum number of messages that can be consumed at a time. In this example, the value is set to 3. The maximum value that you can specify is 16.
                3 // The length of a long polling period. Unit: seconds. In this example, the value is set to 3. The maximum value that you can specify is 30.
            );
        } catch (Throwable e) {
            e.printStackTrace();
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e1) {
                e1.printStackTrace();
            }
        }
        // No messages in the topic are available for consumption.
        if (messages == null || messages.isEmpty()) {
            System.out.println(Thread.currentThread().getName() + ": no new message, continue!");
            continue;
        }
    } while (true);

```

```

    }
    // Specify the message consumption logic.
    for (Message message : messages) {
        System.out.println("Receive message: " + message);
    }
    // If the broker does not receive an acknowledgment (ACK) for a message from the
    // consumer before the delivery retry interval elapses, the broker delivers the message for
    // consumption again.
    // A unique timestamp is specified for the handle of a message each time the message
    // is consumed.
    {
        List<String> handles = new ArrayList<String>();
        for (Message message : messages) {
            handles.add(message.getReceiptHandle());
        }
        try {
            consumer.ackMessage(handles);
        } catch (Throwable e) {
            // If the handle of a message times out, the broker cannot receive an ACK
            // for the message from the consumer.
            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);
}
}

```

6.3.3. Go SDK

6.3.3.1. Prepare the environment

This topic describes how to prepare the environment before you use the HTTP client SDK for Go to send and consume messages.

Environment requirements

Go is installed. For more information, see [Installing Go from source](#).

After Go is installed, you can run the `go version` command to check the version of Go that you installed.

Install the SDK for Go

1. Run the following command to enable Go Modules: For more information about Go Modules, see [Go Modules Reference](#).

```
go env -w GOM11MODULE=on
```

2. Run the following command to configure a Go Modules proxy:

```
go env -w GOPROXY=https://goproxy.cn,direct
```

3. Run the following command to initialize Go Modules and generate the *go.mod* file:

```
go mod init
```

4. Run the following command to install the SDK for Go:

```
go get github.com/aliyunmq/mq-http-go-sdk
```

6.3.3.2. Send and consume normal messages

Normal messages are messages that have no special features in Message Queue for Apache RocketMQ. They are different from featured messages, such as scheduled messages, delayed messages, ordered messages, and transactional messages. This topic provides sample code to show how to use the HTTP client SDK for Go to send and consume normal messages.

Prerequisites

The following operations are performed:

- Install the SDK for Go. For more information about the message routing feature, see [Prepare the environment](#).
- Create resources that you want to specify in the code. For example, you must create the instance, topic, and group that you want to specify in the code in the Message Queue for Apache RocketMQ console in advance. For more information about the message routing feature, see [Create resources](#).

Send normal messages

The following sample code provides an example on how to send normal messages:

```

package main
import (
    "fmt"
    "time"
    "strconv"
    "github.com/aliyunmq/mq-http-go-sdk"
)
func main() {
    // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log on
    // to the console. In the left-side navigation pane, click Instances. On the Instances page,
    // select the name of your instance. Then, view the HTTP endpoint on the Network Management ta
    // b.
    endpoint := "${HTTP_ENDPOINT}"
    // The AccessKey ID that is used for identity verification. You can obtain the AccessKe
    // y ID in the Apsara Uni-manager Operations Console.
    accessKey := "${ACCESS_KEY}"
    // The AccessKey secret that is used for identity verification. You can obtain the Acce
    // ssKey secret in the Apsara Uni-manager Operations Console.
    secretKey := "${SECRET_KEY}"
    // The topic to which you want to send messages. The topic is created in the Message Qu
    // eue for Apache RocketMQ console.
    topic := "${TOPIC}"
    // The ID of the instance to which the topic belongs. The instance is created in the Me
    // ssage Queue for Apache RocketMQ console.
    // If the instance has a namespace, specify the ID of the instance. If the instance doe
    // s not have a namespace, set the instance ID to null or an empty string. You can check wheth
    // er your instance has a namespace on the Instances page in the RocketMQ console.
    instanceId := "${INSTANCE_ID}"
    client := mq_http_sdk.NewAliyunMQClient(endpoint, accessKey, secretKey, "")
    mqProducer := client.GetProducer(instanceId, topic)
    // Cyclically send four messages.
    for i := 0; i < 4; i++ {
        var msg mq_http_sdk.PublishMessageRequest
        msg = mq_http_sdk.PublishMessageRequest{
            MessageBody: "hello mq!",           // The content of the message.
            MessageTag: "",                     // The tag of the message.
            Properties: map[string]string{}, // The properties of the message.
        }
        // The key of the message.
        msg.MessageKey = "MessageKey"
        // The custom property of the message.
        msg.Properties["a"] = strconv.Itoa(i)
        ret, err := mqProducer.PublishMessage(msg)
        if err != nil {
            fmt.Println(err)
            return
        } else {
            fmt.Printf("Publish ---->\n\tMessageId:%s, BodyMD5:%s, \n", ret.MessageId, ret.
            MessageBodyMD5)
        }
        time.Sleep(time.Duration(100) * time.Millisecond)
    }
}

```

Consume normal messages

The following sample code provides an example on how to consume normal messages:

```
package main
import (
    "fmt"
    "github.com/gogap/errors"
    "strings"
    "time"
    "github.com/aliyunmq/mq-http-go-sdk"
)

func main() {
    // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log on
    // to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click I
    // nstances. On the Instances page, select the name of your instance. Then, view the HTTP endp
    // oint on the Network Management tab.
    endpoint := "${HTTP_ENDPOINT}"

    // The AccessKey ID that is used for identity verification. You can obtain the AccessKe
    // y ID in the Apsara Uni-manager Operations Console.
    accessKey := "${ACCESS_KEY}"

    // The AccessKey secret that is used for identity verification. You can obtain the Acce
    // ssKey secret in the Apsara Uni-manager Operations Console.
    secretKey := "${SECRET_KEY}"

    // The topic from which you want to consume messages. The topic is created in the Messa
    // ge Queue for Apache RocketMQ console.
    // Each topic can be used to send and consume messages of a specific type. For example,
    // a topic that is used to send and consume normal messages cannot be used to send and consume
    // messages of other types.
    topic := "${TOPIC}"

    // The ID of the instance to which the topic belongs. The instance is created in the Me
    // ssage Queue for Apache RocketMQ console.
    // If the instance has a namespace, specify the ID of the instance. If the instance doe
    // s not have a namespace, set the instance ID to null or an empty string. You can check wheth
    // er your instance has a namespace on the Instances page in the RocketMQ console.
    instanceId := "${INSTANCE_ID}"

    // The ID of the group that you created in the Message Queue for Apache RocketMQ consol
    // e.
    groupId := "${GROUP_ID}"

    client := mq_http_sdk.NewAliyunMQClient(endpoint, accessKey, secretKey, "")
    mqConsumer := client.GetConsumer(instanceId, topic, groupId, "")
    for {
        endChan := make(chan int)
        respChan := make(chan mq_http_sdk.ConsumeMessageResponse)
        errChan := make(chan error)
        go func() {
            select {
            case resp := <-respChan:
                {
                    // Specify the message consumption logic.
                    var handles []string
                    fmt.Printf("Consume %d messages---->\n", len(resp.Messages))
                    for _, v := range resp.Messages {
                        handles = append(handles, v.ReceiptHandle)
                        fmt.Printf("%+tMessageID: %s PublishTime: %d MessageTag: %s\n",

```

```

        fmt.Println("\tMessageID: %s, PublishTime: %d, MessageTag: %s\n" +
            "\tConsumedTimes: %d, FirstConsumeTime: %d, NextConsumeTime: %d\n" +
            "\tBody: %s\n" +
            "\tProps: %s\n",
            v.MessageId, v.PublishTime, v.MessageTag, v.ConsumedTimes,
            v.FirstConsumeTime, v.NextConsumeTime, v.MessageBody, v.Properties)
    }
    // If the broker does not receive an acknowledgment (ACK) for a message
    from the consumer before the period of time specified by the NextConsumeTime parameter elapses,
    the broker delivers the message for consumption again.
    // A unique timestamp is specified for the handle of a message each time the message is consumed.
    ackerr := mqConsumer.AckMessage(handles)
    if ackerr != nil {
        // If the handle of a message times out, the broker cannot receive
        an ACK for the message from the consumer.
        fmt.Println(ackerr)
        if errAckItems, ok := ackerr.(errors.ErrCode).Context()["Detail"].([]mq_http_sdk.ErrAckItem); ok {
            for _, errAckItem := range errAckItems {
                fmt.Printf("\tErrorHandle:%s, ErrorCode:%s, ErrorMessage:%s\n",
                    errAckItem.ErrorHandle, errAckItem.ErrorCode, errAckItem.ErrorMessage)
            }
        } else {
            fmt.Println("ack err =", ackerr)
        }
        time.Sleep(time.Duration(3) * time.Second)
    } else {
        fmt.Printf("Ack ---->\n\t%s\n", handles)
    }
    endChan <- 1
}
case err := <-errChan:
{
    // No messages in the topic are available for consumption.
    if strings.Contains(err.(errors.ErrCode).Error(), "MessageNotExist") {
        fmt.Println("\nNo new message, continue!")
    } else {
        fmt.Println(err)
        time.Sleep(time.Duration(3) * time.Second)
    }
    endChan <- 1
}
case <-time.After(35 * time.Second):
{
    fmt.Println("Timeout of consumer message ??")
    endChan <- 1
}
}
}()
// In long polling mode, the default network timeout period is 35 seconds.
// In long polling mode, if no message in the topic is available for consumption, the

```

```

// In long polling mode, if no message in the topic is available for consumption, the
// request is suspended on the broker for a specified period of time. If a message becomes
// available for consumption within this period, the broker immediately sends a response to the
// consumer. In this example, the period is set to 3 seconds.
mqConsumer.ConsumeMessage(respChan, errChan,
    3, // The maximum number of messages that can be consumed at a time. In this example,
    // the value is set to 3. The maximum value that you can specify is 16.
    3, // The length of a long polling period. Unit: seconds. In this example, the
    // value is set to 3. The maximum value that you can specify is 30.
)
<-endChan
}
}

```

6.3.3.3. Send and consume ordered messages

Ordered messages are a type of message that is published and consumed in a strict order. Ordered messages in Message Queue for Apache RocketMQ are also known as first-in-first-out (FIFO) messages. This topic provides sample code to show how to use the HTTP client SDK for Go to send and consume ordered messages.

Background information

Ordered messages are classified into the following types:

- Globally ordered message: All messages in a specified topic are published and consumed in first-in-first-out (FIFO) order.
- Partitionally ordered message: All messages in a specified topic are distributed to different partitions by using shard keys. The messages in each partition are published and consumed in FIFO order. A Sharding Key is a key field that is used for ordered messages to identify different partitions. The Sharding Key is different from the key of a normal message.

For more information about the message routing feature, see [Ordered messages](#).

Prerequisites

The following operations are performed:

- Install the SDK for Go. For more information about the message routing feature, see [Prepare the environment](#).
- Create resources that you want to specify in the code. For example, you must create the instance, topic, and group that you want to specify in the code in the Message Queue for Apache RocketMQ console in advance. For more information about the message routing feature, see [Create resources](#).

Send ordered messages

The following sample code provides an example on how to send ordered messages:

```

package main
import (
    "fmt"
    "time"
    "strconv"
    "github.com/aliyunmq/mq-http-go-sdk"
)

```

```

func main() {
    // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log on
    // to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click I
    // nstances. On the Instances page, select the name of your instance. Then, view the HTTP endp
    // oint on the Network Management tab.
    endpoint := "${HTTP_ENDPOINT}"

    // The AccessKey ID that is used for identity verification. You can obtain the AccessKe
    // y ID in the Apsara Uni-manager Operations Console.
    accessKey := "${ACCESS_KEY}"

    // The AccessKey secret that is used for identity verification. You can obtain the Acce
    // ssKey secret in the Apsara Uni-manager Operations Console.
    secretKey := "${SECRET_KEY}"

    // The topic to which you want to send messages. The topic is created in the Message Qu
    // eue for Apache RocketMQ console.
    topic := "${TOPIC}"

    // The ID of the instance to which the topic belongs. The instance is created in the Me
    // ssage Queue for Apache RocketMQ console.
    // If the instance has a namespace, specify the ID of the instance. If the instance doe
    // s not have a namespace, set the instance ID to null or an empty string. You can check wheth
    // er your instance has a namespace on the Instances page in the RocketMQ console.
    instanceId := "${INSTANCE_ID}"

    client := mq_http_sdk.NewAliyunMQClient(endpoint, accessKey, secretKey, "")
    mqProducer := client.GetProducer(instanceId, topic)
    // Cyclically send eight messages.
    for i := 0; i < 8; i++ {
        msg := mq_http_sdk.PublishMessageRequest{
            MessageBody: "hello mq!",           // The content of the message.
            MessageTag: "",                     // The tag of the message.
            Properties: map[string]string{}, // The properties of the message.
        }
        // The key of the message.
        msg.MessageKey = "MessageKey"
        // The custom property of the message.
        msg.Properties["a"] = strconv.Itoa(i)
        // The shard key that is used to distribute ordered messages to a specific partiti
        // o. Shard keys can be used to identify different partitions. A shard key is different from a
        // message key.
        msg.ShardingKey = strconv.Itoa(i % 2)
        ret, err := mqProducer.PublishMessage(msg)
        if err != nil {
            fmt.Println(err)
            return
        } else {
            fmt.Printf("Publish ---->\n\tMessageId:%s, BodyMD5:%s, \n", ret.MessageId, ret.
            MessageBodyMD5)
        }
        time.Sleep(time.Duration(100) * time.Millisecond)
    }
}

```

Consume ordered messages

The following sample code provides an example on how to consume ordered messages:

```

package main
import (
    "fmt"
    "github.com/gogap/errors"
    "strings"
    "time"
    "github.com/aliyunmq/mq-http-go-sdk"
)

func main() {
    // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log on
    // to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click I
    // nstances. On the Instances page, select the name of your instance. Then, view the HTTP endp
    // oint on the Network Management tab.
    endpoint := "${HTTP_ENDPOINT}"
    // The AccessKey ID that is used for identity verification. You can obtain the AccessKe
    // y ID in the Apsara Uni-manager Operations Console.
    accessKey := "${ACCESS_KEY}"
    // The AccessKey secret that is used for identity verification. You can obtain the Acce
    // ssKey secret in the Apsara Uni-manager Operations Console.
    secretKey := "${SECRET_KEY}"
    // The topic from which you want to consume messages. The topic is created in the Messa
    // ge Queue for Apache RocketMQ console.
    topic := "${TOPIC}"
    // The ID of the instance to which the topic belongs. The instance is created in the Me
    // ssage Queue for Apache RocketMQ console.
    // If the instance has a namespace, specify the ID of the instance. If the instance doe
    // s not have a namespace, set the instance ID to null or an empty string. You can check wheth
    // er your instance has a namespace on the Instances page in the RocketMQ console.
    instanceId := "${INSTANCE_ID}"
    // The ID of the group that you created in the Message Queue for Apache RocketMQ consol
    // e.
    groupId := "${GROUP_ID}"
    client := mq_http_sdk.NewAliyunMQClient(endpoint, accessKey, secretKey, "")
    mqConsumer := client.GetConsumer(instanceId, topic, groupId, "")
    for {
        endChan := make(chan int)
        respChan := make(chan mq_http_sdk.ConsumeMessageResponse)
        errChan := make(chan error)
        go func() {
            select {
            case resp := <-respChan:
                {
                    // Specify the message consumption logic.
                    var handles []string
                    fmt.Printf("Consume %d messages---->\n", len(resp.Messages))
                    for _, v := range resp.Messages {
                        handles = append(handles, v.ReceiptHandle)
                        fmt.Printf("\tMessageID: %s, PublishTime: %d, MessageTag: %s\n"+
                            "\tConsumedTimes: %d, FirstConsumeTime: %d, NextConsumeTime: %d
\n"+
                            "\tBody: %s\n"+
                            "\tProps: %s\n"+
                            "\tShardingKey: %s\n",
                                v.MessageId, v.PublishTime, v.MessageTag, v.ConsumedTimes,

```

```

        v.FirstConsumeTime, v.NextConsumeTime, v.MessageBody, v.Properties, v.ShardingKey)
    }
    // If the broker does not receive an acknowledgment (ACK) for a message
    from the consumer before the period of time specified by the NextConsumeTime parameter elapses,
    the broker delivers the message for consumption again.
    // A unique timestamp is specified for the handle of a message each time the message is consumed.
    ackerr := mqConsumer.AckMessage(handles)
    if ackerr != nil {
        // If the handle of a message times out, the broker cannot receive
        an ACK for the message from the consumer.
        fmt.Println(ackerr)
        if errAckItems, ok := ackerr.(errors.ErrCode).Context()["Detail"].([]mq_http_sdk.ErrAckItem); ok {
            for _, errAckItem := range errAckItems {
                fmt.Printf("\tErrorHandle:%s, ErrorCode:%s, ErrorMsg:%s\n",
                    errAckItem.ErrorHandle, errAckItem.ErrorCode, errAckItem.ErrorMessage)
            }
        } else {
            fmt.Println("ack err =", ackerr)
        }
        time.Sleep(time.Duration(3) * time.Second)
    } else {
        fmt.Printf("Ack ---->\n\t%s\n", handles)
    }
    endChan <- 1
}
case err := <-errChan:
{
    // No messages in the topic are available for consumption.
    if strings.Contains(err.(errors.ErrCode).Error(), "MessageNotExist") {
        fmt.Println("\nNo new message, continue!")
    } else {
        fmt.Println(err)
        time.Sleep(time.Duration(3) * time.Second)
    }
    endChan <- 1
}
case <-time.After(35 * time.Second):
{
    fmt.Println("Timeout of consumer message ??")
    endChan <- 1
}
}
}()

// The consumer may pull partitionally ordered messages from multiple partitions.
The consumer consumes the messages in each partition in the order in which the messages are sent.

// A consumer pulls partitionally ordered messages from a partition. If the broker does not receive an ACK for a message after the message is consumed, the consumer consumes the message again.

// The consumer can consume the next batch of messages from a partition only

```

```
y after all messages that are pulled from the partition in the previous batch are acknowledged to be consumed.
    // In long polling mode, the default network timeout period is 35 seconds.
    // In long polling mode, if no message in the topic is available for consumption, the request is suspended on the broker for a specified period of time. If a message becomes available for consumption within this period, the broker immediately sends a response to the consumer. In this example, the period is set to 3 seconds.
    mqConsumer.ConsumeMessageOrderly(respChan, errChan,
        3, // The maximum number of messages that can be consumed at a time. In this example, the value is set to 3. The maximum value that you can specify is 16.
        3, // The length of a long polling period. Unit: seconds. In this example, the value is set to 3. The maximum value that you can specify is 30.
    )
    <-endChan
}
}
```

6.3.3.4. Send and consume scheduled messages and delayed messages

This topic provides sample code to show how to use the HTTP client SDK for Go to send and consume scheduled messages and delayed messages.

Background information

- **Delayed message:** The producer sends the message to the Message Queue for Apache RocketMQ server, but does not expect the message to be delivered immediately. Instead, the message is delivered to the consumer for consumption after a certain period of time. This message is a delayed message.
- **Scheduled message:** A producer sends a message to a Message Queue for Apache RocketMQ broker and expects the message to be delivered to a consumer at a specified point in time. This type of message is called a scheduled message.

If an HTTP client SDK is used, the code configurations of scheduled messages are the same as the code configurations of delayed messages. Both types of messages are delivered to consumers after a specific period of time based on the attributes of the messages.

For more information about the message routing feature, see [Scheduled messages and delayed messages](#).

Prerequisites

The following operations are performed:

- Install the SDK for Go. For more information about the message routing feature, see [Prepare the environment](#).
- Create resources that you want to specify in the code. For example, you must create the instance, topic, and group that you want to specify in the code in the Message Queue for Apache RocketMQ console in advance. For more information about the message routing feature, see [Create resources](#).

Send scheduled messages or delayed messages

The following sample code provides an example on how to send scheduled messages or delayed messages:

```
package main
import (
    "fmt"
    "time"
    "strconv"
    "github.com/aliyunmq/mq-http-go-sdk"
)
func main() {
    // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log on
    // to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click I
    // nstances. On the Instances page, select the name of your instance. Then, view the HTTP endp
    // oint on the Network Management tab.
    endpoint := "${HTTP_ENDPOINT}"
    // The AccessKey ID that is used for identity verification. You can obtain the AccessKe
    // y ID in the Apsara Uni-manager Operations Console.
    accessKey := "${ACCESS_KEY}"
    // The AccessKey secret that is used for identity verification. You can obtain the Acce
    // ssKey secret in the Apsara Uni-manager Operations Console.
    secretKey := "${SECRET_KEY}"
    // The topic to which you want to send messages. The topic is created in the Message Qu
    // eue for Apache RocketMQ console.
    topic := "${TOPIC}"
    // The ID of the instance to which the topic belongs. The instance is created in the Me
    // ssage Queue for Apache RocketMQ console.
    // If the instance has a namespace, specify the ID of the instance. If the instance doe
    // s not have a namespace, set the instance ID to null or an empty string. You can check wheth
    // er your instance has a namespace on the Instances page in the RocketMQ console.
    instanceId := "${INSTANCE_ID}"
    client := mq_http_sdk.NewAliyunMQClient(endpoint, accessKey, secretKey, "")
    mqProducer := client.GetProducer(instanceId, topic)
    // Cyclically send four messages.
    for i := 0; i < 4; i++ {
        var msg mq_http_sdk.PublishMessageRequest
        msg = mq_http_sdk.PublishMessageRequest{
            MessageBody: "hello mq!",           // The content of the message.
            MessageTag: "",                     // The tag of the message.
            Properties: map[string]string{}, // The properties of the message.
        }
        // The key of the message.
        msg.MessageKey = "MessageKey"
        // The custom property of the message.
        msg.Properties["a"] = strconv.Itoa(i)
        // The period of time after which the broker delivers the message. In this exam
        // ple, when the broker receives a message, the broker waits for 10 seconds before it delivers
        // the message to the consumer. Set this parameter to a timestamp in milliseconds.
        // If the producer sends a scheduled message, set the parameter to the time int
        // erval between the scheduled point in time and the current point in time.
        msg.StartDeliverTime = time.Now().UTC().Unix() * 1000 + 10 * 1000
        ret, err := mqProducer.PublishMessage(msg)
        if err != nil {
            fmt.Println(err)
        }
    }
}
```

```

        return
    } else {
        fmt.Printf("Publish ---->\n\tMessageId:%s, BodyMD5:%s, \n", ret.MessageId, ret.
MessageBodyMD5)
    }
    time.Sleep(time.Duration(100) * time.Millisecond)
}
}
}

```

Consume scheduled messages or delayed messages

The following sample code provides an example on how to consume scheduled messages or delayed messages:

```

package main
import (
    "fmt"
    "github.com/gogap/errors"
    "strings"
    "time"
    "github.com/aliyunmq/mq-http-go-sdk"
)
func main() {
    // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log on
    // to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click I
    // nstances. On the Instances page, select the name of your instance. Then, view the HTTP endp
    // oint on the Network Management tab.
    endpoint := "${HTTP_ENDPOINT}"
    // The AccessKey ID that is used for identity verification. You can obtain the AccessKe
    // y ID in the Apsara Uni-manager Operations Console.
    accessKey := "${ACCESS_KEY}"
    // The AccessKey secret that is used for identity verification. You can obtain the Acce
    // ssKey secret in the Apsara Uni-manager Operations Console.
    secretKey := "${SECRET_KEY}"
    // The topic from which you want to consume messages. The topic is created in the Messa
    // ge Queue for Apache RocketMQ console.
    // Each topic can be used to send and consume messages of a specific type. For example,
    // a topic that is used to send and consume normal messages cannot be used to send and consume
    // messages of other types.
    topic := "${TOPIC}"
    // The ID of the instance to which the topic belongs. The instance is created in the Me
    // ssage Queue for Apache RocketMQ console.
    // If the instance has a namespace, specify the ID of the instance. If the instance doe
    // s not have a namespace, set the instance ID to null or an empty string. You can check wheth
    // er your instance has a namespace on the Instances page in the RocketMQ console.
    instanceId := "${INSTANCE_ID}"
    // The ID of the group that you created in the Message Queue for Apache RocketMQ consol
    // e.
    groupId := "${GROUP_ID}"
    client := mq_http_sdk.NewAliyunMQClient(endpoint, accessKey, secretKey, "")
    mqConsumer := client.GetConsumer(instanceId, topic, groupId, "")
    for {
        endChan := make(chan int)
        respChan := make(chan mq_http_sdk.ConsumeMessageResponse)
    }
}

```

```

errChan := make(chan error)
go func() {
    select {
    case resp := <-respChan:
        {
            // Specify the message consumption logic.
            var handles []string
            fmt.Printf("Consume %d messages---->\n", len(resp.Messages))
            for _, v := range resp.Messages {
                handles = append(handles, v.ReceiptHandle)
                fmt.Printf("\tMessageID: %s, PublishTime: %d, MessageTag: %s\n"+
                    "\tConsumedTimes: %d, FirstConsumeTime: %d, NextConsumeTime: %d
\n"+
                    "\tBody: %s\n"+
                    "\tProps: %s\n",
                    v.MessageId, v.PublishTime, v.MessageTag, v.ConsumedTimes,
                    v.FirstConsumeTime, v.NextConsumeTime, v.MessageBody, v.Properties)
            }
            // If the broker does not receive an acknowledgment (ACK) for a message
            from the consumer before the period of time specified by the NextConsumeTime parameter elapses,
            the broker delivers the message for consumption again.
            // A unique timestamp is specified for the handle of a message each time
            the message is consumed.
            ackerr := mqConsumer.AckMessage(handles)
            if ackerr != nil {
                // If the handle of a message times out, the broker cannot receive
                an ACK for the message from the consumer.
                fmt.Println(ackerr)
                if errAckItems, ok := ackerr.(errors.ErrCode).Context()["Detail"].([]mq_http_sdk.ErrAckItem); ok {
                    for _, errAckItem := range errAckItems {
                        fmt.Printf("\tErrorHandle:%s, ErrorCode:%s, ErrorMessage:%s\n",
                            errAckItem.ErrorHandle, errAckItem.ErrorCode, errAckItem.ErrorMessage)
                    }
                } else {
                    fmt.Println("ack err =", ackerr)
                }
                time.Sleep(time.Duration(3) * time.Second)
            } else {
                fmt.Printf("Ack ---->\n\t%s\n", handles)
            }
            endChan <- 1
        }
    case err := <-errChan:
        {
            // No messages in the topic are available for consumption.
            if strings.Contains(err.(errors.ErrCode).Error(), "MessageNotExist") {
                fmt.Println("\nNo new message, continue!")
            } else {
                fmt.Println(err)
                time.Sleep(time.Duration(3) * time.Second)
            }
        }
    }
}

```

```

        endChan <- 1
    }
    case <-time.After(35 * time.Second):
    {
        fmt.Println("Timeout of consumer message ??")
        endChan <- 1
    }
}
}()
// In long polling mode, the default network timeout period is 35 seconds.
// In long polling mode, if no message in the topic is available for consumption, the request is suspended on the broker for a specified period of time. If a message becomes available for consumption within this period, the broker immediately sends a response to the consumer. In this example, the period is set to 3 seconds.
mqConsumer.ConsumeMessage(respChan, errChan,
    3, // The maximum number of messages that can be consumed at a time. In this example, the value is set to 3. The maximum value that you can specify is 16.
    3, // The length of a long polling period. Unit: seconds. In this example, the value is set to 3. The maximum value that you can specify is 30.
)
<-endChan
}
}

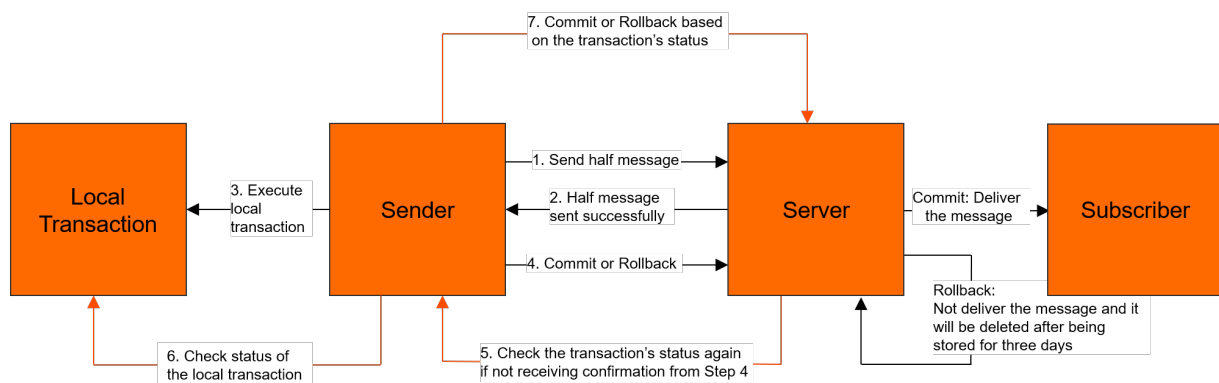
```

6.3.3.5. Send and consume transactional messages

Message Queue for Apache RocketMQ provides a distributed transaction processing feature that is similar to X/Open XA. Message Queue for Apache RocketMQ uses transactional messages to ensure transactional consistency. This topic provides sample code to show how to use the HTTP client SDK for Go to send and consume transactional messages.

Background information

The following figure shows the interaction process of transactional messages.



For more information about the message routing feature, see [Transactional messages](#).

Prerequisites

The following operations are performed:

- Install the SDK for Go. For more information about the message routing feature, see [Prepare the](#)

environment.

- Create resources that you want to specify in the code. For example, you must create the instance, topic, and group that you want to specify in the code in the Message Queue for Apache RocketMQ console in advance. For more information about the message routing feature, see [Create resources](#).

Send transactional messages

The following sample code provides an example on how to send transactional messages:

```
package main
import (
    "fmt"
    "github.com/gogap/errors"
    "strconv"
    "strings"
    "time"
    "github.com/aliyunmq/mq-http-go-sdk"
)
var loopCount = 0
func ProcessError(err error) {
    // If a transactional message is not committed or rolled back before the timeout period
    // specified by the TransCheckImmunityTime parameter for the handle of the transactional message
    // elapses or before the timeout period specified for the handle of consumeHalfMessage elapses,
    // the commit or rollback operation fails. In this example, the timeout period for the handle
    // of consumeHalfMessage is 10 seconds.
    if err == nil {
        return
    }
    fmt.Println(err)
    for _, errAckItem := range err.(errors.ErrCode).Context()["Detail"].([]mq_http_sdk.ErrAckItem) {
        fmt.Printf("\tErrorHandle:%s, ErrorCode:%s, ErrorMsg:%s\n",
            errAckItem.ErrorHandle, errAckItem.ErrorCode, errAckItem.ErrorMsg)
    }
}
func ConsumeHalfMsg(mqTransProducer *mq_http_sdk.MQTransProducer) {
    for {
        if loopCount >= 10 {
            return
        }
        loopCount++
        endChan := make(chan int)
        respChan := make(chan mq_http_sdk.ConsumeMessageResponse)
        errChan := make(chan error)
        go func() {
            select {
            case resp := <-respChan:
                {
                    // Specify the business processing logic.
                    var handles []string
                    fmt.Printf("Consume %d messages---->\n", len(resp.Messages))
                    for _, v := range resp.Messages {
                        handles = append(handles, v.ReceiptHandle)
                        fmt.Printf("\tMessageID: %s, PublishTime: %d, MessageTag: %s\n"+
```

```

        "\tConsumedTimes: %d, FirstConsumeTime: %d, NextConsumeTime: %d\n"
        "\tBody: %s\n"+
        "\tProperties:%s, Key:%s, Timer:%d, Trans:%d\n",
        v.MessageId, v.PublishTime, v.MessageTag, v.ConsumedTimes,
        v.FirstConsumeTime, v.NextConsumeTime, v.MessageBody,
        v.Properties, v.MessageKey, v.StartDeliverTime, v.TransCheckImm
unityTime)

    a, _ := strconv.Atoi(v.Properties["a"])
    var comRollErr error
    if a == 1 {
        // Confirm to commit the transactional message.
        comRollErr = (*mqTransProducer).Commit(v.ReceiptHandle)
        fmt.Println("Commit----->")
    } else if a == 2 && v.ConsumedTimes > 1 {
        // Confirm to commit the transactional message.
        comRollErr = (*mqTransProducer).Commit(v.ReceiptHandle)
        fmt.Println("Commit----->")
    } else if a == 3 {
        // Confirm to roll back the transactional message.
        comRollErr = (*mqTransProducer).Rollback(v.ReceiptHandle)
        fmt.Println("Rollback----->")
    } else {
        // Do not perform operations. Check the status next time.
        fmt.Println("Unknown----->")
    }
    ProcessError(comRollErr)
}
endChan <- 1
}
case err := <-errChan:
{
    // No messages in the topic are available for consumption.
    if strings.Contains(err.(errors.ErrCode).Error(), "MessageNotExist") {
        fmt.Println("\nNo new message, continue!")
    } else {
        fmt.Println(err)
        time.Sleep(time.Duration(3) * time.Second)
    }
    endChan <- 1
}
case <-time.After(35 * time.Second):
{
    fmt.Println("Timeout of consumer message ??")
    return
}
}
}()
// Check the status of half messages in long polling mode.
// In long polling mode, if no message in the topic is available for consumption, t
he request is suspended on the broker for a specified period of time. If a message becomes
available for consumption within this period, the broker immediately sends a response to th
e consumer. In this example, the period is set to 3 seconds.
(*mqTransProducer).ConsumeHalfMessage(respChan, errChan,
    3, // The maximum number of messages that can be consumed at a time. In this ex
ample, the value is set to 3. The maximum value that you can specify is 16

```

```

ample, the value is set to 3. The maximum value that you can specify is 10.
    3, // The length of a long polling period. Unit: seconds. In this example, the
value is set to 3. The maximum value that you can specify is 30.
    )
    <-endChan
}
}
func main() {
    // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log on
to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click I
nstances. On the Instances page, select the name of your instance. Then, view the HTTP endp
oint on the Network Management tab.
    endpoint := "${HTTP_ENDPOINT}"
    // The AccessKey ID that is used for identity verification. You can obtain the AccessKe
y ID in the Apsara Uni-manager Operations Console.
    accessKey := "${ACCESS_KEY}"
    // The AccessKey secret that is used for identity verification. You can obtain the Acce
ssKey secret in the Apsara Uni-manager Operations Console.
    secretKey := "${SECRET_KEY}"
    // The topic to which you want to send messages. The topic is created in the Message Qu
eue for Apache RocketMQ console.
    // Each topic can be used to send and consume messages of a specific type. For example,
a topic that is used to send and consume normal messages cannot be used to send and consume
messages of other types.
    topic := "${TOPIC}"
    // The ID of the instance to which the topic belongs. The instance is created in the Me
ssage Queue for Apache RocketMQ console.
    // If the instance has a namespace, specify the ID of the instance. If the instance doe
s not have a namespace, set the instance ID to null or an empty string. You can check wheth
er your instance has a namespace on the Instances page in the RocketMQ console.
    instanceId := "${INSTANCE_ID}"
    // The ID of the group that you created in the Message Queue for Apache RocketMQ consol
e.
    groupId := "${GROUP_ID}"
    client := mq_http_sdk.NewAliyunMQClient(endpoint, accessKey, secretKey, "")
    mqTransProducer := client.GetTransProducer(instanceId, topic, groupId)
    // The client needs a thread or a process to process unacknowledged transactional messa
ges.
    // Start a goroutine to process unacknowledged transactional messages.
    go ConsumeHalfMsg(&mqTransProducer)
    // Send four transactional messages. Commit one message after the message is sent. Chec
k the status of the half messages that correspond to the other three transactional messages
after the three messages are sent.
    for i := 0; i < 4; i++ {
        msg := mq_http_sdk.PublishMessageRequest{
            MessageBody:"I am transaction msg!",
            Properties: map[string]string{"a":strconv.Itoa(i)},
        }
        // The time interval between the time when the transactional message is sent and th
e start time of the first transaction status check. Unit: seconds. Valid values: 10 to 300.

        // If the message is not committed or rolled back after the first transaction statu
s check is performed, the broker initiates a request to check the status of the local trans
action at an interval of 10 seconds within the next 24 hours.
        msg.TransCheckImmunityTime = 10

```

```

    msg := mqTransProducer.PublishMessage(msg)
    resp, pubErr := mqTransProducer.PublishMessage(msg)
    if pubErr != nil {
        fmt.Println(pubErr)
        return
    }
    fmt.Printf("Publish ---->\n\tMessageId:%s, BodyMD5:%s, Handle:%s\n",
        resp.MessageId, resp.MessageBodyMD5, resp.ReceiptHandle)
    if i == 0 {
        // After the producer sends the transactional message, the broker obtains the handle of the half message that corresponds to the transactional message and commits or rolls back the transactional message based on the status of the handle.
        ackErr := mqTransProducer.Commit(resp.ReceiptHandle)
        fmt.Println("Commit----->")
        ProcessError(ackErr)
    }
}
for ; loopCount < 10 ; {
    time.Sleep(time.Duration(1) * time.Second)
}
}

```

Consume transactional messages

The following sample code provides an example on how to consume transactional messages:

```

package main
import (
    "fmt"
    "github.com/gogap/errors"
    "strings"
    "time"
    "github.com/aliyunmq/mq-http-go-sdk"
)
func main() {
    // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click Instances. On the Instances page, select the name of your instance. Then, view the HTTP endpoint on the Network Management tab.
    endpoint := "${HTTP_ENDPOINT}"
    // The AccessKey ID that is used for identity verification. You can obtain the AccessKey ID in the Apsara Uni-manager Operations Console.
    accessKey := "${ACCESS_KEY}"
    // The AccessKey secret that is used for identity verification. You can obtain the AccessKey secret in the Apsara Uni-manager Operations Console.
    secretKey := "${SECRET_KEY}"
    // The topic from which you want to consume messages. The topic is created in the Message Queue for Apache RocketMQ console.
    // Each topic can be used to send and consume messages of a specific type. For example, a topic that is used to send and consume normal messages cannot be used to send and consume messages of other types.
    topic := "${TOPIC}"
    // The ID of the instance to which the topic belongs. The instance is created in the Message Queue for Apache RocketMQ console.
    // If the instance has a namespace, specify the ID of the instance. If the instance does

```

```

// If the instance has a namespace, specify the ID of the instance. If the instance does
not have a namespace, set the instance ID to null or an empty string. You can check whether
your instance has a namespace on the Instances page in the RocketMQ console.
instanceId := "${INSTANCE_ID}"
// The ID of the group that you created in the Message Queue for Apache RocketMQ console.
groupId := "${GROUP_ID}"
client := mq_http_sdk.NewAliyunMQClient(endpoint, accessKey, secretKey, "")
mqConsumer := client.GetConsumer(instanceId, topic, groupId, "")
for {
    endChan := make(chan int)
    respChan := make(chan mq_http_sdk.ConsumeMessageResponse)
    errChan := make(chan error)
    go func() {
        select {
        case resp := <-respChan:
            {
                // Specify the message consumption logic.
                var handles []string
                fmt.Printf("Consume %d messages---->\n", len(resp.Messages))
                for _, v := range resp.Messages {
                    handles = append(handles, v.ReceiptHandle)
                    fmt.Printf("\tMessageID: %s, PublishTime: %d, MessageTag: %s\n"+
                        "\tConsumedTimes: %d, FirstConsumeTime: %d, NextConsumeTime: %d
\n"+
                        "\tBody: %s\n"+
                        "\tProps: %s\n",
                        v.MessageId, v.PublishTime, v.MessageTag, v.ConsumedTimes,
                        v.FirstConsumeTime, v.NextConsumeTime, v.MessageBody, v.Properties)
                }
                // If the broker does not receive an acknowledgment (ACK) for a message
from the consumer before the period of time specified by the NextConsumeTime parameter elapses,
the broker delivers the message for consumption again.
                // A unique timestamp is specified for the handle of a message each time
the message is consumed.
                ackerr := mqConsumer.AckMessage(handles)
                if ackerr != nil {
                    // If the handle of a message times out, the broker cannot receive
an ACK for the message from the consumer.
                    fmt.Println(ackerr)
                    if errAckItems, ok := ackerr.(errors.ErrCode).Context()["Detail"].([]mq_http_sdk.ErrAckItem); ok {
                        for _, errAckItem := range errAckItems {
                            fmt.Printf("\tErrorHandle:%s, ErrorCode:%s, ErrorMsg:%s\n",
                                errAckItem.ErrorHandle, errAckItem.ErrorCode, errAckItem.ErrorMessage)
                        }
                    } else {
                        fmt.Println("ack err =", ackerr)
                    }
                    time.Sleep(time.Duration(3) * time.Second)
                } else {
                    fmt.Printf("Ack ---->\n\t%s\n", handles)
                }
            }
        }
    }()
}

```

```

        endChan <- 1
    }
    case err := <-errChan:
    {
        // No messages in the topic are available for consumption.
        if strings.Contains(err.(errors.ErrCode).Error(), "MessageNotExist") {
            fmt.Println("\nNo new message, continue!")
        } else {
            fmt.Println(err)
            time.Sleep(time.Duration(3) * time.Second)
        }
        endChan <- 1
    }
    case <-time.After(35 * time.Second):
    {
        fmt.Println("Timeout of consumer message ??")
        endChan <- 1
    }
    }
}()
// In long polling mode, the default network timeout period is 35 seconds.
// In long polling mode, if no message in the topic is available for consumption, the request is suspended on the broker for a specified period of time. If a message becomes available for consumption within this period, the broker immediately sends a response to the consumer. In this example, the period is set to 3 seconds.
mqConsumer.ConsumeMessage(respChan, errChan,
    3, // The maximum number of messages that can be consumed at a time. In this example, the value is set to 3. The maximum value that you can specify is 16.
    3, // The length of a long polling period. Unit: seconds. In this example, the value is set to 3. The maximum value that you can specify is 30.
)
<-endChan
}
}

```

6.3.4. Python SDK


6.3.4.1. Prepare the environment

This topic describes how to prepare the environment before you use the HTTP client SDK for Python to send and consume messages.

Environment requirements

- Install Python. For more information, visit the [official website of Python](#). You must install an appropriate version of Python based on the following instructions:
 - If the version of your SDK is V1.0.0, make sure that the version of Python you installed is 2.5 or is later than 2.5 but earlier than 3.0.
 - If the version of your SDK is later than V1.0.0, make sure that the version of Python you installed is 2.5 or later.

- The pip tool is installed. For more information, see [Install pip](#).

 **Note** The pip tool is provided in Python 3.4 or later by default. If the version of Python you installed is 3.4 or later, you do not need to install the pip tool.

After Python is installed, you can run the `python -V` command to view the version of Python that you installed.

Install the SDK for Python

Run the following command to install the SDK for Python:

```
pip install mq_http_sdk
```

6.3.4.2. Send and consume normal messages

Normal messages are messages that have no special features in Message Queue for Apache RocketMQ. They are different from featured messages, such as scheduled messages, delayed messages, ordered messages, and transactional messages. This topic provides sample code to show how to use the HTTP client SDK for Python to send and consume normal messages.

Prerequisites

The following operations are performed:

- Install the SDK for Python. For more information about the message routing feature, see [Prepare the environment](#).
- Create resources that you want to specify in the code. For example, you must create the instance, topic, and group that you want to specify in the code in the Message Queue for Apache RocketMQ console in advance. For more information about the message routing feature, see [Create resources](#).

Send normal messages

The following sample code provides an example on how to send normal messages:

```

import sys
from mq_http_sdk.mq_exception import MQExceptionBase
from mq_http_sdk.mq_producer import *
from mq_http_sdk.mq_client import *
import time
# Initialize a producer client.
mq_client = MQClient(
    # The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log on to
    # the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click In
    # stances. On the Instances page, select the name of your instance. Then, view the HTTP endpo
    # int on the Network Management tab.
    "${HTTP_ENDPOINT}",
    # The AccessKey ID that is used for identity verification. You can obtain the AccessKey
    # ID in the Apsara Uni-manager Operations Console.
    "${ACCESS_KEY}",
    # The AccessKey secret that is used for identity verification. You can obtain the Acces
    # sKey secret in the Apsara Uni-manager Operations Console.
    "${SECRET_KEY}"
)
# The topic to which you want to send messages. The topic is created in the Message Queue f
# or Apache RocketMQ console.
topic_name = "${TOPIC}"
# The ID of the instance to which the topic belongs. The instance is created in the Message
# Queue for Apache RocketMQ console.
# If the instance has a namespace, specify the ID of the instance. If the instance does not
# have a namespace, set the instance ID to null or an empty string. You can check whether you
# r instance has a namespace on the Instances page in the RocketMQ console.
instance_id = "${INSTANCE_ID}"
producer = mq_client.get_producer(instance_id, topic_name)
# Cyclically send four messages.
msg_count = 4
print("%sPublish Message To %s\nTopicName:%s\nMessageCount:%s\n" % (10 * "=", 10 * "=", top
ic_name, msg_count))
try:
    for i in range(msg_count):
        msg = TopicMessage(
            # The content of the message.
            "I am test message %s.hello" % i,
            # The tag of the message.
            "tag %s" % i
        )
        # The custom property of the message.
        msg.put_property("a", "i")
        # The key of the message.
        msg.set_message_key("MessageKey")
        re_msg = producer.publish_message(msg)
        print("Publish Message Succeed. MessageID:%s, BodyMD5:%s" % (re_msg.message_id,
re_msg.message_body_md5))
    except MQExceptionBase as e:
        if e.type == "TopicNotExist":
            print("Topic not exist, please create it.")
            sys.exit(1)
        print("Publish Message Fail. Exception:%s" % e)

```

Consume normal messages

The following sample code provides an example on how to consume normal messages:

```
from mq_http_sdk.mq_exception import MQExceptionBase
from mq_http_sdk.mq_consumer import *
from mq_http_sdk.mq_client import *
# Initialize a consumer client.
mq_client = MQClient(
    # The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click Instances. On the Instances page, select the name of your instance. Then, view the HTTP endpoint on the Network Management tab.
    "${HTTP_ENDPOINT}",
    # The AccessKey ID that is used for identity verification. You can obtain the AccessKey ID in the Apsara Uni-manager Operations Console.
    "${ACCESS_KEY}",
    # The AccessKey secret that is used for identity verification. You can obtain the AccessKey secret in the Apsara Uni-manager Operations Console.
    "${SECRET_KEY}"
)
# The topic from which you want to consume messages. The topic is created in the Message Queue for Apache RocketMQ console.
topic_name = "${TOPIC}"
# The ID of the group that you created in the Message Queue for Apache RocketMQ console.
group_id = "${GROUP_ID}"
# The ID of the instance to which the topic belongs. The instance is created in the Message Queue for Apache RocketMQ console.
# If the instance has a namespace, specify the ID of the instance. If the instance does not have a namespace, set the instance ID to null or an empty string. You can check whether your instance has a namespace on the Instances page in the RocketMQ console.
instance_id = "${INSTANCE_ID}"
consumer = mq_client.get_consumer(instance_id, topic_name, group_id)
# In long polling mode, if no message in the topic is available for consumption, the request is suspended on the broker for a specified period of time. If a message becomes available for consumption within this period, the broker immediately sends a response to the consumer. In this example, the period is set to 3 seconds.
# The length of a long polling period. Unit: seconds. In this example, the value is set to 3. The maximum value that you can specify is 30.
wait_seconds = 3
# The maximum number of messages that can be consumed at a time. In this example, the value is set to 3. The maximum value that you can specify is 16.
batch = 3
print("%sConsume And Ak Message From Topic%s\nTopicName:%s\nMQConsumer:%s\nWaitSeconds:%s\n" \
      % (10 * "=", 10 * "=", topic_name, group_id, wait_seconds)))
while True:
    try:
        # Consume messages in long polling mode.
        recv_msgs = consumer.consume_message(batch, wait_seconds)
        for msg in recv_msgs:
            print("Receive, MessageId: %s\nMessageBodyMD5: %s \
                  \nMessageTag: %s\nConsumedTimes: %s \
                  \nPublishTime: %s\nBody: %s \
                  \nNextConsumeTime: %s \
                  \nNextConsumeOffset: %s\n" % (msg.message_id, msg.message_body_md5, msg.message_tag, msg.consumed_times, msg.publish_time, msg.body, msg.next_consume_time, msg.next_consume_offset))
```

```

        \nnextConsumeTime: %s \
        \nReceiptHandle: %s \
        \nProperties: %s\n" % \
        (msg.message_id, msg.message_body_md5,
         msg.message_tag, msg.consumed_times,
         msg.publish_time, msg.message_body,
         msg.next_consume_time, msg.receipt_handle, msg.properties)))
    print(msg.get_property(""))
except MQExceptionBase as e:
    # No messages in the topic are available for consumption.
    if e.type == "MessageNotExist":
        print(("No new message! RequestId: %s" % e.req_id))
        continue
    print(("Consume Message Fail! Exception:%s\n" % e))
    time.sleep(2)
    continue

    # If the broker does not receive an acknowledgment (ACK) for a message from the consumer
    # before the period of time specified by the msg.next_consume_time parameter elapses, the broker
    # delivers the message for consumption again.
    # A unique timestamp is specified for the handle of a message each time the message is
    # consumed.
    try:
        receipt_handle_list = [msg.receipt_handle for msg in recv_msgs]
        consumer.ack_message(receipt_handle_list)
        print(("Ak %s Message Succeed.\n\n" % len(receipt_handle_list)))
    except MQExceptionBase as e:
        print(("nAk Message Fail! Exception:%s" % e))
        # If the handle of a message times out, the broker cannot receive an ACK for the message
        # from the consumer.
        if e.sub_errors:
            for sub_error in e.sub_errors:
                print(("tErrorHandle:%s, ErrorCode:%s, ErrorMessage:%s" % \
                    (sub_error["ReceiptHandle"], sub_error["ErrorCode"], sub_error["Error
                    Message"])))

```

6.3.4.3. Send and consume ordered messages

Ordered messages are a type of message that is published and consumed in a strict order. Ordered messages in Message Queue for Apache RocketMQ are also known as first-in-first-out (FIFO) messages. This topic provides sample code to show how to use the HTTP client SDK for Python to send and consume ordered messages.

Background information

Ordered messages are classified into the following types:

- Globally ordered message: All messages in a specified topic are published and consumed in first-in-first-out (FIFO) order.
- Partititionally ordered message: All messages in a specified topic are distributed to different partitions by using shard keys. The messages in each partition are published and consumed in FIFO order. A Sharding Key is a key field that is used for ordered messages to identify different partitions. The Sharding Key is different from the key of a normal message.

For more information about the message routing feature, see [Ordered messages](#).

Prerequisites

The following operations are performed:

- Install the SDK for Python. For more information about the message routing feature, see [Prepare the environment](#).
- Create resources that you want to specify in the code. For example, you must create the instance, topic, and group that you want to specify in the code in the Message Queue for Apache RocketMQ console in advance. For more information about the message routing feature, see [Create resources](#).

Send ordered messages

The following sample code provides an example on how to send ordered messages:

```
import sys
from mq_http_sdk.mq_exception import MQExceptionBase
from mq_http_sdk.mq_producer import *
from mq_http_sdk.mq_client import *
# Initialize a producer client.
mq_client = MQClient(
    # The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log on to
    # the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click In
    # stances. On the Instances page, select the name of your instance. Then, view the HTTP endpo
    # int on the Network Management tab.
    "${HTTP_ENDPOINT}",
    # The AccessKey ID that is used for identity verification. You can obtain the AccessKey
    # ID in the Apsara Uni-manager Operations Console.
    "${ACCESS_KEY}",
    # The AccessKey secret that is used for identity verification. You can obtain the Acces
    # sKey secret in the Apsara Uni-manager Operations Console.
    "${SECRET_KEY}"
)
# The topic to which you want to send messages. The topic is created in the Message Queue f
# or Apache RocketMQ console.
topic_name = "${TOPIC}"
# The ID of the instance to which the topic belongs. The instance is created in the Message
# Queue for Apache RocketMQ console.
# If the instance has a namespace, specify the ID of the instance. If the instance does not
# have a namespace, set the instance ID to null or an empty string. You can check whether you
# r instance has a namespace on the Instances page in the RocketMQ console.
instance_id = "${INSTANCE_ID}"
producer = mq_client.get_producer(instance_id, topic_name)
# Cyclically send eight messages.
msg_count = 8
print("%sPublish Message To %s\nTopicName:%s\nMessageCount:%s\n" % (10 * "=", 10 * "=", top
ic_name, msg_count))
try:
    for i in range(msg_count):
        msg = TopicMessage(
            # The content of the message.
            "I am test message %s.hello" % i,
            # The tag of the message.
            "tag %s" % i
```

```

    )
    # The custom property of the message.
    msg.put_property("a", str(i))
    # The shard key that is used to distribute ordered messages to a specific partition
    . Shard keys can be used to identify different partitions. A shard key is different from a
    message key.
    msg.set_sharding_key(str(i % 3))
    re_msg = producer.publish_message(msg)
    print("Publish Message Succeed. MessageID:%s, BodyMD5:%s" % (re_msg.message_id, re_
    msg.message_body_md5))
except MQExceptionBase as e:
    if e.type == "TopicNotExist":
        print("Topic not exist, please create it.")
        sys.exit(1)
    print("Publish Message Fail. Exception:%s" % e)

```

Consume ordered messages

The following sample code provides an example on how to consume ordered messages:

```

from mq_http_sdk.mq_exception import MQExceptionBase
from mq_http_sdk.mq_consumer import *
from mq_http_sdk.mq_client import *
# Initialize a consumer client.
mq_client = MQClient(
    # The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log on t
    o the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click In
    stances. On the Instances page, select the name of your instance. Then, view the HTTP endpo
    int on the Network Management tab.
    "${HTTP_ENDPOINT}",
    # The AccessKey ID that is used for identity verification. You can obtain the AccessKey
    ID in the Apsara Uni-manager Operations Console.
    "${ACCESS_KEY}",
    # The AccessKey secret that is used for identity verification. You can obtain the Acces
    sKey secret in the Apsara Uni-manager Operations Console.
    "${SECRET_KEY}"
)
# The topic from which you want to consume messages. The topic is created in the Message Qu
    eue for Apache RocketMQ console.
topic_name = "${TOPIC}"
# The ID of the group that you created in the Message Queue for Apache RocketMQ console.
group_id = "${GROUP_ID}"
# The ID of the instance to which the topic belongs. The instance is created in the Message
    Queue for Apache RocketMQ console.
# If the instance has a namespace, specify the ID of the instance. If the instance does not
    have a namespace, set the instance ID to null or an empty string. You can check whether you
    r instance has a namespace on the Instances page in the RocketMQ console.
instance_id = "${INSTANCE_ID}"
consumer = mq_client.get_consumer(instance_id, topic_name, group_id)
# Consume messages in long polling mode. The consumer may pull partitionally ordered messag
    es from multiple partitions. The consumer consumes the messages in the same partition in th
    e order in which the messages are sent.
# A consumer pulls partitionally ordered messages from a partition. If the broker does not

```

```

receive an acknowledgment (ACK) for a message after the message is consumed, the consumer consumes the message again.
# The consumer can consume the next batch of messages from a partition only after all the messages that are pulled from the partition in the previous batch are acknowledged to be consumed.
# In long polling mode, if no message in the topic is available for consumption, the request is suspended on the broker for a specified period of time. If a message becomes available for consumption within this period, the broker immediately sends a response to the consumer. In this example, the period is set to 3 seconds.
wait_seconds = 3
# The maximum number of messages that can be consumed at a time. In this example, the value is set to 3. The maximum value that you can specify is 16.
batch = 3
print(("Consume And Ack Message From Topic%s\nTopicName:%s\nMQConsumer:%s\nWaitSeconds:%s\n" \
      % (10 * "=", 10 * "=", topic_name, group_id, wait_seconds)))
while True:
    try:
        recv_msgs = consumer.consume_message_orderly(batch, wait_seconds)
        print("=====>Receive %d messages:" % len(recv_msgs))
        for msg in recv_msgs:
            print("\tMessageId: %s, MessageBodyMD5: %s,NextConsumeTime: %s,ConsumedTimes: %s,PublishTime: %s\n\tBody: %s \
                  \n\tReceiptHandle: %s \
                  \n\tProperties: %s,ShardingKey: %s\n" % \
                  (msg.message_id, msg.message_body_md5,
                   msg.next_consume_time, msg.consumed_times,
                   msg.publish_time, msg.message_body,
                   msg.receipt_handle, msg.properties, msg.get_sharding_key()))
    except MQExceptionBase as e:
        if e.type == "MessageNotExist":
            print(("No new message! RequestId: %s" % e.req_id))
            continue
        print(("Consume Message Fail! Exception:%s\n" % e))
        time.sleep(2)
        continue

    # If the broker does not receive an ACK for a message from the consumer before the period of time specified by the msg.next_consume_time parameter elapses, the broker delivers the message for consumption again.
    # A unique timestamp is specified for the handle of a message each time the message is consumed.
    try:
        receipt_handle_list = [msg.receipt_handle for msg in recv_msgs]
        consumer.ack_message(receipt_handle_list)
        print("=====>Ack %s Message Succeed.\n\n" % len(receipt_handle_list))
    except MQExceptionBase as e:
        print(("Ack Message Fail! Exception:%s" % e))
        # If the handle of a message times out, the broker cannot receive an ACK for the message from the consumer.
        if e.sub_errors:
            for sub_error in e.sub_errors:
                print(("ErrorHandle:%s,ErrorCode:%s,ErrorMsg:%s" % \
                      (sub_error["ReceiptHandle"], sub_error["ErrorCode"], sub_error["ErrorMessage"])))

```

6.3.4.4. Send and consume scheduled messages and delayed messages

This topic provides sample code to show how to use the HTTP client SDK for Python to send and consume scheduled messages and delayed messages.

Background information

- **Delayed message:** The producer sends the message to the Message Queue for Apache RocketMQ server, but does not expect the message to be delivered immediately. Instead, the message is delivered to the consumer for consumption after a certain period of time. This message is a delayed message.
- **Scheduled message:** A producer sends a message to a Message Queue for Apache RocketMQ broker and expects the message to be delivered to a consumer at a specified point in time. This type of message is called a scheduled message.

If an HTTP client SDK is used, the code configurations of scheduled messages are the same as the code configurations of delayed messages. Both types of messages are delivered to consumers after a specific period of time based on the attributes of the messages.

For more information about the message routing feature, see [Scheduled messages and delayed messages](#).

Prerequisites

The following operations are performed:

- Install the SDK for Python. For more information about the message routing feature, see [Prepare the environment](#).
- Create resources that you want to specify in the code. For example, you must create the instance, topic, and group that you want to specify in the code in the Message Queue for Apache RocketMQ console in advance. For more information about the message routing feature, see [Create resources](#).

Send scheduled messages or delayed messages

The following sample code provides an example on how to send scheduled messages or delayed messages:

```
import sys
from mq_http_sdk.mq_exception import MQExceptionBase
from mq_http_sdk.mq_producer import *
from mq_http_sdk.mq_client import *
import time

# Initialize a producer client.
mq_client = MQClient(
    # The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log on to
    # the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click In
    # stances. On the Instances page, select the name of your instance. Then, view the HTTP endpo
    # int on the Network Management tab.
    "${HTTP_ENDPOINT}",
    # The AccessKey ID that is used for identity verification. You can obtain the AccessKey
    # ID in the Apsara Uni-manager Operations Console.
```

```

    "${ACCESS_KEY}",
    # The AccessKey secret that is used for identity verification. You can obtain the AccessKey secret in the Apsara Uni-manager Operations Console.
    "${SECRET_KEY}"
)
# The topic to which you want to send messages. The topic is created in the Message Queue for Apache RocketMQ console.
topic_name = "${TOPIC}"
# The ID of the instance to which the topic belongs. The instance is created in the Message Queue for Apache RocketMQ console.
# If the instance has a namespace, specify the ID of the instance. If the instance does not have a namespace, set the instance ID to null or an empty string. You can check whether your instance has a namespace on the Instances page in the RocketMQ console.
instance_id = "${INSTANCE_ID}"
producer = mq_client.get_producer(instance_id, topic_name)
# Cyclically send four messages.
msg_count = 4
print("%sPublish Message To %s\nTopicName:%s\nMessageCount:%s\n" % (10 * "=", 10 * "=", topic_name, msg_count))
try:
    for i in range(msg_count):
        msg = TopicMessage(
            # The content of the message.
            "I am test message %s.hello" % i,
            # The tag of the message.
            "tag1"
        )
        # The property of the message.
        msg.put_property("a", "i")
        # The key of the message.
        msg.set_message_key("MessageKey")
        # The period of time after which the broker delivers the message. In this example, when the broker receives a message, the broker waits for 10 seconds before it delivers the message to the consumer. Set this parameter to a timestamp in milliseconds.
        # If the producer sends a scheduled message, set the parameter to the time interval between the scheduled point in time and the current point in time.
        msg.set_start_deliver_time(int(round(time.time() * 1000)) + 10 * 1000)
        re_msg = producer.publish_message(msg)
        print("Publish Timer Message Succeed. MessageID:%s, BodyMD5:%s" % (re_msg.message_id, re_msg.message_body_md5))
except MQExceptionBase as e:
    if e.type == "TopicNotExist":
        print("Topic not exist, please create it.")
        sys.exit(1)
    print("Publish Message Fail. Exception:%s" % e)

```

Consume scheduled messages or delayed messages

The following sample code provides an example on how to consume scheduled messages or delayed messages:

```

from mq_http_sdk.mq_exception import MQExceptionBase
from mq_http_sdk.mq_consumer import *
from mq_http_sdk.mq_client import *

```

```

# Initialize a consumer client.
mq_client = MQClient(
    # The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log on to
    # the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click In
    # stances. On the Instances page, select the name of your instance. Then, view the HTTP endpo
    # int on the Network Management tab.
    "${HTTP_ENDPOINT}",
    # The AccessKey ID that is used for identity verification. You can obtain the AccessKey
    # ID in the Apsara Uni-manager Operations Console.
    "${ACCESS_KEY}",
    # The AccessKey secret that is used for identity verification. You can obtain the Acces
    # sKey secret in the Apsara Uni-manager Operations Console.
    "${SECRET_KEY}"
)

# The topic from which you want to consume messages. The topic is created in the Message Qu
# eue for Apache RocketMQ console.
topic_name = "${TOPIC}"

# The ID of the group that you created in the Message Queue for Apache RocketMQ console.
group_id = "${GROUP_ID}"

# The ID of the instance to which the topic belongs. The instance is created in the Message
# Queue for Apache RocketMQ console.
# If the instance has a namespace, specify the ID of the instance. If the instance does not
# have a namespace, set the instance ID to null or an empty string. You can check whether you
# r instance has a namespace on the Instances page in the RocketMQ console.
instance_id = "${INSTANCE_ID}"

consumer = mq_client.get_consumer(instance_id, topic_name, group_id)

# In long polling mode, if no message in the topic is available for consumption, the reques
# t is suspended on the broker for a specified period of time. If a message becomes available
# for consumption within this period, the broker immediately sends a response to the consumer
# . In this example, the period is set to 3 seconds.
# The length of a long polling period. Unit: seconds. In this example, the value is set to
# 3. The maximum value that you can specify is 30.
wait_seconds = 3

# The maximum number of messages that can be consumed at a time. In this example, the value
# is set to 3. The maximum value that you can specify is 16.
batch = 3

print(("Consuming And Acknowledging Message From Topic%s\nTopicName:%s\nMQConsumer:%s\nWaitSeconds:%s\n"
      % (10 * "=", 10 * "=", topic_name, group_id, wait_seconds)))

while True:
    try:
        # Consume messages in long polling mode.
        recv_msgs = consumer.consume_message(batch, wait_seconds)
        for msg in recv_msgs:
            print(("Receive, MessageId: %s\nMessageBodyMD5: %s \
                  \nMessageTag: %s\nConsumedTimes: %s \
                  \nPublishTime: %s\nBody: %s \
                  \nNextConsumeTime: %s \
                  \nReceiptHandle: %s \
                  \nProperties: %s\n" % \
                  (msg.message_id, msg.message_body_md5,
                   msg.message_tag, msg.consumed_times,
                   msg.publish_time, msg.message_body,
                   msg.next_consume_time, msg.receipt_handle, msg.properties)))
    
```

```

        print(msg.get_property(""))
    except MQExceptionBase as e:
        # No messages in the topic are available for consumption.
        if e.type == "MessageNotExist":
            print(("No new message! RequestId: %s" % e.req_id))
            continue
        print(("Consume Message Fail! Exception:%s\n" % e))
        time.sleep(2)
        continue

    # If the broker does not receive an acknowledgment (ACK) for a message from the consumer
    # before the period of time specified by the msg.next_consume_time parameter elapses, the broker
    # delivers the message for consumption again.

    # A unique timestamp is specified for the handle of a message each time the message is
    # consumed.
    try:
        receipt_handle_list = [msg.receipt_handle for msg in recv_msgs]
        consumer.ack_message(receipt_handle_list)
        print(("Ak %s Message Succeed.\n\n" % len(receipt_handle_list)))
    except MQExceptionBase as e:
        print(("Ak Message Fail! Exception:%s" % e))
        # If the handle of a message times out, the broker cannot receive an ACK for the message
        # from the consumer.
        if e.sub_errors:
            for sub_error in e.sub_errors:
                print(("tErrorHandle:%s, ErrorCode:%s, ErrorMsg:%s" % \
                    (sub_error["ReceiptHandle"], sub_error["ErrorCode"], sub_error["Error
Message"])))

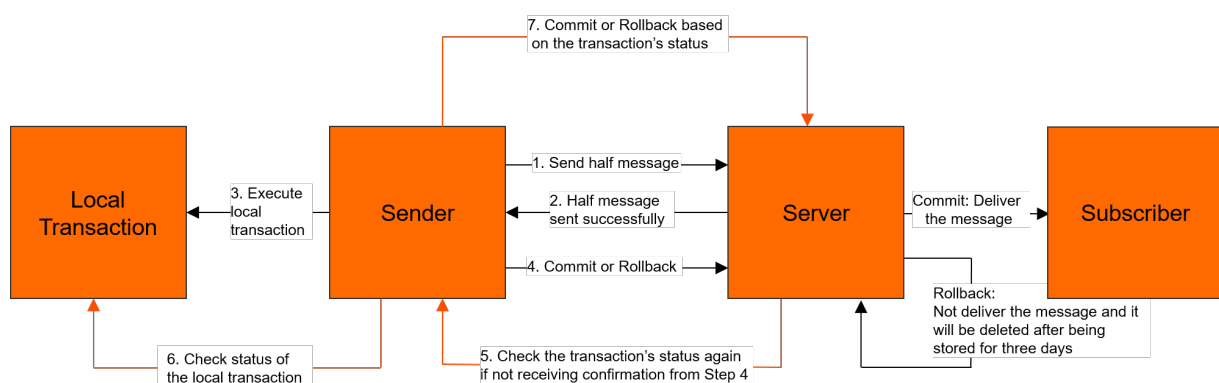
```

6.3.4.5. Send and consume transactional messages

Message Queue for Apache RocketMQ provides a distributed transaction processing feature that is similar to X/Open XA. Message Queue for Apache RocketMQ uses transactional messages to ensure transactional consistency. This topic provides sample code to show how to use the HTTP client SDK for Python to send and consume transactional messages.

Background information

The following figure shows the interaction process of transactional messages.



For more information about the message routing feature, see [Transactional messages](#).

Prerequisites

The following operations are performed:

- Install the SDK for Python. For more information about the message routing feature, see [Prepare the environment](#).
- Create resources that you want to specify in the code. For example, you must create the instance, topic, and group that you want to specify in the code in the Message Queue for Apache RocketMQ console in advance. For more information about the message routing feature, see [Create resources](#).

Send transactional messages

The following sample code provides an example on how to send transactional messages:

```
#!/usr/bin/env python
# coding=utf8
import sys
from mq_http_sdk.mq_exception import MQExceptionBase
from mq_http_sdk.mq_producer import *
from mq_http_sdk.mq_client import *
import time
import threading
# Initialize a producer client.
mq_client = MQClient(
    # The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log on to
    # the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click In
    # stances. On the Instances page, select the name of your instance. Then, view the HTTP endpo
    # int on the Network Management tab.
    "${HTTP_ENDPOINT}",
    # The AccessKey ID that you created in the Resource Access Management (RAM) console. Th
    # e AccessKey ID is used for identity verification.
    "${ACCESS_KEY}",
    # The AccessKey secret that you created in the RAM console. The AccessKey secret is use
    # d for identity verification.
    "${SECRET_KEY}"
)
# The topic to which you want to send messages. The topic is created in the Message Queue f
# or Apache RocketMQ console.
topic_name = "${TOPIC}"
# The ID of the group that you created in the Message Queue for Apache RocketMQ console.
group_id = "${GROUP_ID}"
# The ID of the instance to which the topic belongs. The instance is created in the Message
# Queue for Apache RocketMQ console.
# If the instance has a namespace, specify the ID of the instance. If the instance does not
# have a namespace, set the instance ID to null or an empty string. You can check whether you
# r instance has a namespace on the Instances page in the
# Message Queue for Apache RocketMQ console.
instance_id = "${INSTANCE_ID}"
# Cyclically send four transactional messages.
msg_count = 4
print("%sPublish Transaction Message To %s\nTopicName:%s\nMessageCount:%s\n" \
      % (10 * "=", 10 * "=", topic_name, msg_count))
def process_trans_error(exp):
    print("\nCommit/Roll Transaction Message Fail! Exception:%s" % exp)
    # If a transactional message is not committed or rolled back before the timeout period
```

specified by the TransCheckImmunityTime parameter for the handle of the transactional message elapses or before the timeout period specified for the handle of consumeHalfMessage elapses, the commit or rollback operation fails. In this example, the timeout period for the handle of consumeHalfMessage is 10 seconds.

```

    if exp.sub_errors:
        for sub_error in exp.sub_errors:
            print("\tErrorHandle:%s,ErrorCode:%s,ErrorMsg:%s" % \
                  (sub_error["ReceiptHandle"], sub_error["ErrorCode"], sub_error["ErrorMessage"]))
# The client requires a thread or a process to process unacknowledged transactional messages.
# Start a thread to process unacknowledged transactional messages.
class ConsumeHalfMessageThread(threading.Thread):
    def __init__(self):
        threading.Thread.__init__(self)
        self.count = 0
        # Create another client.
        self.mq_client = MQClient(
            # The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click Instances. On the Instances page, select the name of your instance. Then, view the endpoint in the HTTP Endpoint section on the Network Management tab.
            "${HTTP_ENDPOINT}",
            # The AccessKey ID that is used for identity verification. You can obtain the AccessKey ID in the Apsara Uni-manager Operations Console.
            "${ACCESS_KEY}",
            # The AccessKey secret that is used for identity verification. You can obtain the AccessKey secret in the Apsara Uni-manager Operations Console.
            "${SECRET_KEY}"
        )
        self.trans_producer = self.mq_client.get_trans_producer(instance_id, topic_name, group_id)
    def run(self):
        while 1:
            if self.count == 3:
                break;
            try:
                half_msgs = self.trans_producer.consume_half_message(1, 3)
                for half_msg in half_msgs:
                    print("Receive Half Message, MessageId: %s\nMessageBodyMD5: %s \
                          \nMessageTag: %s\nConsumedTimes: %s \
                          \nPublishTime: %s\nBody: %s \
                          \nNextConsumeTime: %s \
                          \nReceiptHandle: %s \
                          \nProperties: %s" % \
                          (half_msg.message_id, half_msg.message_body_md5,
                           half_msg.message_tag, half_msg.consumed_times,
                           half_msg.publish_time, half_msg.message_body,
                           half_msg.next_consume_time, half_msg.receipt_handle, half_msg.properties))
                    a = int(half_msg.get_property("a"))
                    try:
                        if a == 1:
                            # Confirm to commit the transactional message.

```

```

        self.trans_producer.commit(half_msg.receipt_handle)
        self.count += 1
        print("----->commit")
    elif a == 2 and half_msg.consumed_times > 1:
        # Confirm to commit the transactional message.
        self.trans_producer.commit(half_msg.receipt_handle)
        self.count += 1
        print("----->commit")
    elif a == 3:
        # Confirm to roll back the transactional message.
        self.trans_producer.rollback(half_msg.receipt_handle)
        self.count += 1
        print("----->rollback")
    else:
        # Do not perform operations. Check the status next time.
        print("----->unknown")
except MQExceptionBase as rec_commit_roll_e:
    process_trans_error(rec_commit_roll_e)
except MQExceptionBase as half_e:
    if half_e.type == "MessageNotExist":
        print("No half message! RequestId: %s" % half_e.req_id)
        continue
    print("Consume half message Fail! Exception:%s\n" % half_e)
    break

consume_half_thread = ConsumeHalfMessageThread()
consume_half_thread.setDaemon(True)
consume_half_thread.start()

try:
    trans_producer = mq_client.get_trans_producer(instance_id, topic_name, group_id)
    for i in range(msg_count):
        msg = TopicMessage(
            # The content of the message.
            "I am test message %s." % i,
            # The tag of the message.
            "tagA"
        )
        # The custom property of the message.
        msg.put_property("xy", i)
        # The key of the message.
        msg.set_message_key("MessageKey")
        # The time interval between the time when the transactional message is sent and the
        # start time of the first transaction status check. Unit: seconds. Valid values: 10 to 300.
        # If the message is not committed or rolled back after the first transaction status
        # check is performed, the broker initiates a request to check the status of the local transac
        # tion at an interval of 10 seconds within the next 24 hours.
        msg.set_trans_check_immunity_time(10)
        re_msg = trans_producer.publish_message(msg)
        print("Publish Transaction Message Succeed. MessageID:%s, BodyMD5:%s, Handle:%s" \
              % (re_msg.message_id, re_msg.message_body_md5, re_msg.receipt_handle))
        time.sleep(1)
    if i == 0:
        # After the producer sends the transactional message, the broker obtains the ha
        # ndle of the half message that corresponds to the transactional message and commits or rolls
        # back the transactional message based on the status of the handle.

```

```

        try:
            trans_producer.commit(re_msg.receipt_handle)
        except MQExceptionBase as pub_commit_roll_e:
            process_trans_error(pub_commit_roll_e)
    except MQExceptionBase as pub_e:
        if pub_e.type == "TopicNotExist":
            print("Topic not exist, please create it.")
            sys.exit(1)
        print("Publish Message Fail. Exception:%s" % pub_e)
    while 1:
        if not consume_half_thread.is_alive():
            break
        time.sleep(1)

```

Consume transactional messages

The following sample code provides an example on how to consume transactional messages:

```

from mq_http_sdk.mq_exception import MQExceptionBase
from mq_http_sdk.mq_consumer import *
from mq_http_sdk.mq_client import *
# Initialize a consumer client.
mq_client = MQClient(
    # The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log on to
    # the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click In
    # stances. On the Instances page, select the name of your instance. Then, view the HTTP endpo
    # int on the Network Management tab.
    "${HTTP_ENDPOINT}",
    # The AccessKey ID that is used for identity verification. You can obtain the AccessKey
    # ID in the Apsara Uni-manager Operations Console.
    "${ACCESS_KEY}",
    # The AccessKey secret that is used for identity verification. You can obtain the Acces
    # sKey secret in the Apsara Uni-manager Operations Console.
    "${SECRET_KEY}"
)
# The topic from which you want to consume messages. The topic is created in the Message Qu
# eue for Apache RocketMQ console.
topic_name = "${TOPIC}"
# The ID of the group that you created in the Message Queue for Apache RocketMQ console.
group_id = "${GROUP_ID}"
# The ID of the instance to which the topic belongs. The instance is created in the Message
# Queue for Apache RocketMQ console.
# If the instance has a namespace, specify the ID of the instance. If the instance does not
# have a namespace, set the instance ID to null or an empty string. You can check whether you
# r instance has a namespace on the Instances page in the RocketMQ console.
instance_id = "${INSTANCE_ID}"
consumer = mq_client.get_consumer(instance_id, topic_name, group_id)
# In long polling mode, if no message in the topic is available for consumption, the reques
# t is suspended on the broker for a specified period of time. If a message becomes available
# for consumption within this period, the broker immediately sends a response to the consumer
# . In this example, the period is set to 3 seconds.
# The length of a long polling period. Unit: seconds. In this example, the value is set to
# 3. The maximum value that you can specify is 30.
wait_seconds = 3

```

```

# The maximum number of messages that can be consumed at a time. In this example, the value
is set to 3. The maximum value that you can specify is 16.
batch = 3
print(("Consumed And Ack Message From Topic%s\nTopicName:%s\nMQConsumer:%s\nWaitSeconds:%s\n" \
      % (10 * "=", 10 * "=", topic_name, group_id, wait_seconds)))
while True:
    try:
        # Consume messages in long polling mode.
        recv_msgs = consumer.consume_message(batch, wait_seconds)
        for msg in recv_msgs:
            print(("Receive, MessageId: %s\nMessageBodyMD5: %s \
                  \nMessageTag: %s\nConsumedTimes: %s \
                  \nPublishTime: %s\nBody: %s \
                  \nNextConsumeTime: %s \
                  \nReceiptHandle: %s \
                  \nProperties: %s\n" % \
                  (msg.message_id, msg.message_body_md5,
                   msg.message_tag, msg.consumed_times,
                   msg.publish_time, msg.message_body,
                   msg.next_consume_time, msg.receipt_handle, msg.properties)))
            print(msg.get_property(""))
    except MQExceptionBase as e:
        # No messages in the topic are available for consumption.
        if e.type == "MessageNotExist":
            print(("No new message! RequestId: %s" % e.req_id))
            continue
        print(("Consume Message Fail! Exception:%s\n" % e))
        time.sleep(2)
        continue

    # If the broker does not receive an acknowledgment (ACK) for a message from the consume
    r before the period of time specified by the msg.next_consume_time parameter elapses, the b
    roker delivers the message for consumption again.

    # A unique timestamp is specified for the handle of a message each time the message is
    consumed.
    try:
        receipt_handle_list = [msg.receipt_handle for msg in recv_msgs]
        consumer.ack_message(receipt_handle_list)
        print(("Ack %s Message Succeed.\n\n" % len(receipt_handle_list)))
    except MQExceptionBase as e:
        print(("Ack Message Fail! Exception:%s" % e))
        # If the handle of a message times out, the broker cannot receive an ACK for the me
        ssage from the consumer.
        if e.sub_errors:
            for sub_error in e.sub_errors:
                print(("ErrorHandle:%s, ErrorCode:%s, ErrorMessage:" % \
                      (sub_error["ReceiptHandle"], sub_error["ErrorCode"], sub_error["Error
Message"])))

```

6.3.5. Node.js SDK

6.3.5.1. Prepare the environment

This topic describes how to prepare the environment before you use the HTTP client SDK for Node.js to send and consume messages.

Environment requirements

Node.js 7.6.0 or later is installed. For more information, see [Install Node.js](#).

After Node.js is installed, you can run the `node -v` command to view the version of Node.js that you installed.

Install the SDK for Node.js

Run the following command to install the SDK for Node.js:

```
npm i @aliyunmq/mq-http-sdk
```

6.3.5.2. Send and consume normal messages

Normal messages are messages that have no special features in Message Queue for Apache RocketMQ. They are different from featured messages, such as scheduled messages, delayed messages, ordered messages, and transactional messages. This topic provides sample code to show how to use the HTTP client SDK for Node.js to send and consume normal messages.

Prerequisites

The following operations are performed:

- SDK for Node.js is installed. For more information about the message routing feature, see [Prepare the environment](#).
- Create resources that you want to specify in the code. For example, you must create the instance, topic, and group that you want to specify in the code in the Message Queue for Apache RocketMQ console in advance. For more information about the message routing feature, see [Create resources](#).

Send normal messages

The following sample code provides an example on how to send normal messages:

```

const {
  MQClient,
  MessageProperties
} = require('@aliyunmq/mq-http-sdk');
// The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log on to the
// Message Queue for Apache RocketMQ console. In the left-side navigation pane, click Instances.
// On the Instances page, select the name of your instance. Then, view the HTTP endpoint on the
// Network Management tab.
const endpoint = "${HTTP_ENDPOINT}";
// The AccessKey ID that you created in the Resource Access Management (RAM) console for identity
// verification. The AccessKey ID is used for identity verification.
const accessKeyId = "${ACCESS_KEY}";
// The AccessKey secret that you created in the RAM console for identity verification. The
// AccessKey secret is used for identity verification.
const accessKeySecret = "${SECRET_KEY}";
var client = new MQClient(endpoint, accessKeyId, accessKeySecret);
// The topic to which you want to send messages. The topic is created in the Message Queue
// for Apache RocketMQ console.
const topic = "${TOPIC}";
// The ID of the instance to which the topic belongs. The instance is created in the Message
// Queue for Apache RocketMQ console.
// If the instance has a namespace, specify the ID of the instance. If the instance does not
// have a namespace, set the instance ID to null or an empty string. You can check whether your
// instance has a namespace on the Instances page in the RocketMQ console.
const instanceId = "${INSTANCE_ID}";
const producer = client.getProducer(instanceId, topic);
(async function(){
  try {
    // Cyclically send four messages.
    for(var i = 0; i < 4; i++) {
      let res;
      msgProps = new MessageProperties();
      // The custom property of the message.
      msgProps.putProperty("a", i);
      // The key of the message.
      msgProps.messageKey("MessageKey");
      res = await producer.publishMessage("hello mq.", "", msgProps);
      console.log("Publish message: MessageID:%s,BodyMD5:%s", res.body.MessageId, res.body.
        MessageBodyMD5);
    }
  } catch(e) {
    // Specify the logic that you want to use to resend or persist the message if the message
    // fails to be sent and needs to be sent again.
    console.log(e)
  }
})();

```

Consume normal messages

The following sample code provides an example on how to consume normal messages:

```

const {
  MQClient
} = require('@aliyunmq/mq-http-sdk');

```

```

const client = require('aliyunmq/mq-http-sdk');
// The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log on to the
// Message Queue for Apache RocketMQ console. In the left-side navigation pane, click Instances.
// On the Instances page, select the name of your instance. Then, view the HTTP endpoint on the
// Network Management tab.
const endpoint = "${HTTP_ENDPOINT}";
// The AccessKey ID that is used for identity verification. You can obtain the AccessKey ID in the
// Apsara Uni-manager Operations Console.
const accessKeyId = "${ACCESS_KEY}";
// The AccessKey secret that is used for identity verification. You can obtain the AccessKey secret
// in the Apsara Uni-manager Operations Console.
const accessKeySecret = "${SECRET_KEY}";
var client = new MQClient(endpoint, accessKeyId, accessKeySecret);
// The topic from which you want to consume messages. The topic is created in the Message Queue
// for Apache RocketMQ console.
const topic = "${TOPIC}";
// The ID of the group that you created in the Message Queue for Apache RocketMQ console.
const groupId = "${GROUP_ID}";
// The ID of the instance to which the topic belongs. The instance is created in the Message Queue
// for Apache RocketMQ console.
// If the instance has a namespace, specify the ID of the instance. If the instance does not have a
// namespace, set the instance ID to null or an empty string. You can check whether your instance
// has a namespace on the Instances page in the RocketMQ console.
const instanceId = "${INSTANCE_ID}";
const consumer = client.getConsumer(instanceId, topic, groupId);
(async function(){
    // Cyclically consume messages.
    while(true) {
        try {
            // Consume messages in long polling mode.
            // In long polling mode, if no message in the topic is available for consumption, the
            // request is suspended on the broker for a specified period of time. If a message becomes
            // available for consumption within this period, the broker immediately sends a response to the
            // consumer. In this example, the period is set to 3 seconds.
            res = await consumer.consumeMessage(
                3, // The maximum number of messages that can be consumed at a time. In this example,
                // the value is set to 3. The maximum value that you can specify is 16.
                3 // The length of a long polling period. Unit: seconds. In this example, the value
                // is set to 3. The maximum value that you can specify is 30.
            );
            if (res.code == 200) {
                // Specify the message consumption logic.
                console.log("Consume Messages, requestId:%s", res.requestId);
                const handles = res.body.map((message) => {
                    console.log("\t\tMessageId:%s,Tag:%s,PublishTime:%d,NextConsumeTime:%d,FirstConsumeTime:%d,ConsumedTimes:%d,Body:%s" +
                        "\t\t,Props:%j,MessageKey:%s,Prop-A:%s",
                        message.MessageId, message.MessageTag, message.PublishTime, message.NextConsumeTime,
                        message.FirstConsumeTime, message.ConsumedTimes,
                        message.MessageBody, message.Properties, message.MessageKey, message.Properties.a);
                    return message.ReceiptHandle;
                });
                // If the broker does not receive an acknowledgment (ACK) for a message from the consumer
                // before the period of time specified by the message.NextConsumeTime parameter elapses

```

```

, the broker delivers the message for consumption again.
    // A unique timestamp is specified for the handle of a message each time the message is consumed.
    res = await consumer.ackMessage(handles);
    if (res.code !== 204) {
        // If the handle of a message times out, the broker cannot receive an ACK for the message from the consumer.
        console.log("Ack Message Fail:");
        const failHandles = res.body.map((error) => {
            console.log("\tErrorHandle:%s, Code:%s, Reason:%s\n", error.ReceiptHandle, error.ErrorCode, error.ErrorMessage);
            return error.ReceiptHandle;
        });
        handles.forEach((handle) => {
            if (failHandles.indexOf(handle) < 0) {
                console.log("\tSucHandle:%s\n", handle);
            }
        });
    } else {
        // Obtain an ACK from the consumer.
        console.log("Ack Message suc, RequestId:%s\n\t", res.requestId, handles.join(', '));
    };
}
}
} catch (e) {
    if (e.Code.indexOf("MessageNotExist") > -1) {
        // If no message in the topic is available for consumption, the long polling mode continues to take effect.
        console.log("Consume Message: no new message, RequestId:%s, Code:%s", e.RequestId, e.Code);
    } else {
        console.log(e);
    }
}
}
}
}() );

```

6.3.5.3. Send and consume ordered messages

Ordered messages are a type of message that is published and consumed in a strict order. Ordered messages in Message Queue for Apache RocketMQ are also known as first-in-first-out (FIFO) messages. This topic provides sample code to show how to use the HTTP client SDK for Node.js to send and consume ordered messages.

Background information

Ordered messages are classified into the following types:

- Globally ordered message: All messages in a specified topic are published and consumed in first-in-first-out (FIFO) order.
- Partitionally ordered message: All messages in a specified topic are distributed to different partitions by using shard keys. The messages in each partition are published and consumed in FIFO order. A

Sharding Key is a key field that is used for ordered messages to identify different partitions. The Sharding Key is different from the key of a normal message.

For more information about the message routing feature, see [Ordered messages](#).

Prerequisites

The following operations are performed:

- SDK for Node.js is installed. For more information about the message routing feature, see [Prepare the environment](#).
- Create resources that you want to specify in the code. For example, you must create the instance, topic, and group that you want to specify in the code in the Message Queue for Apache RocketMQ console in advance. For more information about the message routing feature, see [Create resources](#).

Send ordered messages

The following sample code provides an example on how to send ordered messages:

```

const {
  MQClient,
  MessageProperties
} = require('@aliyunmq/mq-http-sdk');
// The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log on to the
// Message Queue for Apache RocketMQ console. In the left-side navigation pane, click Instances.
// On the Instances page, select the name of your instance. Then, view the HTTP endpoint on the
// Network Management tab.
const endpoint = "${HTTP_ENDPOINT}";
// The AccessKey ID that is used for identity verification. You can obtain the AccessKey ID in the
// Apsara Uni-manager Operations Console.
const accessKeyId = "${ACCESS_KEY}";
// The AccessKey secret that is used for identity verification. You can obtain the AccessKey secret in the
// Apsara Uni-manager Operations Console.
const accessKeySecret = "${SECRET_KEY}";
var client = new MQClient(endpoint, accessKeyId, accessKeySecret);
// The topic to which you want to send messages. The topic is created in the Message Queue for
// Apache RocketMQ console.
const topic = "${TOPIC}";
// The ID of the instance to which the topic belongs. The instance is created in the Message Queue
// for Apache RocketMQ console.
// If the instance has a namespace, specify the ID of the instance. If the instance does not have a
// namespace, set the instance ID to null or an empty string. You can check whether your instance
// has a namespace on the Instances page in the RocketMQ console.
const instanceId = "${INSTANCE_ID}";
const producer = client.getProducer(instanceId, topic);
(async function(){
  try {
    // Cyclically send eight messages.
    for(var i = 0; i < 8; i++) {
      msgProps = new MessageProperties();
      // The custom property of the message.
      msgProps.putProperty("a", i);
      // The shard key that is used to distribute ordered messages to a specific partition.
      // Shard keys can be used to identify different partitions. A shard key is different from a message
      // key.
      msgProps.shardingKey(i % 2);
      res = await producer.publishMessage("hello mq.", "", msgProps);
      console.log("Publish message: MessageID:%s,BodyMD5:%s", res.body.MessageId, res.body.
        MessageBodyMD5);
    }
  } catch(e) {
    // Specify the logic that you want to use to resend or persist the message if the message
    // fails to be sent and needs to be sent again.
    console.log(e)
  }
})();

```

Consume ordered messages

The following sample code provides an example on how to consume ordered messages:

```

const {
  MQClient

```

```

MQClient,
} = require('@aliyunmq/mq-http-sdk');
// The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log on to the
// Message Queue for Apache RocketMQ console. In the left-side navigation pane, click Instances.
// On the Instances page, select the name of your instance. Then, view the HTTP endpoint on the
// Network Management tab.
const endpoint = "${HTTP_ENDPOINT}";
// The AccessKey ID that is used for identity verification. You can obtain the AccessKey ID in the
// Apsara Uni-manager Operations Console.
const accessKeyId = "${ACCESS_KEY}";
// The AccessKey secret that is used for identity verification. You can obtain the AccessKey secret
// in the Apsara Uni-manager Operations Console.
const accessKeySecret = "${SECRET_KEY}";
var client = new MQClient(endpoint, accessKeyId, accessKeySecret);
// The topic from which you want to consume messages. The topic is created in the Message Queue
// for Apache RocketMQ console.
const topic = "${TOPIC}";
// The ID of the group that you created in the Message Queue for Apache RocketMQ console.
const groupId = "GID_http";
// The ID of the instance to which the topic belongs. The instance is created in the Message Queue
// for Apache RocketMQ console.
// If the instance has a namespace, specify the ID of the instance. If the instance does not have a
// namespace, set the instance ID to null or an empty string. You can check whether your instance
// has a namespace on the Instances page in the RocketMQ console.
const instanceId = "${INSTANCE_ID}";
const consumer = client.getConsumer(instanceId, topic, groupId);
(async function(){
    // Cyclically consume messages.
    while(true) {
        try {
            // Consume messages in long polling mode. The consumer may pull partitionally ordered
            // messages from multiple partitions. The consumer consumes messages from the same partition
            // in the order in which the messages are sent.
            // A consumer pulls partitionally ordered messages from a partition. If the broker does
            // not receive an acknowledgment (ACK) for a message after the message is consumed, the
            // consumer consumes the message again.
            // The consumer can consume the next batch of messages from a partition only after all
            // messages that are pulled from the partition in the previous batch are acknowledged to
            // be consumed.
            // In long polling mode, if no message in the topic is available for consumption, the
            // request is suspended on the broker for a specified period of time. If a message becomes
            // available for consumption within this period, the broker immediately sends a response
            // to the consumer. In this example, the period is set to 3 seconds.
            res = await consumer.consumeMessageOrderly(
                3, // The maximum number of messages that can be consumed at a time. In this
                // example, the value is set to 3. The maximum value that you can specify is 16.
                3 // The length of a long polling period. Unit: seconds. In this example, the
                // value is set to 3. The maximum value that you can specify is 30.
            );
            if (res.code == 200) {
                // Specify the message consumption logic.
                console.log("Consume Messages, requestId:%s", res.requestId);
                const handles = res.body.map((message) => {
                    console.log("\tMessageId:%s, Tag:%s, PublishTime:%d, NextConsumeTime:%d, FirstConsumeTime:%d, ConsumedTimes:%d, Body:%s" +

```

```

        ", Props:%j, ShardingKey:%s, Prop-A:%s, Tag:%s",
        message.MessageId, message.MessageTag, message.PublishTime, message.NextConsumeTime, message.FirstConsumeTime, message.ConsumedTimes,
        message.MessageBody, message.Properties, message.ShardingKey, message.Properties.a, message.MessageTag);
        return message.ReceiptHandle;
    });
    // If the broker does not receive an ACK for a message from the consumer before the period of time specified by the message.NextConsumeTime parameter elapses, the broker delivers the message for consumption again.
    // A unique timestamp is specified for the handle of a message each time the message is consumed.
    res = await consumer.ackMessage(handles);
    if (res.code !== 204) {
        // If the handle of a message times out, the broker cannot receive an ACK for the message from the consumer.
        console.log("Ack Message Fail:");
        const failHandles = res.body.map((error) => {
            console.log("\tErrorHandle:%s, Code:%s, Reason:%s\n", error.ReceiptHandle, error.ErrorCode, error.ErrorMessage);
            return error.ReceiptHandle;
        });
        handles.forEach((handle) => {
            if (failHandles.indexOf(handle) < 0) {
                console.log("\tSucHandle:%s\n", handle);
            }
        });
    } else {
        // Obtain an ACK from the consumer.
        console.log("Ack Message suc, RequestId:%s\n\t", res.requestId, handles.join(','));
    };
    }
}
} catch (e) {
    if (e.Code.indexOf("MessageNotExist") > -1) {
        // If no message in the topic is available for consumption, the long polling mode continues to take effect.
        console.log("Consume Message: no new message, RequestId:%s, Code:%s", e.RequestId, e.Code);
    } else {
        console.log(e);
    }
}
}
}() );

```

6.3.5.4. Send and consume scheduled messages and delayed messages

This topic provides sample code to show how to use the HTTP client SDK for Node.js to send and consume scheduled messages and delayed messages.

Background information

- **Delayed message:** The producer sends the message to the Message Queue for Apache RocketMQ server, but does not expect the message to be delivered immediately. Instead, the message is delivered to the consumer for consumption after a certain period of time. This message is a delayed message.
- **Scheduled message:** A producer sends a message to a Message Queue for Apache RocketMQ broker and expects the message to be delivered to a consumer at a specified point in time. This type of message is called a scheduled message.

If an HTTP client SDK is used, the code configurations of scheduled messages are the same as the code configurations of delayed messages. Both types of messages are delivered to consumers after a specific period of time based on the attributes of the messages.

For more information about the message routing feature, see [Scheduled messages and delayed messages](#).

Prerequisites

The following operations are performed:

- SDK for Node.js is installed. For more information about the message routing feature, see [Prepare the environment](#).
- Create resources that you want to specify in the code. For example, you must create the instance, topic, and group that you want to specify in the code in the Message Queue for Apache RocketMQ console in advance. For more information about the message routing feature, see [Create resources](#).

Send scheduled messages or delayed messages

The following sample code provides an example on how to send scheduled messages or delayed messages:

```

const {
  MQClient,
  MessageProperties
} = require('@aliyunmq/mq-http-sdk');
// The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log on to the
// Message Queue for Apache RocketMQ console. In the left-side navigation pane, click Instances.
// On the Instances page, select the name of your instance. Then, view the HTTP endpoint on the
// Network Management tab.
const endpoint = "${HTTP_ENDPOINT}";
// The AccessKey ID that is used for identity verification. You can obtain the AccessKey ID in the
// Apsara Uni-manager Operations Console.
const accessKeyId = "${ACCESS_KEY}";
// The AccessKey secret that is used for identity verification. You can obtain the AccessKey secret
// in the Apsara Uni-manager Operations Console.
const accessKeySecret = "${SECRET_KEY}";
var client = new MQClient(endpoint, accessKeyId, accessKeySecret);
// The topic to which you want to send messages. The topic is created in the Message Queue for
// Apache RocketMQ console.
const topic = "${TOPIC}";
// The ID of the instance to which the topic belongs. The instance is created in the Message Queue
// for Apache RocketMQ console.
// If the instance has a namespace, specify the ID of the instance. If the instance does not have a
// namespace, set the instance ID to null or an empty string. You can check whether your instance
// has a namespace on the Instances page in the RocketMQ console.
const instanceId = "${INSTANCE_ID}";
const producer = client.getProducer(instanceId, topic);
(async function(){
  try {
    // Cyclically send four messages.
    for(var i = 0; i < 4; i++) {
      let res;
      msgProps = new MessageProperties();
      // The custom property of the message.
      msgProps.putProperty("a", i);
      // The key of the message.
      msgProps.messageKey("MessageKey");
      // The period of time after which the broker delivers the message. In this example, when the
      // broker receives a message, the broker waits for 10 seconds before it delivers the message to
      // the consumer. Set this parameter to a timestamp in milliseconds.
      // If the producer sends a scheduled message, set the parameter to the time interval between
      // the scheduled point in time and the current point in time.
      msgProps.startDeliverTime(Date.now() + 10 * 1000);
      res = await producer.publishMessage("hello mq. timer msg!", "TagA", msgProps);
      console.log("Publish message: MessageID:%s,BodyMD5:%s", res.body.MessageId, res.body.MessageBodyMD5);
    }
  } catch(e) {
    // Specify the logic that you want to use to resend or persist the message if the message fails
    // to be sent and needs to be sent again.
    console.log(e)
  }
})();

```

Consume scheduled messages or delayed messages

The following sample code provides an example on how to consume scheduled messages or delayed messages:

```
const {
  MQClient
} = require('@aliyunmq/mq-http-sdk');
// The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log on to the
// Message Queue for Apache RocketMQ console. In the left-side navigation pane, click Instances.
// On the Instances page, select the name of your instance. Then, view the HTTP endpoint on the
// Network Management tab.
const endpoint = "${HTTP_ENDPOINT}";
// The AccessKey ID that is used for identity verification. You can obtain the AccessKey ID in the
// Apsara Uni-manager Operations Console.
const accessKeyId = "${ACCESS_KEY}";
// The AccessKey secret that is used for identity verification. You can obtain the AccessKey secret
// in the Apsara Uni-manager Operations Console.
const accessKeySecret = "${SECRET_KEY}";
var client = new MQClient(endpoint, accessKeyId, accessKeySecret);
// The topic from which you want to consume messages. The topic is created in the Message Queue
// for Apache RocketMQ console.
const topic = "${TOPIC}";
// The ID of the group that you created in the Message Queue for Apache RocketMQ console.
const groupId = "${GROUP_ID}";
// The ID of the instance to which the topic belongs. The instance is created in the Message Queue
// for Apache RocketMQ console.
// If the instance has a namespace, specify the ID of the instance. If the instance does not have a
// namespace, set the instance ID to null or an empty string. You can check whether your instance
// has a namespace on the Instances page in the RocketMQ console.
const instanceId = "${INSTANCE_ID}";
const consumer = client.getConsumer(instanceId, topic, groupId);
(async function(){
  // Cyclically consume messages.
  while(true) {
    try {
      // Consume messages in long polling mode.
      // In long polling mode, if no message in the topic is available for consumption, the request
      // is suspended on the broker for a specified period of time. If a message becomes available
      // for consumption within this period, the broker immediately sends a response to the consumer.
      // In this example, the period is set to 3 seconds.
      res = await consumer.consumeMessage(
        3, // The maximum number of messages that can be consumed at a time. In this example, the
        // value is set to 3. The maximum value that you can specify is 16.
        3 // The length of a long polling period. Unit: seconds. In this example, the value is
        // set to 3. The maximum value that you can specify is 30.
      );
      if (res.code == 200) {
        // Specify the message consumption logic.
        console.log("Consume Messages, requestId:%s", res.requestId);
        const handles = res.body.map((message) => {
          console.log("\t\tMessageId:%s, Tag:%s, PublishTime:%d, NextConsumeTime:%d, FirstConsumeTime:%d, ConsumedTimes:%d, Body:%s" +
            "\n\t\tProps:%j, MessageKey:%s, Prop-A:%s",
            message.MessageId, message.MessageTag, message.PublishTime, message.NextConsumeTime, message.FirstConsumeTime, message.ConsumedTimes, message.Body, message.MessageKey, message.PropA);
        });
      }
    } catch (e) {
      console.error(e);
    }
  }
})();
```

```

        message.MessageId, message.MessageTag, message.PublishTime, message.NextConsumeTime, message.FirstConsumeTime, message.ConsumedTimes,
        message.MessageBody, message.Properties, message.MessageKey, message.Properties.
    a);

    return message.ReceiptHandle;
});
// If the broker does not receive an acknowledgment (ACK) for a message from the consumer before the period of time specified by the message.NextConsumeTime parameter elapses, the broker delivers the message for consumption again.
// A unique timestamp is specified for the handle of a message each time the message is consumed.
res = await consumer.ackMessage(handles);
if (res.code !== 204) {
    // If the handle of a message times out, the broker cannot receive an ACK for the message from the consumer.
    console.log("Ack Message Fail:");
    const failHandles = res.body.map((error) => {
        console.log("\tErrorHandle:%s, Code:%s, Reason:%s\n", error.ReceiptHandle, error.ErrorCode, error.ErrorMessage);
        return error.ReceiptHandle;
    });
    handles.forEach((handle) => {
        if (failHandles.indexOf(handle) < 0) {
            console.log("\tSucHandle:%s\n", handle);
        }
    });
} else {
    // Obtain an ACK from the consumer.
    console.log("Ack Message suc, RequestId:%s\n\t", res.requestId, handles.join(','));
};

}
}
} catch (e) {
    if (e.Code.indexOf("MessageNotExist") > -1) {
        // If no message in the topic is available for consumption, the long polling mode continues to take effect.
        console.log("Consume Message: no new message, RequestId:%s, Code:%s", e.RequestId, e.Code);
    } else {
        console.log(e);
    }
}
}
}() );

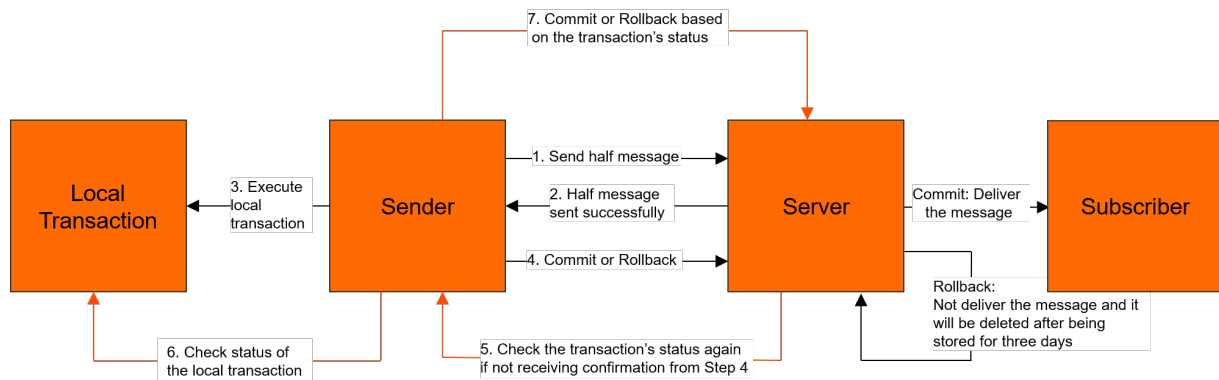
```

6.3.5.5. Send and consume transactional messages

Message Queue for Apache RocketMQ provides a distributed transaction processing feature that is similar to X/Open XA. Message Queue for Apache RocketMQ uses transactional messages to ensure transactional consistency. This topic provides sample code to show how to use the HTTP client SDK for Node.js to send and consume transactional messages.

Background information

The following figure shows the interaction process of transactional messages.



For more information about the message routing feature, see [Transactional messages](#).

Prerequisites

The following operations are performed:

- SDK for Node.js is installed. For more information about the message routing feature, see [Prepare the environment](#).
- Create resources that you want to specify in the code. For example, you must create the instance, topic, and group that you want to specify in the code in the Message Queue for Apache RocketMQ console in advance. For more information about the message routing feature, see [Create resources](#).

Send transactional messages

The following sample code provides an example on how to send transactional messages:

```

const {
  MQClient,
  MessageProperties
} = require('@aliyunmq/mq-http-sdk');
// The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log on to the
// Message Queue for Apache RocketMQ console. In the left-side navigation pane, click Instances.
// On the Instances page, select the name of your instance. Then, view the HTTP endpoint on the
// Network Management tab.
const endpoint = "${HTTP_ENDPOINT}";
// The AccessKey ID that is used for identity verification. You can obtain the AccessKey ID in the
// Apsara Uni-manager Operations Console.
const accessKeyId = "${ACCESS_KEY}";
// The AccessKey secret that is used for identity verification. You can obtain the AccessKey secret
// in the Apsara Uni-manager Operations Console.
const accessKeySecret = "${SECRET_KEY}";
var client = new MQClient(endpoint, accessKeyId, accessKeySecret);
// The topic to which you want to send messages. The topic is created in the Message Queue for
// Apache RocketMQ console.
const topic = "${TOPIC}";
// The ID of the group that you created in the Message Queue for Apache RocketMQ console.
const groupId = "${GROUP_ID}";
// The ID of the instance to which the topic belongs. The instance is created in the Message Queue
// for Apache RocketMQ console.
// If the instance has a namespace, specify the ID of the instance. If the instance does not
// have a namespace, set the instance ID to null or an empty string. You can check whether y
  
```

```

our instance has a namespace on the Instances page in the RocketMQ console.
const instanceId = "${INSTANCE_ID}";
const mqTransProducer = client.getTransProducer(instanceId, topic, groupId);
async function processTransResult(res, msgId) {
    if (!res) {
        return;
    }
    if (res.code !== 204) {
        // If a transactional message is not committed or rolled back before the timeout period specified by the TransCheckImmunityTime parameter for the handle of the transactional message elapses or before the timeout period specified for the handle of consumeHalfMessage elapses, the commit or rollback operation fails. In this example, the timeout period for the handle of consumeHalfMessage is 10 seconds.
        console.log("Commit/Rollback Message Fail:");
        const failHandles = res.body.map((error) => {
            console.log("\tErrorHandle:%s, Code:%s, Reason:%s\n", error.ReceiptHandle, error.ErrorCode, error.ErrorMessage);
            return error.ReceiptHandle;
        });
    } else {
        console.log("Commit/Rollback Message suc!!! %s", msgId);
    }
}
}
var halfMessageCount = 0;
var halfMessageConsumeCount = 0;
(async function(){
    try {
        // Cyclically send four transactional messages.
        for(var i = 0; i < 4; i++) {
            let res;
            msgProps = new MessageProperties();
            // The custom property of the message.
            msgProps.putProperty("a", i);
            // The key of the message.
            msgProps.messageKey("MessageKey");
            // The time interval between the time when the transactional message is sent and the start time of the first transaction status check. Unit: seconds. Valid values: 10 to 300.
            // If the message is not committed or rolled back after the first transaction status check is performed, the broker initiates a request to check the status of the local transaction at an interval of 10 seconds within the next 24 hours.
            msgProps.transCheckImmunityTime(10);
            res = await mqTransProducer.publishMessage("hello mq.", "tagA", msgProps);
            console.log("Publish message: MessageID:%s,BodyMD5:%s,Handle:%s", res.body.MessageId, res.body.MessageBodyMD5, res.body.ReceiptHandle);
            if (res && i == 0) {
                // After the producer sends the transactional message, the broker obtains the handle of the half message that corresponds to the transactional message and commits or rolls back the transactional message based on the status of the handle.
                const msgId = res.body.MessageId;
                res = await mqTransProducer.commit(res.body.ReceiptHandle);
                console.log("Commit msg when publish, %s", msgId);
                // If the transactional message is not committed or rolled back before the timeout period specified by the TransCheckImmunityTime parameter elapses, the commit or rollback operation fails.

```

```

        processTransResult(res, msgId);
    }
}
} catch(e) {
    // Specify the logic that you want to use to resend or persist the message if the message fails to be sent and needs to be sent again.
    console.log(e)
}
})();
// The client needs a thread or a process to process unacknowledged transactional messages.

// Process unacknowledged transactional messages.
(async function() {
    // Cyclically check the status of half messages. This process is similar to consuming normal messages.
    while(halfMessageCount < 3 && halfMessageConsumeCount < 15) {
        try {
            halfMessageConsumeCount++;
            res = await mqTransProducer.consumeHalfMessage(3, 3);
            if (res.code == 200) {
                // Specify the message consumption logic.
                console.log("Consume Messages, requestId:%s", res.requestId);
                res.body.forEach(async (message) => {
                    console.log("\tMessageId:%s,Tag:%s,PublishTime:%d,NextConsumeTime:%d,FirstConsumeTime:%d,ConsumedTimes:%d,Body:%s" +
                        ",Props:%j,MessageKey:%s,Prop-A:%s",
                        message.MessageId, message.MessageTag, message.PublishTime, message.NextConsumeTime, message.FirstConsumeTime, message.ConsumedTimes,
                        message.MessageBody,message.Properties,message.MessageKey,message.Properties.a);
                    var propA = message.Properties && message.Properties.a ? parseInt(message.Properties.a) : 0;

                    var opResp;
                    if (propA == 1 || (propA == 2 && message.ConsumedTimes > 1)) {
                        opResp = await mqTransProducer.commit(message.ReceiptHandle);
                        console.log("Commit msg when check half, %s", message.MessageId);
                        halfMessageCount++;
                    } else if (propA == 3) {
                        opResp = await mqTransProducer.rollback(message.ReceiptHandle);
                        console.log("Rollback msg when check half, %s", message.MessageId);
                        halfMessageCount++;
                    }
                    processTransResult(opResp, message.MessageId);
                });
            }
        } catch(e) {
            if (e.Code && e.Code.indexOf("MessageNotExist") > -1) {
                // If no message in the topic is available for consumption, the long polling mode continues to take effect.
                console.log("Consume Transaction Half msg: no new message, RequestId:%s, Code:%s", e.RequestId, e.Code);
            } else {
                console.log(e);
            }
        }
    }
},

```

```

    }
  }
}() ;

```

Consume transactional messages

The following sample code provides an example on how to consume transactional messages:

```

const {
  MQClient
} = require('@aliyunmq/mq-http-sdk');
// The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log on to the
// Message Queue for Apache RocketMQ console. In the left-side navigation pane, click Instances.
// On the Instances page, select the name of your instance. Then, view the HTTP endpoint on the
// Network Management tab.
const endpoint = "${HTTP_ENDPOINT}";
// The AccessKey ID that is used for identity verification. You can obtain the AccessKey ID in the
// Apsara Uni-manager Operations Console.
const accessKeyId = "${ACCESS_KEY}";
// The AccessKey secret that is used for identity verification. You can obtain the AccessKey secret
// in the Apsara Uni-manager Operations Console.
const accessKeySecret = "${SECRET_KEY}";
var client = new MQClient(endpoint, accessKeyId, accessKeySecret);
// The topic from which you want to consume messages. The topic is created in the Message Queue
// for Apache RocketMQ console.
const topic = "${TOPIC}";
// The ID of the group that you created in the Message Queue for Apache RocketMQ console.
const groupId = "${GROUP_ID}";
// The ID of the instance to which the topic belongs. The instance is created in the Message Queue
// for Apache RocketMQ console.
// If the instance has a namespace, specify the ID of the instance. If the instance does not have a
// namespace, set the instance ID to null or an empty string. You can check whether your instance
// has a namespace on the Instances page in the RocketMQ console.
const instanceId = "${INSTANCE_ID}";
const consumer = client.getConsumer(instanceId, topic, groupId);
(async function(){
  // Cyclically consume messages.
  while(true) {
    try {
      // Consume messages in long polling mode.
      // In long polling mode, if no message in the topic is available for consumption, the request
      // is suspended on the broker for a specified period of time. If a message becomes available
      // for consumption within this period, the broker immediately sends a response to the consumer.
      // In this example, the period is set to 3 seconds.
      res = await consumer.consumeMessage(
        3, // The maximum number of messages that can be consumed at a time. In this example, the
        // value is set to 3. The maximum value that you can specify is 16.
        3 // The length of a long polling period. Unit: seconds. In this example, the value is
        // set to 3. The maximum value that you can specify is 30.
      );
      if (res.code == 200) {
        // Specify the message consumption logic.
        console.log("Consume Messages, requestId:%s", res.requestId);
        const handles = res.body.map((message) => {

```

```

        console.log("\tMessageId:%s,Tag:%s,PublishTime:%d,NextConsumeTime:%d,FirstConsumeTime:%d,ConsumedTimes:%d,Body:%s" +
            ",Props:%j,MessageKey:%s,Prop-A:%s",
            message.MessageId, message.MessageTag, message.PublishTime, message.NextConsumeTime, message.FirstConsumeTime, message.ConsumedTimes,
            message.MessageBody,message.Properties,message.MessageKey,message.Properties.a);

        return message.ReceiptHandle;
    });
    // If the broker does not receive an acknowledgment (ACK) for a message from the consumer before the period of time specified by the message.NextConsumeTime parameter elapses, the broker delivers the message for consumption again.
    // A unique timestamp is specified for the handle of a message each time the message is consumed.
    res = await consumer.ackMessage(handles);
    if (res.code !== 204) {
        // If the handle of a message times out, the broker cannot receive an ACK for the message from the consumer.
        console.log("Ack Message Fail:");
        const failHandles = res.body.map((error)=>{
            console.log("\tErrorHandle:%s, Code:%s, Reason:%s\n", error.ReceiptHandle, error.ErrorCode, error.ErrorMessage);
            return error.ReceiptHandle;
        });
        handles.forEach((handle)=>{
            if (failHandles.indexOf(handle) < 0) {
                console.log("\tSucHandle:%s\n", handle);
            }
        });
    } else {
        // Obtain an ACK from the consumer.
        console.log("Ack Message suc, RequestId:%s\n\t", res.requestId, handles.join(','));
    };
}
}
} catch(e) {
    if (e.Code.indexOf("MessageNotExist") > -1) {
        // If no message in the topic is available for consumption, the long polling mode continues to take effect.
        console.log("Consume Message: no new message, RequestId:%s, Code:%s", e.RequestId, e.Code);
    } else {
        console.log(e);
    }
}
}
}() );

```

6.3.6. PHP SDK

6.3.6.1. Prepare the environment

This topic describes how to prepare the environment before you use the HTTP client SDK for PHP to send and consume messages.

Environment requirements

- PHP 5.5.0 or later is installed. For more information, see [Install PHP](#).
- Composer is installed. For more information, see [Install Composer](#).

After PHP is installed, you can run the `php -v` command to view the version of PHP that you installed.

Install the SDK for PHP

To install the SDK for PHP, perform the following steps:

1. Add the following dependency to the `composer.json` file in your PHP installation directory:

```
{
  "require": {
    "aliyunmq/mq-http-sdk": ">=1.0.3"
  }
}
```

2. Run the following command to use Composer to install the SDK for PHP:

```
composer install
```

6.3.6.2. Send and consume normal messages

Normal messages are messages that have no special features in Message Queue for Apache RocketMQ. They are different from featured messages, such as scheduled messages, delayed messages, ordered messages, and transactional messages. This topic provides sample code to show how to use the HTTP client SDK for PHP to send and consume normal messages.

Prerequisites

The following operations are performed:

- Install the SDK for PHP. For more information about the message routing feature, see [Prepare the environment](#).
- Create resources that you want to specify in the code. For example, you must create the instance, topic, and group that you want to specify in the code in the Message Queue for Apache RocketMQ console in advance. For more information about the message routing feature, see [Create resources](#).

Send normal messages

The following sample code provides an example on how to send normal messages:

```
<?php
require "vendor/autoload.php";
use MQ\Model\TopicMessage;
use MQ\MQClient;
class ProducerTest
{
    private $client;
    private $producer;
```

```

public function __construct()
{
    $this->client = new MQClient(
        // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint,
        log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane,
        click Instances. On the Instances page, select the name of your instance. Then, view the HT
        TP endpoint on the Network Management tab.
        "${HTTP_ENDPOINT}",
        // The AccessKey ID that is used for identity verification. You can obtain the
        AccessKey ID in the Apsara Uni-manager Operations Console.
        "${ACCESS_KEY}",
        // The AccessKey secret that is used for identity verification. You can obtain
        the AccessKey secret in the Apsara Uni-manager Operations Console.
        "${SECRET_KEY}"
    );
    // The topic to which you want to send messages. The topic is created in the Messag
    e Queue for Apache RocketMQ console.
    $topic = "${TOPIC}";
    // The ID of the instance to which the topic belongs. The instance is created in th
    e Message Queue for Apache RocketMQ console.
    // If the instance has a namespace, specify the ID of the instance. If the instance
    does not have a namespace, set the instance ID to null or an empty string. You can check wh
    ether your instance has a namespace on the Instances page in the RocketMQ console.
    $instanceId = "${INSTANCE_ID}";
    $this->producer = $this->client->getProducer($instanceId, $topic);
}
public function run()
{
    try
    {
        for ($i=1; $i<=4; $i++)
        {
            $publishMessage = new TopicMessage(
                // The content of the message.
                "hello mq!"
            );
            // The custom property of the message.
            $publishMessage->putProperty("a", $i);
            // The key of the message.
            $publishMessage->setMessageKey("MessageKey");
            $result = $this->producer->publishMessage($publishMessage);
            print "Send mq message success. msgId is:" . $result->getMessageId() . ", b
            odyMD5 is:" . $result->getMessageBodyMD5() . "\n";
        }
    } catch (\Exception $e) {
        print_r($e->getMessage() . "\n");
    }
}
}
$instance = new ProducerTest();
$instance->run();
?>

```

Consume normal messages

The following sample code provides an example on how to consume normal messages:

```
<?php
require "vendor/autoload.php";
use MQ\Model\TopicMessage;
use MQ\MQClient;
class ConsumerTest
{
    private $client;
    private $producer;
    public function __construct()
    {
        $this->client = new MQClient(
            // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint,
            // log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane,
            // click Instances. On the Instances page, select the name of your instance. Then, view the HT
            // TP endpoint on the Network Management tab.
            "${HTTP_ENDPOINT}",
            // The AccessKey ID that is used for identity verification. You can obtain the
            // AccessKey ID in the Apsara Uni-manager Operations Console.
            "${ACCESS_KEY}",
            // The AccessKey secret that is used for identity verification. You can obtain
            // the AccessKey secret in the Apsara Uni-manager Operations Console.
            "${SECRET_KEY}"
        );
        // The topic from which you want to consume messages. The topic is created in the M
        // essage Queue for Apache RocketMQ console.
        $topic = "${TOPIC}";
        // The ID of the group that you created in the Message Queue for Apache RocketMQ co
        // nsole.
        $groupId = "${GROUP_ID}";
        // The ID of the instance to which the topic belongs. The instance is created in th
        // e Message Queue for Apache RocketMQ console.
        // If the instance has a namespace, specify the ID of the instance. If the instance
        // does not have a namespace, set the instance ID to null or an empty string. You can check wh
        // ether your instance has a namespace on the Instances page in the RocketMQ console.
        $instanceId = "${INSTANCE_ID}";
        $this->consumer = $this->client->getConsumer($instanceId, $topic, $groupId);
    }
    public function run()
    {
        // Cyclically consume messages in the current thread. We recommend that you use mul
        // tiple threads to concurrently consume messages.
        while (True) {
            try {
                // Consume messages in long polling mode.
                // In long polling mode, if no message in the topic is available for consum
                // ption, the request is suspended on the broker for a specified period of time. If a message
                // becomes available for consumption within this period, the broker immediately sends a respon
                // se to the consumer. In this example, the period is set to 3 seconds.
                $messages = $this->consumer->consumeMessage(
                    3, // The maximum number of messages that can be consumed at a time. In
                    // this example, the value is set to 3. The maximum value that you can specify is 16.
```

```

In this example, the value is set to 3. The maximum value that you can specify is 10.
    3 // The length of a long polling period. Unit: seconds. In this example,
    the value is set to 3. The maximum value that you can specify is 30.
    );
    } catch (\Exception $e) {
        if ($e instanceof MQ\Exception\MessageNotExistException) {
            // If no message in the topic is available for consumption, the long polling
            mode continues to take effect.
            printf("No message, continue long polling!RequestId:%s\n", $e->getRequestId());

            continue;
        }
        print_r($e->getMessage() . "\n");
        sleep(3);
        continue;
    }
    print "consume finish, messages:\n";
    // Specify the message consumption logic.
    $receiptHandles = array();
    foreach ($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());
    }
    // If the broker does not receive an acknowledgment (ACK) for a message from the
    consumer before the period of time specified by $message->getNextConsumeTime() elapses, the
    broker delivers the message for consumption again.
    // A unique timestamp is specified for the handle of a message each time the message
    is consumed.
    print_r($receiptHandles);
    try {
        $this->consumer->ackMessage($receiptHandles);
    } catch (\Exception $e) {
        if ($e instanceof MQ\Exception\AckMessageException) {
            // If the handle of a message times out, the broker cannot receive an ACK
            for the message from the consumer.
            printf("Ack Error, RequestId:%s\n", $e->getRequestId());
            foreach ($e->getAckMessageErrorItems() as $errorItem) {
                printf("\tReceiptHandle:%s, ErrorCode:%s, ErrorMessage:%s\n", $errorItem->
                getReceiptHandle(), $errorItem->getErrorCode(), $errorItem->getErrorMessage());
            }
        }
    }
    print "ack finish\n";
}
}

$instance = new ConsumerTest();
$instance->run();

```

```
?>
```

6.3.6.3. Send and consume ordered messages

Ordered messages are a type of message that is published and consumed in a strict order. Ordered messages in Message Queue for Apache RocketMQ are also known as first-in-first-out (FIFO) messages. This topic provides sample code to show how to use the HTTP client SDK for PHP to send and consume ordered messages.

Background information

Ordered messages are classified into the following types:

- Globally ordered message: All messages in a specified topic are published and consumed in first-in-first-out (FIFO) order.
- Partitionally ordered message: All messages in a specified topic are distributed to different partitions by using shard keys. The messages in each partition are published and consumed in FIFO order. A Sharding Key is a key field that is used for ordered messages to identify different partitions. The Sharding Key is different from the key of a normal message.

For more information about the message routing feature, see [Ordered messages](#).

Prerequisites

The following operations are performed:

- Install the SDK for PHP. For more information about the message routing feature, see [Prepare the environment](#).
- Create resources that you want to specify in the code. For example, you must create the instance, topic, and group that you want to specify in the code in the Message Queue for Apache RocketMQ console in advance. For more information about the message routing feature, see [Create resources](#).

Send ordered messages

The following sample code provides an example on how to send ordered messages:

```
<?php
require "vendor/autoload.php";
use MQ\Model\TopicMessage;
use MQ\MQClient;
class ProducerTest
{
    private $client;
    private $producer;
    public function __construct()
    {
        $this->client = new MQClient(
            // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint,
            log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane,
            click Instances. On the Instances page, select the name of your instance. Then, view the HT
            TP endpoint on the Network Management tab.
            "${HTTP_ENDPOINT}",
            // The AccessKey ID that is used for identity verification. You can obtain the
            AccessKey ID in the Apsara Uni-manager Operations Console.
        );
    }
}
```

```

        "${ACCESS_KEY}",
        // The AccessKey secret that is used for identity verification. You can obtain
        the AccessKey secret in the Apsara Uni-manager Operations Console.
        "${SECRET_KEY}"
    );
    // The topic to which you want to send messages. The topic is created in the Message Queue for Apache RocketMQ console.
    $topic = "${TOPIC}";
    // The ID of the instance to which the topic belongs. The instance is created in the Message Queue for Apache RocketMQ console.
    // If the instance has a namespace, specify the ID of the instance. If the instance does not have a namespace, set the instance ID to null or an empty string. You can check whether your instance has a namespace on the Instances page in the RocketMQ console.
    $instanceId = "${INSTANCE_ID}";
    $this->producer = $this->client->getProducer($instanceId, $topic);
}
public function run()
{
    try
    {
        for ($i=1; $i<=4; $i++)
        {
            $publishMessage = new TopicMessage(
                "hello mq!" // The content of the message.
            );
            // The custom property of the message.
            $publishMessage->putProperty("a", $i);
            // The shard key that is used to distribute ordered messages to a specific partition. Shard keys can be used to identify different partitions. A shard key is different from a message key.
            $publishMessage->setShardingKey($i % 2);
            $result = $this->producer->publishMessage($publishMessage);
            print "Send mq message success. msgId is:" . $result->getMessageId() . ", bodyMD5 is:" . $result->getMessageBodyMD5() . "\n";
        }
    } catch (\Exception $e) {
        print_r($e->getMessage() . "\n");
    }
}
$instance = new ProducerTest();
$instance->run();
?>

```

Consume ordered messages

The following sample code provides an example on how to consume ordered messages:

```

<?php
require "vendor/autoload.php";
use MQ\Model\TopicMessage;
use MQ\MQClient;
class ConsumerTest

```

```

{
    private $client;
    private $producer;
    public function __construct()
    {
        $this->client = new MQClient(
            // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint,
            log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane,
            click Instances. On the Instances page, select the name of your instance. Then, view the HT
            TP endpoint on the Network Management tab.
            "${HTTP_ENDPOINT}",
            // The AccessKey ID that is used for identity verification. You can obtain the
            AccessKey ID in the Apsara Uni-manager Operations Console.
            "${ACCESS_KEY}",
            // The AccessKey secret that is used for identity verification. You can obtain
            the AccessKey secret in the Apsara Uni-manager Operations Console.
            "${SECRET_KEY}"
        );
        // The topic from which you want to consume messages. The topic is created in the M
        essage Queue for Apache RocketMQ console.
        $topic = "${TOPIC}";
        // The ID of the group that you created in the Message Queue for Apache RocketMQ co
        nsole.
        $groupId = "${GROUP_ID}";
        // The ID of the instance to which the topic belongs. The instance is created in th
        e Message Queue for Apache RocketMQ console.
        // If the instance has a namespace, specify the ID of the instance. If the instance
        does not have a namespace, set the instance ID to null or an empty string. You can check wh
        ether your instance has a namespace on the Instances page in the RocketMQ console.
        $instanceId = "${INSTANCE_ID}";
        $this->consumer = $this->client->getConsumer($instanceId, $topic, $groupId);
    }
    public function run()
    {
        // Cyclically consume messages in the current thread. We recommend that you use mul
        tiple threads to concurrently consume messages.
        while (True) {
            try {
                // Consume messages in long polling mode. The consumer may pull partitiona
                lly ordered messages from multiple partitions. The consumer consumes messages from the same
                partition in the order in which the messages are sent.
                // A consumer pulls partitionally ordered messages from a partition. If the
                broker does not receive an acknowledgment (ACK) for a message after the message is consumed
                , the consumer consumes the message again.
                // The consumer can consume the next batch of messages from a partition onl
                y after all messages that are pulled from the partition in the previous batch are acknowl
                edged to be consumed.
                // In long polling mode, if no message in the topic is available for consum
                ption, the request is suspended on the broker for a specified period of time. If a message
                becomes available for consumption within this period, the broker immediately sends a respon
                se to the consumer. In this example, the period is set to 3 seconds.
                $messages = $this->consumer->consumeMessageOrderly(
                    3, // The maximum number of messages that can be consumed at a time. In
                    this example, the value is set to 3. The maximum value that you can specify is 16.

```

```

        3 // The length of a long polling period. Unit: seconds. In this example, the value is set to 3. The maximum value that you can specify is 30.
    );
    } catch (\Exception $e) {
        if ($e instanceof MQ\Exception\MessageNotExistException) {
            // If no message in the topic is available for consumption, the long polling mode continues to take effect.
            printf("No message, continue long polling!RequestId:%s\n", $e->getRequestId());

            continue;
        }
        print_r($e->getMessage() . "\n");
        sleep(3);
        continue;
    }
    print "=====>consume finish, messages:\n";
    // Specify the message consumption logic.
    $receiptHandles = array();
    foreach ($messages as $message) {
        $receiptHandles[] = $message->getReceiptHandle();
        printf("MessageID:%s TAG:%s BODY:%s \nPublishTime:%d, FirstConsumeTime:%d, \nConsumedTimes:%d, NextConsumeTime:%d,ShardingKey:%s\n",
            $message->getMessageId(), $message->getMessageTag(), $message->getMessageBody(),
            $message->getPublishTime(), $message->getFirstConsumeTime(), $message->getConsumedTimes(), $message->getNextConsumeTime(),
            $message->getShardingKey());
        print_r($message->getProperties());
    }
    // If the broker does not receive an ACK for a message from the consumer before the period of time specified by $message->getNextConsumeTime() elapses, the broker delivers the message for consumption again.
    // A unique timestamp is specified for the handle of a message each time the message is consumed.
    print_r($receiptHandles);
    try {
        $this->consumer->ackMessage($receiptHandles);
    } catch (\Exception $e) {
        if ($e instanceof MQ\Exception\AckMessageException) {
            // If the handle of a message times out, the broker cannot receive an ACK for the message from the consumer.
            printf("Ack Error, RequestId:%s\n", $e->getRequestId());
            foreach ($e->getAckMessageErrorItems() as $errorItem) {
                printf("\tReceiptHandle:%s, ErrorCode:%s, ErrorMessage:%s\n", $errorItem->getReceiptHandle(), $errorItem->getErrorCode(), $errorItem->getErrorMessage());
            }
        }
    }
    print "=====>ack finish\n";
}
}

$instance = new ConsumerTest();
$instance->run();

```

```
?>
```

6.3.6.4. Send and consume scheduled messages and delayed messages

This topic provides sample code to show how to use the HTTP client SDK for PHP to send and consume scheduled messages and delayed messages.

Background information

- **Delayed message:** The producer sends the message to the Message Queue for Apache RocketMQ server, but does not expect the message to be delivered immediately. Instead, the message is delivered to the consumer for consumption after a certain period of time. This message is a delayed message.
- **Scheduled message:** A producer sends a message to a Message Queue for Apache RocketMQ broker and expects the message to be delivered to a consumer at a specified point in time. This type of message is called a scheduled message.

If an HTTP client SDK is used, the code configurations of scheduled messages are the same as the code configurations of delayed messages. Both types of messages are delivered to consumers after a specific period of time based on the attributes of the messages.

For more information about the message routing feature, see [Scheduled messages and delayed messages](#).

Prerequisites

The following operations are performed:

- Install the SDK for PHP. For more information about the message routing feature, see [Prepare the environment](#).
- Create resources that you want to specify in the code. For example, you must create the instance, topic, and group that you want to specify in the code in the Message Queue for Apache RocketMQ console in advance. For more information about the message routing feature, see [Create resources](#).

Send scheduled messages or delayed messages

The following sample code provides an example on how to send scheduled messages or delayed messages:

```
<?php
require "vendor/autoload.php";
use MQ\Model\TopicMessage;
use MQ\MQClient;
class ProducerTest
{
    private $client;
    private $producer;
    public function __construct()
    {
        $this->client = new MQClient(
            // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint,
            log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane,
```

click Instances. On the Instances page, select the name of your instance. Then, view the HTTP endpoint on the Network Management tab.

```

    "${HTTP_ENDPOINT}",
    // The AccessKey ID that is used for identity verification. You can obtain the
    AccessKey ID in the Apsara Uni-manager Operations Console.
    "${ACCESS_KEY}",
    // The AccessKey secret that is used for identity verification. You can obtain
    the AccessKey secret in the Apsara Uni-manager Operations Console.
    "${SECRET_KEY}"
);
// The topic to which you want to send messages. The topic is created in the Message
Queue for Apache RocketMQ console.
$topic = "${TOPIC}";
// The ID of the instance to which the topic belongs. The instance is created in the
Message Queue for Apache RocketMQ console.
// If the instance has a namespace, specify the ID of the instance. If the instance
does not have a namespace, set the instance ID to null or an empty string. You can check whether
your instance has a namespace on the Instances page in the RocketMQ console.
$instanceId = "${INSTANCE_ID}";
$this->producer = $this->client->getProducer($instanceId, $topic);
}
public function run()
{
    try
    {
        for ($i=1; $i<=4; $i++)
        {
            $publishMessage = new TopicMessage(
                "hello mq!"// The content of the message.
            );
            // The custom property of the message.
            $publishMessage->putProperty("a", $i);
            // The key of the message.
            $publishMessage->setMessageKey("MessageKey");
            // The period of time after which the broker delivers the message. In this
            example, when the broker receives a message, the broker waits for 10 seconds before it delivers
            the message to the consumer. Set this parameter to a timestamp in milliseconds.
            // If the producer sends a scheduled message, set the parameter to the time
            interval between the scheduled point in time and the current point in time.
            $publishMessage->setStartDeliverTime(time() * 1000 + 10 * 1000);
            $result = $this->producer->publishMessage($publishMessage);
            print "Send mq message success. msgId is:" . $result->getMessageId() . ", bodyMD5 is:" . $result->getMessageBodyMD5() . "\n";
        }
    } catch (\Exception $e) {
        print_r($e->getMessage() . "\n");
    }
}
$instance = new ProducerTest();
$instance->run();
?>

```

Consume scheduled messages or delayed messages

The following sample code provides an example on how to consume scheduled messages or delayed messages:

```
<?php
require "vendor/autoload.php";
use MQ\Model\TopicMessage;
use MQ\MQClient;
class ConsumerTest
{
    private $client;
    private $producer;
    public function __construct()
    {
        $this->client = new MQClient(
            // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint,
            // log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane,
            // click Instances. On the Instances page, select the name of your instance. Then, view the HT
            // TP endpoint on the Network Management tab.
            "${HTTP_ENDPOINT}",
            // The AccessKey ID that is used for identity verification. You can obtain the
            // AccessKey ID in the Apsara Uni-manager Operations Console.
            "${ACCESS_KEY}",
            // The AccessKey secret that is used for identity verification. You can obtain
            // the AccessKey secret in the Apsara Uni-manager Operations Console.
            "${SECRET_KEY}"
        );
        // The topic from which you want to consume messages. The topic is created in the M
        // essage Queue for Apache RocketMQ console.
        $topic = "${TOPIC}";
        // The ID of the group that you created in the Message Queue for Apache RocketMQ co
        // nsole.
        $groupId = "${GROUP_ID}";
        // The ID of the instance to which the topic belongs. The instance is created in th
        // e Message Queue for Apache RocketMQ console.
        // If the instance has a namespace, specify the ID of the instance. If the instance
        // does not have a namespace, set the instance ID to null or an empty string. You can check wh
        // ether your instance has a namespace on the Instances page in the RocketMQ console.
        $instanceId = "${INSTANCE_ID}";
        $this->consumer = $this->client->getConsumer($instanceId, $topic, $groupId);
    }
    public function run()
    {
        // Cyclically consume messages in the current thread. We recommend that you use mul
        // tiple threads to concurrently consume messages.
        while (True) {
            try {
                // Consume messages in long polling mode.
                // In long polling mode, if no message in the topic is available for consum
                // ption, the request is suspended on the broker for a specified period of time. If a message
                // becomes available for consumption within this period, the broker immediately sends a respon
                // se to the consumer. In this example, the period is set to 3 seconds.
                $messages = $this->consumer->consumeMessage(
                    3 // The maximum number of messages that can be consumed at a time. To
```

```

        3, // The maximum number of messages that can be consumed at a time. In
this example, the value is set to 3. The maximum value that you can specify is 16.
        3 // The length of a long polling period. Unit: seconds. In this exampl
e, the value is set to 3. The maximum value that you can specify is 30.
    );
    } catch (\Exception $e) {
        if ($e instanceof MQ\Exception\MessageNotExistException) {
            // If no message in the topic is available for consumption, the long po
lling mode continues to take effect.
            printf("No message, continue long polling!RequestId:%s\n", $e->getReques
tId());

            continue;
        }
        print_r($e->getMessage() . "\n");
        sleep(3);
        continue;
    }
    print "consume finish, messages:\n";
    // Specify the message consumption logic.
    $receiptHandles = array();
    foreach ($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->getMessa
geBody(),
            $message->getPublishTime(), $message->getFirstConsumeTime(), $message->
getConsumedTimes(), $message->getNextConsumeTime(),
            $message->getMessageKey());
        print_r($message->getProperties());
    }
    // If the broker does not receive an acknowledgment (ACK) for a message from th
e consumer before the period of time specified by $message->getNextConsumeTime() elapses, t
he broker delivers the message for consumption again.
    // A unique timestamp is specified for the handle of a message each time the me
ssage is consumed.
    print_r($receiptHandles);
    try {
        $this->consumer->ackMessage($receiptHandles);
    } catch (\Exception $e) {
        if ($e instanceof MQ\Exception\AckMessageException) {
            // If the handle of a message times out, the broker cannot receive an A
CK for the message from the consumer.
            printf("Ack Error, RequestId:%s\n", $e->getRequestId());
            foreach ($e->getAckMessageErrorItems() as $errorItem) {
                printf("\tReceiptHandle:%s, ErrorCode:%s, ErrorMessage:%s\n", $errorIte
m->getReceiptHandle(), $errorItem->getErrorCode(), $errorItem->getErrorMessage());
            }
        }
    }
    print "ack finish\n";
}
}
}

$instance = new ConsumerTest();

```

```

$instance->run();
?>

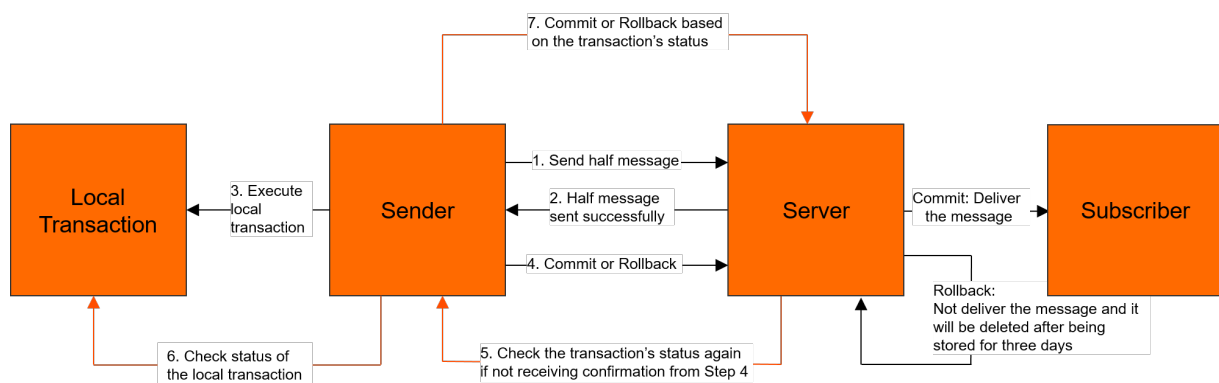
```

6.3.6.5. Send and consume transactional messages

Message Queue for Apache RocketMQ provides a distributed transaction processing feature that is similar to X/Open XA. Message Queue for Apache RocketMQ uses transactional messages to ensure transactional consistency. This topic provides sample code to show how to use the HTTP client SDK for PHP to send and consume transactional messages.

Background information

The following figure shows the interaction process of transactional messages.



For more information about the message routing feature, see [Transactional messages](#).

Prerequisites

The following operations are performed:

- Install the SDK for PHP. For more information about the message routing feature, see [Prepare the environment](#).
- Create resources that you want to specify in the code. For example, you must create the instance, topic, and group that you want to specify in the code in the Message Queue for Apache RocketMQ console in advance. For more information about the message routing feature, see [Create resources](#).

Send transactional messages

The following sample code provides an example on how to send transactional messages:

```

<?php
require "vendor/autoload.php";
use MQ\Model\TopicMessage;
use MQ\MQClient;
class ProducerTest
{
    private $client;
    private $transProducer;
    private $count;
    private $popMsgCount;
    public function __construct()
    {

```

```

        $this->client = new MQClient(
            // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint,
            log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane,
            click Instances. On the Instances page, select the name of your instance. Then, view the HT
            TP endpoint on the Network Management tab.
            "${HTTP_ENDPOINT}",
            // The AccessKey ID that is used for identity verification. You can obtain the
            AccessKey ID in the Apsara Uni-manager Operations Console.
            "${ACCESS_KEY}",
            // The AccessKey secret that is used for identity verification. You can obtain
            the AccessKey secret in the Apsara Uni-manager Operations Console.
            "${SECRET_KEY}"
        );
        // The topic to which you want to send messages. The topic is created in the Messag
        e Queue for Apache RocketMQ console.
        $topic = "${TOPIC}";
        // The ID of the group that you created in the Message Queue for Apache RocketMQ co
        nsole.
        $groupId = "${GROUP_ID}";
        // The ID of the instance to which the topic belongs. The instance is created in th
        e Message Queue for Apache RocketMQ console.
        // If the instance has a namespace, specify the ID of the instance. If the instance
        does not have a namespace, set the instance ID to null or an empty string. You can check wh
        ether your instance has a namespace on the Instances page in the RocketMQ console.
        $instanceId = "${INSTANCE_ID}";
        $this->transProducer = $this->client->getTransProducer($instanceId,$topic, $groupId
    );

    $this->count = 0;
    $this->popMsgCount = 0;
}

function processAckError($e) {
    if ($e instanceof MQ\Exception\AckMessageException) {
        // If a transactional message is not committed or rolled back within the timeou
        t period specified by the TransCheckImmunityTime parameter or the NextConsumeTime parameter
        , the commit or rollback operation fails. The TransCheckImmunityTime parameter specifies a
        timeout period for the handle of transactional messages. The NextConsumeTime parameter spec
        ifies a timeout period for the handle of consumeHalfMessage.
        printf("Commit/Rollback Error, RequestId:%s\n", $e->getRequestId());
        foreach ($e->getAckMessageErrorItems() as $errorItem) {
            printf("\tReceiptHandle:%s, ErrorCode:%s, ErrorMsg:%s\n", $errorItem->getRe
            ceiptHandle(), $errorItem->getErrorCode(), $errorItem->getErrorCode());
        }
    } else {
        print_r($e);
    }
}

function consumeHalfMsg() {
    while($this->count < 3 && $this->popMsgCount < 15) {
        $this->popMsgCount++;
        try {
            $messages = $this->transProducer->consumeHalfMessage(4, 3);
        } catch (\Exception $e) {
            if ($e instanceof MQ\Exception\MessageNotExistException) {
                print "no half transaction message\n";
            }
        }
    }
}

```

```

        continue;
    }
    print_r($e->getMessage() . "\n");
    sleep(3);
    continue;
}
foreach ($messages as $message) {
    printf("ID:%s TAG:%s BODY:%s \nPublishTime:%d, FirstConsumeTime:%d\nConsume
dTimes:%d, NextConsumeTime:%d\nPropA:%s\n",
        $message->getMessageId(), $message->getMessageTag(), $message->getMessa
geBody(),
        $message->getPublishTime(), $message->getFirstConsumeTime(), $message->
getConsumedTimes(), $message->getNextConsumeTime(),
        $message->getProperty("a"));
    print_r($message->getProperties());
    $propA = $message->getProperty("a");
    $consumeTimes = $message->getConsumedTimes();
    try {
        if ($propA == "1") {
            print "\n commit transaction msg: " . $message->getMessageId() . "\
n";

            $this->transProducer->commit($message->getReceiptHandle());
            $this->count++;
        } else if ($propA == "2" && $consumeTimes > 1) {
            print "\n commit transaction msg: " . $message->getMessageId() . "\
n";

            $this->transProducer->commit($message->getReceiptHandle());
            $this->count++;
        } else if ($propA == "3") {
            print "\n rollback transaction msg: " . $message->getMessageId() .
"\n";

            $this->transProducer->rollback($message->getReceiptHandle());
            $this->count++;
        } else {
            print "\n unknown transaction msg: " . $message->getMessageId() . "
\n";
        }
    } catch (\Exception $e) {
        processAckError($e);
    }
}
}

public function run()
{
    // Cyclically send four transactional messages.
    for ($i = 0; $i < 4; $i++) {
        $pubMsg = new TopicMessage("hello,mq");
        // The custom property of the message.
        $pubMsg->putProperty("a", $i);
        // The key of the message.
        $pubMsg->setMessageKey("MessageKey");
        // The time interval between the time when the transactional message is sent an
d the start time of the first transaction status check. Unit: seconds. Valid values: 10 to
200

```

```

300.
        // If the message is not committed or rolled back after the first transaction s
        tatus check is performed, the broker initiates a request to check the status of the local t
        ransaction at an interval of 10 seconds within the next 24 hours.
        $pubMsg->setTransCheckImmunityTime(10);
        $topicMessage = $this->transProducer->publishMessage($pubMsg);
        print "\npublish -> \n\t" . $topicMessage->getMessageId() . " " . $topicMessage
        ->getReceiptHandle() . "\n";
        if ($i == 0) {
            try {
                // After the producer sends the transactional message, the broker obtai
                ns the handle of the half message that corresponds to the transactional message and commits
                or rolls back the transactional message based on the status of the handle.
                $this->transProducer->commit($topicMessage->getReceiptHandle());
                print "\n commit transaction msg when publish: " . $topicMessage->getMe
                ssageId() . "\n";
            } catch (\Exception $e) {
                // If the transactional message is not committed or rolled back before
                the timeout period specified by the TransCheckImmunityTime parameter elapses, the commit or
                rollback operation fails.
                processAckError($e);
            }
        }
        // The client needs a thread or a process to process unacknowledged transactional m
        essages.
        // Process unacknowledged transactional messages.
        $this->consumeHalfMsg();
    }
}
$instance = new ProducerTest();
$instance->run();
?>

```

Consume transactional messages

The following sample code provides an example on how to consume transactional messages:

```

<?php
require "vendor/autoload.php";
use MQ\Model\TopicMessage;
use MQ\MQClient;
class ConsumerTest
{
    private $client;
    private $producer;
    public function __construct()
    {
        $this->client = new MQClient(
            // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint,
            log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane,
            click Instances. On the Instances page, select the name of your instance. Then, view the HT
            TP endpoint on the Network Management tab.
            "{$HTTP_ENDPOINT}",

```

```

        // The AccessKey ID that is used for identity verification. You can obtain the
        AccessKey ID in the Apsara Uni-manager Operations Console.
        "${ACCESS_KEY}",
        // The AccessKey secret that is used for identity verification. You can obtain
        the AccessKey secret in the Apsara Uni-manager Operations Console.
        "${SECRET_KEY}"
    );
    // The topic from which you want to consume messages. The topic is created in the M
    essage Queue for Apache RocketMQ console.
    $topic = "${TOPIC}";
    // The ID of the group that you created in the Message Queue for Apache RocketMQ co
    nsole.
    $groupId = "${GROUP_ID}";
    // The ID of the instance to which the topic belongs. The instance is created in th
    e Message Queue for Apache RocketMQ console.
    // If the instance has a namespace, specify the ID of the instance. If the instance
    does not have a namespace, set the instance ID to null or an empty string. You can check wh
    ether your instance has a namespace on the Instances page in the RocketMQ console.
    $instanceId = "${INSTANCE_ID}";
    $this->consumer = $this->client->getConsumer($instanceId, $topic, $groupId);
}
public function run()
{
    // Cyclically consume messages in the current thread. We recommend that you use mul
    tiple threads to concurrently consume messages.
    while (True) {
        try {
            // Consume messages in long polling mode.
            // In long polling mode, if no message in the topic is available for consum
            ption, the request is suspended on the broker for a specified period of time. If a message
            becomes available for consumption within this period, the broker immediately sends a respon
            se to the consumer. In this example, the period is set to 3 seconds.
            $messages = $this->consumer->consumeMessage(
                3, // The maximum number of messages that can be consumed at a time. In
                this example, the value is set to 3. The maximum value that you can specify is 16.
                3 // The length of a long polling period. Unit: seconds. In this exampl
                e, the value is set to 3. The maximum value that you can specify is 30.
            );
        } catch (\Exception $e) {
            if ($e instanceof MQ\Exception\MessageNotExistException) {
                // If no message in the topic is available for consumption, the long po
                lling mode continues to take effect.
                printf("No message, contine long polling!RequestId:%s\n", $e->getReques
                tId());
                continue;
            }
            print_r($e->getMessage() . "\n");
            sleep(3);
            continue;
        }
        print "consume finish, messages:\n";
        // Specify the message consumption logic.
        $receiptHandles = array();
        foreach ($messages as $message) {
            $receiptHandles[] = $message->getReceiptHandle();
        }
    }
}

```

```

        $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->getMessa
geBody(),
            $message->getPublishTime(), $message->getFirstConsumeTime(), $message->
getConsumedTimes(), $message->getNextConsumeTime(),
            $message->getMessageKey());
        print_r($message->getProperties());
    }
    // If the broker does not receive an acknowledgment (ACK) for a message from th
e consumer before the period of time specified by $message->getNextConsumeTime() elapses, t
he broker delivers the message for consumption again.
    // A unique timestamp is specified for the handle of a message each time the me
ssage is consumed.
    print_r($receiptHandles);
    try {
        $this->consumer->ackMessage($receiptHandles);
    } catch (\Exception $e) {
        if ($e instanceof MQ\Exception\AckMessageException) {
            // If the handle of a message times out, the broker cannot receive an A
CK for the message from the consumer.
            printf("Ack Error, RequestId:%s\n", $e->getRequestId());
            foreach ($e->getAckMessageErrorItems() as $errorItem) {
                printf("\tReceiptHandle:%s, ErrorCode:%s, ErrorMsg:%s\n", $errorIte
m->getReceiptHandle(), $errorItem->getErrorCode(), $errorItem->getErrorCode());
            }
        }
    }
    print "ack finish\n";
}
}
}
$instance = new ConsumerTest();
$instance->run();
?>

```

6.3.7. C# SDK

6.3.7.1. Prepare the environment

This topic describes how to prepare the environment before you use the HTTP client SDK for C# to send and consume messages.

Environment requirements

- .NET is installed. For more information, see [Install .NET](#).
- Visual Studio 2015 or later is installed. For more information, visit the [official website of Visual Studio](#).

After .NET is installed, you can run the `dotnet --version` command to check the version of .NET that you installed.

Install the SDK for C#

To install the SDK for C#, perform the following steps:

1. Download the [SDK for C# and the project file](#) to your on-premises machine and decompress them. *Aliyun_MQ_SDK* is the directory where the SDK is located. *Aliyun_MQ_SDK.sln* is the project file.
2. Use Visual Studio to open the *Aliyun_MQ_SDK.sln* file and import the file to the Aliyun_MQ_SDK project.
3. Run the *Samples.cs* file. In the Aliyun_MQ_SDK project, the *Samples.cs* file appears. This file provides the sample code on how to send and consume messages by using the SDK for C#. Replace the values in the sample code with the actual values that are used in your application. Then, save and run the file.

6.3.7.2. Send and consume normal messages

Normal messages are messages that have no special features in Message Queue for Apache RocketMQ. They are different from featured messages, such as scheduled messages, delayed messages, ordered messages, and transactional messages. This topic provides sample code to show how to use the HTTP client SDK for C# to send and consume normal messages.

Prerequisites

The following operations are performed:

- Install the SDK for C#. For more information about the message routing feature, see [Prepare the environment](#).
- Create resources that you want to specify in the code. For example, you must create the instance, topic, and group that you want to specify in the code in the Message Queue for Apache RocketMQ console in advance. For more information about the message routing feature, see [Create resources](#).

Send normal messages

The following sample code provides an example on how to send normal messages:

```
using System;
using System.Collections.Generic;
using System.Threading;
using Aliyun.MQ.Model;
using Aliyun.MQ.Model.Exp;
using Aliyun.MQ.Util;
namespace Aliyun.MQ.Sample
{
    public class ProducerSample
    {
        // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log
        // on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, clic
        // k Instances. On the Instances page, select the name of your instance. Then, view the HTTP e
        // ndpoint on the Network Management tab.
        private const string _endpoint = "${HTTP_ENDPOINT}";
        // The AccessKey ID that is used for identity verification. You can obtain the Acce
        // ssKey ID in the Apsara Uni-manager Operations Console.
        private const string _accessKeyId = "${ACCESS_KEY}";
        // The AccessKey secret that is used for identity verification. You can obtain the
        // AccessKey secret in the Apsara Uni-manager Operations Console.
```

```

private const string _secretAccessKey = "${SECRET_KEY}";
// The topic to which you want to send messages. The topic is created in the Message Queue for Apache RocketMQ console.
private const string _topicName = "${TOPIC}";
// The ID of the instance to which the topic belongs. The instance is created in the Message Queue for Apache RocketMQ console.
// If the instance has a namespace, specify the ID of the instance. If the instance does not have a namespace, set the instance ID to null or an empty string. You can check whether your instance has a namespace on the Instances page in the RocketMQ console.
private const string _instanceId = "${INSTANCE_ID}";
private static MQClient _client = new Aliyun.MQ.MQClient(_accessKeyId, _secretAccessKey, _endpoint);
static MQProducer producer = _client.GetProducer(_instanceId, _topicName);
static void Main(string[] args)
{
    try
    {
        // Cyclically send four messages.
        for (int i = 0; i < 4; i++)
        {
            TopicMessage sendMsg;
            // The content of the message.
            sendMsg = new TopicMessage("hello mq");
            // The custom property of the message.
            sendMsg.PutProperty("a", i.ToString());
            // The key of the message.
            sendMsg.MessageKey = "MessageKey";
            TopicMessage result = producer.PublishMessage(sendMsg);
            Console.WriteLine("publish message success:" + result);
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex);
    }
}
}

```

Consume normal messages

The following sample code provides an example on how to consume normal messages:

```

using System;
using System.Collections.Generic;
using System.Threading;
using Aliyun.MQ.Model;
using Aliyun.MQ.Model.Exp;
using Aliyun.MQ;
namespace Aliyun.MQ.Sample
{
    public class ConsumerSample
    {
        // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log
    }
}

```

on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click Instances. On the Instances page, select the name of your instance. Then, view the HTTP endpoint on the Network Management tab.

```
private const string _endpoint = "${HTTP_ENDPOINT}";
// The AccessKey ID that is used for identity verification. You can obtain the AccessKey ID in the Apsara Uni-manager Operations Console.
private const string _accessKeyId = "${ACCESS_KEY}";
// The AccessKey secret that is used for identity verification. You can obtain the AccessKey secret in the Apsara Uni-manager Operations Console.
private const string _secretAccessKey = "${SECRET_KEY}";
// The topic from which you want to consume messages. The topic is created in the Message Queue for Apache RocketMQ console.
private const string _topicName = "${TOPIC}";
// The ID of the instance to which the topic belongs. The instance is created in the Message Queue for Apache RocketMQ console.
// If the instance has a namespace, specify the ID of the instance. If the instance does not have a namespace, set the instance ID to null or an empty string. You can check whether your instance has a namespace on the Instances page in the RocketMQ console.
private const string _instanceId = "${INSTANCE_ID}";
// The ID of the group that you created in the Message Queue for Apache RocketMQ console.
private const string _groupId = "${GROUP_ID}";
private static MQClient _client = new Aliyun.MQ.MQClient(_accessKeyId, _secretAccessKey, _endpoint);
static MQConsumer consumer = _client.GetConsumer(_instanceId, _topicName, _groupId, null);

static void Main(string[] args)
{
    // Cyclically consume messages in the current thread. We recommend that you use multiple threads to concurrently consume messages.
    while (true)
    {
        try
        {
            // Consume messages in long polling mode.
            // In long polling mode, if no message in the topic is available for consumption, the request is suspended on the broker for a specified period of time. If a message becomes available for consumption within this period, the broker immediately sends a response to the consumer. In this example, the period is set to 3 seconds.
            List<Message> messages = null;
            try
            {
                messages = consumer.ConsumeMessage(
                    3, // The maximum number of messages that can be consumed at a time. In this example, the value is set to 3. The maximum value that you can specify is 16.

                    3 // The length of a long polling period. Unit: seconds. In this example, the value is set to 3. The maximum value that you can specify is 30.
                );
            }
            catch (Exception exp1)
            {
                if (exp1 is MessageNotExistException)
                {

```

```

        Console.WriteLine(Thread.CurrentThread.Name + " No new message,
" + ((MessageNotExistException)exp1).RequestId);
        continue;
    }
    Console.WriteLine(exp1);
    Thread.Sleep(2000);
}
if (messages == null)
{
    continue;
}
List<string> handlers = new List<string>();
Console.WriteLine(Thread.CurrentThread.Name + " Receive Messages:");
// Specify the message consumption logic.
foreach (Message message in messages)
{
    Console.WriteLine(message);
    Console.WriteLine("Property a is:" + message.GetProperty("a"));
    handlers.Add(message.ReceiptHandle);
}
// If the broker does not receive an acknowledgment (ACK) for a message
from the consumer before the period of time specified by Message.nextConsumeTime elapses, t
he broker delivers the message for consumption again.
// A unique timestamp is specified for the handle of a message each tim
e the message is consumed.
try
{
    consumer.AckMessage(handlers);
    Console.WriteLine("Ack message success:");
    foreach (string handle in handlers)
    {
        Console.WriteLine("\t" + handle);
    }
    Console.WriteLine();
}
catch (Exception exp2)
{
    // If the handle of a message times out, the broker cannot receive
an ACK for the message from the consumer.
    if (exp2 is AckMessageException)
    {
        AckMessageException ackExp = (AckMessageException)exp2;
        Console.WriteLine("Ack message fail, RequestId:" + ackExp.Reque
stId);

        foreach (AckMessageErrorItem errorItem in ackExp.ErrorItems)
        {
            Console.WriteLine("\tErrorHandle:" + errorItem.ReceiptHandl
e + ",ErrorCode:" + errorItem.ErrorCode + ",ErrorMsg:" + errorItem.ErrorMessage);
        }
    }
}
catch (Exception ex)
{

```

```
        Console.WriteLine(ex);  
        Thread.Sleep(2000);  
    }  
}  
}  
}  
}
```

6.3.7.3. Send and consume ordered messages

Ordered messages are a type of message that is published and consumed in a strict order. Ordered messages in Message Queue for Apache RocketMQ are also known as first-in-first-out (FIFO) messages. This topic provides sample code to show how to use the HTTP client SDK for C# to send and consume ordered messages.

Background information

Ordered messages are classified into the following types:

- Globally ordered message: All messages in a specified topic are published and consumed in first-in-first-out (FIFO) order.
- Partitionally ordered message: All messages in a specified topic are distributed to different partitions by using shard keys. The messages in each partition are published and consumed in FIFO order. A Sharding Key is a key field that is used for ordered messages to identify different partitions. The Sharding Key is different from the key of a normal message.

For more information about the message routing feature, see [Ordered messages](#).

Prerequisites

The following operations are performed:

- Install the SDK for C#. For more information about the message routing feature, see [Prepare the environment](#).
- Create resources that you want to specify in the code. For example, you must create the instance, topic, and group that you want to specify in the code in the Message Queue for Apache RocketMQ console in advance. For more information about the message routing feature, see [Create resources](#).

Send ordered messages

The following sample code provides an example on how to send ordered messages:

```
using System;  
using System.Collections.Generic;  
using System.Threading;  
using Aliyun.MQ.Model;  
using Aliyun.MQ.Model.Exp;  
using Aliyun.MQ.Util;  
namespace Aliyun.MQ.Sample  
{  
    public class OrderProducerSample  
    {  
        // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log  
        // on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, clic  
        // k Instances. On the Instances page, select the name of your instance. Then, view the HTTP e
```

instances. On the Instances page, select the name of your instance. Then, view the HTTP endpoint on the Network Management tab.

```
private const string _endpoint = "${HTTP_ENDPOINT}";
// The AccessKey ID that is used for identity verification. You can obtain the AccessKey ID in the Apsara Uni-manager Operations Console.
private const string _accessKeyId = "${ACCESS_KEY}";
// The AccessKey secret that is used for identity verification. You can obtain the AccessKey secret in the Apsara Uni-manager Operations Console.
private const string _secretAccessKey = "${SECRET_KEY}";
// The topic to which you want to send messages. The topic is created in the Message Queue for Apache RocketMQ console.
private const string _topicName = "${TOPIC}";
// The ID of the instance to which the topic belongs. The instance is created in the Message Queue for Apache RocketMQ console.
// If the instance has a namespace, specify the ID of the instance. If the instance does not have a namespace, set the instance ID to null or an empty string. You can check whether your instance has a namespace on the Instances page in the RocketMQ console.
private const string _instanceId = "${INSTANCE_ID}";
private static MQClient _client = new Aliyun.MQ.MQClient(_accessKeyId, _secretAccessKey, _endpoint);
static MQProducer producer = _client.GetProducer(_instanceId, _topicName);
static void Main(string[] args)
{
    try
    {
        // Cyclically send eight messages.
        for (int i = 0; i < 8; i++)
        {
            // The content and tag of the message.
            TopicMessage sendMsg = new TopicMessage("hello mq", "tag");
            // The custom property of the message.
            sendMsg.PutProperty("a", i.ToString());
            // The shard key that is used to distribute ordered messages to a specific partition. Shard keys can be used to identify different partitions. A shard key is different from a message key.
            sendMsg.ShardingKey = (i % 2).ToString();
            TopicMessage result = producer.PublishMessage(sendMsg);
            Console.WriteLine("publish message success:" + result);
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex);
    }
}
}
```

Consume ordered messages

The following sample code provides an example on how to consume ordered messages:

```
using System;
using System.Collections.Generic;
using System.Threading;
```

```

using System.Threading;
using Aliyun.MQ.Model;
using Aliyun.MQ.Model.Exp;
using Aliyun.MQ;
namespace Aliyun.MQ.Sample
{
    public class OrderConsumerSample
    {
        // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log
        // on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, clic
        // k Instances. On the Instances page, select the name of your instance. Then, view the HTTP e
        // ndpoint on the Network Management tab.
        private const string _endpoint = "${HTTP_ENDPOINT}";
        // The AccessKey ID that is used for identity verification. You can obtain the Acce
        // ssKey ID in the Apsara Uni-manager Operations Console.
        private const string _accessKeyId = "${ACCESS_KEY}";
        // The AccessKey secret that is used for identity verification. You can obtain the
        // AccessKey secret in the Apsara Uni-manager Operations Console.
        private const string _secretAccessKey = "${SECRET_KEY}";
        // The topic from which you want to consume messages. The topic is created in the M
        // essage Queue for Apache RocketMQ console.
        private const string _topicName = "${TOPIC}";
        // The ID of the group that you created in the Message Queue for Apache RocketMQ co
        // nsole.
        private const string _groupId = "${GROUP_ID}";
        // The ID of the instance to which the topic belongs. The instance is created in th
        // e Message Queue for Apache RocketMQ console.
        // If the instance has a namespace, specify the ID of the instance. If the instance
        // does not have a namespace, set the instance ID to null or an empty string. You can check wh
        // ether your instance has a namespace on the Instances page in the RocketMQ console.
        private const string _instanceId = "${INSTANCE_ID}";
        private static MQClient _client = new Aliyun.MQ.MQClient(_accessKeyId, _secretAcces
        // sKey, _endpoint);
        static MQConsumer consumer = _client.GetConsumer(_instanceId, _topicName, _groupId,
        // null);
        static void Main(string[] args)
        {
            // Cyclically consume messages in the current thread. We recommend that you use
            // multiple threads to concurrently consume messages.
            while (true)
            {
                try
                {
                    // Consume messages in long polling mode. The consumer may pull partiti
                    // onally ordered messages from multiple partitions. The consumer consumes messages from the s
                    // ame partition in the order in which the messages are sent.
                    // A consumer pulls partitionally ordered messages from a partition. If
                    // the broker does not receive an acknowledgment (ACK) for a message after the message is cons
                    // umed, the consumer consumes the message again.
                    // The consumer can consume the next batch of messages from a partition
                    // only after all messages that are pulled from the partition in the previous batch are acknow
                    // ledged to be consumed.
                    // In long polling mode, if no message in the topic is available for co
                    // nsumption, the request is suspended on the broker for a specified period of time. If a mess
                    // age becomes available for consumption within this period, the broker immediately sends a re

```

age becomes available for consumption within this period, the broker immediately sends a response to the consumer. In this example, the period is set to 3 seconds.

```
List<Message> messages = null;
try
{
    messages = consumer.ConsumeMessageOrderly(
        3, // The maximum number of messages that can be consumed at a
time. In this example, the value is set to 3. The maximum value that you can specify is 16.

        3 // The length of a long polling period. Unit: seconds. In this
example, the value is set to 3. The maximum value that you can specify is 30.
    );
}
catch (Exception exp1)
{
    if (exp1 is MessageNotExistException)
    {
        Console.WriteLine(Thread.CurrentThread.Name + " No new message,
" + ((MessageNotExistException)exp1).RequestId);
        continue;
    }
    Console.WriteLine(exp1);
    Thread.Sleep(2000);
}
if (messages == null)
{
    continue;
}
List<string> handlers = new List<string>();
Console.WriteLine(Thread.CurrentThread.Name + " Receive Messages:");
// Specify the message consumption logic.
foreach (Message message in messages)
{
    Console.WriteLine(message);
    Console.WriteLine("Property a is:" + message.GetProperty("a"));
    handlers.Add(message.ReceiptHandle);
}
// If the broker does not receive an ACK for a message from the consumer
before the period of time specified by Message.NextConsumeTime elapses, the broker delivers
the message for consumption again.
// A unique timestamp is specified for the handle of a message each time
the message is consumed.
try
{
    consumer.AckMessage(handlers);
    Console.WriteLine("Ack message success:");
    foreach (string handle in handlers)
    {
        Console.WriteLine("\t" + handle);
    }
    Console.WriteLine();
}
catch (Exception exp2)
{
    // If the handle of a message times out, the broker cannot receive
```

```
an ACK for the message from the consumer.
        if (exp2 is AckMessageException)
        {
            AckMessageException ackExp = (AckMessageException)exp2;
            Console.WriteLine("Ack message fail, RequestId:" + ackExp.RequestId);

            foreach (AckMessageErrorItem errorItem in ackExp.ErrorItems)
            {
                Console.WriteLine("\tErrorHandle:" + errorItem.ReceiptHandle + ",ErrorCode:" + errorItem.ErrorCode + ",ErrorMsg:" + errorItem.ErrorMessage);
            }
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex);
        Thread.Sleep(2000);
    }
}
}
```

6.3.7.4. Send and consume scheduled messages and delayed messages

This topic provides sample code to show how to use the HTTP client SDK for C# to send and consume scheduled messages and delayed messages.

Background information

- **Delayed message:** The producer sends the message to the Message Queue for Apache RocketMQ server, but does not expect the message to be delivered immediately. Instead, the message is delivered to the consumer for consumption after a certain period of time. This message is a delayed message.
- **Scheduled message:** A producer sends a message to a Message Queue for Apache RocketMQ broker and expects the message to be delivered to a consumer at a specified point in time. This type of message is called a scheduled message.

If an HTTP client SDK is used, the code configurations of scheduled messages are the same as the code configurations of delayed messages. Both types of messages are delivered to consumers after a specific period of time based on the attributes of the messages.

For more information about the message routing feature, see [Scheduled messages and delayed messages](#).

Prerequisites

The following operations are performed:

- Install the SDK for C#. For more information about the message routing feature, see [Prepare the environment](#).
- Create resources that you want to specify in the code. For example, you must create the instance, topic, and group that you want to specify in the code in the Message Queue for Apache RocketMQ console in advance. For more information about the message routing feature, see [Create resources](#).

Send scheduled messages or delayed messages

The following sample code provides an example on how to send scheduled messages or delayed messages:

```
using System;
using System.Collections.Generic;
using System.Threading;
using Aliyun.MQ.Model;
using Aliyun.MQ.Model.Exp;
using Aliyun.MQ.Util;
namespace Aliyun.MQ.Sample
{
    public class ProducerSample
    {
        // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log
        // on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, clic
        // k Instances. On the Instances page, select the name of your instance. Then, view the HTTP e
        // ndpoint on the Network Management tab.
        private const string _endpoint = "${HTTP_ENDPOINT}";
        // The AccessKey ID that is used for identity verification. You can obtain the Acce
        // ssKey ID in the Apsara Uni-manager Operations Console.
        private const string _accessKeyId = "${ACCESS_KEY}";
        // The AccessKey secret that is used for identity verification. You can obtain the
        // AccessKey secret in the Apsara Uni-manager Operations Console.
        private const string _secretAccessKey = "${SECRET_KEY}";
        // The topic to which you want to send messages. The topic is created in the Messag
        // e Queue for Apache RocketMQ console.
        private const string _topicName = "${TOPIC}";
        // The ID of the instance to which the topic belongs. The instance is created in th
        // e Message Queue for Apache RocketMQ console.
        // If the instance has a namespace, specify the ID of the instance. If the instance
        // does not have a namespace, set the instance ID to null or an empty string. You can check wh
        // ether your instance has a namespace on the Instances page in the RocketMQ console.
        private const string _instanceId = "${INSTANCE_ID}";
        private static MQClient _client = new Aliyun.MQ.MQClient(_accessKeyId, _secretAcces
        sKey, _endpoint);
        static MQProducer producer = _client.GetProducer(_instanceId, _topicName);
        static void Main(string[] args)
        {
            try
            {
                // Cyclically send four messages.
                for (int i = 0; i < 4; i++)
                {
                    TopicMessage sendMsg;
                    // The content of the message.
                    sendMsg = new TopicMessage("hello mq");
```

```

        // The custom property of the message.
        sendMsg.PutProperty("a", i.ToString());

        // The period of time after which the broker delivers the message. In this example, when the broker receives a message, the broker waits for 10 seconds before it delivers the message to the consumer. Set this parameter to a timestamp in milliseconds.

        // If the producer sends a scheduled message, set the parameter to the time interval between the scheduled point in time and the current point in time.
        sendMsg.StartDeliverTime = AliyunSDKUtils.GetNowTimeStamp() + 10 * 1000
;

        TopicMessage result = producer.PublishMessage(sendMsg);
        Console.WriteLine("publis message success:" + result);
    }
}
catch (Exception ex)
{
    Console.WriteLine(ex);
}
}
}
}

```

Consume scheduled messages or delayed messages

The following sample code provides an example on how to consume scheduled messages or delayed messages:

```

using System;
using System.Collections.Generic;
using System.Threading;
using Aliyun.MQ.Model;
using Aliyun.MQ.Model.Exp;
using Aliyun.MQ;
namespace Aliyun.MQ.Sample
{
    public class ConsumerSample
    {
        // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click Instances. On the Instances page, select the name of your instance. Then, view the HTTP endpoint on the Network Management tab.
        private const string _endpoint = "${HTTP_ENDPOINT}";
        // The AccessKey ID that is used for identity verification. You can obtain the AccessKey ID in the Apsara Uni-manager Operations Console.
        private const string _accessKeyId = "${ACCESS_KEY}";
        // The AccessKey secret that is used for identity verification. You can obtain the AccessKey secret in the Apsara Uni-manager Operations Console.
        private const string _secretAccessKey = "${SECRET_KEY}";
        // The topic from which you want to consume messages. The topic is created in the Message Queue for Apache RocketMQ console.
        private const string _topicName = "${TOPIC}";
        // The ID of the instance to which the topic belongs. The instance is created in the Message Queue for Apache RocketMQ console.
        // If the instance has a namespace, specify the ID of the instance. If the instance
    }
}

```

does not have a namespace, set the instance ID to null or an empty string. You can check whether your instance has a namespace on the Instances page in the RocketMQ console.

```
private const string _instanceId = "${INSTANCE_ID}";
// The ID of the group that you created in the Message Queue for Apache RocketMQ console.
private const string _groupId = "${GROUP_ID}";
private static MQClient _client = new Aliyun.MQ.MQClient(_accessKeyId, _secretAccessKey, _endpoint);
static MQConsumer consumer = _client.GetConsumer(_instanceId, _topicName, _groupId, null);
static void Main(string[] args)
{
    // Cyclically consume messages in the current thread. We recommend that you use multiple threads to concurrently consume messages.
    while (true)
    {
        try
        {
            // Consume messages in long polling mode.
            // In long polling mode, if no message in the topic is available for consumption, the request is suspended on the broker for a specified period of time. If a message becomes available for consumption within this period, the broker immediately sends a response to the consumer. In this example, the period is set to 3 seconds.
            List<Message> messages = null;
            try
            {
                messages = consumer.ConsumeMessage(
                    3, // The maximum number of messages that can be consumed at a time. In this example, the value is set to 3. The maximum value that you can specify is 16.

                    3 // The length of a long polling period. Unit: seconds. In this example, the value is set to 3. The maximum value that you can specify is 30.
                );
            }
            catch (Exception expl)
            {
                if (expl is MessageNotExistException)
                {
                    Console.WriteLine(Thread.CurrentThread.Name + " No new message, " + ((MessageNotExistException)expl).RequestId);
                    continue;
                }
                Console.WriteLine(expl);
                Thread.Sleep(2000);
            }
            if (messages == null)
            {
                continue;
            }
            List<string> handlers = new List<string>();
            Console.WriteLine(Thread.CurrentThread.Name + " Receive Messages:");
            // Specify the message consumption logic.
            foreach (Message message in messages)
            {
```

```

        Console.WriteLine(message);
        Console.WriteLine("Property a is:" + message.GetProperty("a"));
        handlers.Add(message.ReceiptHandle);
    }
    // If the broker does not receive an acknowledgment (ACK) for a message
    from the consumer before the period of time specified by Message.nextConsumeTime elapses, t
    he broker delivers the message for consumption again.
    // A unique timestamp is specified for the handle of a message each tim
    e the message is consumed.
    try
    {
        consumer.AckMessage(handlers);
        Console.WriteLine("Ack message success:");
        foreach (string handle in handlers)
        {
            Console.Write("\t" + handle);
        }
        Console.WriteLine();
    }
    catch (Exception exp2)
    {
        // If the handle of a message times out, the broker cannot receive
        an ACK for the message from the consumer.
        if (exp2 is AckMessageException)
        {
            AckMessageException ackExp = (AckMessageException)exp2;
            Console.WriteLine("Ack message fail, RequestId:" + ackExp.Reque
            stId);

            foreach (AckMessageErrorItem errorItem in ackExp.ErrorItems)
            {
                Console.WriteLine("\tErrorHandle:" + errorItem.ReceiptHandl
                e + ",ErrorCode:" + errorItem.ErrorCode + ",ErrorMsg:" + errorItem.ErrorMessage);
            }
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex);
        Thread.Sleep(2000);
    }
}
}
}
}

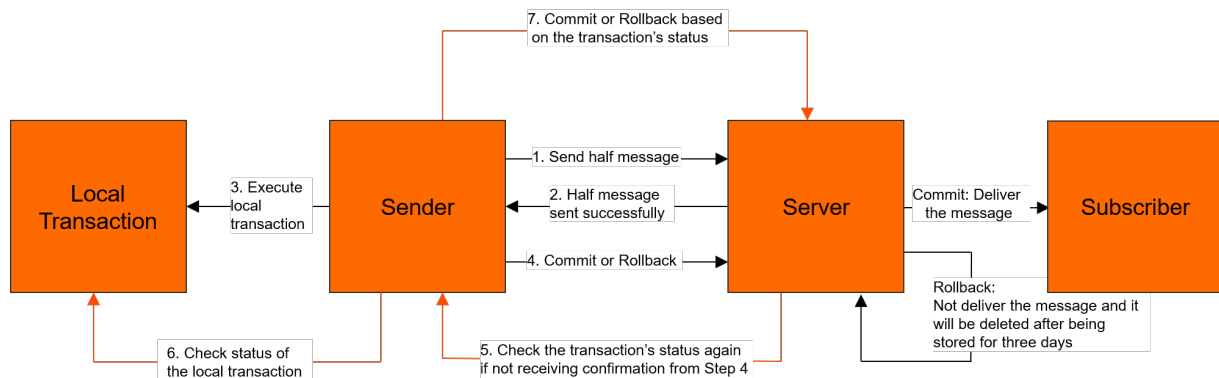
```

6.3.7.5. Send and consume transactional messages

Message Queue for Apache RocketMQ provides a distributed transaction processing feature that is similar to X/Open XA. Message Queue for Apache RocketMQ uses transactional messages to ensure transactional consistency. This topic provides sample code to show how to use the HTTP client SDK for C# to send and consume transactional messages.

Background information

The following figure shows the interaction process of transactional messages.



For more information about the message routing feature, see [Transactional messages](#).

Prerequisites

The following operations are performed:

- Install the SDK for C#. For more information about the message routing feature, see [Prepare the environment](#).
- Create resources that you want to specify in the code. For example, you must create the instance, topic, and group that you want to specify in the code in the Message Queue for Apache RocketMQ console in advance. For more information about the message routing feature, see [Create resources](#).

Send transactional messages

The following sample code provides an example on how to send transactional messages:

```

using System;
using System.Collections.Generic;
using System.Threading;
using Aliyun.MQ.Model;
using Aliyun.MQ.Model.Exp;
using Aliyun.MQ.Util;
namespace Aliyun.MQ.Sample
{
    public class TransProducerSample
    {
        // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log
        // on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, clic
        // k Instances. On the Instances page, select the name of your instance. Then, view the HTTP e
        // ndpoint on the Network Management tab.
        private const string _endpoint = "${HTTP_ENDPOINT}";
        // The AccessKey ID that is used for identity verification. You can obtain the Acce
        // ssKey ID in the Apsara Uni-manager Operations Console.
        private const string _accessKeyId = "${ACCESS_KEY}";
        // The AccessKey secret that is used for identity verification. You can obtain the
        // AccessKey secret in the Apsara Uni-manager Operations Console.
        private const string _secretAccessKey = "${SECRET_KEY}";
        // The topic to which you want to send messages. The topic is created in the Messag
        // e Queue for Apache RocketMQ console.
        private const string _topicName = "${TOPIC}";
    }
}
  
```

```

private const string _topicName = "Topic1";
// The ID of the instance to which the topic belongs. The instance is created in the
// Message Queue for Apache RocketMQ console.
// If the instance has a namespace, specify the ID of the instance. If the instance
// does not have a namespace, set the instance ID to null or an empty string. You can check whether
// your instance has a namespace on the Instances page in the RocketMQ console.
private const string _instanceId = "${INSTANCE_ID}";
// The ID of the group that you created in the Message Queue for Apache RocketMQ console.
private const string _groupId = "${GROUP_ID}";
private static readonly MQClient _client = new Aliyun.MQ.MQClient(_accessKeyId, _secretAccessKey, _endpoint);
private static readonly MQTransProducer transProducer = _client.GetTransProducer(_instanceId, _topicName, _groupId);
static void ProcessAckError(Exception exception)
{
    // If a transactional message is not committed or rolled back before the timeout period
    // specified by the TransCheckImmunityTime parameter for the handle of the transactional
    // message elapses or before the timeout period specified for the handle of consumeHalfMessage
    // elapses, the commit or rollback operation fails. In this example, the timeout period for
    // the handle of consumeHalfMessage is 10 seconds.
    if (exception is AckMessageException)
    {
        AckMessageException ackExp = (AckMessageException)exception;
        Console.WriteLine("Ack message fail, RequestId:" + ackExp.RequestId);
        foreach (AckMessageErrorItem errorItem in ackExp.ErrorItems)
        {
            Console.WriteLine("\tErrorHandle:" + errorItem.ReceiptHandle + ", ErrorCode:"
                + errorItem.ErrorCode + ", ErrorMessage:" + errorItem.ErrorMessage);
        }
    }
}
static void ConsumeHalfMessage()
{
    int count = 0;
    while (true)
    {
        if (count == 3)
            break;
        try
        {
            // Check the status of half messages. This process is similar to consuming
            // normal messages.
            List<Message> messages = null;
            try
            {
                messages = transProducer.ConsumeHalfMessage(3, 3);
            } catch (Exception exp1) {
                if (exp1 is MessageNotExistException)
                {
                    Console.WriteLine(Thread.CurrentThread.Name + " No half message
, " + ((MessageNotExistException)exp1).RequestId);
                    continue;
                }
            }
            Console.WriteLine(exp1);
        }
    }
}

```

```

        Thread.Sleep(2000);
    }
    if (messages == null)
        continue;
    // Specify the business processing logic.
    foreach (Message message in messages)
    {
        Console.WriteLine(message);
        int a = int.Parse(message.GetProperty("a"));
        uint consumeTimes = message.ConsumedTimes;
        try {
            if (a == 1) {
                // Confirm to commit the transactional message.
                transProducer.Commit(message.ReceiptHandle);
                count++;
                Console.WriteLine("Id:" + message.Id + ", commit");
            } else if (a == 2 && consumeTimes > 1) {
                // Confirm to commit the transactional message.
                transProducer.Commit(message.ReceiptHandle);
                count++;
                Console.WriteLine("Id:" + message.Id + ", commit");
            } else if (a == 3) {
                // Confirm to roll back the transactional message.
                transProducer.Rollback(message.ReceiptHandle);
                count++;
                Console.WriteLine("Id:" + message.Id + ", rollback");
            } else {
                // Do not perform operations. Check the status next time.
                Console.WriteLine("Id:" + message.Id + ", unknown");
            }
        } catch (Exception ackError) {
            ProcessAckError(ackError);
        }
    }
}
catch (Exception ex)
{
    Console.WriteLine(ex);
    Thread.Sleep(2000);
}
}

static void Main(string[] args)
{
    // The client needs a thread or a process to process unacknowledged transactional messages.
    // Start a thread to process unacknowledged transactional messages.
    Thread consumeHalfThread = new Thread(ConsumeHalfMessage);
    consumeHalfThread.Start();
    try
    {
        // Cyclically send four transactional messages. Among the four messages, commit the first message after the message is sent, and process the other three messages based on the specified conditions.
    }
}

```

```

        for (int i = 0; i < 4; i++)
        {
            TopicMessage sendMsg = new TopicMessage("trans_msg");
            sendMsg.MessageTag = "a";
            sendMsg.MessageKey = "MessageKey";
            sendMsg.PutProperty("a", i.ToString());

            // The time interval between the time when the transactional message is
            sent and the start time of the first transaction status check. Unit: seconds. Valid values:
            10 to 300.

            // If the message is not committed or rolled back after the first trans
            action status check is performed, the broker initiates a request to check the status of the
            local transaction at an interval of 10 seconds within the next 24 hours.
            sendMsg.TransCheckImmunityTime = 10;
            TopicMessage result = transProducer.PublishMessage(sendMsg);
            Console.WriteLine("publis message success:" + result);
            try {
                if (!string.IsNullOrEmpty(result.ReceiptHandle) && i == 0)
                {
                    // After the producer sends the transactional message, the brok
                    er obtains the handle of the half message that corresponds to the transactional message and
                    commits or rolls back the transactional message based on the status of the handle.
                    transProducer.Commit(result.ReceiptHandle);
                    Console.WriteLine("Id:" + result.Id + ", commit");
                }
            } catch (Exception ackError) {
                ProcessAckError(ackError);
            }
        }
    } catch (Exception ex) {
        Console.Write(ex);
    }
    consumeHalfThread.Join();
}
}

```

Consume transactional messages

The following sample code provides an example on how to consume transactional messages:

```

using System;
using System.Collections.Generic;
using System.Threading;
using Aliyun.MQ.Model;
using Aliyun.MQ.Model.Exp;
using Aliyun.MQ;
namespace Aliyun.MQ.Sample
{
    public class ConsumerSample
    {
        // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log
        on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, clic
        k Instances. On the Instances page, select the name of your instance. Then, view the HTTP e
        ndpoint on the Network Management tab.
    }
}

```

```

        private const string _endpoint = "${HTTP_ENDPOINT}";
        // The AccessKey ID that is used for identity verification. You can obtain the AccessKey ID in the Apsara Uni-manager Operations Console.
        private const string _accessKeyId = "${ACCESS_KEY}";
        // The AccessKey secret that is used for identity verification. You can obtain the AccessKey secret in the Apsara Uni-manager Operations Console.
        private const string _secretAccessKey = "${SECRET_KEY}";
        // The topic from which you want to consume messages. The topic is created in the Message Queue for Apache RocketMQ console.
        private const string _topicName = "${TOPIC}";
        // The ID of the instance to which the topic belongs. The instance is created in the Message Queue for Apache RocketMQ console.
        // If the instance has a namespace, specify the ID of the instance. If the instance does not have a namespace, set the instance ID to null or an empty string. You can check whether your instance has a namespace on the Instances page in the RocketMQ console.
        private const string _instanceId = "${INSTANCE_ID}";
        // The ID of the group that you created in the Message Queue for Apache RocketMQ console.
        private const string _groupId = "${GROUP_ID}";
        private static MQClient _client = new Aliyun.MQ.MQClient(_accessKeyId, _secretAccessKey, _endpoint);
        static MQConsumer consumer = _client.GetConsumer(_instanceId, _topicName, _groupId, null);

        static void Main(string[] args)
        {
            // Cyclically consume messages in the current thread. We recommend that you use multiple threads to concurrently consume messages.
            while (true)
            {
                try
                {
                    // Consume messages in long polling mode.
                    // In long polling mode, if no message in the topic is available for consumption, the request is suspended on the broker for a specified period of time. If a message becomes available for consumption within this period, the broker immediately sends a response to the consumer. In this example, the period is set to 3 seconds.
                    List<Message> messages = null;
                    try
                    {
                        messages = consumer.ConsumeMessage(
                            3, // The maximum number of messages that can be consumed at a time. In this example, the value is set to 3. The maximum value that you can specify is 16.

                            3 // The length of a long polling period. Unit: seconds. In this example, the value is set to 3. The maximum value that you can specify is 30.
                        );
                    }
                    catch (Exception exp1)
                    {
                        if (exp1 is MessageNotExistException)
                        {
                            Console.WriteLine(Thread.CurrentThread.Name + " No new message, " + ((MessageNotExistException)exp1).RequestId);
                            continue;
                        }
                    }
                }
            }
        }

```

```

        }
        Console.WriteLine(exp1);
        Thread.Sleep(2000);
    }
    if (messages == null)
    {
        continue;
    }
    List<string> handlers = new List<string>();
    Console.WriteLine(Thread.CurrentThread.Name + " Receive Messages:");
    // Specify the message consumption logic.
    foreach (Message message in messages)
    {
        Console.WriteLine(message);
        Console.WriteLine("Property a is:" + message.GetProperty("a"));
        handlers.Add(message.ReceiptHandle);
    }
    // If the broker does not receive an acknowledgment (ACK) for a message
    from the consumer before the period of time specified by Message.nextConsumeTime elapses, t
    he broker delivers the message for consumption again.
    // A unique timestamp is specified for the handle of a message each tim
    e the message is consumed.
    try
    {
        consumer.AckMessage(handlers);
        Console.WriteLine("Ack message success:");
        foreach (string handle in handlers)
        {
            Console.WriteLine("\t" + handle);
        }
        Console.WriteLine();
    }
    catch (Exception exp2)
    {
        // If the handle of a message times out, the broker cannot receive
        an ACK for the message from the consumer.
        if (exp2 is AckMessageException)
        {
            AckMessageException ackExp = (AckMessageException)exp2;
            Console.WriteLine("Ack message fail, RequestId:" + ackExp.Reque
            stId);

            foreach (AckMessageErrorItem errorItem in ackExp.ErrorItems)
            {
                Console.WriteLine("\tErrorHandle:" + errorItem.ReceiptHandl
                e + ",ErrorCode:" + errorItem.ErrorCode + ",ErrorMsg:" + errorItem.ErrorMessage);
            }
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex);
        Thread.Sleep(2000);
    }

```

```
}  
}  
}  
}
```

6.3.8. C++ SDK

6.3.8.1. Prepare the environment

This topic describes how to prepare the environment before you use the HTTP client SDK for C++ to send and consume messages.

Environment requirements

- SCons is installed. For more information, visit the [official website of SCons](#).
- Before you can use SCons, make sure that Python 3.5 or later is installed. For more information, visit the [official website of Python](#).
- Visual Studio 2015 or later is installed. For more information, visit the [official website of Visual Studio](#).

 **Note** Visual Studio is required only in Windows environments. In this topic, Visual Studio 2019 is used in the example.

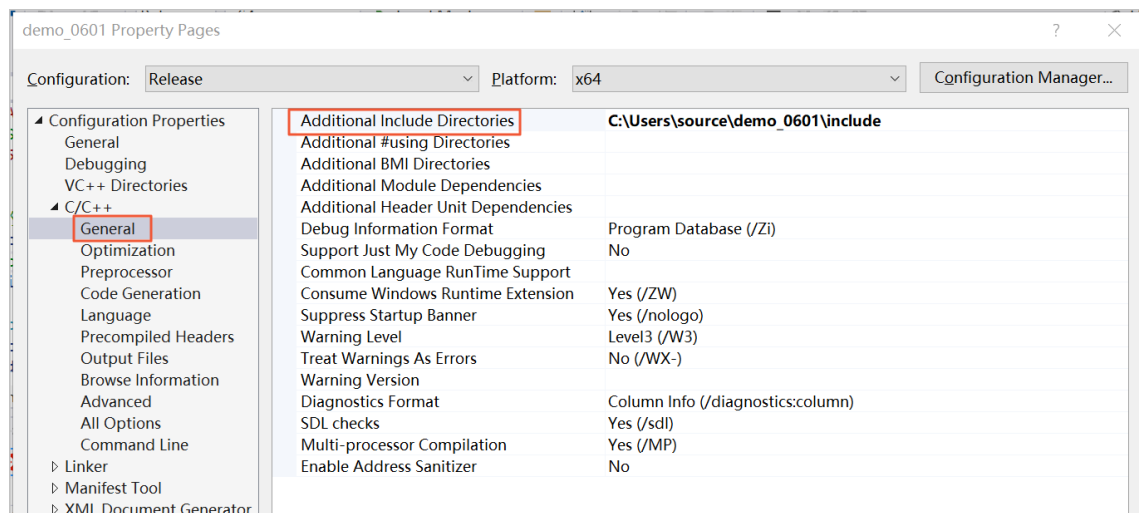
Install the SDK for C++ in a Windows environment

1. Download the SDK for C++ to your on-premises machine and decompress the package. For more information about the download link to the SDK, see [Overview](#).
2. In the SDK directory, run the following command to compile your C++ project:

```
scons
```

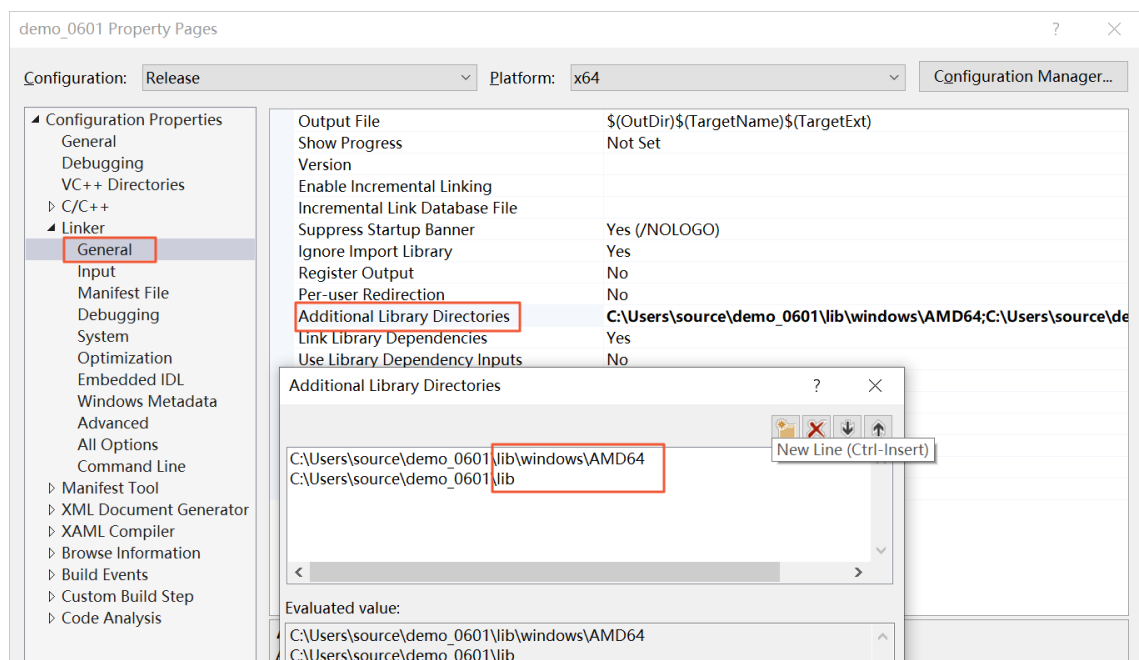
3. After the project is compiled, copy the *include* and *lib* folders in the SDK directory to the C++ project directory that you created on your on-premises machine.
4. Configure project properties in Visual Studio. Right-click your project and select **Properties**.
 - **Set the Additional Include Directories property**

In the Property Pages dialog box of your project, choose **Configuration Properties > C/C++ > General** in the left-side navigation pane. On the right side, set **Additional Include Directories** to the path of the *include* folder that you copied in Step 3.



◦ Set the Additional Library Directories property

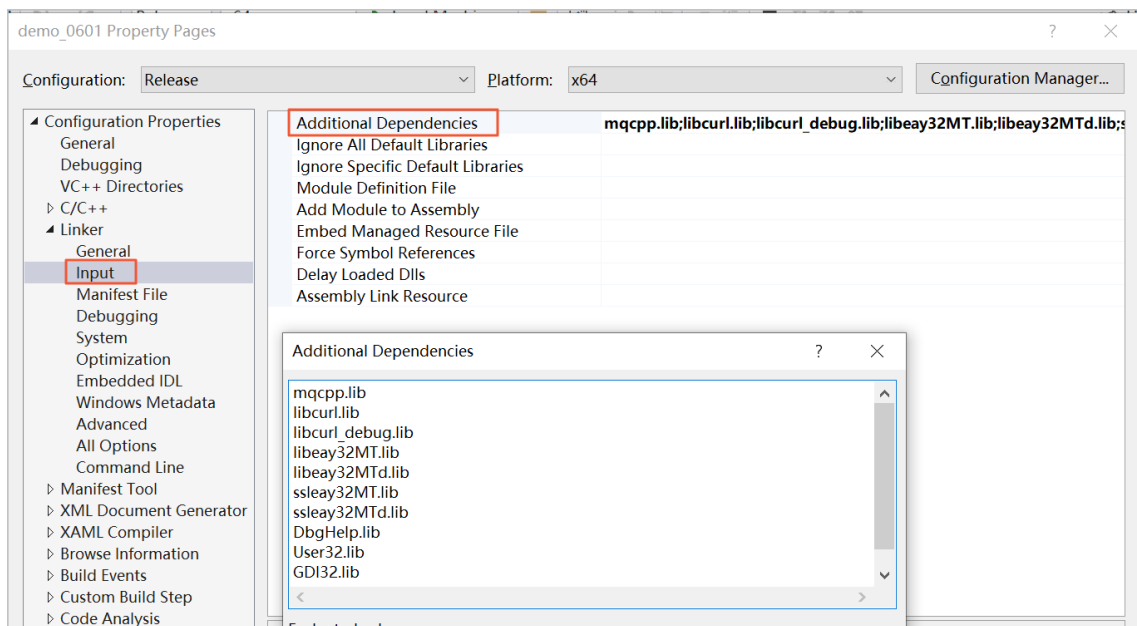
In the Property Pages dialog box of your project, choose **Configuration Properties > Linker > General** in the left-side navigation pane. On the right side, set **Additional Library Directories** to the path of the *lib* folder that you copied in Step 3 and the path of the *lib\windows\{Platform directory}* folder. Configure the *{Platform directory}* variable based on the OS that you use. If you use a 64-bit OS, set this variable to *AMD64*. If you use a 32-bit OS, set this variable to *IA86*.




◦ Set the Additional Dependencies property


In the Property Pages dialog box of your project, choose **Configuration Properties > Linker > Input** in the left-side navigation pane. On the right side, add the following content to the **Additional Dependencies** field:

```
mqcpp.lib
libcurl.lib
libcurl_debug.lib
libeay32MT.lib
libeay32MTd.lib
ssleay32MT.lib
ssleay32MTd.lib
DbgHelp.lib
User32.lib
GDI32.lib
Advapi32.lib
```



5. Copy the sample code to the project file, change the parameter values based on the comments in the code, and then save the changes. For more information about the sample code, see [Sample code](#).
6. Click the  icon to compile the project.

Install the SDK for C++ in a Linux environment

 **Note** The following procedure provides an example on how to install the SDK for C++ in CentOS.

1. Download the SDK for C++ to your on-premises machine and decompress the package. For more information about the download link to the SDK, see [Overview](#).
2. Run the following commands to install all the `libcurl-devel` and `openssl-devel` libraries:

```
yum install libcurl-devel
```

```
yum install openssl-devel
```

3. In the SDK directory, run the following command to compile your C++ project:

```
scons
```

4. After the project is compiled, copy the *include* and *lib* folders in the SDK directory to the C++ project directory that you created on your on-premises machine.
5. Copy the sample code to the project file on your on-premises machine, change the parameter values based on the comments in the code, and then save the changes. For more information about the sample code, see [Sample code](#).
6. Run the following command to compile the project:

```
# Replace producer.cpp with the name of the project file that you created on your on-premises machine.
g++ producer.cpp -o producer lib/libmqcpp.a -I include/ -lcurl -lcrypto
```

6.3.8.2. Send and consume normal messages

Normal messages are messages that have no special features in Message Queue for Apache RocketMQ. They are different from featured messages, such as scheduled messages, delayed messages, ordered messages, and transactional messages. This topic provides sample code to show how to use the HTTP client SDK for C++ to send and consume normal messages.

Prerequisites

The following operations are performed:

- Install the SDK for C++. For more information about the message routing feature, see [Prepare the environment](#).
- Create resources that you want to specify in the code. For example, you must create the instance, topic, and group that you want to specify in the code in the Message Queue for Apache RocketMQ console in advance. For more information about the message routing feature, see [Create resources](#).

Send normal messages

The following sample code provides an example on how to send normal messages:

```
#include <fstream>
#include <time.h>
#include "mq_http_sdk/mq_client.h"
using namespace std;
using namespace mq::http::sdk;
int main() {
    MQClient mqClient(
        // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint,
        // log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane,
        // click Instances. On the Instances page, select the name of your instance. Then, view the HTTP
        // endpoint on the Network Management tab.
        "${HTTP_ENDPOINT}",
        // The AccessKey ID that is used for identity verification. You can obtain the
        // AccessKey ID in the Apsara Uni-manager Operations Console.
        "${ACCESS_KEY}",
        // The AccessKey secret that is used for identity verification. You can obtain
        // the AccessKey secret in the Apsara Uni-manager Operations Console.
        "${SECRET_KEY}"
    );
```

```

// The topic to which you want to send messages. The topic is created in the Message Queue
// for Apache RocketMQ console.
string topic = "${TOPIC}";
// The ID of the instance to which the topic belongs. The instance is created in the Message Queue
// for Apache RocketMQ console.
// If the instance has a namespace, specify the ID of the instance. If the instance does not have a
// namespace, set the instance ID to null or an empty string. You can check whether your instance
// has a namespace on the Instances page in the RocketMQ console.
string instanceId = "${INSTANCE_ID}";
MQProducerPtr producer;
if (instanceId == "") {
    producer = mqClient.getProducerRef(topic);
} else {
    producer = mqClient.getProducerRef(instanceId, topic);
}
try {
    // Cyclically send four messages.
    for (int i = 0; i < 4; i++)
    {
        PublishMessageResponse pmResp;
        // The content of the message.
        TopicMessage pubMsg("Hello, mq!have key!");
        // The custom property of the message.
        pubMsg.putProperty("a", std::to_string(i));
        // The key of the message.
        pubMsg.setMessageKey("MessageKey" + std::to_string(i));
        producer->publishMessage(pubMsg, pmResp);
        cout << "Publish mq message success. Topic is: " << topic
              << ", msgId is:" << pmResp.getMessageId()
              << ", bodyMD5 is:" << pmResp.getMessageBodyMD5() << endl;
    }
} catch (MQServerException& me) {
    cout << "Request Failed: " + me.GetErrorCode() << ", requestId is:" << me.GetRequestId() << endl;
    return -1;
} catch (MQExceptionBase& mb) {
    cout << "Request Failed: " + mb.ToString() << endl;
    return -2;
}
return 0;
}

```

Consume normal messages

The following sample code provides an example on how to consume normal messages:

```

#include <vector>
#include <fstream>
#include "mq_http_sdk/mq_client.h"
#ifdef _WIN32
#include <windows.h>
#else
#include <unistd.h>
#endif

```

```

using namespace std;
using namespace mq::http::sdk;
int main() {
    MQClient mqClient(
        // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint,
        log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane,
        click Instances. On the Instances page, select the name of your instance. Then, view the HT
        TP endpoint on the Network Management tab.
        "${HTTP_ENDPOINT}",
        // The AccessKey ID that is used for identity verification. You can obtain the
        AccessKey ID in the Apsara Uni-manager Operations Console.
        "${ACCESS_KEY}",
        // The AccessKey secret that is used for identity verification. You can obtain
        the AccessKey secret in the Apsara Uni-manager Operations Console.
        "${SECRET_KEY}"
    );
    // The topic from which you want to consume messages. The topic is created in the Messa
    ge Queue for Apache RocketMQ console.
    string topic = "${TOPIC}";
    // The ID of the group that you created in the Message Queue for Apache RocketMQ consol
    e.
    string groupId = "${GROUP_ID}";
    // The ID of the instance to which the topic belongs. The instance is created in the Me
    ssage Queue for Apache RocketMQ console.
    // If the instance has a namespace, specify the ID of the instance. If the instance doe
    s not have a namespace, set the instance ID to null or an empty string. You can check wheth
    er your instance has a namespace on the Instances page in the RocketMQ console.
    string instanceId = "${INSTANCE_ID}";
    MQConsumerPtr consumer;
    if (instanceId == "") {
        consumer = mqClient.getConsumerRef(topic, groupId);
    } else {
        consumer = mqClient.getConsumerRef(instanceId, topic, groupId, "");
    }
    do {
        try {
            std::vector<Message> messages;
            // Consume messages in long polling mode.
            // In long polling mode, if no message in the topic is available for consumptio
            n, the request is suspended on the broker for a specified period of time. If a message beco
            mes available for consumption within this period, the broker immediately sends a response t
            o the consumer. In this example, the period is set to 3 seconds.
            consumer->consumeMessage(
                3, // The maximum number of messages that can be consumed at a time. In
                this example, the value is set to 3. The maximum value that you can specify is 16.
                3, // The length of a long polling period. Unit: seconds. In this exampl
                e, the value is set to 3. The maximum value that you can specify is 30.
                messages
            );
            cout << "Consume: " << messages.size() << " Messages!" << endl;
            // Specify the message consumption logic.
            std::vector<std::string> receiptHandles;
            for (std::vector<Message>::iterator iter = messages.begin();
                iter != messages.end(); ++iter)

```

```

        {
            cout << "MessageId: " << iter->getMessageId()
                << " PublishTime: " << iter->getPublishTime()
                << " Tag: " << iter->getMessageTag()
                << " Body: " << iter->getMessageBody()
                << " FirstConsumeTime: " << iter->getFirstConsumeTime()
                << " NextConsumeTime: " << iter->getNextConsumeTime()
                << " ConsumedTimes: " << iter->getConsumedTimes()
                << " Properties: " << iter->getPropertiesAsString()
                << " Key: " << iter->getMessageKey() << endl;
            receiptHandles.push_back(iter->getReceiptHandle());
        }
        // Obtain an acknowledgment (ACK) from the consumer.
        // If the broker does not receive an ACK for a message from the consumer before
        the period of time that is specified by the Message.NextConsumeTime parameter elapses, the
        broker delivers the message for consumption again.
        // A unique timestamp is specified for the handle of a message each time the me
        ssage is consumed.
        AckMessageResponse bdmResp;
        consumer->ackMessage(receiptHandles, bdmResp);
        if (!bdmResp.isSuccess()) {
            // If the handle of a message times out, the broker cannot receive an ACK f
            or the message from the consumer.
            const std::vector<AckMessageFailedItem>& failedItems =
                bdmResp.getAckMessageFailedItem();
            for (std::vector<AckMessageFailedItem>::const_iterator iter = failedItems.b
            egin();

                iter != failedItems.end(); ++iter)
            {
                cout << "AckFailedItem: " << iter->errorCode
                    << " " << iter->receiptHandle << endl;
            }
        } else {
            cout << "Ack: " << messages.size() << " messages suc!" << endl;
        }
    } catch (MQServerException& me) {
        if (me.GetErrorCode() == "MessageNotExist") {
            cout << "No message to consume! RequestId: " + me.GetRequestId() << endl;
            continue;
        }
        cout << "Request Failed: " + me.GetErrorCode() + ".RequestId: " + me.GetRequest
        Id() << endl;
#ifdef _WIN32
        Sleep(2000);
#else
        usleep(2000 * 1000);
#endif
    } catch (MQExceptionBase& mb) {
        cout << "Request Failed: " + mb.ToString() << endl;
#ifdef _WIN32
        Sleep(2000);
#else
        usleep(2000 * 1000);
#endif
    }
}

```

```
    } while(true);  
}
```

6.3.8.3. Send and consume ordered messages

Ordered messages are a type of message that is published and consumed in a strict order. Ordered messages in Message Queue for Apache RocketMQ are also known as first-in-first-out (FIFO) messages. This topic provides sample code to show how to use the HTTP client SDK for C++ to send and consume ordered messages.

Background information

Ordered messages are classified into the following types:

- Globally ordered message: All messages in a specified topic are published and consumed in first-in-first-out (FIFO) order.
- Partititionally ordered message: All messages in a specified topic are distributed to different partitions by using shard keys. The messages in each partition are published and consumed in FIFO order. A Sharding Key is a key field that is used for ordered messages to identify different partitions. The Sharding Key is different from the key of a normal message.

For more information about the message routing feature, see [Ordered messages](#).

Prerequisites

The following operations are performed:

- Install the SDK for C++. For more information about the message routing feature, see [Prepare the environment](#).
- Create resources that you want to specify in the code. For example, you must create the instance, topic, and group that you want to specify in the code in the Message Queue for Apache RocketMQ console in advance. For more information about the message routing feature, see [Create resources](#).

Send ordered messages

The following sample code provides an example on how to send ordered messages:

```
//#include <iostream>  
#include <fstream>  
#include <time.h>  
#include "mq_http_sdk/mq_client.h"  
using namespace std;  
using namespace mq::http::sdk;  
int main() {  
    MQClient mqClient(  
        // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint,  
        // log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane,  
        // click Instances. On the Instances page, select the name of your instance. Then, view the HT  
        // TP endpoint on the Network Management tab.  
        "${HTTP_ENDPOINT}",  
        // The AccessKey ID that is used for identity verification. You can obtain the  
        // AccessKey ID in the Apsara Uni-manager Operations Console.  
        "${ACCESS_KEY}",  
        // The AccessKey secret that is used for identity verification. You can obtain
```

```

the AccessKey secret in the Apsara Uni-manager Operations Console.
    "${SECRET_KEY}"
);
// The topic to which you want to send messages. The topic is created in the Message Queue for Apache RocketMQ console.
string topic = "${TOPIC}";
// The ID of the instance to which the topic belongs. The instance is created in the Message Queue for Apache RocketMQ console.
// If the instance has a namespace, specify the ID of the instance. If the instance does not have a namespace, set the instance ID to null or an empty string. You can check whether your instance has a namespace on the Instances page in the RocketMQ console.
string instanceId = "${INSTANCE_ID}";
MQProducerPtr producer;
if (instanceId == "") {
    producer = mqClient.getProducerRef(topic);
} else {
    producer = mqClient.getProducerRef(instanceId, topic);
}
try {
    // Cyclically send four messages.
    for (int i = 0; i < 8; i++)
    {
        PublishMessageResponse pmResp;
        // The content of the message.
        TopicMessage pubMsg("Hello, mq!order msg!");
        // The shard key that is used to distribute ordered messages to a specific partition. Shard keys can be used to identify different partitions. A shard key is different from a message key.
        pubMsg.setShardingKey(std::to_string(i % 2));
        // The custom property of the message.
        pubMsg.putProperty("a", std::to_string(i));
        producer->publishMessage(pubMsg, pmResp);
        cout << "Publish mq message success. Topic is: " << topic
            << ", msgId is:" << pmResp.getMessageId()
            << ", bodyMD5 is:" << pmResp.getMessageBodyMD5() << endl;
    }
} catch (MQServerException& me) {
    cout << "Request Failed: " + me.GetErrorCode() << ", requestId is:" << me.GetRequestId() << endl;
    return -1;
} catch (MQExceptionBase& mb) {
    cout << "Request Failed: " + mb.ToString() << endl;
    return -2;
}
return 0;
}

```

Consume ordered messages

The following sample code provides an example on how to consume ordered messages:

```

#include <vector>
#include <fstream>
#include "mq-http-sdk/mq-client.h"

```

```

#ifdef _WIN32
#include <windows.h>
#else
#include <unistd.h>
#endif
using namespace std;
using namespace mq::http::sdk;
int main() {
    MQClient mqClient(
        // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint,
        // log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane,
        // click Instances. On the Instances page, select the name of your instance. Then, view the HT
        // TP endpoint on the Network Management tab.
        "${HTTP_ENDPOINT}",
        // The AccessKey ID that is used for identity verification. You can obtain the
        // AccessKey ID in the Apsara Uni-manager Operations Console.
        "${ACCESS_KEY}",
        // The AccessKey secret that is used for identity verification. You can obtain
        // the AccessKey secret in the Apsara Uni-manager Operations Console.
        "${SECRET_KEY}"
    );

    // The topic from which you want to consume messages. The topic is created in the Messa
    // ge Queue for Apache RocketMQ console.
    string topic = "${TOPIC}";
    // The ID of the group that you created in the Message Queue for Apache RocketMQ consol
    // e.
    string groupId = "${GROUP_ID}";
    // The ID of the instance to which the topic belongs. The instance is created in the Me
    // ssage Queue for Apache RocketMQ console.
    // If the instance has a namespace, specify the ID of the instance. If the instance doe
    // s not have a namespace, set the instance ID to null or an empty string. You can check wheth
    // er your instance has a namespace on the Instances page in the RocketMQ console.
    string instanceId = "${INSTANCE_ID}";
    MQConsumerPtr consumer;
    if (instanceId == "") {
        consumer = mqClient.getConsumerRef(topic, groupId);
    } else {
        consumer = mqClient.getConsumerRef(instanceId, topic, groupId, "");
    }
    do {
        try {
            std::vector<Message> messages;
            // Consume messages in long polling mode. The consumer may pull partitionally o
            // rdered messages from multiple partitions. The consumer consumes messages from the same part
            // ition in the order in which the messages are sent.
            // A consumer pulls partitionally ordered messages from a partition. If the bro
            // ker does not receive an acknowledgment (ACK) for a message after the message is consumed, t
            // he consumer consumes the message again.
            // The consumer can consume the next batch of messages from a partition only af
            // ter all messages that are pulled from the partition in the previous batch are acknowledged
            // to be consumed.
            // In long polling mode, if no message in the topic is available for consumptio
            // n, the request is suspended on the broker for a specified period of time. If a message beco
            // mes available for consumption within this period, the broker immediately sends a response t

```

o the consumer. In this example, the period is set to 3 seconds.

```

    consumer->consumeMessageOrderly(
        3, // The maximum number of messages that can be consumed at a time. In
this example, the value is set to 3. The maximum value that you can specify is 16.
        3, // The length of a long polling period. Unit: seconds. In this exampl
le, the value is set to 3. The maximum value that you can specify is 30.
        messages
    );
    cout << "Consume: " << messages.size() << " Messages!" << endl;
    // Specify the message consumption logic.
    std::vector<std::string> receiptHandles;
    for (std::vector<Message>::iterator iter = messages.begin();
        iter != messages.end(); ++iter)
    {
        cout << "MessageId: " << iter->getMessageId()
            << " PublishTime: " << iter->getPublishTime()
            << " Tag: " << iter->getMessageTag()
            << " Body: " << iter->getMessageBody()
            << " FirstConsumeTime: " << iter->getFirstConsumeTime()
            << " NextConsumeTime: " << iter->getNextConsumeTime()
            << " ConsumedTimes: " << iter->getConsumedTimes()
            << " Properties: " << iter->getPropertiesAsString()
            << " ShardingKey: " << iter->getShardingKey() << endl;
        receiptHandles.push_back(iter->getReceiptHandle());
    }
    // Obtain an ACK from the consumer.
    // If the broker does not receive an ACK for a message from the consumer before
the period of time that is specified by the Message.NextConsumeTime parameter elapses, the
broker delivers the message for consumption again.
    // A unique timestamp is specified for the handle of a message each time the me
ssage is consumed.
    AckMessageResponse bdmResp;
    consumer->ackMessage(receiptHandles, bdmResp);
    if (!bdmResp.isSuccess()) {
        // If the handle of a message times out, the broker cannot receive an ACK f
or the message from the consumer.
        const std::vector<AckMessageFailedItem>& failedItems =
            bdmResp.getAckMessageFailedItem();
        for (std::vector<AckMessageFailedItem>::const_iterator iter = failedItems.b
egin();

            iter != failedItems.end(); ++iter)
        {
            cout << "AckFailedItem: " << iter->errorCode
                << " " << iter->receiptHandle << endl;
        }
    } else {
        cout << "Ack: " << messages.size() << " messages suc!" << endl;
    }
} catch (MQServerException& me) {
    if (me.GetErrorCode() == "MessageNotExist") {
        cout << "No message to consume! RequestId: " + me.GetRequestId() << endl;
        continue;
    }
    cout << "Request Failed: " + me.GetErrorCode() + ".RequestId: " + me.GetRequest

```

```
Id() << endl;
#ifdef _WIN32
    Sleep(2000);
#else
    usleep(2000 * 1000);
#endif
    } catch (MQExceptionBase& mb) {
        cout << "Request Failed: " + mb.ToString() << endl;
#ifdef _WIN32
        Sleep(2000);
#else
        usleep(2000 * 1000);
#endif
    }
    } while(true);
}
```

6.3.8.4. Send and consume scheduled messages and delayed messages

This topic provides sample code to show how to use the HTTP client SDK for C++ to send and consume scheduled messages and delayed messages.

Background information

- **Delayed message:** The producer sends the message to the Message Queue for Apache RocketMQ server, but does not expect the message to be delivered immediately. Instead, the message is delivered to the consumer for consumption after a certain period of time. This message is a delayed message.
- **Scheduled message:** A producer sends a message to a Message Queue for Apache RocketMQ broker and expects the message to be delivered to a consumer at a specified point in time. This type of message is called a scheduled message.

If an HTTP client SDK is used, the code configurations of scheduled messages are the same as the code configurations of delayed messages. Both types of messages are delivered to consumers after a specific period of time based on the attributes of the messages.

For more information about the message routing feature, see [Scheduled messages and delayed messages](#).

Prerequisites

The following operations are performed:

- Install the SDK for C++. For more information about the message routing feature, see [Prepare the environment](#).
- Create resources that you want to specify in the code. For example, you must create the instance, topic, and group that you want to specify in the code in the Message Queue for Apache RocketMQ console in advance. For more information about the message routing feature, see [Create resources](#).

Send scheduled messages or delayed messages

The following sample code provides an example on how to send scheduled messages or delayed messages:

```
#include <fstream>
#include <time.h>
#include "mq_http_sdk/mq_client.h"
using namespace std;
using namespace mq::http::sdk;
int main() {
    MQClient mqClient(
        // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint,
        // log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane,
        // click Instances. On the Instances page, select the name of your instance. Then, view the HT
        // TP endpoint on the Network Management tab.
        "${HTTP_ENDPOINT}",
        // The AccessKey ID that you created in the Resource Access Management (RAM) co
        // nsole. The AccessKey ID is used for identity verification.
        "${ACCESS_KEY}",
        // The AccessKey secret that you created in the RAM console. The AccessKey secr
        // et is used for identity verification.
        "${SECRET_KEY}"
    );
    // The topic to which you want to send messages. The topic is created in the Message Qu
    // eue for Apache RocketMQ console.
    string topic = "${TOPIC}";
    // The ID of the instance to which the topic belongs. The instance is created in the Me
    // ssage Queue for Apache RocketMQ console.
    // If the instance has a namespace, specify the ID of the instance. If the instance doe
    // s not have a namespace, set the instance ID to null or an empty string. You can check wheth
    // er your instance has a namespace on the Instances page in the RocketMQ console.
    string instanceId = "${INSTANCE_ID}";
    MQProducerPtr producer;
    if (instanceId == "") {
        producer = mqClient.getProducerRef(topic);
    } else {
        producer = mqClient.getProducerRef(instanceId, topic);
    }
    try {
        // Cyclically send four messages.
        for (int i = 0; i < 4; i++)
        {
            PublishMessageResponse pmResp;
            // The content of the message.
            TopicMessage pubMsg("Hello, mq!have key!");
            // The custom property of the message.
            pubMsg.putProperty("a",std::to_string(i));
            // The key of the message.
            pubMsg.setMessageKey("MessageKey" + std::to_string(i));
            // The period of time after which the broker delivers the message. In this exam
            // ple, when the broker receives a message, the broker waits for 10 seconds before it delivers
            // the message to the consumer. Set this parameter to a timestamp in milliseconds.
            // If the producer sends a scheduled message, set the parameter to the time int
            // erval between the scheduled point in time and the current point in time.
            pubMsg.setStartDeliverTime(time(NULL) * 1000 + 10 * 1000);
```

```

        producer->publishMessage(pubMsg, pmResp);
        cout << "Publish mq message success. Topic is: " << topic
              << ", msgId is:" << pmResp.getMessageId()
              << ", bodyMD5 is:" << pmResp.getMessageBodyMD5() << endl;
    }
} catch (MQServerException& me) {
    cout << "Request Failed: " + me.GetErrorCode() << ", requestId is:" << me.GetRequestId() << endl;
    return -1;
} catch (MQExceptionBase& mb) {
    cout << "Request Failed: " + mb.ToString() << endl;
    return -2;
}
return 0;
}

```

Consume scheduled messages or delayed messages

The following sample code provides an example on how to consume scheduled messages or delayed messages:

```

#include <vector>
#include <fstream>
#include "mq_http_sdk/mq_client.h"
#ifdef _WIN32
#include <windows.h>
#else
#include <unistd.h>
#endif
using namespace std;
using namespace mq::http::sdk;
int main() {
    MQClient mqClient(
        // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint,
        // log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane,
        // click Instances. On the Instances page, select the name of your instance. Then, view the HTTP
        // endpoint on the Network Management tab.
        "${HTTP_ENDPOINT}",
        // The AccessKey ID that is used for identity verification. You can obtain the
        // AccessKey ID in the Apsara Uni-manager Operations Console.
        "${ACCESS_KEY}",
        // The AccessKey secret that is used for identity verification. You can obtain
        // the AccessKey secret in the Apsara Uni-manager Operations Console.
        "${SECRET_KEY}"
    );

    // The topic from which you want to consume messages. The topic is created in the Message
    // Queue for Apache RocketMQ console.
    string topic = "${TOPIC}";
    // The ID of the group that you created in the Message Queue for Apache RocketMQ console.
    string groupId = "${GROUP_ID}";
    // The ID of the instance to which the topic belongs. The instance is created in the Message
    // Queue for Apache RocketMQ console.
    // If the instance has a namespace, specify the ID of the instance. If the instance doe

```

s not have a namespace, set the instance ID to null or an empty string. You can check whether your instance has a namespace on the Instances page in the RocketMQ console.

```
string instanceId = "${INSTANCE_ID}";
MQConsumerPtr consumer;
if (instanceId == "") {
    consumer = mqClient.getConsumerRef(topic, groupId);
} else {
    consumer = mqClient.getConsumerRef(instanceId, topic, groupId, "");
}
do {
    try {
        std::vector<Message> messages;
        // Consume messages in long polling mode.
        // In long polling mode, if no message in the topic is available for consumption, the request is suspended on the broker for a specified period of time. If a message becomes available for consumption within this period, the broker immediately sends a response to the consumer. In this example, the period is set to 3 seconds.
        consumer->consumeMessage(
            3, // The maximum number of messages that can be consumed at a time. In this example, the value is set to 3. The maximum value that you can specify is 16.
            3, // The length of a long polling period. Unit: seconds. In this example, the value is set to 3. The maximum value that you can specify is 30.
            messages
        );
        cout << "Consume: " << messages.size() << " Messages!" << endl;
        // Specify the message consumption logic.
        std::vector<std::string> receiptHandles;
        for (std::vector<Message>::iterator iter = messages.begin();
            iter != messages.end(); ++iter)
        {
            cout << "MessageId: " << iter->getMessageId()
                << " PublishTime: " << iter->getPublishTime()
                << " Tag: " << iter->getMessageTag()
                << " Body: " << iter->getMessageBody()
                << " FirstConsumeTime: " << iter->getFirstConsumeTime()
                << " NextConsumeTime: " << iter->getNextConsumeTime()
                << " ConsumedTimes: " << iter->getConsumedTimes()
                << " Properties: " << iter->getPropertiesAsString()
                << " Key: " << iter->getMessageKey() << endl;
            receiptHandles.push_back(iter->getReceiptHandle());
        }
        // Obtain an acknowledgment (ACK) from the consumer.
        // If the broker does not receive an ACK for a message from the consumer before the period of time that is specified by the Message.NextConsumeTime parameter elapses, the broker delivers the message for consumption again.
        // A unique timestamp is specified for the handle of a message each time the message is consumed.
        AckMessageResponse bdmResp;
        consumer->ackMessage(receiptHandles, bdmResp);
        if (!bdmResp.isSuccess()) {
            // If the handle of a message times out, the broker cannot receive an ACK for the message from the consumer.
            const std::vector<AckMessageFailedItem>& failedItems =
                bdmResp.getAckMessageFailedItem();
```

```

        for (std::vector<AckMessageFailedItem>::const_iterator iter = failedItems.begin();
            iter != failedItems.end(); ++iter)
        {
            cout << "AckFailedItem: " << iter->errorCode
                << " " << iter->receiptHandle << endl;
        }
    } else {
        cout << "Ack: " << messages.size() << " messages suc!" << endl;
    }
} catch (MQServerException& me) {
    if (me.GetErrorCode() == "MessageNotExist") {
        cout << "No message to consume! RequestId: " + me.GetRequestId() << endl;
        continue;
    }
    cout << "Request Failed: " + me.GetErrorCode() + ".RequestId: " + me.GetRequestId() << endl;
#ifdef _WIN32
    Sleep(2000);
#else
    usleep(2000 * 1000);
#endif
    } catch (MQExceptionBase& mb) {
        cout << "Request Failed: " + mb.ToString() << endl;
#ifdef _WIN32
        Sleep(2000);
#else
        usleep(2000 * 1000);
#endif
    }
} while(true);
}

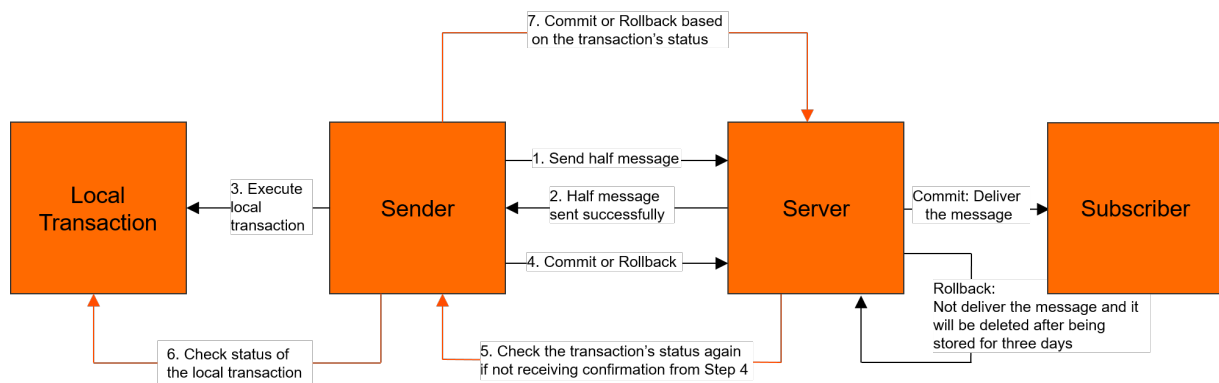
```

6.3.8.5. Send and consume transactional messages

Message Queue for Apache RocketMQ provides a distributed transaction processing feature that is similar to X/Open XA. Message Queue for Apache RocketMQ uses transactional messages to ensure transactional consistency. This topic provides sample code to show how to use the HTTP client SDK for C++ to send and consume transactional messages.

Background information

The following figure shows the interaction process of transactional messages.



For more information about the message routing feature, see [Transactional messages](#).

Prerequisites

The following operations are performed:

- Install the SDK for C++. For more information about the message routing feature, see [Prepare the environment](#).
- Create resources that you want to specify in the code. For example, you must create the instance, topic, and group that you want to specify in the code in the Message Queue for Apache RocketMQ console in advance. For more information about the message routing feature, see [Create resources](#).

Send transactional messages

The following sample code provides an example on how to send transactional messages:

```

//#include <iostream>
#include <fstream>
#ifdef _WIN32
#include <windows.h>
#include <process.h>
#else
#include "pthread.h"
#endif
#include "mq_http_sdk/mq_client.h"
using namespace std;
using namespace mq::http::sdk;
const int32_t pubMsgCount = 4;
const int32_t halfCheckCount = 3;
void processCommitRollError(AckMessageResponse& bdmResp, const std::string& messageId) {
    if (bdmResp.isSuccess()) {
        cout << "Commit/Roll Transaction Suc: " << messageId << endl;
        return;
    }
    const std::vector<AckMessageFailedItem>& failedItems =
        bdmResp.getAckMessageFailedItem();
    for (std::vector<AckMessageFailedItem>::const_iterator iter = failedItems.begin();
        iter != failedItems.end(); ++iter)
    {
        cout << "Commit/Roll Transaction ERROR: " << iter->errorCode
            << " " << iter->receiptHandle << endl;
    }
}
  
```

```

}
#ifdef WIN32
unsigned __stdcall consumeHalfMessageThread(void *arg)
#else
void* consumeHalfMessageThread(void *arg)
#endif
{
    MQTransProducerPtr transProducer = *(MQTransProducerPtr*)(arg);
    int count = 0;
    do {
        std::vector<Message> halfMsgs;
        try {
            // Consume messages in long polling mode.
            // In long polling mode, if no message in the topic is available for consumption, the request is suspended on the broker for a specified period of time. If a message becomes available for consumption within this period, the broker immediately sends a response to the consumer. In this example, the period is set to 3 seconds.
            transProducer->consumeHalfMessage(
                1, // The maximum number of messages that can be consumed at a time. In this example, the value is set to 1. The maximum value that you can specify is 16.
                3, // The length of a long polling period. Unit: seconds. In this example, the value is set to 3. The maximum value that you can specify is 30.
                halfMsgs
            );
        } catch (MQServerException& me) {
            if (me.GetErrorCode() == "MessageNotExist") {
                cout << "No half message to consume! RequestId: " + me.GetRequestId() << endl;
                continue;
            }
            cout << "Request Failed: " + me.GetErrorCode() + ".RequestId: " + me.GetRequestId() << endl;
        }
        if (halfMsgs.size() == 0) {
            continue;
        }
        cout << "Consume Half: " << halfMsgs.size() << " Messages!" << endl;
        // Process half messages.
        std::vector<std::string> receiptHandles;
        for (std::vector<Message>::iterator iter = halfMsgs.begin();
            iter != halfMsgs.end(); ++iter)
        {
            cout << "MessageId: " << iter->getMessageId()
                << " PublishTime: " << iter->getPublishTime()
                << " Tag: " << iter->getMessageTag()
                << " Body: " << iter->getMessageBody()
                << " FirstConsumeTime: " << iter->getFirstConsumeTime()
                << " NextConsumeTime: " << iter->getNextConsumeTime()
                << " ConsumedTimes: " << iter->getConsumedTimes()
                << " Properties: " << iter->getPropertiesAsString()
                << " Key: " << iter->getMessageKey() << endl;
            int32_t consumedTimes = iter->getConsumedTimes();
            const std::string propA = iter->getProperty("a");
            const std::string handle = iter->getReceiptHandle();

```

```

AckMessageResponse bdmResp;
if (propA == "1") {
    cout << "Commit msg.." << endl;
    transProducer->commit(handle, bdmResp);
    count++;
} else if(propA == "2") {
    if (consumedTimes > 1) {
        cout << "Commit msg.." << endl;
        transProducer->commit(handle, bdmResp);
        count++;
    } else {
        cout << "Commit Later!!!" << endl;
    }
} else if(propA == "3") {
    cout << "Rollback msg.." << endl;
    transProducer->rollback(handle, bdmResp);
    count++;
} else {
    transProducer->commit(handle, bdmResp);
    cout << "Unkown msg.." << endl;
}

// If the transactional message is not committed or rolled back before the period of time specified by the NextConsumeTime parameter elapses, the commit or rollback operation fails.
processCommitRollError(bdmResp, iter->getMessageId());
}
} while(count < halfCheckCount);
#ifdef WIN32
    return 0;
#else
    return NULL;
#endif
}

int main() {
    MQClient mqClient(
        // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint, log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click Instances. On the Instances page, select the name of your instance. Then, view the HTTP endpoint on the Network Management tab.
        "${HTTP_ENDPOINT}",
        // The AccessKey ID that is used for identity verification. You can obtain the AccessKey ID in the Apsara Uni-manager Operations Console.
        "${ACCESS_KEY}",
        // The AccessKey secret that is used for identity verification. You can obtain the AccessKey secret in the Apsara Uni-manager Operations Console.
        "${SECRET_KEY}"
    );

    // The topic to which you want to send messages. The topic is created in the Message Queue for Apache RocketMQ console.
    string topic = "${TOPIC}";

    // The ID of the instance to which the topic belongs. The instance is created in the Message Queue for Apache RocketMQ console.

    // If the instance has a namespace, specify the ID of the instance. If the instance does not have a namespace, set the instance ID to null or an empty string. You can check wheth

```

```

er your instance has a namespace on the Instances page in the RocketMQ console.
    string instanceId = "${INSTANCE_ID}";
    // The ID of the group that you created in the Message Queue for Apache RocketMQ console.
e.
    string groupId = "${GROUP_ID}";
    MQTransProducerPtr transProducer;
    if (instanceId == "") {
        transProducer = mqClient.getTransProducerRef(topic, groupId);
    } else {
        transProducer = mqClient.getTransProducerRef(instanceId, topic, groupId);
    }
    // The client needs a thread or a process to process unacknowledged transactional messages.
    // Start a thread to process unacknowledged transactional messages.
#ifdef WIN32
    HANDLE thread;
    unsigned int threadId;
    thread = (HANDLE)_beginthreadex(NULL, 0, consumeHalfMessageThread, &transProducer, 0, &
threadId);
#else
    pthread_t thread;
    pthread_create(&thread, NULL, consumeHalfMessageThread, static_cast<void *>(&transProducer));
#endif
    try {
        for (int i = 0; i < pubMsgCount; i++)
        {
            PublishMessageResponse pmResp;
            TopicMessage pubMsg("Hello, mq, trans_msg!");
            pubMsg.putProperty("a", std::to_string(i));
            pubMsg.setMessageKey("ImKey");
            pubMsg.setTransCheckImmunityTime(10);
            transProducer->publishMessage(pubMsg, pmResp);
            cout << "Publish mq message success. Topic:" << topic
                << ", msgId:" << pmResp.getMessageId()
                << ", bodyMD5:" << pmResp.getMessageBodyMD5()
                << ", Handle:" << pmResp.getReceiptHandle() << endl;
            if (i == 0) {
                // After the producer sends the transactional message, the broker obtains the handle of the half message that corresponds to the transactional message and commits or rolls back the transactional message based on the status of the handle.
                // If a transactional message is not committed or rolled back after the period of time specified by the TransCheckImmunityTime parameter elapses, the commit or rollback operation fails.
                AckMessageResponse bdmResp;
                transProducer->commit(pmResp.getReceiptHandle(), bdmResp);
                processCommitRollError(bdmResp, pmResp.getMessageId());
            }
        }
    } catch (MQServerException& me) {
        cout << "Request Failed: " + me.GetErrorCode() << ", requestId is:" << me.GetRequestId() << endl;
    } catch (MQExceptionBase& mb) {
        cout << "Request Failed: " + mb.ToString() << endl;
    }

```

```

    }
#ifdef WIN32
    WaitForSingleObject(thread, INFINITE);
    CloseHandle(thread);
#else
    pthread_join(thread, NULL);
#endif
    return 0;
}

```

Consume transactional messages

The following sample code provides an example on how to consume transactional messages:

```

#include <vector>
#include <fstream>
#include "mq_http_sdk/mq_client.h"
#ifdef _WIN32
#include <windows.h>
#else
#include <unistd.h>
#endif
using namespace std;
using namespace mq::http::sdk;
int main() {
    MQClient mqClient(
        // The HTTP endpoint to which you want to connect. To obtain the HTTP endpoint,
        // log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane,
        // click Instances. On the Instances page, select the name of your instance. Then, view the HT
        // TP endpoint on the Network Management tab.
        "${HTTP_ENDPOINT}",
        // The AccessKey ID that is used for identity verification. You can obtain the
        // AccessKey ID in the Apsara Uni-manager Operations Console.
        "${ACCESS_KEY}",
        // The AccessKey secret that is used for identity verification. You can obtain
        // the AccessKey secret in the Apsara Uni-manager Operations Console.
        "${SECRET_KEY}"
    );
    // The topic from which you want to consume messages. The topic is created in the Messa
    // ge Queue for Apache RocketMQ console.
    string topic = "${TOPIC}";
    // The ID of the group that you created in the Message Queue for Apache RocketMQ consol
    // e.
    string groupId = "${GROUP_ID}";
    // The ID of the instance to which the topic belongs. The instance is created in the Me
    // ssage Queue for Apache RocketMQ console.
    // If the instance has a namespace, specify the ID of the instance. If the instance doe
    // s not have a namespace, set the instance ID to null or an empty string. You can check wheth
    // er your instance has a namespace on the Instances page in the RocketMQ console.
    string instanceId = "${INSTANCE_ID}";
    MQConsumerPtr consumer;
    if (instanceId == "") {
        consumer = mqClient.getConsumerRef(topic, groupId);
    } else {

```

```

    consumer = mqClient.getConsumerRef(instanceId, topic, groupId, "");
}
do {
    try {
        std::vector<Message> messages;
        // Consume messages in long polling mode.
        // In long polling mode, if no message in the topic is available for consumption, the request is suspended on the broker for a specified period of time. If a message becomes available for consumption within this period, the broker immediately sends a response to the consumer. In this example, the period is set to 3 seconds.
        consumer->consumeMessage(
            3, // The maximum number of messages that can be consumed at a time. In this example, the value is set to 3. The maximum value that you can specify is 16.
            3, // The length of a long polling period. Unit: seconds. In this example, the value is set to 3. The maximum value that you can specify is 30.
            messages
        );
        cout << "Consume: " << messages.size() << " Messages!" << endl;
        // Specify the message consumption logic.
        std::vector<std::string> receiptHandles;
        for (std::vector<Message>::iterator iter = messages.begin();
            iter != messages.end(); ++iter)
        {
            cout << "MessageId: " << iter->getMessageId()
                << " PublishTime: " << iter->getPublishTime()
                << " Tag: " << iter->getMessageTag()
                << " Body: " << iter->getMessageBody()
                << " FirstConsumeTime: " << iter->getFirstConsumeTime()
                << " NextConsumeTime: " << iter->getNextConsumeTime()
                << " ConsumedTimes: " << iter->getConsumedTimes()
                << " Properties: " << iter->getPropertiesAsString()
                << " Key: " << iter->getMessageKey() << endl;
            receiptHandles.push_back(iter->getReceiptHandle());
        }
        // Obtain an acknowledgment (ACK) from the consumer.
        // If the broker does not receive an ACK for a message from the consumer before the period of time that is specified by the Message.NextConsumeTime parameter elapses, the broker delivers the message for consumption again.
        // A unique timestamp is specified for the handle of a message each time the message is consumed.
        AckMessageResponse bdmResp;
        consumer->ackMessage(receiptHandles, bdmResp);
        if (!bdmResp.isSuccess()) {
            // If the handle of a message times out, the broker cannot receive an ACK for the message from the consumer.
            const std::vector<AckMessageFailedItem>& failedItems =
                bdmResp.getAckMessageFailedItem();
            for (std::vector<AckMessageFailedItem>::const_iterator iter = failedItems.begin();
                iter != failedItems.end(); ++iter)
            {
                cout << "AckFailedItem: " << iter->errorCode
                    << " " << iter->receiptHandle << endl;
            }
        }
    }
} while (true);

```

```
        } else {
            cout << "Ack: " << messages.size() << " messages suc!" << endl;
        }
    } catch (MQServerException& me) {
        if (me.GetErrorCode() == "MessageNotExist") {
            cout << "No message to consume! RequestId: " + me.GetRequestId() << endl;
            continue;
        }
        cout << "Request Failed: " + me.GetErrorCode() + ".RequestId: " + me.GetRequest
Id() << endl;
#ifdef _WIN32
        Sleep(2000);
#else
        usleep(2000 * 1000);
#endif
    } catch (MQExceptionBase& mb) {
        cout << "Request Failed: " + mb.ToString() << endl;
#ifdef _WIN32
        Sleep(2000);
#else
        usleep(2000 * 1000);
#endif
    }
} while(true);
}
```

7. Best practices

7.1. Clustering consumption and broadcasting consumption

This topic describes the terms, scenarios, and usage notes of clustering consumption and broadcasting consumption in Message Queue for Apache RocketMQ.

Terms

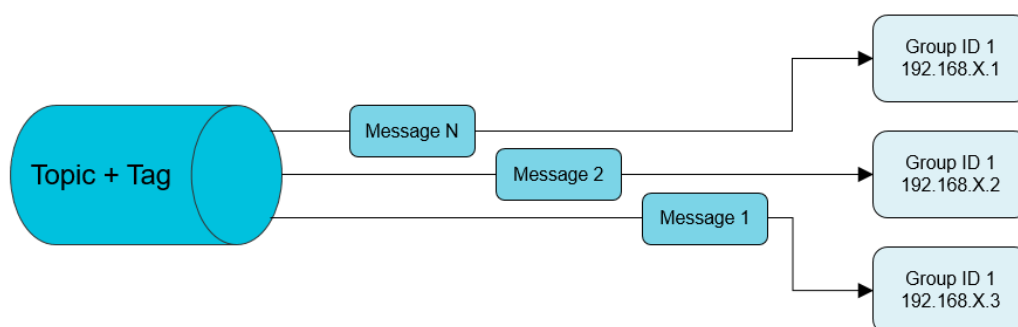
Message Queue for Apache RocketMQ is a messaging system that is based on the publish-subscribe model. In Message Queue for Apache RocketMQ, a consumer subscribes to a topic to obtain and consume messages. In most cases, consumer applications use distributed systems. Multiple machines are deployed in one cluster. Therefore, Message Queue for Apache RocketMQ defines the following terms:

- **Cluster:** Consumers identified by the same group ID belong to the same cluster. These consumers must have the same consumption logic that also involves tags. These consumers can be considered logically as one consumption node.
- **Clustering consumption:** In this mode, a message needs to be processed only by a consumer in the cluster.
- **Broadcasting consumption:** In this mode, Message Queue for Apache RocketMQ broadcasts each message to all consumers registered in the cluster to ensure that the message is consumed by each consumer at least once.

Scenarios

- **Clustering consumption mode:**

Clustering consumption mode



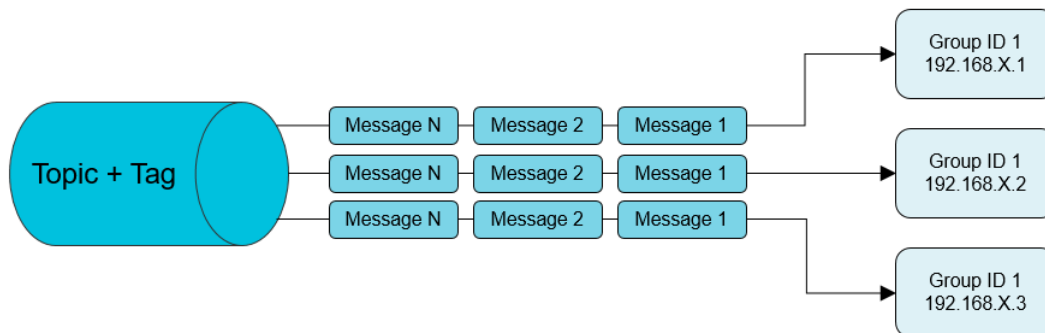
Scenarios and usage notes:

- Consumers are deployed in a cluster and each message needs to be processed only once.
- The consumption progress is recorded on the Message Queue for Apache RocketMQ broker. Therefore, the reliability is high.
- In clustering consumption mode, each message is delivered to only one consumer in the cluster for processing. If a message needs to be processed by each consumer in the cluster, use the broadcasting consumption mode.

- In clustering consumption mode, it is not guaranteed that a failed message can be routed to the same consumer each time the message is redelivered. Therefore, no definitive assumptions can be made during message processing.

- **Broadcasting consumption mode:**

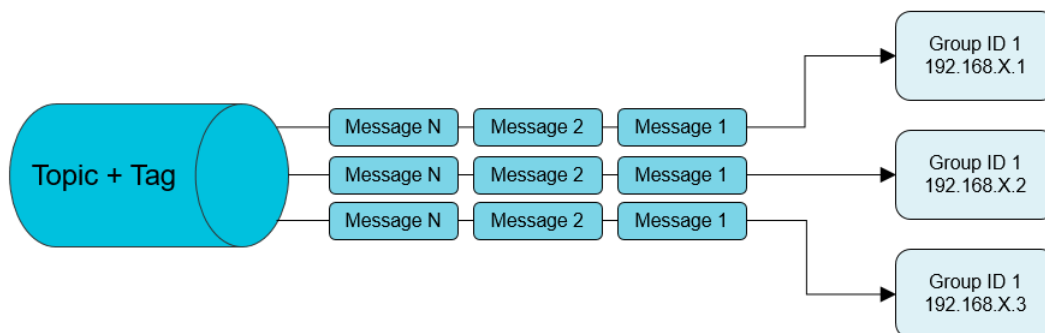
Broadcasting consumption mode



Scenarios and usage notes:

- Ordered messages are not supported in broadcasting consumption mode.
 - Consumer offsets cannot be reset in broadcasting consumption mode.
 - Each message needs to be processed by multiple consumers that are subject to the same logic.
 - The consumption progress is recorded on the consumer. Duplicate messages are more likely to occur in broadcasting consumption mode than in clustering consumption mode.
 - In broadcasting consumption mode, Message Queue for Apache RocketMQ ensures that each message is consumed at least once by each consumer, but does not resend a message that fails to be consumed. Therefore, you need to pay attention to consumption failures.
 - In broadcasting consumption mode, a consumer starts consumption from the latest message each time the consumer is restarted. The consumer automatically skips the messages that are sent to the Message Queue for Apache RocketMQ broker when the consumer is stopped. Therefore, use this mode with caution.
 - In broadcasting consumption mode, each message is repeatedly processed by many consumers. Therefore, we recommend that you use the clustering consumption mode whenever possible.
 - Only Java clients support the broadcasting consumption mode.
 - In broadcasting consumption mode, the Message Queue for Apache RocketMQ broker does not record the consumption progress. In this mode, you cannot query accumulated messages, configure message accumulation alerts, or query subscriptions in the Message Queue for Apache RocketMQ console.
- **Use the clustering consumption mode to simulate the broadcasting consumption mode**
If the broadcasting consumption mode is required for your business, you can create multiple group IDs to subscribe to the same topic.

Use the clustering consumption mode to simulate the broadcasting consumption mode



Scenarios and usage notes:

- Each message needs to be processed by multiple consumers, and the logic of the consumers can be the same or different.
- The consumption progress is recorded on the Message Queue for Apache RocketMQ broker. Therefore, the reliability is higher than that in broadcasting consumption mode.
- For each group ID, one or more consumer instances can be deployed. When multiple consumer instances are deployed, these instances compose a cluster and work together to consume messages. Assume that three consumer instances C1, C2, and C3 are deployed for Group ID 1. These instances share the messages sent from the Message Queue for Apache RocketMQ broker to Group ID 1. In addition, these instances must subscribe to the same topics and same tags.

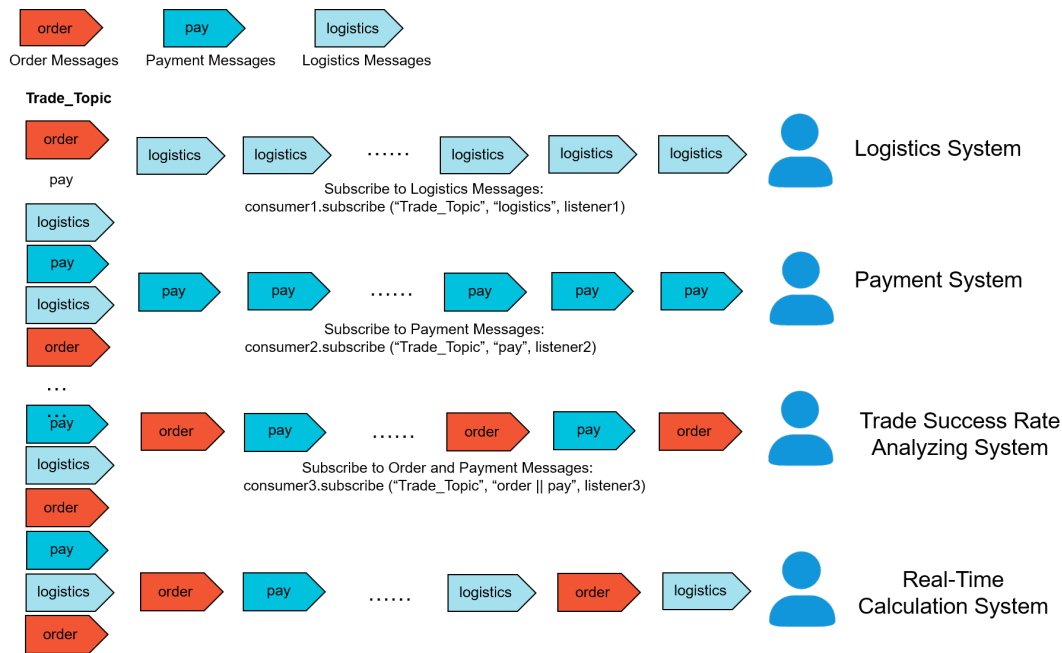
7.2. Message filtering

This topic describes how Message Queue for Apache RocketMQ consumers filter messages on the Message Queue for Apache RocketMQ broker based on tags.

A tag is used to classify messages in a topic into different types. Message Queue for Apache RocketMQ allows consumers to filter messages by using tags. This ensures that the consumers consume only messages that they are concerned with.

The following figure shows an example in the e-commerce transaction scenario. In the process from placing an order to receiving the product by the customer, a series of messages including order messages, payment messages, and logistics messages are generated. These messages are sent to the Trade_Topic topic and subscribed to by different systems, such as the payment system, analysis system for transaction success rate, and real-time computing system. Among these systems, the logistics system receives only logistics messages and the real-time computing system receives all transaction-related messages, including the order messages, payment messages, and logistics messages.

Filter messages



Note To classify messages, you can create multiple topics, or create multiple tags in the same topic. In most cases, messages in one topic are irrelevant to those in another topic. Tags are used to distinguish between relevant messages in the same topic. For example, you can create different tags in the same topic to distinguish between a set and its subsets or distinguish between processes in sequence.

Examples

- Send messages

Specify a tag for each message before the message is sent.

```
Message msg = new Message("MQ_TOPIC", "TagA", "Hello MQ".getBytes());
```

- Subscribe to messages

- Consumption method 1

If a consumer needs to subscribe to messages of all types in a topic, use an asterisk (*) to represent all tags.

```
consumer.subscribe("MQ_TOPIC", "*", new MessageListener() {
    public Action consume(Message message, ConsumeContext context) {
        System.out.println(message.getMsgID());
        return Action.CommitMessage;
    }
});
```

- Consumption method 2

If a consumer needs to subscribe to messages of a specific type in a topic, specify the corresponding tag.

```
consumer.subscribe("MQ_TOPIC", "TagA", new MessageListener() {  
    public Action consume(Message message, ConsumeContext context) {  
        System.out.println(message.getMsgID());  
        return Action.CommitMessage;  
    }  
});
```

- Consumption method 3

If a consumer needs to subscribe to messages of multiple types in a topic, separate the corresponding tags with two vertical bars (||).

```
consumer.subscribe("MQ_TOPIC", "TagA||TagB", new MessageListener() {  
    public Action consume(Message message, ConsumeContext context) {  
        System.out.println(message.getMsgID());  
        return Action.CommitMessage;  
    }  
});
```

- Consumption method 4 (error example)

If a consumer subscribes to messages with specific tags in a topic multiple times, the tags in the last subscription prevail:

```
// In the following error code, the consumer can receive only messages with TagB in MQ_  
// TOPIC and cannot receive messages with TagA.  
consumer.subscribe("MQ_TOPIC", "TagA", new MessageListener() {  
    public Action consume(Message message, ConsumeContext context) {  
        System.out.println(message.getMsgID());  
        return Action.CommitMessage;  
    }  
});  
consumer.subscribe("MQ_TOPIC", "TagB", new MessageListener() {  
    public Action consume(Message message, ConsumeContext context) {  
        System.out.println(message.getMsgID());  
        return Action.CommitMessage;  
    }  
});
```

7.3. Subscription consistency

In Message Queue for Apache RocketMQ, a group ID represents a consumer instance group. For most distributed applications, multiple consumer instances are attached to the same group ID. Subscription consistency means that the processing logic of all consumer instances identified by the same group ID must be identical. If the subscriptions of the consumer instances are inconsistent, errors occur in the message consumption logic and messages may be lost.

Subscriptions in Message Queue for Apache RocketMQ involve topics and tags. Therefore, all consumer instances identified by the same group ID must be consistent in the following two aspects to ensure subscription consistency:

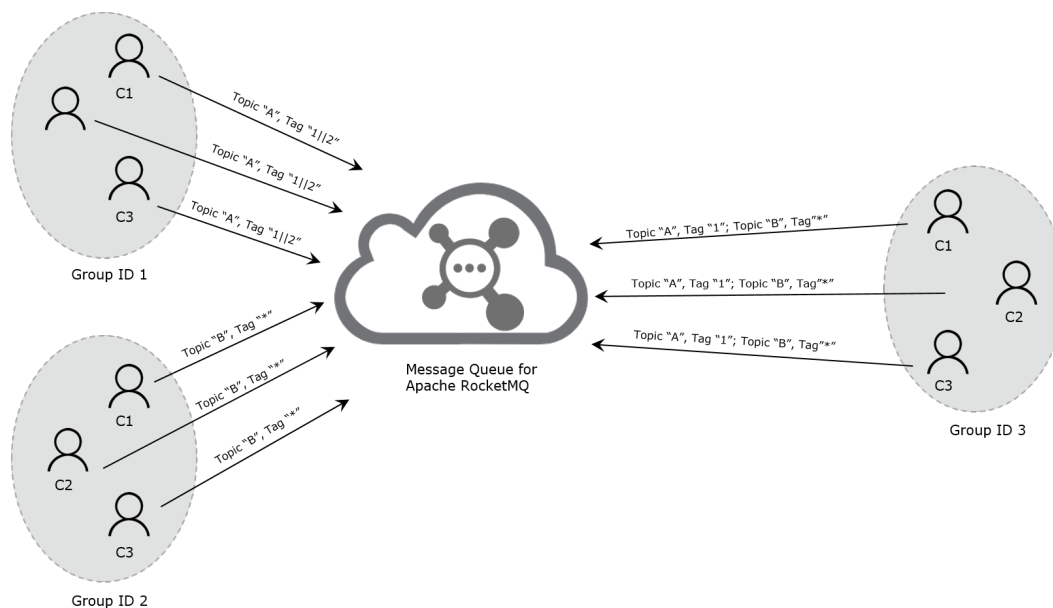
- The topics to which the consumer instances subscribe must be the same.
- The tags of the topics to which the consumer instances subscribe must be the same.

Examples of subscriptions

- Consistent subscriptions

Multiple group IDs subscribe to multiple topics, and the subscriptions of different consumer instances identified by the same group ID are consistent, as shown in the following figure.

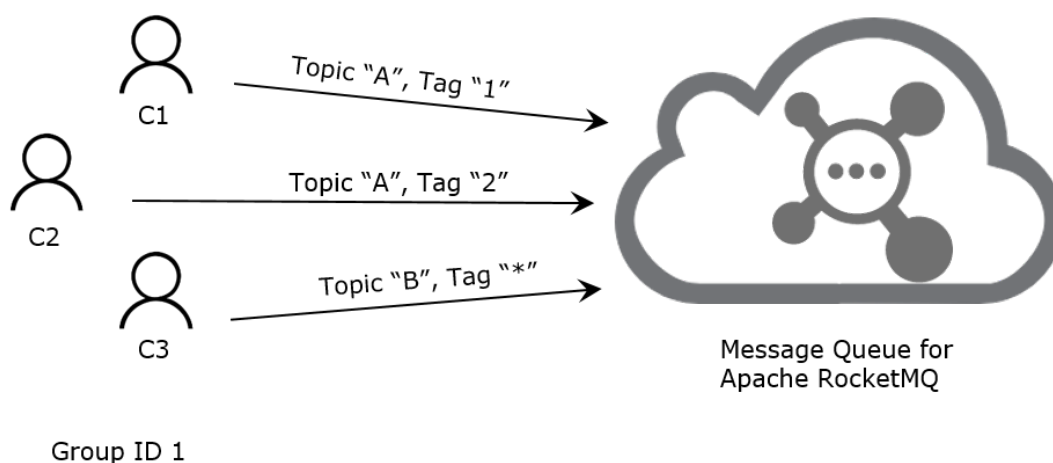
Consistent subscriptions



- Inconsistent subscriptions

One group ID subscribes to multiple topics, but the subscriptions of different consumer instances identified by the group ID are inconsistent, as shown in the following figure.

Inconsistent subscriptions



Sample code of subscriptions

Sample code of inconsistent subscriptions

- Example 1

In the following example, two consumer instances identified by the same group ID subscribe to different topics.

Consumer instance 1-1:

```
Properties properties = new Properties();
properties.put(PropertyKeyConst.GROUP_ID, "GID_jodie_test_1");
Consumer consumer = ONSFactory.createConsumer(properties);
consumer.subscribe("jodie_test_A", "*", new MessageListener() {
    public Action consume(Message message, ConsumeContext context) {
        System.out.println(message.getMsgID());
        return Action.CommitMessage;
    }
});
```

Consumer instance 1-2:

```
Properties properties = new Properties();
properties.put(PropertyKeyConst.GROUP_ID, "GID_jodie_test_1");
Consumer consumer = ONSFactory.createConsumer(properties);
consumer.subscribe("jodie_test_B ", "*", new MessageListener() {
    public Action consume(Message message, ConsumeContext context) {
        System.out.println(message.getMsgID());
        return Action.CommitMessage;
    }
});
```

- Example 2

In the following example, two consumer instances identified by the same group ID subscribe to the same topic but subscribe to different numbers of tags of the topic. Consumer instance 2-1 has subscribed to Tag A, whereas consumer instance 2-2 has not specified a tag.

Consumer instance 2-1:

```
Properties properties = new Properties();
properties.put(PropertyKeyConst.GROUP_ID, "GID_jodie_test_2");
Consumer consumer = ONSFactory.createConsumer(properties);
consumer.subscribe("jodie_test_A", "TagA", new MessageListener() {
    public Action consume(Message message, ConsumeContext context) {
        System.out.println(message.getMsgID());
        return Action.CommitMessage;
    }
});
```

Consumer instance 2-2:

```
Properties properties = new Properties();
properties.put(PropertyKeyConst.GROUP_ID, "GID_jodie_test_2");
Consumer consumer = ONSFactory.createConsumer(properties);
consumer.subscribe("jodie_test_A", "*", new MessageListener() {
    public Action consume(Message message, ConsumeContext context) {
        System.out.println(message.getMsgID());
        return Action.CommitMessage;
    }
});
```

- Example 3

In this example, the subscriptions are inconsistent due to the following reasons:

- Two consumer instances identified by the same group ID subscribe to different numbers of topics.
- Both the consumer instances subscribe to one same topic but subscribe to different tags of the topic.

Consumer instance 3-1:

```
Properties properties = new Properties();
properties.put(PropertyKeyConst.GROUP_ID, "GID_jodie_test_3");
Consumer consumer = ONSFactory.createConsumer(properties);
consumer.subscribe("jodie_test_A", "TagA", new MessageListener() {
    public Action consume(Message message, ConsumeContext context) {
        System.out.println(message.getMsgID());
        return Action.CommitMessage;
    }
});
consumer.subscribe("jodie_test_B", "TagB", new MessageListener() {
    public Action consume(Message message, ConsumeContext context) {
        System.out.println(message.getMsgID());
        return Action.CommitMessage;
    }
});
```

Consumer instance 3-2:

```
Properties properties = new Properties();
properties.put(PropertyKeyConst.GROUP_ID, "GID_jodie_test_3");
Consumer consumer = ONSFactory.createConsumer(properties);
consumer.subscribe("jodie_test_A", "TagB", new MessageListener() {
    public Action consume(Message message, ConsumeContext context) {
        System.out.println(message.getMsgID());
        return Action.CommitMessage;
    }
});
```

7.4. Consumption idempotence

After a Message Queue for Apache RocketMQ consumer receives messages, the consumer needs to perform idempotent processing on these messages based on the unique business-specific keys of the messages.

Necessity for consumption idempotence

In Internet applications, duplicate messages may occur in Message Queue for Apache RocketMQ especially if Internet connection is unstable. Duplicate messages may occur in the following two scenarios:

- A producer repeatedly sends a message to the Message Queue for RocketMQ broker.

If a network disconnection occurs or the producer breaks down after a message is sent to and persisted in the Message Queue for Apache RocketMQ broker, the broker fails to respond to the producer. If the producer realizes that the message failed to be sent and resends the message, the consumer subsequently receives two messages that have the same content and message ID.

- The Message Queue for Apache RocketMQ broker repeatedly delivers a message to a consumer.

A message is delivered to a consumer and is processed by the consumer. However, a network disconnection occurs when the consumer sends a response to the Message Queue for Apache RocketMQ broker. To ensure that the message is consumed at least once, the Message Queue for Apache RocketMQ broker redelivers the previously processed message after the network connection recovers. The consumer subsequently receives two messages that have the same content and message ID.

- Duplicate messages are generated when rebalancing is triggered in scenarios such as network jitter, broker restart, and consumer application restart.

Traffic is rebalanced if the Message Queue for Apache RocketMQ broker or consumer client is restarted or scaled. In this case, a consumer may receive duplicate messages.

Solution

Message IDs may be duplicate. Therefore, we recommend that you do not implement idempotent processing based on message IDs. The best practice is to use unique business-specific keys for idempotent processing. The following sample code provides an example on how to specify a unique business-specific key for a message:

```
Message message = new Message();
message.setKey("ORDERID_100");
SendResult sendResult = producer.send(message);
```

The following sample code provides an example on how a consumer performs idempotent processing after it receives a message:

```
consumer.subscribe("ons_test", "*", new MessageListener() {
    public Action consume(Message message, ConsumeContext context) {
        String key = message.getKey()
        // Use the unique business-specific key for idempotent processing.
    }
});
```

7.5. Active geo-redundancy

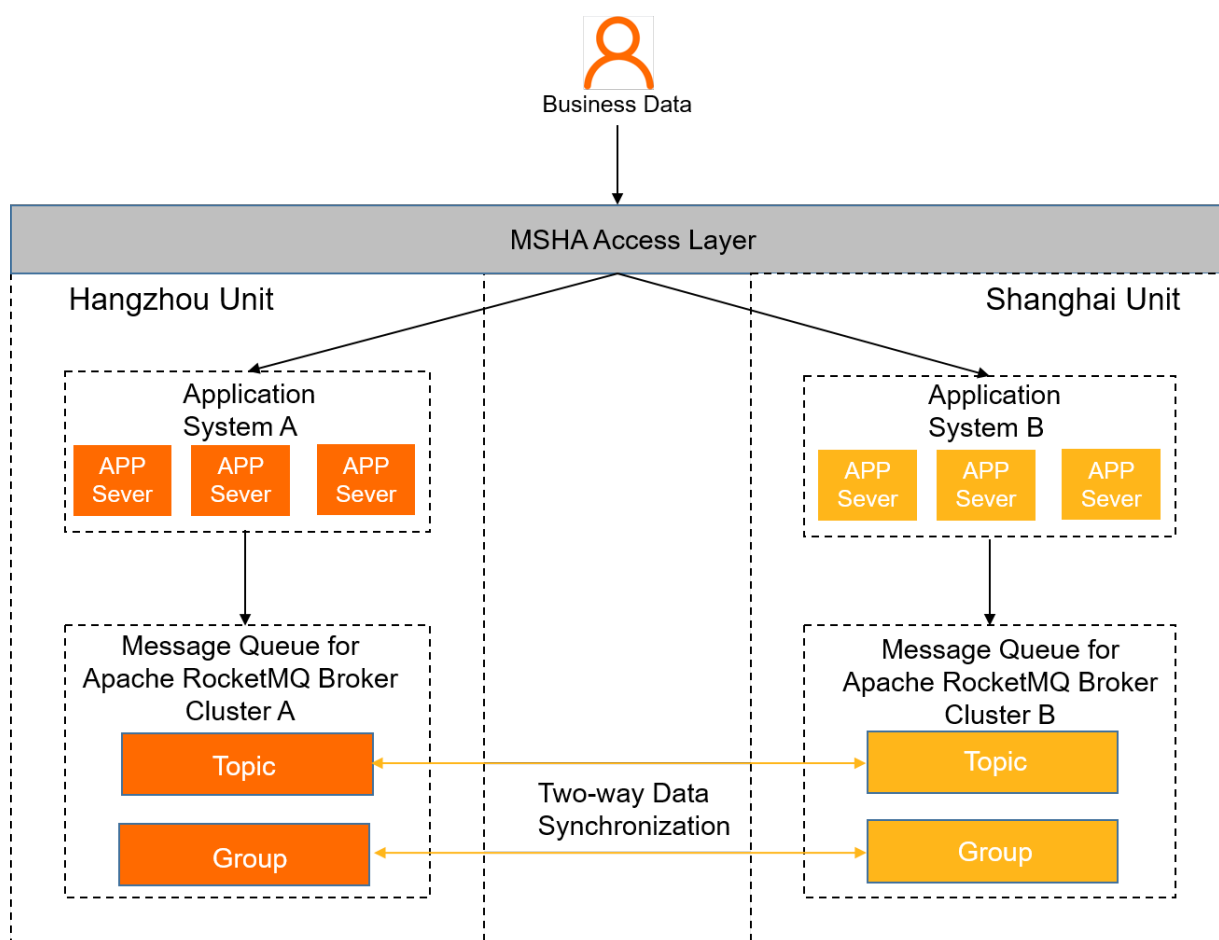
Message Queue for Apache RocketMQ leverages Alibaba Cloud Express Connect and Multi-Site High Availability (MSHA) to support active geo-redundancy. MSHA allows you to implement two-way data synchronization and business traffic switchover across instances. This way, business recovery is decoupled from fault recovery. If a fault occurs, business continuity can be ensured. This topic introduces the concept of MSHA and describes the common scenarios in which MSHA is suitable. This topic also describes the benefits of MSHA.

What is MSHA?

MSHA is an active geo-redundancy solution that was developed in the e-commerce business environment of Alibaba Group. MSHA can help decouple business recovery from fault recovery. MSHA provides capabilities such as flexible scheduling based on traffic rules, cross-region and cross-cloud management, and data protection. If a fault occurs, fast failover and recovery operations can be performed.

Message Queue for Apache RocketMQ leverages Alibaba Cloud Express Connect and MSHA to implement two-way synchronization of message data across instances in the same region or different regions. MSHA is different from a traditional disaster recovery solution. MSHA allows Message Queue for Apache RocketMQ clusters that are deployed in different units to provide services at the same time. This helps implement disaster recovery, improve business continuity, and achieve remote resource scaling.

The following figure shows how to use MSHA to implement active geo-redundancy for Message Queue for Apache RocketMQ.



- A complete business system is deployed in the Hangzhou and Shanghai units.
- The MSHA access layer routes business data to the two units based on traffic rules. The application systems and Message Queue for Apache RocketMQ broker clusters in the Hangzhou and Shanghai units process business data in their local regions.
- The broker clusters in the two units are configured to support active geo-redundancy. Data is synchronized between Broker Cluster A and Broker Cluster B in a two-way manner. The data includes topics, groups, and consumer offsets. In normal cases, the business systems in the Hangzhou and Shanghai units process business data only in their local regions, and synchronize message data of the local unit to the cluster of the remote unit for disaster recovery and backup.

- Assume that a disaster occurs in the Hangzhou unit and the entire business system in the Hangzhou unit fails. In this case, MSHA switches the business data of the Hangzhou unit to the Shanghai unit. As a result, Broker Cluster B of the Shanghai unit stores the business data of the Hangzhou unit and can continue to process the unfinished message data. This allows you to troubleshoot and fix the fault without service downtime. This way, the business can be recovered before the fault is rectified.
- After the Hangzhou unit is recovered from the fault, MSHA switches the business data of the Hangzhou unit back to the business system in the Hangzhou unit. During the entire process, users are not aware of the fault, and the user experience is not affected.

Common scenarios

MSHA can be used in the following common business scenarios:

- Business scenarios in which workloads are divided into different units by region, such as logistics workloads. You can divide the logistics workloads based on the regions in which logistics orders are placed and send business data to production centers in different regions. This way, the data can be simultaneously processed. This helps improve resource utilization and business concurrency.
- Business scenarios that have strict requirements for the reliability of business data, such as financial and securities systems. If a system fault occurs, the transaction results are negatively affected. In this case, you can use MSHA to switch the workloads to the disaster recovery site. The disaster recovery site can continue to process unfinished message data based on the synchronized data.

Benefits

- **High availability**

Compared with a traditional disaster recovery solution, MSHA implements two-way data synchronization across production centers. All production centers can provide services at the same time. This implements traffic balancing and improves resource utilization.

- **Fast fault recovery**

MSHA effectively ensures business continuity. MSHA decouples business recovery from fault recovery. When one of the production centers fails, MSHA immediately switches the business to other healthy production centers to ensure business continuity. This is different from a traditional solution in which the fault must be identified and fixed before the business can be recovered.

- **Remote resource scaling**

The limited resources in a single data center or region may not meet the requirements as the business rapidly develops. In addition, the business may face bottlenecks such as limited storage and computing performance. The horizontal scaling capability of Message Queue for Apache RocketMQ allows the business to be expanded in other data centers or regions to improve cost efficiency.

7.6. Message routing

The message routing feature provided by Message Queue for Apache RocketMQ can be used to synchronize message data across clusters to ensure message consistency and integrity between clusters. This topic introduces the concept of message routing and describes the common scenarios in which the message routing feature is suitable. This topic also describes the benefits of the feature and how to configure the feature.

What is message routing?

The message routing feature of Message Queue for Apache RocketMQ is used to synchronize messages across clusters. You can configure routing rules to dynamically plan the synchronization path of messages so that messages can be synchronized from the source node to the destination node based on filter conditions. This implements remote message synchronization and allows you to synchronize messages across clusters within milliseconds. This way, data consistency and integrity across clusters are ensured.

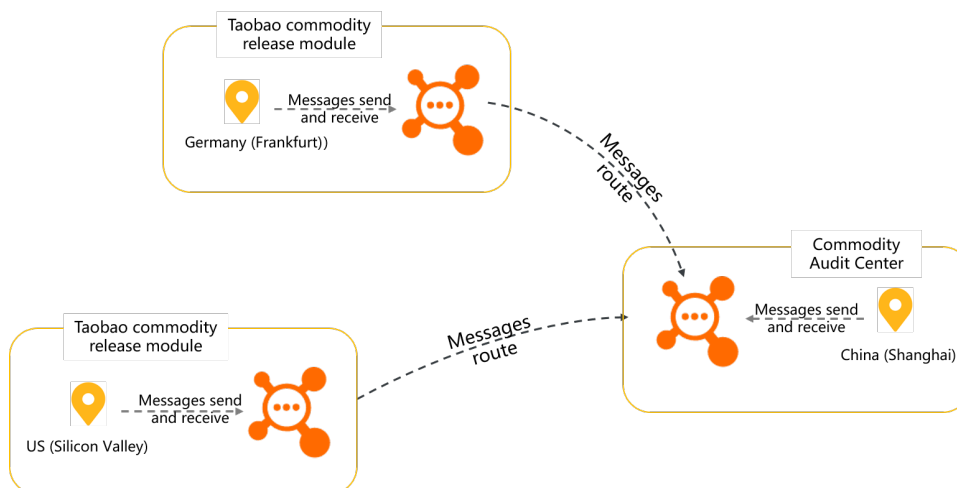
The following figure shows how the message routing feature works in Message Queue for Apache RocketMQ. In the figure, one-way synchronization is performed based on topics to synchronize messages from a specified source topic in the source instance to a specified destination topic in the destination instance.

Common scenarios

The message routing feature can be used in the following common scenarios:

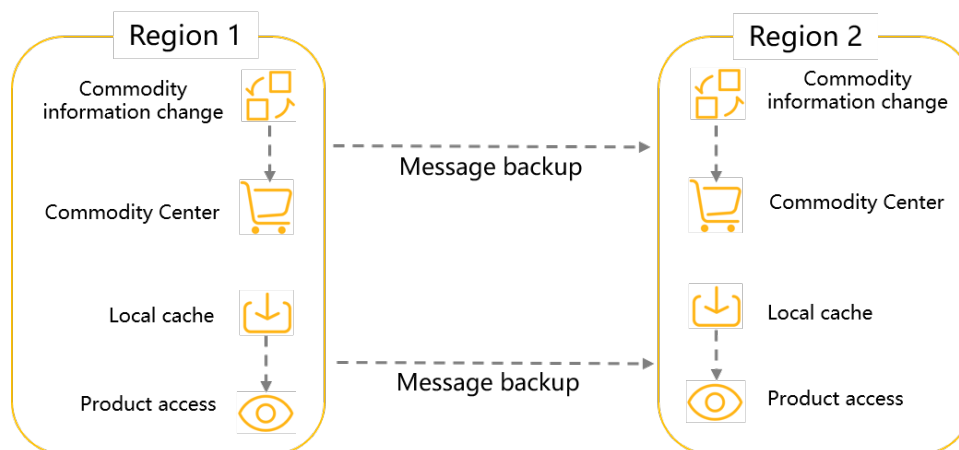
- **Data synchronization**

Taobao and Tmall serve users all over the world. If sellers in a country or a region want to publish products, the products must be reviewed before they can be available for sale. However, the product review systems of Alibaba Group are central systems deployed only in the cities of China, such as Shanghai. In this scenario, messages from the regions outside China must be synchronized to the product review systems in China to achieve cross-region data synchronization.



- **Disaster recovery and backup**

For core trading systems, changes in the details and prices of various commodities need to be updated across all business systems in real time. Multi-region disaster recovery and backup solutions can be used to ensure high availability. This way, service continuity is ensured even if a region becomes unavailable due to issues such as the cut of optical fibers.



Benefits

- **High performance**

The message routing feature provided by Message Queue for Apache RocketMQ enables real-time message synchronization within milliseconds and supports millions of transactions per second (TPS).

- **Low costs**

You do not need to purchase additional leased lines, perform upgrades, or modify application code to use the message routing feature. The message routing process is transparent to your application.

- **Ease of use**

The message routing feature supports GUI-based configuration. You can create and manage routing tasks in the Message Queue for Apache RocketMQ console.

Limits

The message routing feature is available only for instances that have namespaces. You can check whether an instance has a namespace on the **Instance Information** tab of the details page of the instance in the Message Queue for Apache RocketMQ console.

Configure message routing

This section describes the procedure that is used to configure the message routing feature for a Message Queue for Apache RocketMQ instance. For more information, see [Configure message routing](#).

- **Step 1: Create a destination cloud**

Before you create a routing task, you must specify information to create a cloud where your Message Queue for Apache RocketMQ cluster is deployed. The information includes the endpoint of your Message Queue for Apache RocketMQ instance and the AccessKey ID and AccessKey secret of the account to which the cloud belongs. Message Queue for Apache RocketMQ obtains the permissions that are required to access Message Queue for Apache RocketMQ resources across clouds based on the cloud information that you specified.

- **Step 2: Create a routing task**

Specify the message source and the message destination, and configure relevant information. For example, specify filter conditions and set the start offset of message synchronization.

8. Service usage FAQ

8.1. FAQ

8.1.1. Quick start

This topic provides answers to questions frequently asked by new users when they use Message Queue for Apache RocketMQ.

1. Where do consumers identified by a new group ID start to consume?
 - If a consumer identified by the group ID is started for the first time, the consumer ignores the messages that are sent before the consumer is started. This means that the consumer ignores historical messages and starts to consume messages that are sent after the consumer is started.
 - If the consumer is started for the second time, the consumer starts consumption from the previous consumer offset.
 - If you want the consumer to start consumption from a specific offset, you can reset the previous consumer offset in the Message Queue for Apache RocketMQ console to specify a point in time from which the consumer starts to consume messages. Each reset affects only the specific topic under the specific group ID but does not affect other group IDs.

2. How does the Message Queue for Apache RocketMQ broker redeliver a message if the message fails to be consumed?

- **Clustering consumption**

In clustering consumption mode, if `Action.ReconsumerLater` or `NULL` is returned or an error occurs during consumption, the Message Queue for Apache RocketMQ broker attempts to redeliver the message for up to 16 times. If the message still fails to be consumed after the 16 delivery retries, the message is discarded. The following table describes the intervals between delivery retries.

Nth delivery retry	Interval
1	10 seconds
2	30 seconds
3	1 minute
4	2 minutes
5	3 minutes
6	4 minutes
7	5 minutes
8	6 minutes
9	7 minutes
10	8 minutes

Nth delivery retry	Interval
11	9 minutes
12	10 minutes
13	20 minutes
14	30 minutes
15	1 hour
16	2 hours

The `message.getReconsumeTimes()` method can be called to query the serial number of a delivery retry.

- **Broadcasting consumption**

In broadcasting consumption mode, Message Queue for Apache RocketMQ guarantees that a message can be consumed at least once. If the message fails to be consumed, the Message Queue for Apache RocketMQ broker does not redeliver the message.

3. What do I do if a sent message is not received?

Message Queue for Apache RocketMQ provides the following methods for **Message query**:

- Specify a topic and time range to query all messages received by this topic within the specified time range.
- Specify a topic and message ID to query messages by using exact match.
- Specify a topic and message key to query messages with the same message key.

You can use the preceding methods to query the specific content and consumption information of messages. To track the time and location of each role from the producer to the consumer in the entire trace of a message, you can use the message tracing feature provided by Message Queue for Apache RocketMQ. For more information, see **Query the message trace**.

4. Can Message Queue for Apache RocketMQ ensure that no duplicate messages are delivered to consumers?

In most cases, Message Queue for Apache RocketMQ can ensure that no duplicate messages are delivered to consumers. As a distributed messaging middleware, Message Queue for Apache RocketMQ cannot ensure that no duplicate messages are delivered to consumers when exceptions such as network jitter and application processing timeout occur. However, Message Queue for Apache RocketMQ can ensure that no messages are lost.

8.1.2. Configurations

This topic provides answers to frequently asked questions about Message Queue for Apache RocketMQ configurations.

1. How long can messages be retained on the Message Queue for Apache RocketMQ broker?

Messages can be retained on the Message Queue for Apache RocketMQ broker for up to three days. The system automatically deletes the unconsumed after the three days.

2. What is the maximum message body size in Message Queue for Apache RocketMQ?

The maximum message body size in Message Queue for Apache RocketMQ varies with the message type. The following information shows the maximum message body size for different types of messages:

- A normal or ordered message: 4 MB
- A transactional, scheduled, or delayed message: 64 KB

3. How do I set the number of consumer threads on a Message Queue for Apache RocketMQ consumer?

To set the number of consumer threads on a Message Queue for Apache RocketMQ consumer, set the `ConsumeThreadNums` attribute when you start the consumer. The following sample code provides an example on how to set the number of consumer threads:

```
public static void main(String[] args) {
    Properties properties = new Properties();
    properties.put(PropertyKeyConst.GROUP_ID, "GID_001");
    properties.put(PropertyKeyConst.AccessKey, "xxxxxxxxxxxxx");
    properties.put(PropertyKeyConst.SecretKey, "xxxxxxxxxxxxx");
    /**
     * Set the number of consumer threads to 20.
     */
    properties.put(PropertyKeyConst.ConsumeThreadNums, 20);
    Consumer consumer = ONSFactory.createConsumer(properties);
    consumer.subscribe("TestTopic", "*", new MessageListener() {
        public Action consume(Message message, ConsumeContext context) {
            System.out.println("Receive: " + message);
            return Action.CommitMessage;
        }
    });
    consumer.start();
    System.out.println("Consumer Started");
}
```

4. What do I do if an error in loading DLL or another running error occurs due to invalid .NET client configuration?

For more information, see *SDK_GUIDE.pdf* in the compressed package of SDK for .NET to verify that the project configuration is the same as that described in the document.

8.1.3. Message tracing

This topic provides answers to frequently asked questions about the message tracing feature of Message Queue for Apache RocketMQ.

1. Why is trace data not found?

If no trace data is found based on the specified query conditions, check whether the following requirements are met:

- i. Only Java clients of version 1.2.2 or later support the message tracing feature.
- ii. Check whether the query conditions are properly specified. This means that you need to check whether the topic name, message ID, and message key are properly entered.

- iii. Check whether the query time range is correct. To accelerate the query, you must specify the range of the message sending time. If you still cannot retrieve the data, expand the time range and try again.
 - iv. If the preceding settings are correct but the trace data is still not found, contact the technical support and provide the related log file. The path to the log file is `/home/{user}/logs/ons.log`.
 - v. If the preceding settings are correct but the trace data is still not found, submit a ticket to seek help from the technical support and provide the log file. The path to the log file is `/home/{user}/logs/ons.log`.
2. What do I do if the consumption information about a consumed message is not included in the trace data and the client IP address and group ID in the trace data are wrong?

This problem occurs because the client is not updated to the version that supports the message tracing feature. Therefore, the message tracing module of Message Queue for Apache RocketMQ can obtain only some trace data, and the displayed result is abnormal. We recommend that you upgrade your client as soon as possible. For more information about the message tracing feature, see [Query the message trace](#).

3. Why is my group ID not shown in the list of consumers?

The possible cause is that a large number of downstream consumers have subscribed to messages, and the space in the tracing map is insufficient to display all the data. Move the pointer over the scroll bar and scroll down to see all the data.

4. Why are previous query tasks not displayed?

A large number of historical query tasks affect the display result. Therefore, Message Queue for Apache RocketMQ regularly cleans up historical query tasks and retains only query tasks created within the recent seven days. If you cannot find a historical task, query it again.

8.1.4. Alert handling

Alert handling is unavailable for Message Queue for Apache RocketMQ.

Apsara Stack provides an isolated cloud-based environment and cannot be connected to the APIs of Internet services, such as the short message service (SMS) gateway. Therefore, the monitoring and alerting module in the console is unavailable.

8.1.5. Ordered messages

This topic provides answers to frequently asked questions about ordered messages in Message Queue for Apache RocketMQ.

1. Do ordered messages support clustering consumption and broadcasting consumption?

Ordered messages support clustering consumption but do not support broadcasting consumption.

2. Can a message be an ordered message, a scheduled message, and a transactional message at the same time?

No, a message cannot be an ordered message, a scheduled message, and a transactional message at the same time. Ordered messages, scheduled messages, and transactional messages are different and mutually exclusive message types.

3. What is the usage scope of ordered messages?

Ordered messages are messages that are guaranteed to be consumed in the order they are sent within the same topic. Ordered messages are classified into globally ordered messages and partially ordered messages.

4. Why is the performance of globally ordered messages mediocre?

Globally ordered messages are processed in first-in-first-out (FIFO) order. If the previous message is not consumed, the next message will be stored in a queue of the corresponding topic until the previous message is consumed. To improve the transactions per second (TPS) of globally ordered messages, upgrade the specifications of the host that runs the message client, and reduce as much as possible the time required by the message client application to process the local business logic.

5. What transmission modes do ordered messages support?

Ordered messages support only the reliable synchronous transmission mode.

8.2. Exceptions

8.2.1. Usage-related exceptions

This topic describes the exceptions that may occur when you use Message Queue for Apache RocketMQ. This topic also provides solutions.

1. The producer or consumer failed to be started, or duplicate group IDs exist.

Cause:

You attempt to start multiple producer or consumer instances identified by the same group ID in one JVM process. This results in client startup failures.

Solution:

Perform the following steps:

- i. Make sure that only one producer instance identified by a group ID and one consumer instance identified by a group ID are started in one JVM process. This means that you cannot start multiple producer instances identified by the same group ID or multiple consumer instances identified by the same group ID in the same JVM process.
 - ii. Restart your application.
2. In broadcasting consumption mode, an error occurred when the JSON file is loaded for consumer startup.

Cause:

The Fastjson version is much earlier. In broadcasting consumption mode, the consumer failed to load the local *offsets.json* file and failed to be started.

Solution:

Update Fastjson to a version supported by ons-client and make sure that the *offsets.json* file can be normally loaded. By default, the *offsets.json* file is located in the */home/{user}/.rocketmq_offsets/* directory.

3. The queue list failed to be obtained when the consumer subscribes to messages.

Cause:

You did not create this topic in the console. As a result, the consumer failed to obtain the queue information of the topic during startup.

Solution:

Perform the following steps:

- i. [Log on to the Message Queue for Apache RocketMQ console](#). In the left-side navigation pane, click **Topics**. On the Topics page, click **Create Topic**.
 - ii. In the left-side navigation pane, click **Groups**. On the Groups page, click **Create Group ID** to create a group ID as prompted.
 - iii. Restart your application.
4. The message failed to be sent.

The message failed to be sent after multiple delivery retries.

Cause:

- i. The Message Queue for Apache RocketMQ broker returned an error code to the producer. For more information about the error code, see the nested exception that corresponds to this exception.
- ii. After the Message Queue for Apache RocketMQ broker unexpectedly fails and before the producer detects the latest broker list, this exception temporarily occurs.
- iii. The producer timed out when it attempted to send a message. This problem may be caused by heavy load on the broker or unstable network connectivity.

Solution:

Perform the following steps:

- i. Try again later. This exception is temporary. The temporary timeout might be caused by the restart of the Message Queue for Apache RocketMQ broker or heavy load on the broker.
 - ii. If the problem persists after you try for several times, contact technical support engineers.
5. No exception is recorded.

Problem description:

No exception is recorded.

Solution:

Contact technical support engineers.

6. The status of the message is Consumed, but the consumer is not aware of this.

The status of the message is Consumed, but the consumer log shows that the message is not received. This problem is due to the following three reasons:

- o The business code defines that the message is not immediately printed after the message is received.

If the business logic is directly executed after a message is received, the message information is not recorded in the log if the code misses a specific logic branch. This leads to the false symptom that the message is not received.

We recommend that you immediately print the message information after you receive a message to keep the information such as `messageId`, `timestamp`, and `reconsumeTime`.

- o Multiple consumer instances are deployed.

A consumer is often restarted multiple times at the debugging stage. If the previous process does not exit before the next process starts, multiple consumption processes coexist. In this case, multiple consumer instances share the message information. This scenario is similar to clustering consumption. A message that fails to be received by one consumer is received by another consumer.

Log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click **Groups**. On the Groups page, select your instance and click **Consumer Status** in the Actions column. In the Consumer Status panel, view **Connection Information**. The deployment information of consumer instances is displayed, including the number of consumer instances and the IP address of each instance. You can check for the problem based on the information.

- An exception that failed to be caught occurred during the consumption of a message. As a result, the message is redelivered.

```
public class MessageListenerImpl implements MessageListener {
    @Override
    public Action consume(Message message, ConsumeContext context) {
        // The message processing logic throws an exception. The message will be redelivered.

        doConsumeMessage(message);
        // If an exception that is not caught occurs in the doConsumeMessage() method,
        this line of log is not printed.
        log.info("Receive Message, messageId:", message.getMsgID());
        return Action.CommitMessage;
    }
}
```

If the problem persists, contact technical support engineers and provide the local SDK logs.

8.2.2. Nonexistent resources

This topic describes exceptions related to nonexistent resources and provides solutions.

1. Nonexistent group ID

Cause:

The group ID is not created in the Message Queue for Apache RocketMQ console. As a result, when the group ID is used to connect to the Message Queue for Apache RocketMQ broker, verification fails on the broker.

Solution:

Perform the following steps:

- Log on to the Message Queue for Apache RocketMQ console.** In the left-side navigation pane, click **Groups**.
 - If the group ID already exists, proceed to the next step.
 - If the group ID does not exist, create the group ID. Then, perform the next step.
- Restart your application.

2. Nonexistent hostname

Cause:

A possible cause is that the correct hostname or host IP address cannot be retrieved. To verify this assumption, run the **hostname** command.

If the correct hostname cannot be retrieved, this assumption is true. Otherwise, this problem may be due to another reason. In this case, contact technical support engineers.

Solution:

Perform the following steps:

- i. On the host for which the error is reported, run the following command to check the hostname:

```
[root@iz231wxgt6mZ ~]# hostname
iz231wxgt6mZ
```

If an error is returned, check whether an alias is defined for the hostname. For example, an alias can be `alias xxx='hostname'` in `.bash_profile` or `.bashrc`. Another possible cause is that the command path does not point to `$PATH`.

- ii. Ping the host.

```
[root@iz231wxgt6mZ ~]# ping iz231wxgt6mZ
```

If the hostname cannot be pinged, add the local IP address to the `/etc/hosts` file. By default, each Elastic Compute Service (ECS) instance establishes a binding relationship between the local IP address and the hostname. Do not manually remove the relationship.

- iii. Check the system configurations.

Check whether the hostname recorded in `/etc/sysconfig/network` is the same as that added to `/etc/hosts`. If the hostname is not the same as that added to `/etc/hosts`, modify the hostname. If you modify the content in `/etc/sysconfig/network`, you must restart the host after you modify the content. This way, the modification can take effect. Exercise caution when you modify configurations in a system file, because this operation may cause other exceptions.

After the preceding three steps are performed, `UnknownHostException` will no longer be returned when your client starts.

8.2.3. Inconsistent status

This topic describes the exceptions related to inconsistent status and provides solutions.

1. Invalid messages

Cause:

The message attribute or content is invalid in the following scenarios:

- o The message is empty.
- o The message content is empty.
- o The message content is 0 character in length.
- o The length of the message content exceeds the limit.

Solution:

Check whether the preceding exceptions occur to the message and handle the exceptions as prompted.

2. Invalid parameters

Cause:

The following table lists the cases in which the parameters are invalid.

Nested exception	Description
consumeThreadMin Out of range [1, 1000]	The specified number of consumer threads is inappropriate.
consumeThreadMax Out of range [1, 1000]	The specified number of consumer threads is inappropriate.
messageListener is null	messageListener is not configured.
consumerGroup is null	The group ID is not specified.
msg delay time more than 40 day	The delay for the delivery of a scheduled message cannot exceed 40 days.

Solution:

Perform the following steps:

- Modify the parameter settings for the client as prompted and make sure that the new parameter values are within the valid ranges.
- Restart your application.

3. Abnormal client status

Cause:

- After the consumer or producer is created, the return code does not show that the start() method is called to start the consumer or producer.
- After the consumer or producer is created, the consumer or producer fails to start due to an exception in the start() process.
- After the consumer or producer is created and the start() method is called, the return code shows that the shutdown() method is called to shut down the consumer or producer.

Solution:

Perform the following steps:

- Make sure that the start() method is called after the group ID is created. Make sure that the producer or consumer is started.
- Check *ons.log* for exceptions that occur during the startup of the producer or consumer.

4. Subscription inconsistency

Problem description:

Multiple consumer instances are started in different JVM processes. Consumer instances identified by the same group ID subscribe to different topics, or subscribe to the same topic but different tags. As a result, the subscriptions of the consumer instances are inconsistent, and messages cannot be received as expected.

Sample code of inconsistent subscriptions:

- Example 1: The consumer instance on JVM 1 and the consumer instance on JVM 2 use the same group ID GID-MQ-FAQ. The two consumer instances subscribe to different topics. The consumer instance on JVM 1 subscribes to MQ-FAQ-TOPIC-1, whereas the consumer instance on JVM 2 subscribes to MQ-FAQ-TOPIC-2.

Code on JVM-1:

```
Properties properties = new Properties();
properties.put(PropertyKeyConst.GROUP_ID, "GID-MQ-FAQ");
Consumer consumer = ONSFactory.createConsumer(properties);
consumer.subscribe("MQ-FAQ-TOPIC-1", "NM-MQ-FAQ", new MessageListener() {
    public Action consume(Message message, ConsumeContext context) {
        System.out.println("Receive: " + message);
        return Action.CommitMessage;
    }
});
consumer.start();
```

Code on JVM-2:

```
Properties properties = new Properties();
properties.put(PropertyKeyConst.GROUP_ID, "GID-MQ-FAQ");
Consumer consumer = ONSFactory.createConsumer(properties);
consumer.subscribe("MQ-FAQ-TOPIC-2", "NM-MQ-FAQ", new MessageListener() {
    public Action consume(Message message, ConsumeContext context) {
        System.out.println("Receive: " + message);
        return Action.CommitMessage;
    }
});
consumer.start();
```

- Example 2: The consumer instance on JVM 1 and the consumer instance on JVM 2 use the same group ID GID-MQ-FAQ and subscribe to the same topic. However, the two consumer instances subscribe to different tags. The consumer instance on JVM 1 subscribes to NM-MQ-FAQ-1, whereas the consumer instance on JVM 2 subscribes to NM-MQ-FAQ-2.

Code on JVM-1:

```
Properties properties = new Properties();
properties.put(PropertyKeyConst.GROUP_ID, "GID-MQ-FAQ");
Consumer consumer = ONSFactory.createConsumer(properties);
consumer.subscribe("MQ-FAQ-TOPIC-1", "NM-MQ-FAQ-1", new MessageListener() {
    public Action consume(Message message, ConsumeContext context) {
        System.out.println("Receive: " + message);
        return Action.CommitMessage;
    }
});
consumer.start();
```

Code on JVM-2:

```
Properties properties = new Properties();
properties.put(PropertyKeyConst.GROUP_ID, "GID-MQ-FAQ");
Consumer consumer = ONSFactory.createConsumer(properties);
consumer.subscribe("MQ-FAQ-TOPIC-1", "NM-MQ-FAQ-2", new MessageListener() {
    public Action consume(Message message, ConsumeContext context) {
        System.out.println("Receive: " + message);
        return Action.CommitMessage;
    }
});
consumer.start();
```

Solution:

If you start multiple consumer instances identified by the same group ID in different JVM processes, make sure that the topics and tags to which the consumer instances subscribe are the same.

8.3. Troubleshooting

8.3.1. Unexpected consumer connections

This topic describes the symptoms of an unexpected consumer connection, analyzes causes, provides a solution, and verifies the solution.

Problem description

- [Symptom 1]: Some messages are sent but not received. After you query message traces in the Message Queue for Apache RocketMQ console, the returned information shows that some messages are sent to the Message Queue for Apache RocketMQ broker, but the broker does not deliver the messages to consumers. To query message traces, log on to Message Queue for Apache RocketMQ console. In the left-navigation pane, click **Message Tracing**. On the Message Tracing page, click **Create Query Task**. In the Create Query Task dialog box, click the **By Message ID** tab.
- [Symptom 2]: Some consumer IP addresses are not within the expected range and messages are accumulated on the consumers that correspond to these IP addresses. To query connection information about consumers, log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click **Groups**. On the Groups page, find the group ID whose connection information you want to view and click **Consumer Status** in the Actions column. In the Consumer Status panel, view the connection information in the **Connection Information** section.

Problem analysis

Analysis: Log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click **Groups**. On the Groups page, find the group ID whose connection information you want to view and click **Consumer Status** in the Actions column. In the Consumer Status panel, view the connection information in the **Connection Information** section. The connection information about all consumers identified by the group ID are displayed. You can check the IP address and process ID of the unexpected consumer and check whether the configurations loaded by the process are valid. The configurations include the AccessKey ID, AccessKey secret, topic, and group ID. If the configurations are invalid, the consumer process occupies some queues but cannot properly consume messages.

Cause: In the same environment, if a consumer identified by the group ID and configured with an invalid AccessKey ID, AccessKey secret, and topic is started, this consumer process may occupy some queues of the topic but cannot properly consume messages. As a result, messages are accumulated on the Message Queue for Apache RocketMQ broker and cannot be properly delivered to downstream consumers whose IP addresses are within the expected range.

- **Confirmation:** Locate the faulty process based on the connection status and check the AccessKey ID, AccessKey secret, and topic of the process based on the `/{user.home}/logs/ons.log` file or program code.
- **Solution:** This is a quick solution. Shut down the faulty consumer process first. Then, the accumulated messages will be immediately rebalanced and delivered to proper consumers. After the fault is rectified, restart the faulty process.

Verification

Log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click **Groups**. On the Groups page, find the group ID that you want to view and click **Consumer Status** in the Actions column. In the Consumer Status panel, view the connection information of consumers identified by the group ID in the **Connection Information** section. The displayed information shows that IP addresses of all consumers are within the expected range and the value of **Consistent Subscription** is **Yes**.

8.3.2. Inconsistent subscriptions

This topic describes the symptoms of inconsistent subscriptions, analyzes causes, provides a solution, and verifies the solution.

Problem description

- Consumers identified by a group ID failed to receive some messages to which they want to subscribe. To query messages, log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click **Message Query**. On the Message Query page, click the **By Message ID** tab. Specify the corresponding topic and message ID. The displayed information shows that the message has been consumed at least once. However, the message is considered unconsumed based on the consumption logic.
- The subscriptions of consumers identified by the group ID are inconsistent. To check whether the subscriptions of consumers are consistent, log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click **Groups**. On the Groups page, find the group ID and click **Consumer Status** in the Actions column. In the Consumer Status panel, the value of **Consistent Subscription** is **No**.

Problem analysis

In Message Queue for Apache RocketMQ, a group ID represents a consumer instance group. For most distributed applications, multiple consumer instances are attached to the same group ID. Subscription consistency means that the topics and tags of all consumer instances identified by the same group ID must be identical.

If the consumer instances identified by the same group ID subscribe to different topics, or subscribe to the same topic but different tags, the subscriptions are inconsistent. If the subscriptions are inconsistent, errors occur in the message consumption logic and messages may be lost.

- **[Cause 1]:** The topics subscribed to by consumers with the same group ID are different.

Example 1: Two consumers identified by the group ID GID-MQ-FAQ subscribe to different topics: MQ-FAQ-TOPIC-1 and MQ-FAQ-TOPIC-2.

Code on JVM-1:

```
Properties properties = new Properties();
properties.put(PropertyKeyConst.GROUP_ID, "GID-MQ-FAQ");
Consumer consumer = ONSFactory.createConsumer(properties);
consumer.subscribe("MQ-FAQ-TOPIC-1", "NM-MQ-FAQ", new MessageListener() {
    public Action consume(Message message, ConsumeContext context) {
        System.out.println("Receive: " + message);
        return Action.CommitMessage;
    }
});
consumer.start();
```

Code on JVM-2:

```
Properties properties = new Properties();
properties.put(PropertyKeyConst.GROUP_ID, "GID-MQ-FAQ");
Consumer consumer = ONSFactory.createConsumer(properties);
consumer.subscribe("MQ-FAQ-TOPIC-2", "NM-MQ-FAQ", new MessageListener() {
    public Action consume(Message message, ConsumeContext context) {
        System.out.println("Receive: " + message);
        return Action.CommitMessage;
    }
});
consumer.start();
```

- **[Cause 2]:** Two consumers identified by the same group ID subscribe to the same topic but different tags.

Example: Two consumers identified by the group ID GID-MQ-FAQ subscribe to the same topic but different tags: NM-MQ-FAQ-1 and NM-MQ-FAQ-2.

Code on JVM-1:

```
Properties properties = new Properties();
properties.put(PropertyKeyConst.GROUP_ID, "GID-MQ-FAQ");
Consumer consumer = ONSFactory.createConsumer(properties);
consumer.subscribe("MQ-FAQ-TOPIC-1", "NM-MQ-FAQ-1", new MessageListener() {
    public Action consume(Message message, ConsumeContext context) {
        System.out.println("Receive: " + message);
        return Action.CommitMessage;
    }
});
consumer.start();
```

Code on JVM-2:

```
Properties properties = new Properties();
properties.put(PropertyKeyConst.GROUP_ID, "GID-MQ-FAQ");
Consumer consumer = ONSFactory.createConsumer(properties);
consumer.subscribe("MQ-FAQ-TOPIC-1", "NM-MQ-FAQ-2", new MessageListener() {
    public Action consume(Message message, ConsumeContext context) {
        System.out.println("Receive: " + message);
        return Action.CommitMessage;
    }
});
consumer.start();
```

Solution

Perform the following steps:

1. Check the subscription code of different consumers. Make sure that the subscriptions of all consumers identified by the same group ID are consistent. This means that the topics and tags subscribed to by the consumers are all identical.
2. Restart all consumer applications.

Verification

- Consumers can receive messages as expected.
- Log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click **Groups**. On the Groups page, find the group ID that you want to view and click **Consumer Status** in the Actions column. In the Consumer Status panel, the value of **Consistent Subscription** is **Yes**.

8.3.3. Message accumulation

This topic describes the symptoms of message accumulation, analyzes causes, provides a solution, and verifies the solution.

Problem description

- The value of **Accumulated Messages** is higher than expected. To query the number of accumulated messages, log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click **Groups**. On the Groups page, find the group ID that you want to view and click **Consumer Status** in the Actions column. In the Consumer Status panel, check the value of **Accumulated Messages** in the Connection Information section.
- Some messages have been sent to the Message Queue for Apache RocketMQ broker but are not delivered to consumers. To query message traces, log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click **Message Tracing**. On the Message Tracing page, click **Create Query Task**. In the Create Query Task dialog box, click the **By Message ID** tab. Specify the corresponding topic and message ID to query the trace of a message.

Problem analysis

In Message Queue for Apache RocketMQ, messages are first sent to the broker. Then, consumers identified by the group ID pull some messages from the broker to the on-premises machine for consumption based on the current consumer offset. In the consumption process, it may take a long time to consume a single message due to various reasons, such as access to locked shared resources, competition for I/O and network resources, and no timeout set for HTTP calls. As a result, messages start to accumulate on the broker.

If messages are not accumulated, check whether the threshold value is excessively small and causes alerts on message accumulation.

Solution

Perform the following operations for troubleshooting:

- Log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click **Resource Statistics**. On the Resource Statistics page, click the **Message Consumption** tab. Enter the information to query historical consumption records. If message writing is faster than message consumption, modify the code or scale out the consumer.
- Print the Jstack information `jstack -l {PID} | grep ConsumeMessageThread` in the application. If messages are blocked, print the Jstack information for five consecutive times and identify the spot where the consumer thread is stuck. Then, rectify the fault and restart the application. Check whether messages can be consumed.

Verification

- Print the Jstack information `jstack -l {PID} | grep ConsumeMessageThread` in the application. Verify that no consumer thread is blocked.
- Log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click **Groups**. On the Groups page, find the group ID that you want to view and click **Consumer Status** in the Actions column. In the Consumer Status panel, check whether the value of **Real-time Consumption Speed** increases and the value of **Accumulated Messages** decreases.

8.3.4. Message accumulation in Java processes

Problem description

In the **Consumer Status** panel of the Message Queue for Apache RocketMQ console, the number of real-time accumulated messages of the group ID is higher than expected, and the performance is much lower.

Cause

The number of real-time accumulated messages of the group ID is higher than expected due to an excessive number of messages accumulated in the Java process.

Solution

Procedure

- [Log on to the Message Queue for Apache RocketMQ console](#). Navigate to the Consumer Status panel, obtain the host IP address of the consumer instance that has accumulated messages, and then log on to the host or container.
- Run one of the following commands to view the PID of the Java process and record the PID:

```
ps -ef |grep java
```

```
jps -lm
```

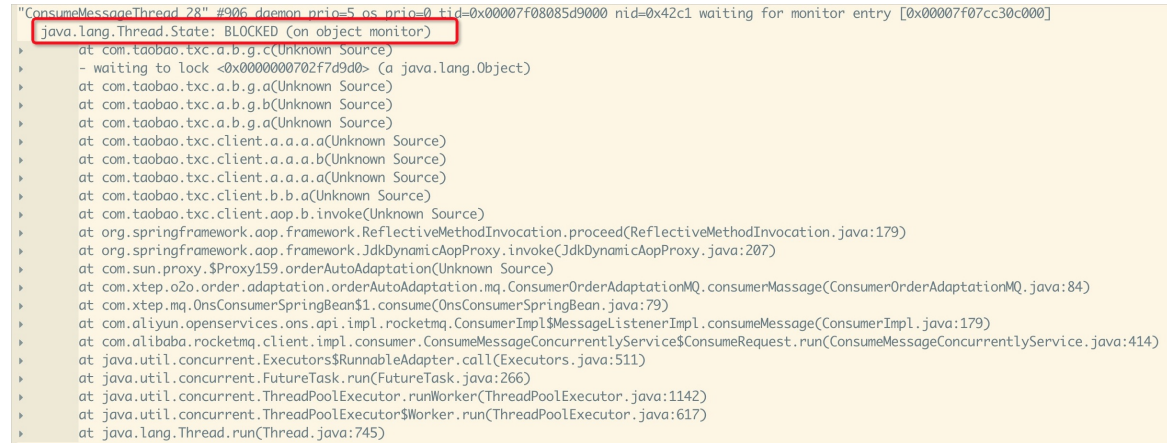
- Run the following command to view the stack information:

```
jstack -l pid > /tmp/pid.jstack
```

4. Run the following command to view the information about `ConsumeMessageThread` and focus on the thread status and stack:

```
cat /tmp/pid.jstack | grep ConsumeMessageThread -A 10 --color
```

The following figure shows an example of command output.



```
"ConsumeMessageThread_28" #906 daemon prio=5 os_prio=0 tid=0x00007f08085d9000 nid=0x42c1 waiting for monitor entry [0x00007f07cc30c000]
 java.lang.Thread.State: BLOCKED (on object monitor)
   at com.taobao.txc.a.b.g.c(Unknown Source)
   - waiting to lock <0x00000000702f7d9d0> (a java.lang.Object)
   at com.taobao.txc.a.b.g.a(Unknown Source)
   at com.taobao.txc.a.b.g.b(Unknown Source)
   at com.taobao.txc.a.b.g.a(Unknown Source)
   at com.taobao.txc.client.a.a.a.a(Unknown Source)
   at com.taobao.txc.client.a.a.a.b(Unknown Source)
   at com.taobao.txc.client.a.a.a.a(Unknown Source)
   at com.taobao.txc.client.a.a.a.a(Unknown Source)
   at com.taobao.txc.client.b.b.a(Unknown Source)
   at com.taobao.txc.client.aop.b.invoke(Unknown Source)
   at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:179)
   at org.springframework.aop.framework.JdkDynamicAopProxy.invoke(JdkDynamicAopProxy.java:207)
   at com.sun.proxy.$Proxy159.orderAutoAdaptation(Unknown Source)
   at com.xtep.o2o.order.adaptation.orderAutoAdaptation.mq.ConsumerOrderAdaptationMQ.consumerMessage(ConsumerOrderAdaptationMQ.java:84)
   at com.xtep.mq.OnsConsumerSpringBean$1.consume(OnsConsumerSpringBean.java:79)
   at com.aliyun.openservices.ons.api.impl.rocketmq.ConsumerImpl$MessageListenerImpl.consumeMessage(ConsumerImpl.java:179)
   at org.alibaba.rocketmq.client.impl.consumer.ConsumeMessageConcurrentlyService$ConsumeRequest.run(ConsumeMessageConcurrentlyService.java:414)
   at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:511)
   at java.util.concurrent.FutureTask.run(FutureTask.java:266)
   at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
   at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
   at java.lang.Thread.run(Thread.java:745)
```

For more information about the thread status, see [official Java documentation](#).

Note Message Queue for Apache RocketMQ can support 1 billion accumulated messages without compromising the performance. If the problem of compromised performance is not solved after you perform the preceding steps, contact O&M engineers and provide the following information:

- The `heap.bin` file. Run the `jmap -dump:format=b,file=heap.bin [PID]` command to obtain this file. Then, run the `gzip heap.bin` command to generate a compressed package. `[PID]` represents the PID of the Java process recorded in Step 2.
- The local `ons.log` file of the consumer client where messages are accumulated.
- The version of the consumer client.

8.3.5. Application OOM due to message caching on the client

This topic describes the symptoms of application out of memory (OOM), analyzes causes, provides solutions, and verifies each solution.

Problem description

- [Symptom 1]: The memory is exhausted on the machine where the application is deployed.
- [Symptom 2]: The keyword `OutOfMemory` can be found in `{user.home}/logs/ons.log`.
- [Symptom 3]: In the Message Queue for Apache RocketMQ console, **Real-time Accumulated Messages** in the **Consumer Status** panel of the **Groups** page shows that a large number of messages are accumulated. The **Connection Information** section displays the number of accumulated messages for each connected consumer client. In addition, the result of check by running the `jstack` command indicates that `ConsumeMessageThread_` is not blocked.

Analysis

Analysis: A Message Queue for Apache RocketMQ consumer proactively pulls messages from the Message Queue for Apache RocketMQ broker and caches them to the client. Then, the messages are consumed based on the consumption logic of the client. In versions earlier than 1.7.0.Final, the client caches up to 1,000 messages for each queue of each topic by default. Assume that each topic has 16 queues (two primary brokers and two secondary brokers, and eight queues on each broker). The average size of a message in this topic is 64 KB. The final message size cached for this topic on the client is calculated by using the following formula: $16 \times 1000 \times 64 \text{ KB} = 1 \text{ GB}$. If you subscribe to eight topics at the same time and caches messages of all these topics in the client memory, the memory consumed will exceed the memory size specified in the JVM configuration. In this case, OOM occurs.

- [Cause 1]: An ons-client version earlier than 1.7.0.Final is depended on, and the average size of a message in each topic exceeds 4 KB. In addition, message consumption is slow. This is prone to message caching in the memory of the client.
 - **Confirmation:** Check whether the keyword OutOfMemory can be found in `{user.home}/logs/ons.log`, or run the `jmap -dump:live,format=b,file=heap.bin <pid>` command to detect the objects that occupy a large amount of memory.
 - **Solution:** Update the ons-client version to 1.7.0.Final or later and set the `com.aliyun.openservices.ons.api.PropertyKeyConst#MaxCachedMessageSizeInMiB` parameter to an appropriate value for the corresponding ConsumerBean. Then, restart the application.
- [Cause 2]: ons-client-1.7.0.Final or later is depended on, and the default maximum memory consumed is 512 MB, which is the total cache capacity of all topics to which consumer instances identified by a group ID subscribe. If the application still suffers OOM, set the `com.aliyun.openservices.ons.api.PropertyKeyConst#MaxCachedMessageSizeInMiB` parameter to a value within the valid range from 16 MB to 2048 MB to customize the maximum memory that can be consumed during the startup of ConsumerBean.
 - **Confirmation:** Check the ons-client version used by the application and check the memory size allocated to the process based on the JVM configuration.
 - **Solution:** Set the `com.aliyun.openservices.ons.api.PropertyKeyConst#MaxCachedMessageSizeInMiB` parameter for the corresponding ConsumerBean, based on the memory usage of the machine where the application runs. Then, restart the application.

Verification

- [Verification 1]: The keyword OutOfMemory disappears from `{user.home}/logs/ons.log`.
- [Verification 2]: Log on to the Message Queue for Apache RocketMQ console. In the left-side navigation pane, click **Groups**. On the Groups page, select your instance and click **Consumer Status** in the Actions column. In the Consumer Status panel, the value of **Real-time Consumption Speed** increases, whereas the value of **Real-time Accumulated Messages** decreases.

8.3.6. AuthenticationException reported due to failure in sending or receiving messages

Problem description

The application cannot send messages and `AuthenticationException` is reported in the `{user.home}/logs/ons.log` log of the host.

Cause

A wrong AccessKey ID or AccessKey secret is used.

Solution

Procedure

1. Check whether you use an Apsara Stack tenant account or a Resource Access Management (RAM) user.

The following table describes the Apsara Stack tenant account and RAM user.

Apsara Uni-manager	RocketMQ
Organization administrator	Apsara Stack tenant account
Resource user	RAM user

You can create roles in the Apsara Uni-manager Management Console. If you want a role to become a resource user, the selected permissions must be consistent with the default configuration in the system.

2. Check the permissions of the user who creates resources.

The Apsara Stack tenant account can create a topic and a group ID in the Message Queue for Apache RocketMQ console. The created resources are of the current organization level. A RAM user cannot create a topic, but can create a group ID. The created resource is of the RAM user level.

- If you need to use a RAM user to send and receive messages, use the Apsara Stack tenant account to create a topic in the Message Queue for Apache RocketMQ console. For example, you can create a topic named Topic_bumen. Then, grant the permissions on the topic to a RAM user. At this point, the RAM user can view Topic_bumen in the Message Queue for Apache RocketMQ console. The RAM user can create its own group ID, for example, GiD_zizhanghao. Then, the messaging program of the client can send and receive messages by using Topic_bumen, GiD_zizhanghao, and the AccessKey ID and AccessKey secret of the RAM user.
- If you need to use the topics and group IDs created by the Apsara Stack tenant account to send and receive messages, the AccessKey ID and AccessKey secret of the organization level must be configured because the topics and group IDs created by the Apsara Stack tenant account are of the organizational level and do not belong to the account itself.