分布式消息服务 RocketMQ 版

开发指南

文档版本	01
发布日期	2025-01-17

华为云计算技术有限公司

商标声明


和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为云计算技术有限公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为云计算技术有限公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

目 录

1 概述

第二章介绍如何获取RocketMQ实例连接信息。

第三章~**第七章**介绍Java、Go和Python客户端访问分布式消息服务RocketMQ版的示例代码，具体如**表1-1**所示。

表 1-1 示例代码

客户端语言	示例代码
Java（TCP协议）	• **收发普通消息** • **收发顺序消息** • **收发事务消息** • **发送定时消息** • **使用ACL权限访问** • **消费端限流**
Java（gRPC协议）	• **收发普通消息** • **收发顺序消息** • **收发事务消息** • **发送定时消息**
Go（TCP协议）	• **收发普通消息** • **收发顺序消息** • **收发事务消息** • **发送定时消息** • **使用ACL权限访问**
Go（gRPC协议）	• **收发普通消息** • **收发顺序消息** • **收发事务消息** • **发送定时消息**

客户端语言	示例代码
Python（TCP协议）	• 收发普通消息 • 收发顺序消息 • 收发事务消息 • 发送定时消息 • 使用ACL权限访问

2 收集连接信息

约束与限制

仅RocketMQ实例5.x版本支持gRPC协议，4.8.0版本不支持。

获取实例连接信息

- 实例连接地址和端口

  实例创建后，从RocketMQ实例控制台的“基本信息”页面获取，在客户端配置时，可将地址都配上。

  - 使用TCP协议通过内网连接RocketMQ实例时，获取“连接地址”。
  - 使用gRPC协议通过内网连接RocketMQ实例时，获取“grpc连接地址”。
  - 使用TCP协议通过公网连接RocketMQ实例时，获取“公网连接地址”。
  - 使用gRPC协议通过公网连接RocketMQ实例时，获取“grpc公网连接地址”。

  图 2-1 查看实例的连接地址和端口（5.x 版本）

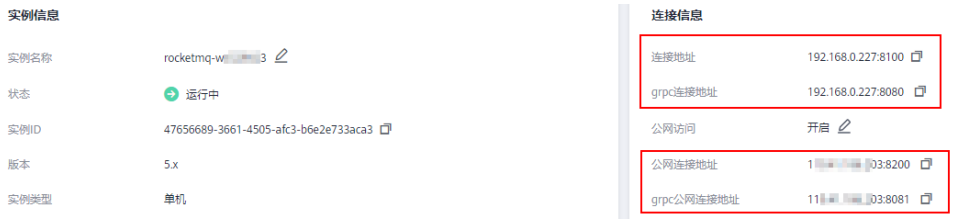
  

  图 2-2 查看实例的连接地址和端口（4.8.0 版本）

- Topic名称

  从RocketMQ实例控制台的“Topic管理”页签中获取Topic名称。

- 消费组名称

  从RocketMQ实例控制台的“消费组管理”页签中获取消费组名称。

- 用户名和用户密钥
  从RocketMQ实例控制台的“用户管理”页面获取用户名，在用户详情页获取用户密钥。

3 Java（TCP 协议）

3.1 收发普通消息

本章节介绍普通消息的收发方法和示例代码。其中，普通消息发送方式分为同步发送和异步发送。

- 同步发送：消息发送方发出一条消息到服务端，服务端接收并处理消息，然后返回响应给发送方，发送方收到响应后才会发送下一条消息。
- 异步发送：消息发送方发出一条消息后，不等服务端返回响应，接着发送下一条消息。

收发普通消息前，请参考**收集连接信息**收集RocketMQ所需的连接信息。

约束与限制

客户端连接RocketMQ实例5.x版本收发普通消息前，需要确保Topic的消息类型为“普通”。

准备环境

开源的Java客户端支持连接分布式消息服务RocketMQ版，推荐使用的客户端版本为**4.9.8**。

通过以下任意一种方式引入依赖：

- 使用Maven方式引入依赖。

```
<dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-client</artifactId>
    <version>4.9.8</version>
</dependency>
```

- 下载**依赖JAR包**。

同步发送

同步发送是指消息发送方发出一条消息到服务端，服务端接收并处理消息，然后返回响应给发送方，发送方收到响应后才会发送下一条消息的通讯方式。

参考如下示例代码，或者通过**Producer.java**获取更多示例代码。

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

public class Main {
    public static void main(String[] args) {
        DefaultMQProducer producer = new DefaultMQProducer("ProducerGroupName");
        //填入连接地址
        producer.setNamesrvAddr("192.168.0.1:8100");
        //producer.setUseTLS(true); //创建实例时，如果开启了SSL，请添加此行代码。
        try {
            producer.start();
            Message msg = new Message("TopicTest",
                "TagA",
                "OrderID188",
                "Hello world".getBytes(RemotingHelper.DEFAULT_CHARSET));
            SendResult sendResult = producer.send(msg);
            System.out.printf("%s%n", sendResult);

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

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

异步发送

异步发送是指消息发送方发出一条消息后，不等服务端返回响应，接着发送下一条消息的通讯方式。

使用异步发送需要客户端实现异步发送回调接口（SendCallback）。即消息发送方在发送了一条消息后，不需要等待服务端响应接着发送第二条消息。发送方通过回调接口接收服务端响应，并处理响应结果。

参考如下示例代码，或者通过**AsyncProducer.java**获取更多示例代码。

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


public class Main {
    public static void main(String[] args) throws InterruptedException {
        DefaultMQProducer producer = new DefaultMQProducer("ProducerGroupName");
        //填入连接地址
        producer.setNamesrvAddr("192.168.120.45:8100;192.168.123.150:8100");
        //producer.setUseTLS(true);    //创建实例时，如果开启了SSL，请增加此行代码。
        try {
            producer.start();
            Message msg = new Message("TopicTest",
                    "TagA",
                    "OrderID188",
                    "Hello world".getBytes(RemotingHelper.DEFAULT_CHARSET));
            producer.send(msg, new SendCallback() {
                @Override
                public void onSuccess(SendResult result) {
                    // 消息发送成功。
                    System.out.println("send message success. msgId= " + result.getMsgId());
                }


                @Override
                public void onException(Throwable throwable) {
```

```
                // 消息发送失败，需要进行重试处理，可重新发送这条消息或持久化这条数据进行补偿处理。
                System.out.println("send message failed.");
                throwable.printStackTrace();
            }
        });


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

订阅普通消息

参考如下示例代码，或者通过**PushConsumer.java**获取更多示例代码。

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

public class PushConsumer {

    public static void main(String[] args) throws InterruptedException, MQClientException {
        DefaultMQPushConsumer consumer = new
DefaultMQPushConsumer("please_rename_unique_group_name");
        //填入连接地址
        consumer.setNamesrvAddr("192.168.0.1:8100");
        //consumer.setUseTLS(true);    //创建实例时，如果开启了SSL，请添加此行代码。
        consumer.subscribe("TopicTest", "*");
        consumer.registerMessageListener(new MessageListenerConcurrently() {
            @Override
            public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs,
ConsumeConcurrentlyContext context) {
                System.out.printf("%s Receive New Messages: %s %n", Thread.currentThread().getName(),
msgs);
                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
            }
        });
        consumer.start();
        System.out.printf("Consumer Started.%n");
    }
}
```

3.2 收发顺序消息

顺序消息是分布式消息服务RocketMQ版提供的一种严格按照顺序来发布和消费的消息类型。

顺序消息分为全局顺序消息和分区顺序消息：

- 全局顺序消息：对于指定的一个Topic，将队列数量设置为1，这个队列内所有消息按照严格的先入先出FIFO（First In First Out）的顺序进行发布和订阅。
- 分区顺序消息：对于指定的一个Topic，同一个队列内的消息按照严格的FIFO顺序进行发布和订阅。生产者指定分区选择算法，保证需要按顺序消费的消息被分配到同一个队列。

全局顺序消息和分区顺序消息的区别仅为队列数量不同，代码没有区别。

收发顺序消息前，请参考**收集连接信息**收集RocketMQ所需的连接信息。

约束与限制

客户端连接RocketMQ实例5.x版本收发顺序消息前，需要确保Topic的消息类型为“顺序”。

准备环境

开源的Java客户端支持连接分布式消息服务RocketMQ版，推荐使用的客户端版本为**4.9.8**。

通过以下任意一种方式引入依赖：

- 使用Maven方式引入依赖。

```
<dependency>
   <groupId>org.apache.rocketmq</groupId>
   <artifactId>rocketmq-client</artifactId>
   <version>4.9.8</version>
</dependency>
```

- 下载**依赖JAR包**。

发送顺序消息

参考如下示例代码，或者通过**Producer.java**获取更多示例代码。

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

public class Producer {

    public static void main(String[] args) {
        try {
            DefaultMQProducer producer = new DefaultMQProducer("please_rename_unique_group_name");
            //填入连接地址
            producer.setNamesrvAddr("192.168.0.1:8100");
            //producer.setUseTLS(true);    //创建实例时，如果开启了SSL，请增加此行代码。
            producer.start();

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

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

```
        }

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

上述代码中，相同orderId的消息需要保证顺序，不同orderId的消息不需要保证顺序，所以在分区选择算法中以“orderId/队列个数的余数”作为消息发送的队列。

订阅顺序消息

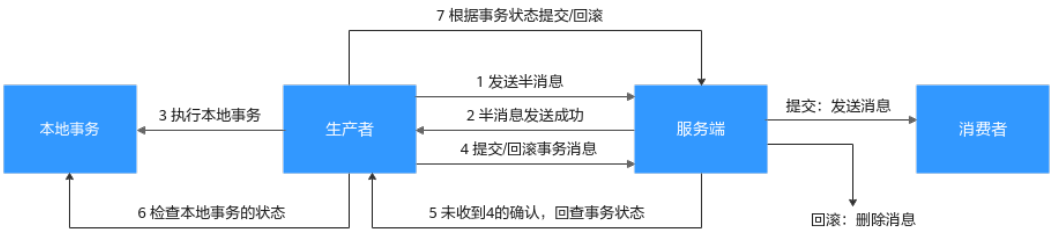
参考如下示例代码，或者通过Consumer.java获取更多示例代码。

```
import java.util.List;
import org.apache.rocketmq.client.consumer.DefaultMQPushConsumer;
import org.apache.rocketmq.client.consumer.listener.ConsumeOrderlyContext;
import org.apache.rocketmq.client.consumer.listener.ConsumeOrderlyStatus;
import org.apache.rocketmq.client.consumer.listener.MessageListenerOrderly;
import org.apache.rocketmq.client.exception.MQClientException;
import org.apache.rocketmq.common.message.MessageExt;
public class Consumer {
    public static void main(String[] args) throws MQClientException {
        DefaultMQPushConsumer consumer = new
DefaultMQPushConsumer("please_rename_unique_group_name_3");
        //填入连接地址
        consumer.setNamesrvAddr("192.168.0.1:8100");
        //consumer.setUseTLS(true);    //创建实例时，如果开启了SSL，请增加此行代码。
        consumer.subscribe("TopicTest", "*");
        consumer.registerMessageListener(new MessageListenerOrderly() {
            @Override
            public ConsumeOrderlyStatus consumeMessage(List<MessageExt> msgs, ConsumeOrderlyContext
context) {
                context.setAutoCommit(true);
                System.out.printf("%s Receive New Messages: %s %n", Thread.currentThread().getName(),
msgs);
                return ConsumeOrderlyStatus.SUCCESS;
            }
        });
        consumer.start();
        System.out.printf("Consumer Started.%n");
    }
}
```

3.3 收发事务消息

分布式消息服务RocketMQ版的事务消息支持在业务逻辑与发送消息之间提供事务保证，通过两阶段的方式提供对事务消息的支持，事务消息交互流程如图3-1所示。

图 3-1 事务消息交互流程



事务消息生产者首先发送半消息，然后执行本地事务。如果执行成功，则发送事务提交，否则发送事务回滚。服务端在一段时间后如果一直收不到提交或回滚，则发起回

查，生产者在收到回查后重新发送事务提交或回滚。消息只有在提交之后才投递给消费者，消费者对回滚的消息不可见。

收发事务消息前，请参考**收集连接信息**收集RocketMQ所需的连接信息。

约束与限制

客户端连接RocketMQ实例5.x版本收发事务消息前，需要确保Topic的消息类型为“事务”。

准备环境

开源的Java客户端支持连接分布式消息服务RocketMQ版，推荐使用的客户端版本为**4.9.8**。

通过以下任意一种方式引入依赖：

- 使用Maven方式引入依赖。

```
<dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-client</artifactId>
    <version>4.9.8</version>
</dependency>
```

- 下载**依赖JAR包**。

发送事务消息

参考如下示例代码，或者通过**TransactionProducer.java**获取更多示例代码。

```
import org.apache.rocketmq.client.exception.MQClientException;
import org.apache.rocketmq.client.producer.LocalTransactionState;
import org.apache.rocketmq.client.producer.SendResult;
import org.apache.rocketmq.client.producer.TransactionListener;
import org.apache.rocketmq.client.producer.TransactionMQProducer;
import org.apache.rocketmq.common.message.Message;
import org.apache.rocketmq.common.message.MessageExt;
import org.apache.rocketmq.remoting.common.RemotingHelper;

import java.io.UnsupportedEncodingException;

public class Main {
    public static void main(String[] args) throws MQClientException, UnsupportedEncodingException {
        TransactionListener transactionListener = new TransactionListener() {
            @Override
            public LocalTransactionState executeLocalTransaction(Message message, Object o) {
                System.out.println("开始执行本地事务： " + message);
                return LocalTransactionState.COMMIT_MESSAGE;
            }

            @Override
            public LocalTransactionState checkLocalTransaction(MessageExt messageExt) {
                System.out.println("收到回查，重新查询事务状态： " + messageExt);
                return LocalTransactionState.COMMIT_MESSAGE;
            }
        };

        TransactionMQProducer producer = new
TransactionMQProducer("please_rename_unique_group_name");
        //填入连接地址
        producer.setNamesrvAddr("192.168.0.1:8100");
        //producer.setUseTLS(true);    //创建实例时，如果开启了SSL，请增加此行代码。
        producer.setTransactionListener(transactionListener);
        producer.start();
```

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

    producer.shutdown();
}}
```

事务消息生产者需要实现两个回调函数，其中executeLocalTransaction回调函数在发送完半事务消息后被调用，即上图中的第3阶段，checkLocalTransaction回调函数在收到回查时调用，即上图中的第6阶段。两个回调函数均可返回3种事务状态：

- LocalTransactionState.COMMIT_MESSAGE：提交事务，允许消费者消费该消息。
- LocalTransactionState.ROLLBACK_MESSAGE：回滚事务，消息将被丢弃不允许消费。
- LocalTransactionState.UNKNOW：无法判断状态，期待服务端向生产者再次回查该消息的状态。

订阅事务消息

订阅事务消息的代码与**订阅普通消息的代码**相同。

3.4 发送定时消息

分布式消息服务RocketMQ版支持**任意时间**的定时消息，最大推迟时间可达到1年。同时也支持定时消息的取消。

定时消息即生产者生产消息到分布式消息服务RocketMQ版后，消息不会立即被消费，而是延迟到设定的时间点后才会发送给消费者进行消费。

发送定时消息前，请参考**收集连接信息**收集RocketMQ所需的连接信息。

约束与限制

- 2022年3月30日及以后购买的实例支持定时消息功能，在此之前购买的实例不支持此功能。
- 客户端连接RocketMQ实例5.x版本收发定时消息前，需要确保Topic的消息类型为“定时”。

适用场景

定时消息适用于以下场景：

- 消息对应的业务逻辑有时间窗口要求，如电商交易中超时未支付关闭订单的场景。在订单创建时发送一条定时消息，5分钟以后投递给消费者，消费者收到此消息后需要判断对应订单是否完成支付，如果未完成支付，则关闭订单。如果已完成，则忽略。
- 通过消息触发定时任务的场景，如在某些固定时间点向用户发送提醒消息。

注意事项

- 定时消息的最大延迟时间为1年，延迟超过1年的消息将会发送失败。

- 定时消息的定时时间如果被设置成当前时间戳之前的某个时刻，消息将立刻投递给消费者。
- 在理想情况下，定时消息设定的时间与实际发送时间的误差在0.1s以内。但在定时消息投递压力过大时，会触发定时消息投递流控机制，精度会变差。
- 在0.1s的精度内，不保证消息投递的顺序性。即如果两条定时消息的定时时间差距小于0.1s，他们投递的顺序与他们发送的顺序无法确保一致。
- 无法确保定时消息仅投递一次，定时消息可能会重复投递。
- 定时消息的定时时间是服务端开始向消费端投递的时间。如果消费者当前有消息堆积，那么定时消息会排在堆积消息后面，将不能严格按照配置的时间进行投递。
- 由于客户端和服务端可能存在时间差，消息的实际投递时间与客户端设置的投递时间之间可能存在偏差，以服务端时间为准。
- 设置定时消息的投递时间后，依然受消息老化时间限制，默认消息老化时间为2天。例如，设置定时消息5天后才能被消费，如果第5天后一直没被消费，那么这条消息将在第7天被删除。
- 定时消息将占用普通消息约3倍的存储空间，大量使用定时消息时需要注意存储空间占用。

准备环境

开源的Java客户端支持连接分布式消息服务RocketMQ版，推荐使用的客户端版本为**4.9.8**。

通过以下任意一种方式引入依赖：

- 使用Maven方式引入依赖。

```
<dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-client</artifactId>
    <version>4.9.8</version>
</dependency>
```

- 下载**依赖JAR包**。

发送定时消息

发送定时消息的示例代码如下：

```
import java.nio.charset.StandardCharsets;
import java.time.Instant;
import org.apache.rocketmq.client.exception.MQBrokerException;
import org.apache.rocketmq.client.exception.MQClientException;
import org.apache.rocketmq.client.producer.DefaultMQProducer;
import org.apache.rocketmq.client.producer.SendResult;
import org.apache.rocketmq.common.UtilAll;
import org.apache.rocketmq.common.message.Message;
import org.apache.rocketmq.remoting.exception.RemotingException;


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


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


        DefaultMQProducer producer = new DefaultMQProducer("ProducerGroupName");
        // 填入连接地址
```

```
        producer.setNamesrvAddr("192.168.0.1:8100");
        //producer.setUseTLS(true);   //创建实例时，如果开启了SSL，请增加此行代码。
        producer.start();


        // 定时消息投递时间戳，该消息10秒后投递
        final long deliverTimestamp = Instant.now().plusSeconds(10).toEpochMilli();
        // 创建消息对象
        Message msg = new Message(TOPIC_NAME,
            "TagA",
            "KEY",
            "scheduled message".getBytes(StandardCharsets.UTF_8));
        // 设置消息定时投递的时间戳属性
        msg.putUserProperty("__STARTDELIVERTIME", String.valueOf(deliverTimestamp));
        // 发送消息，该消息将会在10秒后投递
        SendResult sendResult = producer.send(msg);
        // 打印发送结果和预计投递时间
        System.out.printf("%s %s%n", sendResult, UtilAll.timeMillisToHumanString2(deliverTimestamp));


        producer.shutdown();
    }
}
```

取消定时消息

取消定时消息的示例代码如下：

```
import java.nio.charset.StandardCharsets;
import java.time.Instant;
import org.apache.rocketmq.client.exception.MQBrokerException;
import org.apache.rocketmq.client.exception.MQClientException;
import org.apache.rocketmq.client.producer.DefaultMQProducer;
import org.apache.rocketmq.client.producer.SendResult;
import org.apache.rocketmq.common.UtilAll;
import org.apache.rocketmq.common.message.Message;
import org.apache.rocketmq.remoting.exception.RemotingException;

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

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

        DefaultMQProducer producer = new DefaultMQProducer("ProducerGroupName");
        // 填入连接地址
        producer.setNamesrvAddr("192.168.0.1:8100");
        //producer.setUseTLS(true);   //创建实例时，如果开启了SSL，请增加此行代码。
        producer.start();

        // 定时消息投递时间戳，该消息10秒后投递
        final long deliverTimestamp = Instant.now().plusSeconds(10).toEpochMilli();
        // 创建消息对象
        Message msg = new Message(TOPIC_NAME,
            "TagA",
            "KEY",
            "scheduled message".getBytes(StandardCharsets.UTF_8));
        // 设置消息定时投递的时间戳属性
        msg.putUserProperty("__STARTDELIVERTIME", String.valueOf(deliverTimestamp));
        // 发送消息，该消息将会在10秒后投递
        SendResult sendResult = producer.send(msg);
        // 打印发送结果和预计投递时间
        System.out.printf("%s %s%n", sendResult, UtilAll.timeMillisToHumanString2(deliverTimestamp));

        // ====== 发送取消消息逻辑 ======

        // 创建取消消息对象
        Message cancelMsg = new Message(TOPIC_NAME,
```

```
            """,
            """,
            "cancel".getBytes(StandardCharsets.UTF_8));
        // 设置取消消息的时间戳，该时间戳必须与要取消的定时消息的定时时间戳一致
        cancelMsg.putUserProperty("__STARTDELIVERTIME", String.valueOf(deliverTimestamp));
        // 设置要取消消息的ID，为发送消息的唯一ID（UNIQUE_KEY），可以从发送消息的结果中获取
        cancelMsg.putUserProperty("__CANCEL_SCHEDULED_MSG", sendResult.getMsgId());
        // 发送取消消息，必须在定时消息被投递之前发送才可以取消
        SendResult cancelSendResult = producer.send(cancelMsg, sendResult.getMessageQueue());
        System.out.printf("cancel %s%n", cancelSendResult);

        producer.shutdown();
    }
}
```

3.5 使用 ACL 权限访问

实例开启ACL访问控制后，消息生产者和消费者都需要增加用户认证信息。

准备环境

开源的Java客户端支持连接分布式消息服务RocketMQ版，推荐使用的客户端版本为**4.9.8**。

通过以下任意一种方式引入依赖：

- 使用Maven方式引入依赖。

```
<dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-client</artifactId>
    <version>4.9.8</version>
</dependency>

<dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-acl</artifactId>
    <version>4.9.8</version>
</dependency>
```

- 下载**依赖JAR包**。

生产者增加用户认证信息

生产者初始化时需要增加“rpcHook”参数。

- 普通消息、顺序消息和定时消息，增加如下代码。

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

- 事务消息，增加如下代码。

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

ACL_ACCESS_KEY为用户名，ACL_SECRET_KEY为用户的密钥。创建用户的步骤，请参见**创建用户**。为了确保用户名和密钥的安全性，建议对用户名和密钥进行加密处理。

消费者增加用户认证信息

消费者初始化时需要增加“rpcHook”参数。无论是普通消息、顺序消息、定时消息，还是事务消息，都增加如下代码。

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

```
DefaultMQPushConsumer consumer = new DefaultMQPushConsumer(null, "ConsumerGroupName",
rpcHook);
```

ACL_ACCESS_KEY为用户名，ACL_SECRET_KEY为用户的密钥。创建用户的步骤，请参见**创建用户**。为了确保用户名和密钥的安全性，建议对用户名和密钥进行加密处理。

3.6 消费端限流

在分布式消息服务RocketMQ版中，消费者消费消息时，可能会出现消费过快导致下游业务来不及处理的情况，进而影响系统的稳定性。本章节介绍在消费端进行限流的示例代码，以保障系统的稳定。

准备环境

开源的Java客户端支持连接分布式消息服务RocketMQ版，推荐使用的客户端版本为**4.9.8**。

使用Maven方式引入依赖。

```
<dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-client</artifactId>
    <version>4.9.8</version>
</dependency>

<dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-acl</artifactId>
    <version>4.9.8</version>
</dependency>

<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>29.0-jre</version>
</dependency>
```

示例代码

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

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

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

public class PushConsumer {

    public static void main(String[] args) throws InterruptedException, MQClientException {
        DefaultMQPushConsumer consumer = new
DefaultMQPushConsumer("please_rename_unique_group_name");
        //填入连接地址
        consumer.setNamesrvAddr("192.168.0.1:8100");
        consumer.subscribe("TopicTest", "*");
        consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_OFFSET);
        RateLimiter rateLimiter = RateLimiter.create(200);
        consumer.registerMessageListener(new MessageListenerConcurrently() {
```

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

3.7 RocketMQ-Spring 的使用

本文介绍如何使用RocketMQ-Spring连接华为云RocketMQ实例进行消息的生产和消费。相关代码您可以从**rocketmq-springboot-demo**中获取。

下文所有RocketMQ的配置信息，如实例连接地址、Topic名称、用户信息等，请参考**收集连接信息**获取。

在 pom.xml 文件中引入 RocketMQ-Spring 依赖

```
<dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-spring-boot-starter</artifactId>
    <version>${RELEASE.VERSION}</version>
</dependency>
```

在 application.properties 文件中填写配置

```
#=============== 生产者配置 =======================
## 替换成真实RocketMQ的NameServer地址与端口
rocketmq.name-server=127.0.0.1:9876
rocketmq.producer.group=my-group
## 是否开启SSL
rocketmq.producer.tls-enable=true
#=============== 消费者配置 =======================
## 替换成真实RocketMQ的NameServer地址与端口
rocketmq.name-server=127.0.0.1:9876
```

生产消息

生产消息的示例代码如下（以下加粗内容需要替换为实例自有信息，请根据实际情况替换）。

```
@SpringBootApplication
public class ProduceDemoApplication implements CommandLineRunner {
  @Resource
  private RocketMQTemplate rocketMQTemplate;

  public static void main(String[] args) {
      SpringApplication.run(ProduceDemoApplication.class, args);
  }

  @Override
  public void run(String... args) throws Exception {
      // send message synchronously
      rocketMQTemplate.convertAndSend("topic", "Hello, World!");
      // send spring message
      rocketMQTemplate.send(
```

```
            "topic", MessageBuilder.withPayload("Hello, World! I'm from spring message").build());
        // send messgae asynchronously
        rocketMQTemplate.asyncSend(
                "topic",
                MessageBuilder.withPayload("Hello, World! I'm from spring message").build(),
                new SendCallback() {
                    @Override
                    public void onSuccess(SendResult var1) {
                        System.out.printf("async onSucess SendResult=%s %n", var1);
                    }

                    @Override
                    public void onException(Throwable var1) {
                        System.out.printf("async onException Throwable=%s %n", var1);
                    }
                });
        // Send messages orderly
        rocketMQTemplate.syncSendOrderly(
                "topic", MessageBuilder.withPayload("Hello, World").build(), "hashkey");
    }
}
```

消费消息

消费消息的示例代码如下（以下加粗内容需要替换为实例自有信息，请根据实际情况替换）。

```
@SpringBootApplication
public class ConsumeDemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(ConsumeDemoApplication.class, args);
    }

    @Service
    @RocketMQMessageListener(topic = "topic", consumerGroup = "group", tlsEnable = "true")
    public static class MyConsumer implements RocketMQListener<String> {
        public void onMessage(String message) {
            System.out.printf("received message: %s", message);
        }
    }
}
```

4 Java（gRPC 协议）

4.1 收发普通消息

本章节介绍普通消息的收发方法和示例代码。其中，普通消息发送方式分为同步发送和异步发送。RocketMQ提供PushConsumer和SimpleConsumer类型的消费者，PushConsumer消费者订阅普通消息时，代码不区分同步订阅和异步订阅。SimpleConsumer消费者订阅普通消息时，代码需要区分同步订阅和异步订阅。

普通消息发送方式分为同步发送和异步发送。

- 同步发送：消息发送方发出一条消息到服务端，服务端接收并处理消息，然后返回响应给发送方，发送方收到响应后才会发送下一条消息。
- 异步发送：消息发送方发出一条消息后，不等服务端返回响应，接着发送下一条消息。

表 4-1 普通消息收发方式

消息类型	发送消息	订阅消息（PushConsumer）	订阅消息（SimpleConsumer）
普通消息	• 同步发送 • 异步发送	不分区同步订阅和异步订阅	• 同步订阅 • 异步订阅

收发消息前，请参考**收集连接信息**收集RocketMQ所需的连接信息。

约束与限制

- 仅RocketMQ实例5.x版本支持gRPC协议，4.8.0版本不支持。
- 客户端连接RocketMQ实例5.x版本收发普通消息前，需要确保Topic的消息类型为“普通”。

准备环境

开源的Java客户端支持连接分布式消息服务RocketMQ版，推荐使用的客户端版本为**5.0.5**。

使用Maven方式引入依赖。

```
<dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-client-java</artifactId>
    <version>5.0.5</version>
</dependency>
```

同步发送

同步发送是指消息发送方发出一条消息到服务端，服务端接收并处理消息，然后返回响应给发送方，发送方收到响应后才会发送下一条消息的通讯方式。

参考如下示例代码，或者通过**ProducerNormalMessageExample.java**获取更多示例代码。

```
import org.apache.rocketmq.client.apis.ClientConfiguration;
import org.apache.rocketmq.client.apis.ClientException;
import org.apache.rocketmq.client.apis.ClientServiceProvider;
import org.apache.rocketmq.client.apis.SessionCredentialsProvider;
import org.apache.rocketmq.client.apis.StaticSessionCredentialsProvider;
import org.apache.rocketmq.client.apis.message.Message;
import org.apache.rocketmq.client.apis.producer.Producer;
import org.apache.rocketmq.client.apis.producer.SendReceipt;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.io.IOException;
import java.nio.charset.StandardCharsets;

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

    public static void main(String[] args) throws ClientException, IOException {
        final ClientServiceProvider provider = ClientServiceProvider.loadService();

        String topic = "yourNormalTopics";
        // 填入grpc连接地址/grpc公网连接地址
        String endpoints = "yourEndpoints";
        // 创建实例时，如果开启了ACL才需要添加以下代码。
        String accessKey = System.getenv("ACL_ACCESS_KEY");
        String secretKey = System.getenv("ACL_SECRET_KEY");
        //ACL_ACCESS_KEY为用户名，ACL_SECRET_KEY为用户的密钥。创建用户的步骤，请参见创建用户。用户名和密钥直接硬编码到代码中或者明文存储都存在很大的风险，建议在配置文件或者环境变量中密文存放，使用时解密。
        SessionCredentialsProvider sessionCredentialsProvider =
                new StaticSessionCredentialsProvider(accessKey, secretKey);

        ClientConfiguration clientConfiguration = ClientConfiguration.newBuilder()
                .setEndpoints(endpoints)
                // .enableSsl(false)  // 创建实例时，如果将“SSL”配置为“PLAINTEXT”，则请增加此行代码。如果将“SSL”配置为“PERMISSIVE”，则请根据实际情况选择是否增加此行代码。
                // .setCredentialProvider(sessionCredentialsProvider)  // 创建实例时，如果开启了ACL，请添加此行代码。
                .build();

        final Producer producer = provider.newProducerBuilder()
                .setClientConfiguration(clientConfiguration)
                .setTopics(topic)
                .build();
        byte[] body = "This is a normal message for Apache RocketMQ".getBytes(StandardCharsets.UTF_8);
        String tag = "yourMessageTagA";
        final Message message = provider.newMessageBuilder()
                .setTopic(topic)
                .setTag(tag)
                .setKeys("yourMessageKey")
                .setBody(body)
                .build();
        try {
```

```
            final SendReceipt sendReceipt = producer.send(message);
            log.info("Send message successfully, messageId={}", sendReceipt.getMessageId());
        } catch (Throwable t) {
            log.error("Failed to send message", t);
        }
        // 不再使用后，手动关闭producer。
        producer.close();
    }
}
```

异步发送

异步发送是指消息发送方发出一条消息后，不等服务端返回响应，接着发送下一条消息的通讯方式。

参考如下示例代码，或者通过**AsyncProducerExample.java**获取更多示例代码。

```
import org.apache.rocketmq.client.apis.ClientConfiguration;
import org.apache.rocketmq.client.apis.ClientException;
import org.apache.rocketmq.client.apis.ClientServiceProvider;
import org.apache.rocketmq.client.apis.SessionCredentialsProvider;
import org.apache.rocketmq.client.apis.StaticSessionCredentialsProvider;
import org.apache.rocketmq.client.apis.message.Message;
import org.apache.rocketmq.client.apis.producer.Producer;
import org.apache.rocketmq.client.apis.producer.SendReceipt;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

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

    private AsyncProducerExample() {
    }

    public static void main(String[] args) throws ClientException, InterruptedException, IOException {
        final ClientServiceProvider provider = ClientServiceProvider.loadService();

        String topic = "yourNormalTopics";
        // 填入grpc连接地址/grpc公网连接地址
        String endpoints = "yourEndpoints";
        // 创建实例时，如果开启了ACL才需要添加以下代码。
        String accessKey = System.getenv("ACL_ACCESS_KEY");
        String secretKey = System.getenv("ACL_SECRET_KEY");
        //ACL_ACCESS_KEY为用户名，ACL_SECRET_KEY为用户的密钥。创建用户的步骤，请参见创建用户。用户名和密钥直接硬编码到代码中或者明文存储都存在很大的风险，建议在配置文件或者环境变量中密文存放，使用时解密。
        SessionCredentialsProvider sessionCredentialsProvider =
                new StaticSessionCredentialsProvider(accessKey, secretKey);

        ClientConfiguration clientConfiguration = ClientConfiguration.newBuilder()
                .setEndpoints(endpoints)
                // .enableSsl(false)  // 创建实例时，如果将“SSL”配置为“PLAINTEXT”，则请增加此行代码。如果将“SSL”配置为“PERMISSIVE”，则请根据实际情况选择是否增加此行代码。
                // .setCredentialProvider(sessionCredentialsProvider)  // 创建实例时，如果开启了ACL，请添加此行代码。
                .build();
        final Producer producer = provider.newProducerBuilder()
                .setClientConfiguration(clientConfiguration)
                .setTopics(topic)
                .build();

        byte[] body = "This is a normal message for Apache RocketMQ".getBytes(StandardCharsets.UTF_8);
```

```
        String tag = "yourMessageTagA";
        final Message message = provider.newMessageBuilder()
                .setTopic(topic)
                .setTag(tag)
                .setKeys("yourMessageKey")
                .setBody(body)
                .build();

        final CompletableFuture<SendReceipt> future = producer.sendAsync(message);
        // 使用线程池去处理异步发送回调
        ExecutorService sendCallbackExecutor = Executors.newCachedThreadPool();
        future.whenCompleteAsync((sendReceipt, throwable) -> {
            if (null != throwable) {
                log.error("Failed to send message", throwable);
                return;
            }
            log.info("Send message successfully, messageId={}", sendReceipt.getMessageId());
        }, sendCallbackExecutor);
        // 阻塞主线程，不需要在生产环境中使用。
        Thread.sleep(Long.MAX_VALUE);
        // 不再使用后，手动关闭producer。
        producer.close();
    }
}
```

订阅普通消息（PushConsumer）

参考如下示例代码，或者通过**PushConsumerExample.java**获取更多示例代码。

```
import org.apache.rocketmq.client.apis.ClientConfiguration;
import org.apache.rocketmq.client.apis.ClientException;
import org.apache.rocketmq.client.apis.ClientServiceProvider;
import org.apache.rocketmq.client.apis.SessionCredentialsProvider;
import org.apache.rocketmq.client.apis.StaticSessionCredentialsProvider;
import org.apache.rocketmq.client.apis.consumer.ConsumeResult;
import org.apache.rocketmq.client.apis.consumer.FilterExpression;
import org.apache.rocketmq.client.apis.consumer.FilterExpressionType;
import org.apache.rocketmq.client.apis.consumer.PushConsumer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.io.IOException;
import java.util.Collections;

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

    private PushConsumerExample() {
    }

    public static void main(String[] args) throws ClientException, InterruptedException, IOException {
        final ClientServiceProvider provider = ClientServiceProvider.loadService();

        // 填入grpc连接地址/grpc公网连接地址
        String endpoints = "yourEndpoints";
        // 创建实例时，如果开启了ACL才需要添加以下代码。
        String accessKey = System.getenv("ACL_ACCESS_KEY");
        String secretKey = System.getenv("ACL_SECRET_KEY");
        //ACL_ACCESS_KEY为用户名，ACL_SECRET_KEY为用户的密钥。创建用户的步骤，请参见创建用户。用户名和密钥直接硬编码到代码中或者明文存储都存在很大的风险，建议在配置文件或者环境变量中密文存放，使用时解密。
        SessionCredentialsProvider sessionCredentialsProvider =
                new StaticSessionCredentialsProvider(accessKey, secretKey);

        ClientConfiguration clientConfiguration = ClientConfiguration.newBuilder()
                .setEndpoints(endpoints)
                // .enableSsl(false)  // 创建实例时，如果将“SSL”配置为“PLAINTEXT”，则请增加此行代码。如果将“SSL”配置为“PERMISSIVE”，则请根据实际情况选择是否增加此行代码。
```

```
            // .setCredentialProvider(sessionCredentialsProvider)  // 创建实例时，如果开启了ACL，请添加此行代码。
            .build();
        String tag = "yourMessageTagA";
        FilterExpression filterExpression = new FilterExpression(tag, FilterExpressionType.TAG);
        String consumerGroup = "yourConsumerGroup";
        String topic = "yourTopic";
        // 在多数场景下，推荐使用单例模式创建consumer。
        PushConsumer pushConsumer = provider.newPushConsumerBuilder()
            .setClientConfiguration(clientConfiguration)
            .setConsumerGroup(consumerGroup)
            // 设置订阅关系
            .setSubscriptionExpressions(Collections.singletonMap(topic, filterExpression))
            // 设置监听器，用于处理接收到的消息，并返回消费结果。
            .setMessageListener(messageView -> {
                log.info("Consume message={}", messageView);
                return ConsumeResult.SUCCESS;
            })
            .build();
        // 阻塞主线程，不需要在生产环境中使用。
        Thread.sleep(Long.MAX_VALUE);
        // 不再使用后，手动关闭producer。
        pushConsumer.close();
    }
}
```

同步订阅普通消息（SimpleConsumer）

参考如下示例代码，或者通过**SimpleConsumerExample.java**获取更多示例代码。

```
import org.apache.rocketmq.client.apis.ClientConfiguration;
import org.apache.rocketmq.client.apis.ClientException;
import org.apache.rocketmq.client.apis.ClientServiceProvider;
import org.apache.rocketmq.client.apis.SessionCredentialsProvider;
import org.apache.rocketmq.client.apis.StaticSessionCredentialsProvider;
import org.apache.rocketmq.client.apis.consumer.FilterExpression;
import org.apache.rocketmq.client.apis.consumer.FilterExpressionType;
import org.apache.rocketmq.client.apis.consumer.SimpleConsumer;
import org.apache.rocketmq.client.apis.message.MessageId;
import org.apache.rocketmq.client.apis.message.MessageView;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.time.Duration;
import java.util.Collections;
import java.util.List;

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

    private SimpleConsumerExample() {
    }

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

        // 填入grpc连接地址/grpc公网连接地址
        String endpoints = "yourEndpoints";
        // 创建实例时，如果开启了ACL才需要添加以下代码。
        String accessKey = System.getenv("ACL_ACCESS_KEY");
        String secretKey = System.getenv("ACL_SECRET_KEY");
        //ACL_ACCESS_KEY为用户名，ACL_SECRET_KEY为用户的密钥。创建用户的步骤，请参见创建用户。用户名和密钥直接硬编码到代码中或者明文存储都存在很大的风险，建议在配置文件或者环境变量中密文存放，使用时解密。
        SessionCredentialsProvider sessionCredentialsProvider =
            new StaticSessionCredentialsProvider(accessKey, secretKey);
```

```
        ClientConfiguration clientConfiguration = ClientConfiguration.newBuilder()
                .setEndpoints(endpoints)
                // .enableSsl(false)  // 创建实例时，如果将“SSL”配置为“PLAINTEXT”，则请增加此行代码。如果将“SSL”配置为“PERMISSIVE”，则请根据实际情况选择是否增加此行代码。
                // .setCredentialProvider(sessionCredentialsProvider)  // 创建实例时，如果开启了ACL，请添加此行代码。
                .build();
        String consumerGroup = "yourConsumerGroup";
        Duration awaitDuration = Duration.ofSeconds(30);
        String tag = "yourMessageTagA";
        String topic = "yourTopic";
        FilterExpression filterExpression = new FilterExpression(tag, FilterExpressionType.TAG);
        // 在多数场景下，推荐使用单例模式创建consumer。
        SimpleConsumer consumer = provider.newSimpleConsumerBuilder()
                .setClientConfiguration(clientConfiguration)
                .setConsumerGroup(consumerGroup)
                // 设置长轮询接收消息请求（long-polling receive requests）的最大等待时间
                .setAwaitDuration(awaitDuration)
                // 设置订阅关系
                .setSubscriptionExpressions(Collections.singletonMap(topic, filterExpression))
                .build();
        // 设置每次长轮询能接收的最大消息数
        int maxMessageNum = 16;
        // 设置消息不可见时间，在消息被接收后对其他消费者不可见，直到超时。
        Duration invisibleDuration = Duration.ofSeconds(15);
        // 接收消息，推荐使用多线程的方式。
        while (true) {
            // 如果存在可用的消息会立即返回，否则在等待超时后，返回空。
            final List<MessageView> messages = consumer.receive(maxMessageNum, invisibleDuration);
            log.info("Received {} message(s)", messages.size());
            for (MessageView message : messages) {
                final MessageId messageId = message.getMessageId();
                try {
                    // 处理接收的消息，消费成功后提交消息。
                    consumer.ack(message);
                    log.info("Message is acknowledged successfully, messageId={}", messageId);
                } catch (Throwable t) {
                    log.error("Message is failed to be acknowledged, messageId={}", messageId, t);
                }
            }
        }
        // 不再使用后，手动关闭consumer。
        // consumer.close();
    }
}
```

异步订阅普通消息（SimpleConsumer）

参考如下示例代码，或者通过AsyncSimpleConsumerExample.java获取更多示例代码。

```
import org.apache.rocketmq.client.apis.ClientConfiguration;
import org.apache.rocketmq.client.apis.ClientException;
import org.apache.rocketmq.client.apis.ClientServiceProvider;
import org.apache.rocketmq.client.apis.SessionCredentialsProvider;
import org.apache.rocketmq.client.apis.StaticSessionCredentialsProvider;
import org.apache.rocketmq.client.apis.consumer.FilterExpression;
import org.apache.rocketmq.client.apis.consumer.FilterExpressionType;
import org.apache.rocketmq.client.apis.consumer.SimpleConsumer;
import org.apache.rocketmq.client.apis.message.MessageId;
import org.apache.rocketmq.client.apis.message.MessageView;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.time.Duration;
import java.util.Collections;
import java.util.List;
import java.util.Map;
```

```
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;
import java.util.stream.Collectors;

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

    private AsyncSimpleConsumerExample() {
    }

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

        // 填入grpc连接地址/grpc公网连接地址
        String endpoints = "yourEndpoints";
        // 创建实例时，如果开启了ACL才需要添加以下代码。
        String accessKey = System.getenv("ACL_ACCESS_KEY");
        String secretKey = System.getenv("ACL_SECRET_KEY");
        //ACL_ACCESS_KEY为用户名，ACL_SECRET_KEY为用户的密钥。创建用户的步骤，请参见创建用户。用户名和密钥直接硬编码到代码中或者明文存储都存在很大的风险，建议在配置文件或者环境变量中密文存放，使用时解密。
        SessionCredentialsProvider sessionCredentialsProvider =
                new StaticSessionCredentialsProvider(accessKey, secretKey);

        ClientConfiguration clientConfiguration = ClientConfiguration.newBuilder()
                .setEndpoints(endpoints)
                // .enableSsl(false)  // 创建实例时，如果将“SSL”配置为“PLAINTEXT”，则请增加此行代码。如果将“SSL”配置为“PERMISSIVE”，则请根据实际情况选择是否增加此行代码。
                // .setCredentialProvider(sessionCredentialsProvider)  // 创建实例时，如果开启了ACL，请添加此行代码。
                .build();
        String consumerGroup = "yourConsumerGroup";
        Duration awaitDuration = Duration.ofSeconds(30);
        String tag = "yourMessageTagA";
        String topic = "yourTopic";
        FilterExpression filterExpression = new FilterExpression(tag, FilterExpressionType.TAG);
        // 在多数场景下，推荐使用单例模式创建consumer。
        SimpleConsumer consumer = provider.newSimpleConsumerBuilder()
                .setClientConfiguration(clientConfiguration)
                .setConsumerGroup(consumerGroup)
                // 设置长轮询接收消息请求（long-polling receive requests）的最大等待时间
                .setAwaitDuration(awaitDuration)
                // 设置订阅关系
                .setSubscriptionExpressions(Collections.singletonMap(topic, filterExpression))
                .build();
        // 设置每次长轮询能接收的最大消息数
        int maxMessageNum = 16;
        // 设置消息不可见时间，在消息被接收后对其他消费者不可见，直到超时。
        Duration invisibleDuration = Duration.ofSeconds(15);
        // 设置允许的最大长轮询请求数
        int maxLongPollingSize = 32;
        Semaphore semaphore = new Semaphore(maxLongPollingSize);
        // 使用线程池处理接收消息回调
        ExecutorService receiveCallbackExecutor = Executors.newCachedThreadPool();
        // 使用线程池处理提交消息回调
        ExecutorService ackCallbackExecutor = Executors.newCachedThreadPool();
        // 接收消息
        while (true) {
            semaphore.acquire();
            // 异步提交消息，如果存在可用的消息会立即返回触发回调，否则在等待超时后，返回空。
            final CompletableFuture<List<MessageView>> future0 = consumer.receiveAsync(maxMessageNum,
                    invisibleDuration);
            future0.whenCompleteAsync(((messages, throwable) -> {
                // 处理接收到的消息
                semaphore.release();
                if (null != throwable) {
```

```
                log.error("Failed to receive message from remote", throwable);
                return;
            }
            log.info("Received {} message(s)", messages.size());
            // 异步提交消息，并使用messageView作为键，因为message id可能重复。
            final Map<MessageView, CompletableFuture<Void>> map =
                    messages.stream().collect(Collectors.toMap(message -> message, consumer::ackAsync));
            for (Map.Entry<MessageView, CompletableFuture<Void>> entry : map.entrySet()) {
                final MessageId messageId = entry.getKey().getMessageId();
                final CompletableFuture<Void> future = entry.getValue();
                future.whenCompleteAsync((v, t) -> {
                    // 处理提交消息回调
                    if (null != t) {
                        log.error("Message is failed to be acknowledged, messageId={}", messageId, t);
                        return;
                    }
                    log.info("Message is acknowledged successfully, messageId={}", messageId);
                }, ackCallbackExecutor);
            }

        }), receiveCallbackExecutor);
    }
    // 不再使用后，手动关闭consumer。
    // consumer.close();
  }
}
```

4.2 收发顺序消息

顺序消息是分布式消息服务RocketMQ版提供的一种严格按照顺序来发布和消费的消息类型。

顺序消息分为全局顺序消息和分区顺序消息：

- 全局顺序消息：对于指定的一个Topic，将队列数量设置为1，这个队列内所有消息按照严格的先入先出FIFO（First In First Out）的顺序进行发布和订阅。
- 分区顺序消息：对于指定的一个Topic，同一个队列内的消息按照严格的FIFO顺序进行发布和订阅。生产者为每一条消息指定消息组，相同消息组的消息会被分配到同一个队列。

全局顺序消息和分区顺序消息的区别仅为队列数量不同，代码没有区别。

收发顺序消息前，请参考**收集连接信息**收集RocketMQ所需的连接信息。

约束与限制

- 仅RocketMQ实例5.x版本支持gRPC协议，4.8.0版本不支持。
- 客户端连接RocketMQ实例5.x版本收发顺序消息前，需要确保Topic的消息类型为“顺序”。
- 使用gRPC协议连接RocketMQ实例时，消费者是否顺序消费消息，取决于消费组中是否开启顺序消费，并非在消费代码中设置，顺序消费消息的代码与普通消费的代码相同。

准备环境

开源的Java客户端支持连接分布式消息服务RocketMQ版，推荐使用的客户端版本为**5.0.5**。

使用Maven方式引入依赖。

```
<dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-client-java</artifactId>
    <version>5.0.5</version>
</dependency>
```

发送顺序消息

参考如下示例代码，或者通过**ProducerFifoMessageExample.java**获取更多示例代码。

```
import org.apache.rocketmq.client.apis.ClientConfiguration;
import org.apache.rocketmq.client.apis.ClientException;
import org.apache.rocketmq.client.apis.ClientServiceProvider;
import org.apache.rocketmq.client.apis.SessionCredentialsProvider;
import org.apache.rocketmq.client.apis.StaticSessionCredentialsProvider;
import org.apache.rocketmq.client.apis.message.Message;
import org.apache.rocketmq.client.apis.producer.Producer;
import org.apache.rocketmq.client.apis.producer.SendReceipt;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.io.IOException;
import java.nio.charset.StandardCharsets;


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

    public static void main(String[] args) throws ClientException, IOException {
        final ClientServiceProvider provider = ClientServiceProvider.loadService();

        String topic = "yourNormalTopics";
        // 填入grpc连接地址/grpc公网连接地址
        String endpoints = "yourEndpoints";
        // 创建实例时，如果开启了ACL才需要添加以下代码。
        String accessKey = System.getenv("ACL_ACCESS_KEY");
        String secretKey = System.getenv("ACL_SECRET_KEY");
        //ACL_ACCESS_KEY为用户名，ACL_SECRET_KEY为用户的密钥。创建用户的步骤，请参见创建用户。用户名和密钥直接硬编码到代码中或者明文存储都存在很大的风险，建议在配置文件或者环境变量中密文存放，使用时解密。
        SessionCredentialsProvider sessionCredentialsProvider =
                new StaticSessionCredentialsProvider(accessKey, secretKey);

        ClientConfiguration clientConfiguration = ClientConfiguration.newBuilder()
                .setEndpoints(endpoints)
                // .enableSsl(false)  // 创建实例时，如果将“SSL”配置为“PLAINTEXT”，则请增加此行代码。如果将“SSL”配置为“PERMISSIVE”，则请根据实际情况选择是否增加此行代码。
                // .setCredentialProvider(sessionCredentialsProvider)  // 创建实例时，如果开启了ACL，请添加此行代码。
                .build();

        final Producer producer = provider.newProducerBuilder()
                .setClientConfiguration(clientConfiguration)
                .setTopics(topic)
                .build();
        byte[] body = "This is a FIFO message for Apache RocketMQ".getBytes(StandardCharsets.UTF_8);
        String tag = "yourMessageTagA";
        final Message message = provider.newMessageBuilder()
                .setTopic(topic)
                .setTag(tag)
                .setKeys("yourMessageKey")
                // 指定消息组，相同消息组的消息会被分配到同一个队列。
                .setMessageGroup("yourMessageGroup0")
                .setBody(body)
                .build();
        try {
            final SendReceipt sendReceipt = producer.send(message);
            log.info("Send message successfully, messageId={}", sendReceipt.getMessageId());
```

```
        } catch (Throwable t) {
            log.error("Failed to send message", t);
        }

        // 不再使用后，手动关闭producer。
        producer.close();
    }
}
```

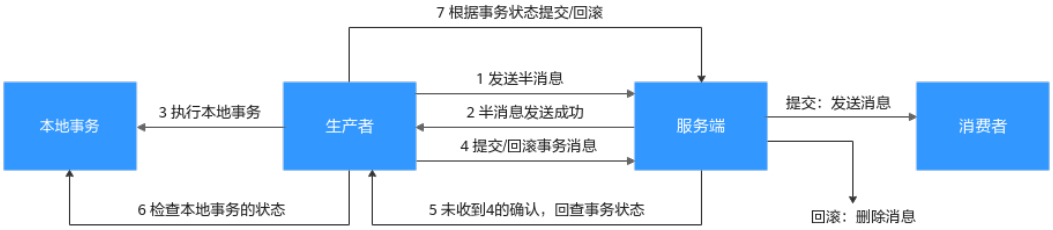
订阅顺序消息

订阅顺序消息的代码与**订阅普通消息**相同。

4.3 收发事务消息

分布式消息服务RocketMQ版的事务消息支持在业务逻辑与发送消息之间提供事务保证，通过两阶段的方式提供对事务消息的支持，事务消息交互流程如**图4-1**所示。

图 4-1 事务消息交互流程



事务消息生产者首先发送半消息，然后执行本地事务。如果执行成功，则发送事务提交，否则发送事务回滚。服务端在一段时间后如果一直收不到提交或回滚，则发起回查，生产者在收到回查后重新发送事务提交或回滚。消息只有在提交之后才投递给消费者，消费者对回滚的消息不可见。

收发事务消息前，请参考**收集连接信息**收集RocketMQ所需的连接信息。

约束与限制

- 仅RocketMQ实例5.x版本支持gRPC协议，4.8.0版本不支持。
- 客户端连接RocketMQ实例5.x版本收发事务消息前，需要确保Topic的消息类型为“事务”。

准备环境

开源的Java客户端支持连接分布式消息服务RocketMQ版，推荐使用的客户端版本为**5.0.5**。

使用Maven方式引入依赖。

```
<dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-client-java</artifactId>
    <version>5.0.5</version>
</dependency>
```

发送事务消息

参考如下示例代码，或者通过**ProducerTransactionMessageExample.java**获取更多示例代码。

```
import org.apache.rocketmq.client.apis.ClientConfiguration;
import org.apache.rocketmq.client.apis.ClientException;
import org.apache.rocketmq.client.apis.ClientServiceProvider;
import org.apache.rocketmq.client.apis.SessionCredentialsProvider;
import org.apache.rocketmq.client.apis.StaticSessionCredentialsProvider;
import org.apache.rocketmq.client.apis.message.Message;
import org.apache.rocketmq.client.apis.producer.Producer;
import org.apache.rocketmq.client.apis.producer.SendReceipt;
import org.apache.rocketmq.client.apis.producer.Transaction;
import org.apache.rocketmq.client.apis.producer.TransactionChecker;
import org.apache.rocketmq.client.apis.producer.TransactionResolution;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.io.IOException;
import java.nio.charset.StandardCharsets;

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

    public static void main(String[] args) throws ClientException, IOException {
        final ClientServiceProvider provider = ClientServiceProvider.loadService();

        String topic = "yourTransactionTopic";
        // 填入grpc连接地址/grpc公网连接地址
        String endpoints = "yourEndpoints";
        // 创建实例时，如果开启了ACL才需要添加以下代码。
        String accessKey = System.getenv("ACL_ACCESS_KEY");
        String secretKey = System.getenv("ACL_SECRET_KEY");
        //ACL_ACCESS_KEY为用户名，ACL_SECRET_KEY为用户的密钥。创建用户的步骤，请参见创建用户。用户名和密钥直接硬编码到代码中或者明文存储都存在很大的风险，建议在配置文件或者环境变量中密文存放，使用时解密。
        SessionCredentialsProvider sessionCredentialsProvider =
                new StaticSessionCredentialsProvider(accessKey, secretKey);

        ClientConfiguration clientConfiguration = ClientConfiguration.newBuilder()
                .setEndpoints(endpoints)
                // .enableSsl(false)  // 创建实例时，如果将“SSL”配置为“PLAINTEXT”，则请增加此行代码。如果将“SSL”配置为“PERMISSIVE”，则请根据实际情况选择是否增加此行代码。
                // .setCredentialProvider(sessionCredentialsProvider)  // 创建实例时，如果开启了ACL，请添加此行代码。
                .build();

        TransactionChecker checker = messageView -> {
            log.info("Receive transactional message check, message={}", messageView);
            // Return the transaction resolution according to your business logic.
            // 检查本地事务并返回本地事务状态
            return TransactionResolution.COMMIT;
        };
        final Producer producer = provider.newProducerBuilder()
                .setClientConfiguration(clientConfiguration)
                .setTopics(topic)
                // 事务消息需要生产者构建一个事务检查器，用于检查确认异常半事务的中间状态。
                .setTransactionChecker(checker)
                .build();
        byte[] body = "This is a transaction message for Apache RocketMQ".getBytes(StandardCharsets.UTF_8);
        String tag = "yourMessageTagA";
        final Message message = provider.newMessageBuilder()
                .setTopic(topic)
                .setTag(tag)
                .setKeys("yourMessageKey")
                .setBody(body)
                .build();
```

```
        // 开启事务分支
        final Transaction transaction;
        try {
            transaction = producer.beginTransaction();
        } catch (ClientException e) {
            log.error("Failed to begin transaction", e);
            //事务分支开启失败，直接退出。
            return;
        }
        try {
            final SendReceipt sendReceipt = producer.send(message, transaction);
            log.info("Send transaction message successfully, messageId={}", sendReceipt.getMessageId());
        } catch (Throwable t) {
            log.error("Failed to send message", t);
            return;
        }
        /**
         * 执行本地事务，并确定本地事务结果。
         * 1. 如果本地事务提交成功，则提交消息事务。
         * 2. 如果本地事务提交失败，则回滚消息事务。
         * 3. 如果本地事务未知异常，则不处理，等待事务消息回查。
         *
         */
        transaction.commit();
        // transaction.rollback();

        // 不再使用后，手动关闭producer。
        producer.close();
    }
}
```

事务消息需要生产者构建一个事务检查器，用于检查确认异常半事务的中间状态，可以返回3种事务状态：

- TransactionResolution.COMMIT：提交事务，允许消费者消费该消息。
- TransactionResolution.ROLLBACK：回滚事务，消息将被丢弃不允许消费。
- TransactionResolution.UNKNOW：无法判断状态，期待服务端向生产者再次回查该消息的状态。

订阅事务消息

订阅事务消息的代码与**订阅普通消息**相同。

4.4 发送定时消息

分布式消息服务RocketMQ版支持**任意时间**的定时消息，最大推迟时间可达到1年。

定时消息即生产者生产消息到分布式消息服务RocketMQ版后，消息不会立即被消费，而是延迟到设定的时间点后才会发送给消费者进行消费。

发送定时消息前，请参考**收集连接信息**收集RocketMQ所需的连接信息。

约束与限制

- 仅RocketMQ实例5.x版本支持gRPC协议，4.8.0版本不支持。
- 客户端连接RocketMQ实例5.x版本收发定时消息前，需要确保Topic的消息类型为“定时”。

适用场景

定时消息适用于以下场景：

- 消息对应的业务逻辑有时间窗口要求，如电商交易中超时未支付关闭订单的场景。在订单创建时发送一条定时消息，5分钟以后投递给消费者，消费者收到此消息后需要判断对应订单是否完成支付，如果未完成支付，则关闭订单。如果已完成，则忽略。
- 通过消息触发定时任务的场景，如在某些固定时间点向用户发送提醒消息。

注意事项

- 定时消息的最大延迟时间为1年，延迟超过1年的消息将会发送失败。
- 定时消息的定时时间如果被设置成当前时间戳之前的某个时刻，消息将立刻投递给消费者。
- 在理想情况下，定时消息设定的时间与实际发送时间的误差在0.1s以内。但在定时消息投递压力过大时，会触发定时消息投递流控机制，精度会变差。
- 在0.1s的精度内，不保证消息投递的顺序性。即如果两条定时消息的定时时间差距小于0.1s，他们投递的顺序与他们发送的顺序无法确保一致。
- 无法确保定时消息仅投递一次，定时消息可能会重复投递。
- 定时消息的定时时间是服务端开始向消费端投递的时间。如果消费者当前有消息堆积，那么定时消息会排在堆积消息后面，将不能严格按照配置的时间进行投递。
- 由于客户端和服务端可能存在时间差，消息的实际投递时间与客户端设置的投递时间之间可能存在偏差，以服务端时间为准。
- 设置定时消息的投递时间后，依然受消息老化时间限制，默认消息老化时间为2天。例如，设置定时消息5天后才能被消费，如果第5天后一直没被消费，那么这条消息将在第7天被删除。
- 定时消息将占用普通消息约3倍的存储空间，大量使用定时消息时需要注意存储空间占用。

准备环境

开源的Java客户端支持连接分布式消息服务RocketMQ版，推荐使用的客户端版本为**5.0.5**。

使用Maven方式引入依赖。

```
<dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-client-java</artifactId>
    <version>5.0.5</version>
</dependency>
```

发送定时消息

发送定时消息的示例代码如下，或者通过**ProducerDelayMessageExample.java**获取更多示例代码。

```
import org.apache.rocketmq.client.apis.ClientConfiguration;
import org.apache.rocketmq.client.apis.ClientException;
import org.apache.rocketmq.client.apis.ClientServiceProvider;
import org.apache.rocketmq.client.apis.SessionCredentialsProvider;
import org.apache.rocketmq.client.apis.StaticSessionCredentialsProvider;
import org.apache.rocketmq.client.apis.message.Message;
```

```
import org.apache.rocketmq.client.apis.producer.Producer;
import org.apache.rocketmq.client.apis.producer.SendReceipt;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.time.Duration;

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

    private ProducerDelayMessageExample() {
    }

    public static void main(String[] args) throws ClientException, IOException {
        final ClientServiceProvider provider = ClientServiceProvider.loadService();

        String topic = "yourDelayTopic";
        // 填入grpc连接地址/grpc公网连接地址
        String endpoints = "yourEndpoints";
        // 创建实例时，如果开启了ACL才需要添加以下代码。
        String accessKey = System.getenv("ACL_ACCESS_KEY");
        String secretKey = System.getenv("ACL_SECRET_KEY");
        //ACL_ACCESS_KEY为用户名，ACL_SECRET_KEY为用户的密钥。创建用户的步骤，请参见创建用户。用户名和密钥直接硬编码到代码中或者明文存储都存在很大的风险，建议在配置文件或者环境变量中密文存放，使用时解密。
        SessionCredentialsProvider sessionCredentialsProvider =
                new StaticSessionCredentialsProvider(accessKey, secretKey);

        ClientConfiguration clientConfiguration = ClientConfiguration.newBuilder()
                .setEndpoints(endpoints)
                // .enableSsl(false)  // 创建实例时，如果将“SSL”配置为“PLAINTEXT”，则请增加此行代码。如果将“SSL”配置为“PERMISSIVE”，则请根据实际情况选择是否增加此行代码。
                // .setCredentialProvider(sessionCredentialsProvider)  // 创建实例时，如果开启了ACL，请添加此行代码。
                .build();
        final Producer producer = provider.newProducerBuilder()
                .setClientConfiguration(clientConfiguration)
                .setTopics(topic)
                .build();

        byte[] body = "This is a delay message for Apache RocketMQ".getBytes(StandardCharsets.UTF_8);
        String tag = "yourMessageTagA";
        Duration messageDelayTime = Duration.ofSeconds(10);
        final Message message = provider.newMessageBuilder()
                .setTopic(topic)
                .setTag(tag)
                .setKeys("yourMessageKey")
                // 设置定时消息投递时间戳
                .setDeliveryTimestamp(System.currentTimeMillis() + messageDelayTime.toMillis())
                .setBody(body)
                .build();
        try {
            final SendReceipt sendReceipt = producer.send(message);
            log.info("Send message successfully, messageId={}", sendReceipt.getMessageId());
        } catch (Throwable t) {
            log.error("Failed to send message", t);
        }

        // 不再使用后，手动关闭producer。
        producer.close();
    }
}
```

5 Go（TCP 协议）

5.1 收发普通消息

本章节介绍普通消息的收发方法和示例代码。其中，普通消息发送方式分为同步发送和异步发送。

- 同步发送：消息发送方发出一条消息到服务端，服务端接收并处理消息，然后返回响应给发送方，发送方收到响应后才会发送下一条消息。
- 异步发送：消息发送方发出一条消息后，不等服务端返回响应，接着发送下一条消息。

收发普通消息前，请参考**收集连接信息**收集RocketMQ所需的连接信息。

准备环境

1. 执行以下命令，检查是否已安装Go。
   ```
   go version
   ```
   返回如下回显时，说明Go已经安装。
   ```
   go version go1.16.5 linux/amd64
   ```
   如果未安装Go，请**下载并安装**。
2. 进入Go脚本所在的bin目录下。
3. 执行“touch go.mod”命令新建一个“go.mod”，并增加以下代码，添加依赖。
   ```
   module rocketmq-example-go

   go 1.13

   require (
       github.com/apache/rocketmq-client-go/v2 v2.1.2
   )
   ```
4. 执行如下命令增加代理。
   ```
   export GOPROXY=https://goproxy.cn,direct
   ```
5. 执行如下命令下载依赖。
   ```
   go mod tidy
   ```

同步发送

同步发送是指消息发送方发出一条消息到服务端，服务端接收并处理消息，然后返回响应给发送方，发送方收到响应后才会发送下一条消息的通讯方式。

参考如下示例代码（以下加粗内容需要替换为实例自有信息，请根据实际情况替换）。

```
package main

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

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

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

示例代码中的参数说明如下，请参考**收集连接信息**获取参数值。

- 192.168.0.1:8100：表示实例连接地址和端口。
- topic1：表示Topic名称。

异步发送

异步发送是指消息发送方发出一条消息后，不等服务端返回响应，接着发送下一条消息的通讯方式。

使用异步发送需要客户端实现异步发送回调接口（SendCallback）。即消息发送方在发送了一条消息后，不需要等待服务端响应接着发送第二条消息。发送方通过回调接口接收服务端响应，并处理响应结果。

参考如下示例代码（以下加粗内容需要替换为实例自有信息，请根据实际情况替换）。

```
package main

import (
    "context"
    "fmt"
```

```
    "os"
    "sync"

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

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

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

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

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

示例代码中的参数说明如下，请参考**收集连接信息**获取参数值。

- 192.168.0.1:8100：表示实例连接地址和端口。
- test：表示Topic名称。

订阅普通消息

参考如下示例代码（以下加粗内容需要替换为实例自有信息，请根据实际情况替换）。

```
package main

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

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

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

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

示例代码中的参数说明如下，请参考**收集连接信息**获取参数值。

- testGroup：表示消费组名称。
- 192.168.0.1:8100：表示实例连接地址和端口。
- test：表示Topic名称。

5.2 收发顺序消息

顺序消息是分布式消息服务RocketMQ版提供的一种严格按照顺序来发布和消费的消息类型。

顺序消息分为全局顺序消息和分区顺序消息：

- 全局顺序消息：对于指定的一个Topic，将队列数量设置为1，这个队列内所有消息按照严格的先入先出FIFO（First In First Out）的顺序进行发布和订阅。
- 分区顺序消息：对于指定的一个Topic，同一个队列内的消息按照严格的FIFO顺序进行发布和订阅。生产者指定分区选择算法，保证需要按顺序消费的消息被分配到同一个队列。

全局顺序消息和分区顺序消息的区别仅为队列数量不同，代码没有区别。

收发顺序消息前，请参考**收集连接信息**收集RocketMQ所需的连接信息。

准备环境

1. 执行以下命令，检查是否已安装Go。
   ```
   go version
   ```
   返回如下回显时，说明Go已经安装。
   ```
   go version go1.16.5 linux/amd64
   ```
   如果未安装Go，请**下载并安装**。

2. 进入Go脚本所在的bin目录下。
3. 执行“touch go.mod”命令新建一个“go.mod”，并增加以下代码，添加依赖。

```
module rocketmq-example-go

go 1.13

require (
    github.com/apache/rocketmq-client-go/v2 v2.1.2
)
```

4. 执行如下命令增加代理。

```
export GOPROXY=https://goproxy.cn,direct
```

5. 执行如下命令下载依赖。

```
go mod tidy
```

发送顺序消息

参考如下示例代码（以下加粗内容需要替换为实例自有信息，请根据实际情况替换）。

```
package main

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

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

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

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

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

示例代码中的参数说明如下，请参考**收集连接信息**获取参数值。

- 192.168.0.1:8100：表示实例连接地址和端口。
- test：表示Topic名称。

上述代码中，相同orderId的消息需要保证顺序，不同orderId的消息不需要保证顺序，所以将orderId作为选择队列的sharding key。

订阅顺序消息

只需要在订阅普通消息的代码基础上增加consumer.WithConsumerOrder(true)，参考如下示例代码（以下加粗内容需要替换为实例自有信息，请根据实际情况替换）。

```
package main

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

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

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

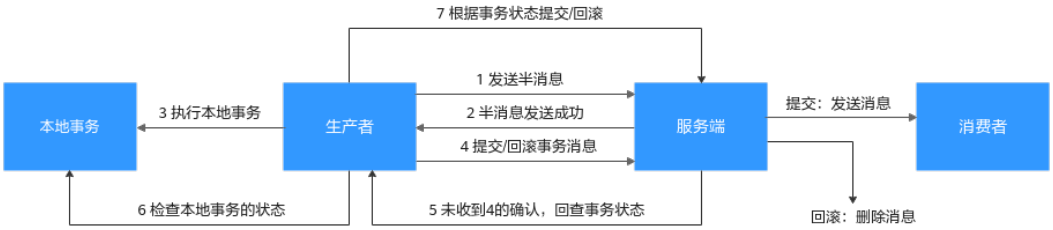
示例代码中的参数说明如下，请参考**收集连接信息**获取参数值。

- testGroup：表示消费组名称。
- 192.168.0.1:8100：表示实例连接地址和端口。
- test：表示Topic名称。

5.3 收发事务消息

分布式消息服务RocketMQ版的事务消息支持在业务逻辑与发送消息之间提供事务保证，通过两阶段的方式提供对事务消息的支持，事务消息交互流程如图5-1所示。

图 5-1 事务消息交互流程



事务消息生产者首先发送半消息，然后执行本地事务。如果执行成功，则发送事务提交，否则发送事务回滚。服务端在一段时间后如果一直收不到提交或回滚，则发起回查，生产者在收到回查后重新发送事务提交或回滚。消息只有在提交之后才投递给消费者，消费者对回滚的消息不可见。

收发事务消息前，请参考**收集连接信息**收集RocketMQ所需的连接信息。

准备环境

1. 执行以下命令，检查是否已安装Go。
   ```
   go version
   ```
   返回如下回显时，说明Go已经安装。
   ```
   go version go1.16.5 linux/amd64
   ```
   如果未安装Go，请**下载并安装**。
2. 进入Go脚本所在的bin目录下。
3. 执行“touch go.mod”命令新建一个“go.mod”，并增加以下代码，添加依赖。
   ```
   module rocketmq-example-go

   go 1.13

   require (
       github.com/apache/rocketmq-client-go/v2 v2.1.2
   )
   ```
4. 执行如下命令增加代理。
   ```
   export GOPROXY=https://goproxy.cn,direct
   ```
5. 执行如下命令下载依赖。
   ```
   go mod tidy
   ```

发送事务消息

参考如下示例代码（以下加粗内容需要替换为实例自有信息，请根据实际情况替换）。

```
package main

import (
    "context"
    "fmt"
```

```
    "os"
    "strconv"
    "sync"
    "sync/atomic"
    "time"

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

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

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

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

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

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

func main() {
    p, _ := rocketmq.NewTransactionProducer(
        NewDemoListener(),
        producer.WithNsResolver(primitive.NewPassthroughResolver([]string{"192.168.0.1:8100"})),
        producer.WithRetry(1),
    )
    err := p.Start()
    if err != nil {
        fmt.Printf("start producer error: %s\n", err.Error())
        os.Exit(1)
    }
```

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

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

示例代码中的参数说明如下，请参考**收集连接信息**获取参数值。

- 192.168.0.1:8100：表示实例连接地址和端口。
- topic1：表示Topic名称。

事务消息生产者需要实现两个回调函数，其中ExecuteLocalTransaction回调函数在发送完半事务消息后被调用，即上图中的第3阶段，CheckLocalTransaction回调函数在收到回查时调用，即上图中的第6阶段。两个回调函数均可返回3种事务状态：

- primitive.CommitMessageState：提交事务，允许消费者消费该消息。
- primitive.RollbackMessageState：回滚事务，消息将被丢弃不允许消费。
- primitive.UnknowState：无法判断状态，期待服务端向生产者再次回查该消息的状态。

订阅事务消息

订阅事务消息的代码与**订阅普通消息的代码**相同。

5.4 发送定时消息

分布式消息服务RocketMQ版支持**任意时间**的定时消息，最大推迟时间可达到1年。

定时消息即生产者生产消息到分布式消息服务RocketMQ版后，消息不会立即被消费，而是延迟到设定的时间点后才会发送给消费者进行消费。

发送定时消息前，请参考**收集连接信息**收集RocketMQ所需的连接信息。

适用场景

定时消息适用于以下场景：

- 消息对应的业务逻辑有时间窗口要求，如电商交易中超时未支付关闭订单的场景。在订单创建时发送一条定时消息，5分钟以后投递给消费者，消费者收到此消息后需要判断对应订单是否完成支付，如果未完成支付，则关闭订单。如果已完成，则忽略。
- 通过消息触发定时任务的场景，如在某些固定时间点向用户发送提醒消息。

注意事项

- 定时消息的最大延迟时间为1年，延迟超过1年的消息将会发送失败。

- 定时消息的定时时间如果被设置成当前时间戳之前的某个时刻，消息将立刻投递给消费者。
- 在理想情况下，定时消息设定的时间与实际发送时间的误差在0.1s以内。但在定时消息投递压力过大时，会触发定时消息投递流控机制，精度会变差。
- 在0.1s的精度内，不保证消息投递的顺序性。即如果两条定时消息的定时时间差距小于0.1s，他们投递的顺序与他们发送的顺序无法确保一致。
- 无法确保定时消息仅投递一次，定时消息可能会重复投递。
- 定时消息的定时时间是服务端开始向消费端投递的时间。如果消费者当前有消息堆积，那么定时消息会排在堆积消息后面，将不能严格按照配置的时间进行投递。
- 由于客户端和服务端可能存在时间差，消息的实际投递时间与客户端设置的投递时间之间可能存在偏差，以服务端时间为准。
- 设置定时消息的投递时间后，依然受消息老化时间限制，默认消息老化时间为2天。例如，设置定时消息5天后才能被消费，如果第5天后一直没被消费，那么这条消息将在第7天被删除。
- 定时消息将占用普通消息约3倍的存储空间，大量使用定时消息时需要注意存储空间占用。

准备环境

1. 执行以下命令，检查是否已安装Go。
   ```
   go version
   ```
   返回如下回显时，说明Go已经安装。
   ```
   go version go1.16.5 linux/amd64
   ```
   如果未安装Go，请**下载并安装**。
2. 进入Go脚本所在的bin目录下。
3. 执行“touch go.mod”命令新建一个“go.mod”，并增加以下代码，添加依赖。
   ```
   module rocketmq-example-go

   go 1.13

   require (
       github.com/apache/rocketmq-client-go/v2 v2.1.2
   )
   ```
4. 执行如下命令增加代理。
   ```
   export GOPROXY=https://goproxy.cn,direct
   ```
5. 执行如下命令下载依赖。
   ```
   go mod tidy
   ```

发送定时消息

发送定时消息的示例代码如下（以下加粗内容需要替换为实例自有信息，请根据实际情况替换）。

```
package main

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

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

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

示例代码中的参数说明如下，请参考**收集连接信息**获取参数值。

- 192.168.0.1:8100：表示实例连接地址和端口。
- test：表示Topic名称。

5.5 使用 ACL 权限访问

实例开启ACL访问控制后，消息生产者和消费者都需要增加用户认证信息。

生产者增加用户认证信息

- 普通消息、顺序消息和定时消息，参考如下代码。以下加粗内容需要替换为实例自有信息，请根据实际情况替换。

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

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

func main() {
    p, err := rocketmq.NewProducer(
        producer.WithNsResolver(primitive.NewPassthroughResolver([]string{"192.168.0.1:8100"})),
        producer.WithRetry(2),
        producer.WithCredentials(primitive.Credentials{
            AccessKey: os.Getenv("ACL_ACCESS_KEY"), //用户名和密钥直接硬编码到代码中或者明文存储都存在很大的风险，建议在配置文件或者环境变量中密文存放,使用时解密。
            SecretKey: os.Getenv("ACL_SECRET_KEY"),
        }),
    )

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

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

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

示例代码中的参数说明如下，请参考**收集连接信息**获取参数值。

- 192.168.0.1:8100：表示实例连接地址和端口。
- AccessKey：表示用户名。创建用户的步骤，请参见**创建用户**。
- SecretKey：表示用户的密钥。
- test：表示Topic名称。

- 事务消息，参考如下代码。以下加粗内容需要替换为实例自有信息，请根据实际情况替换。

```
package main

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

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

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

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

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

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

func (dl *DemoListener) CheckLocalTransaction(msg *primitive.MessageExt)
primitive.LocalTransactionState {
```

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

func main() {
    p, _ := rocketmq.NewTransactionProducer(
        NewDemoListener(),
        producer.WithNsResolver(primitive.NewPassthroughResolver([]string{"192.168.0.1:8100"})),
        producer.WithRetry(2),
        producer.WithCredentials(primitive.Credentials{
            AccessKey: os.Getenv("ACL_ACCESS_KEY"), //用户名和密钥直接硬编码到代码中或者明文存储都存在很大的风险，建议在配置文件或者环境变量中密文存放,使用时解密。
            SecretKey: os.Getenv("ACL_SECRET_KEY"),
        }),
    )
    err := p.Start()
    if err != nil {
        fmt.Printf("start producer error: %s\n", err.Error())
        os.Exit(1)
    }

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

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

示例代码中的参数说明如下，请参考**收集连接信息**获取参数值。

- 192.168.0.1:8100：表示实例连接地址和端口。
- AccessKey：表示用户名。创建用户的步骤，请参见**创建用户**。
- SecretKey：表示用户的密钥。
- topic1：表示Topic名称。

消费者增加用户认证信息

无论是普通消息、顺序消息、定时消息，还是事务消息，都参考如下代码。以下加粗内容需要替换为实例自有信息，请根据实际情况替换。

```
package main

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

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

func main() {
    c, err := rocketmq.NewPushConsumer(
        consumer.WithGroupName("testGroup"),
        consumer.WithNsResolver(primitive.NewPassthroughResolver([]string{"192.168.0.1:8100"})),
        consumer.WithCredentials(primitive.Credentials{
            AccessKey: os.Getenv("ACL_ACCESS_KEY"), //用户名和密钥直接硬编码到代码中或者明文存储都存在很大的风险，建议在配置文件或者环境变量中密文存放,使用时解密。
            SecretKey: os.Getenv("ACL_SECRET_KEY"),
        }),
    )
    if err != nil {
        fmt.Println("init consumer error: " + err.Error())
        os.Exit(0)
    }

    err = c.Subscribe("test", consumer.MessageSelector{}, func(ctx context.Context,
        msgs ...*primitive.MessageExt) (consumer.ConsumeResult, error) {
        fmt.Printf("subscribe callback: %v \n", msgs)
        return consumer.ConsumeSuccess, nil
    })
    if err != nil {
        fmt.Println(err.Error())
    }
    // Note: start after subscribe
    err = c.Start()
    if err != nil {
        fmt.Println(err.Error())
        os.Exit(-1)
    }
    time.Sleep(time.Hour)
    err = c.Shutdown()
    if err != nil {
        fmt.Printf("Shutdown Consumer error: %s", err.Error())
    }
}
```

示例代码中的参数说明如下，请参考**收集连接信息**获取参数值。

- testGroup：表示消费组名称。
- 192.168.0.1:8100：表示实例连接地址和端口。
- AccessKey：表示用户名。创建用户的步骤，请参见**创建用户**。
- SecretKey：表示用户的密钥。
- test：表示Topic名称。

6 Go（gRPC 协议）

6.1 收发普通消息

本章节介绍普通消息的收发方法和示例代码。普通消息发送方式分为同步发送和异步发送。

- 同步发送：消息发送方发出一条消息到服务端，服务端接收并处理消息，然后返回响应给发送方，发送方收到响应后才会发送下一条消息。
- 异步发送：消息发送方发出一条消息后，不等服务端返回响应，接着发送下一条消息。

收发消息前，请参考**收集连接信息**收集RocketMQ所需的连接信息。

约束与限制

- 仅RocketMQ实例5.x版本支持gRPC协议，4.8.0版本不支持。
- 客户端连接RocketMQ实例5.x版本收发普通消息前，需要确保Topic的消息类型为“普通”。

准备环境

1. 执行以下命令，检查是否已安装Go。
   ```
   go version
   ```
   返回如下回显时，说明Go已经安装。
   ```
   go version go1.16.5 linux/amd64
   ```
   如果未安装Go，请**下载并安装**。
2. 在“go.mod”中增加以下代码，添加依赖。
   ```
   module rocketmq-example-go

   go 1.13

   require (
       github.com/apache/rocketmq-clients/golang/v5
   )
   ```

同步发送

同步发送是指消息发送方发出一条消息到服务端，服务端接收并处理消息，然后返回响应给发送方，发送方收到响应后才会发送下一条消息的通讯方式。

参考如下示例代码（以下加粗内容需要替换为实例自有信息，请根据实际情况替换）。

```
package main

import (
    "context"
    "fmt"
    "log"
    "os"
    "strconv"
    "time"

    "github.com/apache/rocketmq-clients/golang"
    "github.com/apache/rocketmq-clients/golang/credentials"
)

const (
    Topic     = "topic01"
    Endpoint  = "192.168.xx.xx:8080"
    AccessKey = os.Getenv("ACL_ACCESS_KEY")
    SecretKey = os.Getenv("ACL_SECRET_KEY")
)
//ACL_ACCESS_KEY为用户名，ACL_SECRET_KEY为用户的密钥。创建用户的步骤，请参见创建用户。用户名和密钥直接硬编码到代码中或者明文存储都存在很大的风险，建议在配置文件或者环境变量中密文存放，使用时解密。
func main() {
    os.Setenv("mq.consoleAppender.enabled", "true")
    golang.ResetLogger()
    producer, err := golang.NewProducer(&golang.Config{
        Endpoint: Endpoint,
        Credentials: &credentials.SessionCredentials{
            AccessKey:    AccessKey,
            AccessSecret: SecretKey,
        },
    },
        golang.WithTopics(Topic),
    )
    if err != nil {
        log.Fatal(err)
    }
    err = producer.Start()
    if err != nil {
        log.Fatal(err)
    }
    defer producer.GracefulStop()

    for i := 0; i < 10; i++ {
        msg := &golang.Message{
            Topic: Topic,
            Body:  []byte("this is a message : " + strconv.Itoa(i)),
        }
        // 设置消息的Key和Tag
        msg.SetKeys("a", "b")
        msg.SetTag("ab")
        resp, err := producer.Send(context.TODO(), msg)
        if err != nil {
            log.Fatal(err)
        }
        for i := 0; i < len(resp); i++ {
            fmt.Printf("%#v\n", resp[i])
        }

        time.Sleep(time.Second * 1)
```

```
    }
}
```

示例代码中的参数说明如下，请参考**收集连接信息**获取参数值。

- Topic：输入Topic名称。
- Endpoint：输入grpc连接地址/grpc公网连接地址。
- AccessKey：创建实例时，如果开启了ACL，需要输入用户名。
- SecretKey：创建实例时，如果开启了ACL，需要输入用户密钥。
- SetKeys：输入消息的Key。
- SetTag：输入消息的Tag。

异步发送

异步发送是指消息发送方发出一条消息后，不等服务端返回响应，接着发送下一条消息的通讯方式。

使用异步发送需要客户端实现异步发送回调接口（SendCallback）。即消息发送方在发送了一条消息后，不需要等待服务端响应接着发送第二条消息。发送方通过回调接口接收服务端响应，并处理响应结果。

参考如下示例代码（以下加粗内容需要替换为实例自有信息，请根据实际情况替换）。

```
package main

import (
    "context"
    "fmt"
    "log"
    "os"
    "strconv"
    "time"

    "github.com/apache/rocketmq-clients/golang"
    "github.com/apache/rocketmq-clients/golang/credentials"
)

const (
    Topic     = "topic01"
    Endpoint  = "192.168.xx.xx:8080"
    AccessKey = os.Getenv("ACL_ACCESS_KEY")
    SecretKey = os.Getenv("ACL_SECRET_KEY")
)
//ACL_ACCESS_KEY为用户名，ACL_SECRET_KEY为用户的密钥。创建用户的步骤，请参见创建用户。用户名和密钥直接硬编码到代码中或者明文存储都存在很大的风险，建议在配置文件或者环境变量中密文存放，使用时解密。
func main() {
    os.Setenv("mq.consoleAppender.enabled", "true")
    golang.ResetLogger()
    producer, err := golang.NewProducer(&golang.Config{
        Endpoint: Endpoint,
        Credentials: &credentials.SessionCredentials{
            AccessKey:    AccessKey,
            AccessSecret: SecretKey,
        },
    },
        golang.WithTopics(Topic),
    )
    if err != nil {
        log.Fatal(err)
    }
    err = producer.Start()
```

```
    if err != nil {
        log.Fatal(err)
    }
    defer producer.GracefulStop()
    for i := 0; i < 10; i++ {
        msg := &golang.Message{
            Topic: Topic,
            Body:  []byte("this is a message : " + strconv.Itoa(i)),
        }
        // 设置消息的Key和Tag
        msg.SetKeys("a", "b")
        msg.SetTag("ab")
        producer.SendAsync(context.TODO(), msg, func(ctx context.Context, resp []*golang.SendReceipt, err
error) {
            if err != nil {
                log.Fatal(err)
            }
            for i := 0; i < len(resp); i++ {
                fmt.Printf("%#v\n", resp[i])
            }
        })

        time.Sleep(time.Second * 1)
    }
}
```

示例代码中的参数说明如下，请参考**收集连接信息**获取参数值。

- Topic：输入Topic名称。
- Endpoint：输入grpc连接地址/grpc公网连接地址。
- AccessKey：创建实例时，如果开启了ACL，需要输入用户名。
- SecretKey：创建实例时，如果开启了ACL，需要输入用户密钥。
- SetKeys：输入消息的Key。
- SetTag：输入消息的Tag。

订阅普通消息

参考如下示例代码（以下加粗内容需要替换为实例自有信息，请根据实际情况替换）。

```
package main

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

    "github.com/apache/rocketmq-clients/golang"
    "github.com/apache/rocketmq-clients/golang/credentials"
)

const (
    Topic     = "topic01"
    GroupName = "groupname"
    Endpoint  = "192.168.xx.xx:8080"
    AccessKey = os.Getenv("ACL_ACCESS_KEY")
    SecretKey = os.Getenv("ACL_SECRET_KEY")
)
//ACL_ACCESS_KEY为用户名，ACL_SECRET_KEY为用户的密钥。创建用户的步骤，请参见创建用户。用户名和密钥直接硬编码到代码中或者明文存储都存在很大的风险，建议在配置文件或者环境变量中密文存放，使用时解密。
var (
```

```
    // 接收消息请求的最大等待时间
    awaitDuration = time.Second * 5
    // 每次能接收的最大消息数
    maxMessageNum int32 = 16
    // 消息不可见时间，在消息被接收后对其他消费者不可见，直到超时。
    invisibleDuration = time.Second * 20
)

func main() {
    os.Setenv("mq.consoleAppender.enabled", "true")
    golang.ResetLogger()
    simpleConsumer, err := golang.NewSimpleConsumer(&golang.Config{
        Endpoint: Endpoint,
        Group:    GroupName,
        Credentials: &credentials.SessionCredentials{
            AccessKey:    AccessKey,
            AccessSecret: SecretKey,
        },
    },
        golang.WithAwaitDuration(awaitDuration),
        golang.WithSubscriptionExpressions(map[string]*golang.FilterExpression{
            Topic: golang.SUB_ALL,
        }),
    )
    if err != nil {
        log.Fatal(err)
    }
    err = simpleConsumer.Start()
    if err != nil {
        log.Fatal(err)
    }
    defer simpleConsumer.GracefulStop()

    go func() {
        for {
            fmt.Println("start recevie message")
            mvs, err := simpleConsumer.Receive(context.TODO(), maxMessageNum, invisibleDuration)
            if err != nil {
                fmt.Println(err)
            }
            for _, mv := range mvs {
                simpleConsumer.Ack(context.TODO(), mv)
                fmt.Println(mv)
            }
            fmt.Println("wait a moment")
            fmt.Println()
            time.Sleep(time.Second * 3)
        }
    }()

    time.Sleep(time.Minute)
}
```

示例代码中的参数说明如下，请参考**收集连接信息**获取参数值。

- Topic：输入Topic名称。
- GroupName：输入消费组名称。
- Endpoint：输入grpc连接地址/grpc公网连接地址。
- AccessKey：创建实例时，如果开启了ACL，需要输入用户名。
- SecretKey：创建实例时，如果开启了ACL，需要输入用户密钥。

6.2 收发顺序消息

顺序消息是分布式消息服务RocketMQ版提供的一种严格按照顺序来发布和消费的消息类型。

顺序消息分为全局顺序消息和分区顺序消息：

- 全局顺序消息：对于指定的一个Topic，将队列数量设置为1，这个队列内所有消息按照严格的先入先出FIFO（First In First Out）的顺序进行发布和订阅。
- 分区顺序消息：对于指定的一个Topic，同一个队列内的消息按照严格的FIFO顺序进行发布和订阅。生产者为每一条消息指定消息组，相同消息组的消息会被分配到同一个队列。

全局顺序消息和分区顺序消息的区别仅为队列数量不同，代码没有区别。

收发顺序消息前，请参考**收集连接信息**收集RocketMQ所需的连接信息。

约束与限制

- 仅RocketMQ实例5.x版本支持gRPC协议，4.8.0版本不支持。
- 客户端连接RocketMQ实例5.x版本收发顺序消息前，需要确保Topic的消息类型为“顺序”。
- 使用gRPC协议连接RocketMQ实例时，消费者是否顺序消费消息，取决于消费组中是否开启顺序消费，并非在消费代码中设置，顺序消费消息的代码与普通消费的代码相同。

准备环境

1. 执行以下命令，检查是否已安装Go。
   ```
   go version
   ```
   返回如下回显时，说明Go已经安装。
   ```
   go version go1.16.5 linux/amd64
   ```
   如果未安装Go，请**下载并安装**。
2. 在“go.mod”中增加以下代码，添加依赖。
   ```
   module rocketmq-example-go

   go 1.13

   require (
       github.com/apache/rocketmq-clients/golang/v5
   )
   ```

发送顺序消息

参考如下示例代码（以下加粗内容需要替换为实例自有信息，请根据实际情况替换）。

```
package main

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

```
    "time"

    "github.com/apache/rocketmq-clients/golang"
    "github.com/apache/rocketmq-clients/golang/credentials"
)

const (
    Topic     = "topic01"
    Endpoint  = "192.168.xx.xx:8080"
    AccessKey = os.Getenv("ACL_ACCESS_KEY")
    SecretKey = os.Getenv("ACL_SECRET_KEY")
)
//ACL_ACCESS_KEY为用户名，ACL_SECRET_KEY为用户的密钥。创建用户的步骤，请参见创建用户。用户名和密钥直接硬编码到代码中或者明文存储都存在很大的风险，建议在配置文件或者环境变量中密文存放，使用时解密。
func main() {
    os.Setenv("mq.consoleAppender.enabled", "true")
    golang.ResetLogger()
    producer, err := golang.NewProducer(&golang.Config{
        Endpoint: Endpoint,
        Credentials: &credentials.SessionCredentials{
            AccessKey:    AccessKey,
            AccessSecret: SecretKey,
        },
    },
        golang.WithTopics(Topic),
    )
    if err != nil {
        log.Fatal(err)
    }
    err = producer.Start()
    if err != nil {
        log.Fatal(err)
    }
    defer producer.GracefulStop()
    for i := 0; i < 10; i++ {
        msg := &golang.Message{
            Topic: Topic,
            Body:  []byte("this is a message : " + strconv.Itoa(i)),
        }
        // 设置消息的Key和Tag
        msg.SetKeys("a", "b")
        msg.SetTag("ab")
        msg.SetMessageGroup("yourMessageGroup0")
        resp, err := producer.Send(context.TODO(), msg)
        if err != nil {
            log.Fatal(err)
        }
        for i := 0; i < len(resp); i++ {
            fmt.Printf("%#v\n", resp[i])
        }

        time.Sleep(time.Second * 1)
    }
}
```

示例代码中的参数说明如下，请参考**收集连接信息**获取参数值。

- Topic：输入Topic名称。
- Endpoint：输入grpc连接地址/grpc公网连接地址。
- AccessKey：创建实例时，如果开启了ACL，需要输入用户名。
- SecretKey：创建实例时，如果开启了ACL，需要输入用户密钥。
- SetKeys：输入消息的Key。
- SetTag：输入消息的Tag。

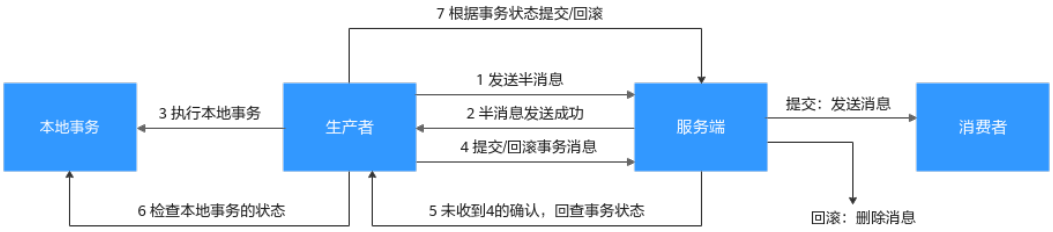
订阅顺序消息

订阅顺序消息的代码与**订阅普通消息**相同。

6.3 收发事务消息

分布式消息服务RocketMQ版的事务消息支持在业务逻辑与发送消息之间提供事务保证，通过两阶段的方式提供对事务消息的支持，事务消息交互流程如**图6-1**所示。

图 6-1 事务消息交互流程



事务消息生产者首先发送半消息，然后执行本地事务。如果执行成功，则发送事务提交，否则发送事务回滚。服务端在一段时间后如果一直收不到提交或回滚，则发起回查，生产者在收到回查后重新发送事务提交或回滚。消息只有在提交之后才投递给消费者，消费者对回滚的消息不可见。

收发事务消息前，请参考**收集连接信息**收集RocketMQ所需的连接信息。

约束与限制

- 仅RocketMQ实例5.x版本支持gRPC协议，4.8.0版本不支持。
- 客户端连接RocketMQ实例5.x版本收发事务消息前，需要确保Topic的消息类型为“事务”。

准备环境

1. 执行以下命令，检查是否已安装Go。
   ```
   go version
   ```
   返回如下回显时，说明Go已经安装。
   ```
   go version go1.16.5 linux/amd64
   ```
   如果未安装Go，请**下载并安装**。
2. 在“go.mod”中增加以下代码，添加依赖。
   ```
   module rocketmq-example-go

   go 1.13

   require (
       github.com/apache/rocketmq-clients/golang/v5
   )
   ```

发送事务消息

参考如下示例代码（以下加粗内容需要替换为实例自有信息，请根据实际情况替换）。

```
package main

import (
    "context"
    "fmt"
    "log"
    "os"
    "strconv"
    "time"

    "github.com/apache/rocketmq-clients/golang"
    "github.com/apache/rocketmq-clients/golang/credentials"
)

const (
    Topic     = "topic01"
    Endpoint  = "192.168.xx.xx:8080"
    AccessKey = os.Getenv("ACL_ACCESS_KEY")
    SecretKey = os.Getenv("ACL_SECRET_KEY")
)
//ACL_ACCESS_KEY为用户名，ACL_SECRET_KEY为用户的密钥。创建用户的步骤，请参见创建用户。用户名和密钥直接硬编码到代码中或者明文存储都存在很大的风险，建议在配置文件或者环境变量中密文存放，使用时解密。
func main() {
    os.Setenv("mq.consoleAppender.enabled", "true")
    golang.ResetLogger()
    producer, err := golang.NewProducer(&golang.Config{
        Endpoint: Endpoint,
        Credentials: &credentials.SessionCredentials{
            AccessKey:    AccessKey,
            AccessSecret: SecretKey,
        },
    },
        golang.WithTransactionChecker(&golang.TransactionChecker{
            Check: func(msg *golang.MessageView) golang.TransactionResolution {
                log.Printf("check transaction message: %v", msg)
                // 检查本地事务并返回本地事务状态
                return golang.COMMIT
            },
        }),
        golang.WithTopics(Topic),
    )
    if err != nil {
        log.Fatal(err)
    }
    err = producer.Start()
    if err != nil {
        log.Fatal(err)
    }
    defer producer.GracefulStop()
    for i := 0; i < 10; i++ {
        msg := &golang.Message{
            Topic: Topic,
            Body:  []byte("this is a message : " + strconv.Itoa(i)),
        }
        // 设置消息的Key和Tag
        msg.SetKeys("a", "b")
        msg.SetTag("ab")
        // 开启事务分支
        transaction := producer.BeginTransaction()
        resp, err := producer.SendWithTransaction(context.TODO(), msg, transaction)
        if err != nil {
            log.Fatal(err)
        }
        for i := 0; i < len(resp); i++ {
            fmt.Printf("%#v\n", resp[i])
        }
        /**
         * 执行本地事务，并确定本地事务结果。
```

```
        * 1. 如果本地事务提交成功，则提交消息事务。
        * 2. 如果本地事务提交失败，则回滚消息事务。
        * 3. 如果本地事务未知异常，则不处理，等待事务消息回查。
        *
        */
       err = transaction.Commit()
       if err != nil {
          log.Fatal(err)
       }

       time.Sleep(time.Second * 1)
    }
}
```

示例代码中的参数说明如下，请参考**收集连接信息**获取参数值。

- Topic：输入Topic名称。
- Endpoint：输入grpc连接地址/grpc公网连接地址。
- AccessKey：创建实例时，如果开启了ACL，需要输入用户名。
- SecretKey：创建实例时，如果开启了ACL，需要输入用户密钥。
- SetKeys：输入消息的Key。
- SetTag：输入消息的Tag。

事务消息需要生产者构建一个事务检查器，用于检查确认异常半事务的中间状态，可以返回3种事务状态：

- TransactionResolution.COMMIT：提交事务，允许消费者消费该消息。
- TransactionResolution.ROLLBACK：回滚事务，消息将被丢弃不允许消费。
- TransactionResolution.UNKNOW：无法判断状态，期待服务端向生产者再次回查该消息的状态。

订阅事务消息

订阅事务消息的代码与**订阅普通消息**相同。

6.4 发送定时消息

分布式消息服务RocketMQ版支持**任意时间**的定时消息，最大推迟时间可达到1年。

定时消息即生产者生产消息到分布式消息服务RocketMQ版后，消息不会立即被消费，而是延迟到设定的时间点后才会发送给消费者进行消费。

发送定时消息前，请参考**收集连接信息**收集RocketMQ所需的连接信息。

约束与限制

- 仅RocketMQ实例5.x版本支持gRPC协议，4.8.0版本不支持。
- 客户端连接RocketMQ实例5.x版本收发定时消息前，需要确保Topic的消息类型为“定时”。

适用场景

定时消息适用于以下场景：

- 消息对应的业务逻辑有时间窗口要求，如电商交易中超时未支付关闭订单的场景。在订单创建时发送一条定时消息，5分钟以后投递给消费者，消费者收到此消息后需要判断对应订单是否完成支付，如果未完成支付，则关闭订单。如果已完成，则忽略。
- 通过消息触发定时任务的场景，如在某些固定时间点向用户发送提醒消息。

注意事项

- 定时消息的最大延迟时间为1年，延迟超过1年的消息将会发送失败。
- 定时消息的定时时间如果被设置成当前时间戳之前的某个时刻，消息将立刻投递给消费者。
- 在理想情况下，定时消息设定的时间与实际发送时间的误差在0.1s以内。但在定时消息投递压力过大时，会触发定时消息投递流控机制，精度会变差。
- 在0.1s的精度内，不保证消息投递的顺序性。即如果两条定时消息的定时时间差距小于0.1s，他们投递的顺序与他们发送的顺序无法确保一致。
- 无法确保定时消息仅投递一次，定时消息可能会重复投递。
- 定时消息的定时时间是服务端开始向消费端投递的时间。如果消费者当前有消息堆积，那么定时消息会排在堆积消息后面，将不能严格按照配置的时间进行投递。
- 由于客户端和服务端可能存在时间差，消息的实际投递时间与客户端设置的投递时间之间可能存在偏差，以服务端时间为准。
- 设置定时消息的投递时间后，依然受消息老化时间限制，默认消息老化时间为2天。例如，设置定时消息5天后才能被消费，如果第5天后一直没被消费，那么这条消息将在第7天被删除。
- 定时消息将占用普通消息约3倍的存储空间，大量使用定时消息时需要注意存储空间占用。

准备环境

开源的Java客户端支持连接分布式消息服务RocketMQ版，推荐使用的客户端版本为**5.0.5**。

使用Maven方式引入依赖。

```
<dependency>
   <groupId>org.apache.rocketmq</groupId>
   <artifactId>rocketmq-client-java</artifactId>
   <version>5.0.5</version>
</dependency>
```

发送定时消息

发送定时消息的示例代码如下（以下加粗内容需要替换为实例自有信息，请根据实际情况替换）。

```
package main

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

```
    "github.com/apache/rocketmq-clients/golang"
    "github.com/apache/rocketmq-clients/golang/credentials"
)

const (
    Topic     = "topic01"
    Endpoint  = "192.168.xx.xx:8080"
    AccessKey = os.Getenv("ACL_ACCESS_KEY")
    SecretKey = os.Getenv("ACL_SECRET_KEY")
)
//ACL_ACCESS_KEY为用户名，ACL_SECRET_KEY为用户的密钥。创建用户的步骤，请参见创建用户。用户名和密钥直接硬编码到代码中或者明文存储都存在很大的风险，建议在配置文件或者环境变量中密文存放，使用时解密。
func main() {
    os.Setenv("mq.consoleAppender.enabled", "true")
    golang.ResetLogger()
    producer, err := golang.NewProducer(&golang.Config{
        Endpoint: Endpoint,
        Credentials: &credentials.SessionCredentials{
            AccessKey:    AccessKey,
            AccessSecret: SecretKey,
        },
    },
        golang.WithTopics(Topic),
    )
    if err != nil {
        log.Fatal(err)
    }
    err = producer.Start()
    if err != nil {
        log.Fatal(err)
    }
    defer producer.GracefulStop()
    for i := 0; i < 10; i++ {
        msg := &golang.Message{
            Topic: Topic,
            Body:  []byte("this is a message : " + strconv.Itoa(i)),
        }
        // 消息的Key和Tag
        msg.SetKeys("a", "b")
        msg.SetTag("ab")
        // 设置定时消息投递时间戳
        msg.SetDelayTimestamp(time.Now().Add(time.Second * 10))
        // send message in sync
        resp, err := producer.Send(context.TODO(), msg)
        if err != nil {
            log.Fatal(err)
        }
        for i := 0; i < len(resp); i++ {
            fmt.Printf("%#v\n", resp[i])
        }

        time.Sleep(time.Second * 1)
    }
}
```

示例代码中的参数说明如下，请参考**收集连接信息**获取参数值。

- Topic：输入Topic名称。
- Endpoint：输入grpc连接地址/grpc公网连接地址。
- AccessKey：创建实例时，如果开启了ACL，需要输入用户名。
- SecretKey：创建实例时，如果开启了ACL，需要输入用户密钥。
- SetKeys：输入消息的Key。
- SetTag：输入消息的Tag。

7 Python（TCP 协议）

7.1 收发普通消息

本章节介绍普通消息的收发方法和示例代码。其中，普通消息发送方式分为同步发送和异步发送。

- 同步发送：消息发送方发出一条消息到服务端，服务端接收并处理消息，然后返回响应给发送方，发送方收到响应后才会发送下一条消息。
- 异步发送：消息发送方发出一条消息后，不等服务端返回响应，接着发送下一条消息。

以下示例仅介绍同步发送的示例代码。

收发普通消息前，请参考**收集连接信息**收集RocketMQ所需的连接信息。

约束与限制

客户端连接RocketMQ实例5.x版本收发普通消息前，需要确保Topic的消息类型为“普通”。

RocketMQ TCP协议的Python SDK只支持Linux系统。

准备环境

1. 在命令行输入**python**，检查是否已安装Python。得到如下回显，说明Python已安装。

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

   如果未安装Python，请使用以下命令安装：

```
yum install python
```

2. 安装librocketmq库和rocketmq-client-python，具体操作请参考**rocketmq-client-python**。

   说明

   建议下载**rocketmq-client-cpp-2.2.0**，获取librocketmq库。

3. 将librocketmq.so添加到系统动态库搜索路径。
   a. 查找librocketmq.so的路径。
   ```
   find / -name librocketmq.so
   ```
   b. 将librocketmq.so添加到系统动态库搜索路径。
   ```
   ln -s /查找到的librocketmq.so路径/librocketmq.so /usr/lib
   sudo ldconfig
   ```

同步发送

同步发送是指消息发送方发出一条消息到服务端，服务端接收并处理消息，然后返回响应给发送方，发送方收到响应后才会发送下一条消息的通讯方式。

参考如下示例代码（以下加粗内容需要替换为实例自有信息，请根据实际情况替换）。

```
from rocketmq.client import Producer, Message

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


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


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


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

示例代码中的参数说明如下，请参考**收集连接信息**获取参数值。

- topic：表示Topic名称。
- gid：表示生产者组名称，请根据业务实际情况输入生产者组名称。
- name_srv：表示实例连接地址和端口。

订阅普通消息

参考如下示例代码（以下加粗内容需要替换为实例自有信息，请根据实际情况替换）。

```
import time

from rocketmq.client import PushConsumer, ConsumeStatus


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

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

    while True:
        time.sleep(3600)


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

示例代码中的参数说明如下，请参考**收集连接信息**获取参数值。

- consumer_group：表示消费组名称。
- 192.168.0.1:8100：表示实例连接地址和端口。
- TopicTest：表示Topic名称。

7.2 收发顺序消息

顺序消息是分布式消息服务RocketMQ版提供的一种严格按照顺序来发布和消费的消息类型。

顺序消息分为全局顺序消息和分区顺序消息：

- 全局顺序消息：对于指定的一个Topic，将队列数量设置为1，这个队列内所有消息按照严格的先入先出FIFO（First In First Out）的顺序进行发布和订阅。
- 分区顺序消息：对于指定的一个Topic，同一个队列内的消息按照严格的FIFO顺序进行发布和订阅。生产者指定分区选择算法，保证需要按顺序消费的消息被分配到同一个队列。

全局顺序消息和分区顺序消息的区别仅为队列数量不同，代码没有区别。

收发顺序消息前，请参考**收集连接信息**收集RocketMQ所需的连接信息。

约束与限制

客户端连接RocketMQ实例5.x版本收发顺序消息前，需要确保Topic的消息类型为“顺序”。

RocketMQ TCP协议的Python SDK只支持Linux系统。

发送顺序消息

参考如下示例代码（以下加粗内容需要替换为实例自有信息，请根据实际情况替换）。

```
from rocketmq.client import Producer, Message

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


def create_message():
    msg = Message(topic)
```

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


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


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

示例代码中的参数说明如下，请参考**收集连接信息**获取参数值。

- topic：表示Topic名称。
- gid：表示生产者组名称，请根据业务实际情况输入生产者组名称。
- name_srv：表示实例连接地址和端口。

上述代码中，相同orderId的消息需要保证顺序，不同orderId的消息不需要保证顺序，所以将orderId作为选择队列的sharding key。

订阅顺序消息

只需要在订阅普通消息的代码基础上增加orderly=True，参考如下示例代码（以下加粗内容需要替换为实例自有信息，请根据实际情况替换）。

```
import time

from rocketmq.client import PushConsumer, ConsumeStatus


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


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

    while True:
        time.sleep(3600)


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

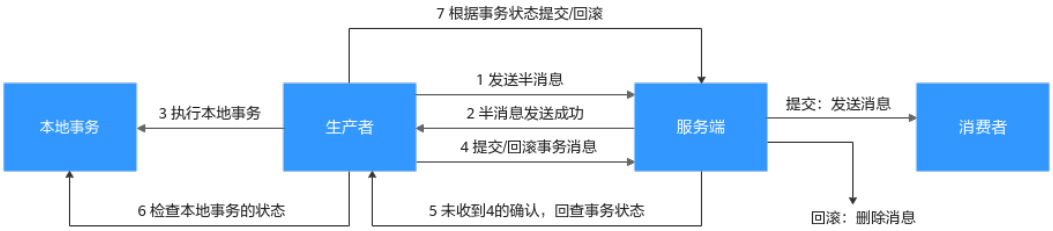
示例代码中的参数说明如下，请参考**收集连接信息**获取参数值。

- consumer_group：表示消费组名称。
- 192.168.0.1:8100：表示实例连接地址和端口。
- TopicTest：表示Topic名称。

7.3 收发事务消息

分布式消息服务RocketMQ版的事务消息支持在业务逻辑与发送消息之间提供事务保证，通过两阶段的方式提供对事务消息的支持，事务消息交互流程如图7-1所示。

图 7-1 事务消息交互流程



事务消息生产者首先发送半消息，然后执行本地事务。如果执行成功，则发送事务提交，否则发送事务回滚。服务端在一段时间后如果一直收不到提交或回滚，则发起回查，生产者在收到回查后重新发送事务提交或回滚。消息只有在提交之后才投递给消费者，消费者对回滚的消息不可见。

收发事务消息前，请参考**收集连接信息**收集RocketMQ所需的连接信息。

约束与限制

客户端连接RocketMQ实例5.x版本收发事务消息前，需要确保Topic的消息类型为“事务”。

RocketMQ TCP协议的Python SDK只支持Linux系统。

发送事务消息

参考如下示例代码（以下加粗内容需要替换为实例自有信息，请根据实际情况替换）。

```
import time

from rocketmq.client import Message, TransactionMQProducer, TransactionStatus

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


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


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

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


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

    while True:
        time.sleep(3600)


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

示例代码中的参数说明如下，请参考**收集连接信息**获取参数值。

- topic：表示Topic名称。
- gid：表示生产者组名称，请根据业务实际情况输入生产者组名称。
- name_srv：表示实例连接地址和端口。

事务消息生产者需要实现两个回调函数，其中local_execute回调函数在发送完半事务消息后被调用，即上图中的第3阶段，check_callback回调函数在收到回查时调用，即上图中的第6阶段。两个回调函数均可返回3种事务状态：

- TransactionStatus.COMMIT：提交事务，允许消费者消费该消息。
- TransactionStatus.ROLLBACK：回滚事务，消息将被丢弃不允许消费。
- TransactionStatus.UNKNOWN：无法判断状态，期待服务端向生产者再次回查该消息的状态。

订阅事务消息

订阅事务消息的代码与**订阅普通消息的代码**相同。

7.4 发送定时消息

分布式消息服务RocketMQ版支持**任意时间**的定时消息，最大推迟时间可达到1年。

定时消息即生产者生产消息到分布式消息服务RocketMQ版后，消息不会立即被消费，而是延迟到设定的时间点后才会发送给消费者进行消费。

发送定时消息前，请参考**收集连接信息**收集RocketMQ所需的连接信息。

约束与限制

- 2022年3月30日及以后购买的实例支持定时消息功能，在此之前购买的实例不支持此功能。
- 客户端连接RocketMQ实例5.x版本收发定时消息前，需要确保Topic的消息类型为“定时”。
- RocketMQ TCP协议的Python SDK只支持Linux系统。

适用场景

定时消息适用于以下场景：

- 消息对应的业务逻辑有时间窗口要求，如电商交易中超时未支付关闭订单的场景。在订单创建时发送一条定时消息，5分钟以后投递给消费者，消费者收到此消息后需要判断对应订单是否完成支付，如果未完成支付，则关闭订单。如果已完成，则忽略。
- 通过消息触发定时任务的场景，如在某些固定时间点向用户发送提醒消息。

注意事项

- 定时消息的最大延迟时间为1年，延迟超过1年的消息将会发送失败。
- 定时消息的定时时间如果被设置成当前时间戳之前的某个时刻，消息将立刻投递给消费者。
- 在理想情况下，定时消息设定的时间与实际发送时间的误差在0.1s以内。但在定时消息投递压力过大时，会触发定时消息投递流控机制，精度会变差。
- 在0.1s的精度内，不保证消息投递的顺序性。即如果两条定时消息的定时时间差距小于0.1s，他们投递的顺序与他们发送的顺序无法确保一致。
- 无法确保定时消息仅投递一次，定时消息可能会重复投递。
- 定时消息的定时时间是服务端开始向消费端投递的时间。如果消费者当前有消息堆积，那么定时消息会排在堆积消息后面，将不能严格按照配置的时间进行投递。
- 由于客户端和服务端可能存在时间差，消息的实际投递时间与客户端设置的投递时间之间可能存在偏差，以服务端时间为准。
- 设置定时消息的投递时间后，依然受消息老化时间限制，默认消息老化时间为2天。例如，设置定时消息5天后才能被消费，如果第5天后一直没被消费，那么这条消息将在第7天被删除。
- 定时消息将占用普通消息约3倍的存储空间，大量使用定时消息时需要注意存储空间占用。

准备环境

1. 在命令行输入**python**，检查是否已安装Python。得到如下回显，说明Python已安装。

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

   如果未安装Python，请使用以下命令安装：

   ```
   yum install python
   ```

2. 安装librocketmq库和rocketmq-client-python，具体操作请参考**rocketmq-client-python**。

   说明

   建议下载**rocketmq-client-cpp-2.2.0**，获取librocketmq库。

3. 将librocketmq.so添加到系统动态库搜索路径。

   a. 查找librocketmq.so的路径。

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

b. 将librocketmq.so添加到系统动态库搜索路径。

```
ln -s /查找到的librocketmq.so路径/librocketmq.so /usr/lib
sudo ldconfig
```

发送定时消息

发送定时消息的示例代码如下（以下加粗内容需要替换为实例自有信息，请根据实际情况替换）。

```
import time

from rocketmq.client import Producer, Message

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


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


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


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

示例代码中的参数说明如下，请参考**收集连接信息**获取参数值。

- topic：表示Topic名称。
- gid：表示生产者组名称，请根据业务实际情况输入生产者组名称。
- name_srv：表示实例连接地址和端口。

7.5 使用 ACL 权限访问

实例开启ACL访问控制后，消息生产者和消费者都需要增加用户认证信息。

约束与限制

RocketMQ TCP协议的Python SDK只支持Linux系统。

生产者增加用户认证信息

- 普通消息、顺序消息和定时消息，参考如下代码。以下加粗内容需要替换为实例自有信息，请根据实际情况替换。

```
from rocketmq.client import Producer, Message
```

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


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


def send_message_sync():
    producer = Producer(gid)
    producer.set_name_server_address(name_srv)
    # 设置权限（角色名和密钥）
    producer.set_session_credentials(
        "ACL_ACCESS_KEY", # 角色名称
        "ACL_SECRET_KEY", # 角色密钥
        ''
    )#用户名和密钥直接硬编码到代码中或者明文存储都存在很大的风险，建议在配置文件或者环境变量中密文存放，使用时解密。

    producer.start()
    msg = create_message()
    ret = producer.send_sync(msg)
    print('send message status: ' + str(ret.status) + ' msgId: ' + ret.msg_id)
    producer.shutdown()


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

示例代码中的参数说明如下，请参考**收集连接信息**获取参数值。

- topic：表示Topic名称。
- gid：表示生产者组名称，请根据业务实际情况输入生产者组名称。
- name_srv：表示实例连接地址和端口。
- ACL_ACCESS_KEY：表示用户名。创建用户的步骤，请参见**创建用户**。
- ACL_SECRET_KEY：表示用户的密钥。

- 事务消息，参考如下代码。以下加粗内容需要替换为实例自有信息，请根据实际情况替换。

```
import time

from rocketmq.client import Message, TransactionMQProducer, TransactionStatus

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


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


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

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


def send_transaction_message(count):
    producer = TransactionMQProducer(gid, check_callback)
    producer.set_name_server_address(name_srv)
    # 设置权限（角色名和密钥）
    producer.set_session_credentials(
        "ACL_ACCESS_KEY", # 角色名称
        "ACL_SECRET_KEY", # 角色密钥
        ''
    )#用户名和密钥直接硬编码到代码中或者明文存储都存在很大的风险，建议在配置文件或者环境变量中密文存放，使用时解密。

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

    while True:
        time.sleep(3600)


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

示例代码中的参数说明如下，请参考**收集连接信息**获取参数值。

- topic：表示Topic名称。
- gid：表示生产者组名称，请根据业务实际情况输入生产者组名称。
- name_srv：表示实例连接地址和端口。
- ACL_ACCESS_KEY：表示用户名。创建用户的步骤，请参见**创建用户**。
- ACL_SECRET_KEY：表示用户的密钥。

消费者增加用户认证信息

无论是普通消息、顺序消息、定时消息，还是事务消息，都参考如下代码。以下加粗内容需要替换为实例自有信息，请根据实际情况替换。

```
import time

from rocketmq.client import PushConsumer, ConsumeStatus


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


def start_consume_message():
    consumer = PushConsumer('consumer_group')
    consumer.set_name_server_address('192.168.0.1:8100')
    # 设置权限（角色名和密钥）
    consumer.set_session_credentials(
        "ACL_ACCESS_KEY", # 角色名称
        "ACL_SECRET_KEY", # 角色密钥
        ''
    )#用户名和密钥直接硬编码到代码中或者明文存储都存在很大的风险，建议在配置文件或者环境变量中密文存放，使用时解密。

    consumer.subscribe('TopicTest', callback)
    print('start consume message')
```

```
    consumer.start()

    while True:
        time.sleep(3600)


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

示例代码中的参数说明如下，请参考**收集连接信息**获取参数值。

- consumer_group：表示消费组名称。
- 192.168.0.1:8100：表示实例连接地址和端口。
- ACL_ACCESS_KEY：表示用户名。创建用户的步骤，请参见**创建用户**。
- ACL_SECRET_KEY：表示用户的密钥。
- TopicTest：表示Topic名称。