RocketMQ事务消费和顺序消费详解

一、RocketMq有3中消息类型

1.普通消费

2. 顺序消费

3.事务消费

- **顺序消费场景**

在网购的时候，我们需要下单，那么下单需要假如有三个顺序，第一、创建订单 ，第二：订单付款，第三：订单完成。也就是这个三个环节要有顺序，这个订单才有意义。RocketMQ可以保证顺序消费。

- rocketMq实现顺序消费的原理

 produce在发送消息的时候，把消息发到同一个队列（queue）中,消费者注册消息监听器为MessageListenerOrderly，这样就可以保证消费端只有一个线程去消费消息

注意：是把把消息发到同一个队列（queue），不是同一个topic，默认情况下一个topic包括4个queue

单个节点（Producer端1个、Consumer端1个）

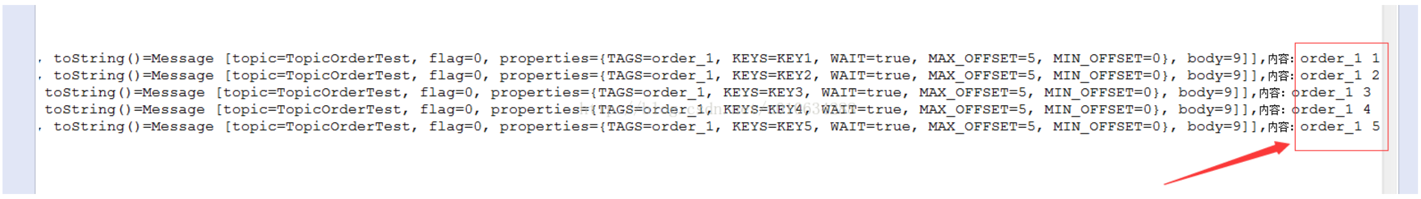
1、Producer.java

```java
package order;

import java.util.List;

import com.alibaba.rocketmq.client.exception.MQBrokerException;
import com.alibaba.rocketmq.client.exception.MQClientException;
import com.alibaba.rocketmq.client.producer.DefaultMQProducer;
import com.alibaba.rocketmq.client.producer.MessageQueueSelector;
import com.alibaba.rocketmq.client.producer.SendResult;
import com.alibaba.rocketmq.common.message.Message;
import com.alibaba.rocketmq.common.message.MessageQueue;
import com.alibaba.rocketmq.remoting.exception.RemotingException;

/**
 * Producer，发送顺序消息
 */
public class Producer {
    public static void main(String[] args) {
        try {
            DefaultMQProducer producer = new DefaultMQProducer("order_Producer");

producer.setNamesrvAddr("192.168.100.145:9876;192.168.100.146:9876;192.168.100.149:9876;192.168.100.239:9876");

            producer.start();

            // String[] tags = new String[] { "TagA", "TagB", "TagC", "TagD",
            // "TagE" };

            for (int i = 1; i <= 5; i++) {

                Message msg = new Message("TopicOrderTest", "order_1", "KEY" + i, ("order_1 " + i).getBytes());

                SendResult sendResult = producer.send(msg, new MessageQueueSelector() {
                    public MessageQueue select(List<MessageQueue> mqs, Message msg, Object arg) {
                        Integer id = (Integer) arg;
                        int index = id % mqs.size();
                        return mqs.get(index);
                    }
                }, 0);

                System.out.println(sendResult);
            }

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

2、Consumer.java

```
package order;
import java.util.List;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicLong;
import com.alibaba.rocketmq.client.consumer.DefaultMQPushConsumer;
import com.alibaba.rocketmq.client.consumer.listener.ConsumeOrderlyContext;
import com.alibaba.rocketmq.client.consumer.listener.ConsumeOrderlyStatus;
import com.alibaba.rocketmq.client.consumer.listener.MessageListenerOrderly;
import com.alibaba.rocketmq.client.exception.MQClientException;
import com.alibaba.rocketmq.common.consumer.ConsumeFromWhere;
import com.alibaba.rocketmq.common.message.MessageExt;

/**
 * 顺序消息消费，带事务方式（应用可控制Offset什么时候提交）
 */
public class Consumer1 {

    public static void main(String[] args) throws MQClientException {
        DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("order_Consumer");
        consumer.setNamesrvAddr("192.168.100.145:9876;192.168.100.146:9876;192.168.100.149:9876;192.168.100.239:9876");

        /**
         * 设置Consumer第一次启动是从队列头部开始消费还是队列尾部开始消费<br>
         * 如果非第一次启动，那么按照上次消费的位置继续消费
         */
        consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_OFFSET);

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

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

            public ConsumeOrderlyStatus consumeMessage(List<MessageExt> msgs, ConsumeOrderlyContext context) {
                // 设置自动提交
                context.setAutoCommit(true);
                for (MessageExt msg : msgs) {
                    System.out.println(msg + ",内容：" + new String(msg.getBody()));
                }

                try {
                    TimeUnit.SECONDS.sleep(5L);
                } catch (InterruptedException e) {

                    e.printStackTrace();
                }
                ;

                return ConsumeOrderlyStatus.SUCCESS;
            }
        });

        consumer.start();

        System.out.println("Consumer1 Started.");
    }
}
```

结果如下图所示：



```
, toString()=Message [topic=TopicOrderTest, flag=0, properties={TAGS=order_1, KEYS=KEY1, WAIT=true, MAX_OFFSET=5, MIN_OFFSET=0}, body=9]],内容: order_1 1
, toString()=Message [topic=TopicOrderTest, flag=0, properties={TAGS=order_1, KEYS=KEY2, WAIT=true, MAX_OFFSET=5, MIN_OFFSET=0}, body=9]],内容: order_1 2
 toString()=Message [topic=TopicOrderTest, flag=0, properties={TAGS=order_1, KEYS=KEY3, WAIT=true, MAX_OFFSET=5, MIN_OFFSET=0}, body=9]],内容: order_1 3
 toString()=Message [topic=TopicOrderTest, flag=0, properties={TAGS=order_1, KEYS=KEY4, WAIT=true, MAX_OFFSET=5, MIN_OFFSET=0}, body=9]],内容: order_1 4
, toString()=Message [topic=TopicOrderTest, flag=0, properties={TAGS=order_1, KEYS=KEY5, WAIT=true, MAX_OFFSET=5, MIN_OFFSET=0}, body=9]],内容: order_1 5
```

这个五条数据被顺序消费了

- **多个节点（Producer端1个、Consumer端2个）**

Producer.java

```java
package order;

import java.util.List;

import com.alibaba.rocketmq.client.exception.MQBrokerException;
import com.alibaba.rocketmq.client.exception.MQClientException;
import com.alibaba.rocketmq.client.producer.DefaultMQProducer;
import com.alibaba.rocketmq.client.producer.MessageQueueSelector;
import com.alibaba.rocketmq.client.producer.SendResult;
import com.alibaba.rocketmq.common.message.Message;
import com.alibaba.rocketmq.common.message.MessageQueue;
import com.alibaba.rocketmq.remoting.exception.RemotingException;

/**
 * Producer，发送顺序消息
 */
public class Producer {
    public static void main(String[] args) {
        try {
            DefaultMQProducer producer = new DefaultMQProducer("order_Producer");

producer.setNamesrvAddr("192.168.100.145:9876;192.168.100.146:9876;192.168.100.149:9876;192.168.100.239:9876");

            producer.start();

            // String[] tags = new String[] { "TagA", "TagB", "TagC", "TagD",
            // "TagE" };

            for (int i = 1; i <= 5; i++) {

                Message msg = new Message("TopicOrderTest", "order_1", "KEY" + i, ("order_1 " + i).getBytes());

                SendResult sendResult = producer.send(msg, new MessageQueueSelector() {
                    public MessageQueue select(List<MessageQueue> mqs, Message msg, Object arg) {
                        Integer id = (Integer) arg;
                        int index = id % mqs.size();
                        return mqs.get(index);
                    }
                }, 0);

                System.out.println(sendResult);
            }
            for (int i = 1; i <= 5; i++) {

                Message msg = new Message("TopicOrderTest", "order_2", "KEY" + i, ("order_2 " + i).getBytes());

                SendResult sendResult = producer.send(msg, new MessageQueueSelector() {
                    public MessageQueue select(List<MessageQueue> mqs, Message msg, Object arg) {
                        Integer id = (Integer) arg;
                        int index = id % mqs.size();
                        return mqs.get(index);
                    }
                }, 1);

                System.out.println(sendResult);
            }
            for (int i = 1; i <= 5; i++) {

                Message msg = new Message("TopicOrderTest", "order_3", "KEY" + i, ("order_3 " + i).getBytes());

                SendResult sendResult = producer.send(msg, new MessageQueueSelector() {
                    public MessageQueue select(List<MessageQueue> mqs, Message msg, Object arg) {
                        Integer id = (Integer) arg;
                        int index = id % mqs.size();
                        return mqs.get(index);
                    }
                }, 2);

                System.out.println(sendResult);
            }

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

Consumer1.java

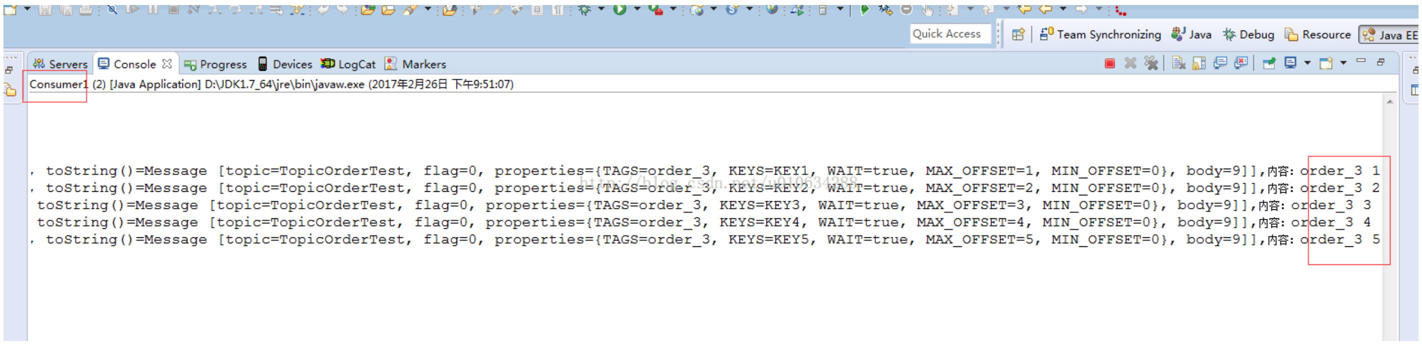
```
/**
 * 顺序消息消费，带事务方式（应用可控制Offset什么时候提交）
 */
public class Consumer1 {

    public static void main(String[] args) throws MQClientException {
        DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("order_Consumer");
        consumer.setNamesrvAddr("192.168.100.145:9876;192.168.100.146:9876;192.168.100.149:9876;192.168.100.239:9876");

        /**
         * 设置Consumer第一次启动是从队列头部开始消费还是队列尾部开始消费<br>
         * 如果非第一次启动，那么按照上次消费的位置继续消费
         */
        consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_OFFSET);

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

        /**
         * 实现了MessageListenerOrderly表示一个队列只会被一个线程取到
         *，第二个线程无法访问这个队列
         */
        consumer.registerMessageListener(new MessageListenerOrderly() {
            AtomicLong consumeTimes = new AtomicLong(0);

            public ConsumeOrderlyStatus consumeMessage(List<MessageExt> msgs, ConsumeOrderlyContext context) {
                // 设置自动提交
                context.setAutoCommit(true);
                for (MessageExt msg : msgs) {
                    System.out.println(msg + ",内容：" + new String(msg.getBody()));
                }

                try {
                    TimeUnit.SECONDS.sleep(5L);
```

```java
                } catch (InterruptedException e) {

                    e.printStackTrace();
                }
                ;

                return ConsumeOrderlyStatus.SUCCESS;
            }
        });

        consumer.start();

        System.out.println("Consumer1 Started.");
    }

}
```

Consumer2.java

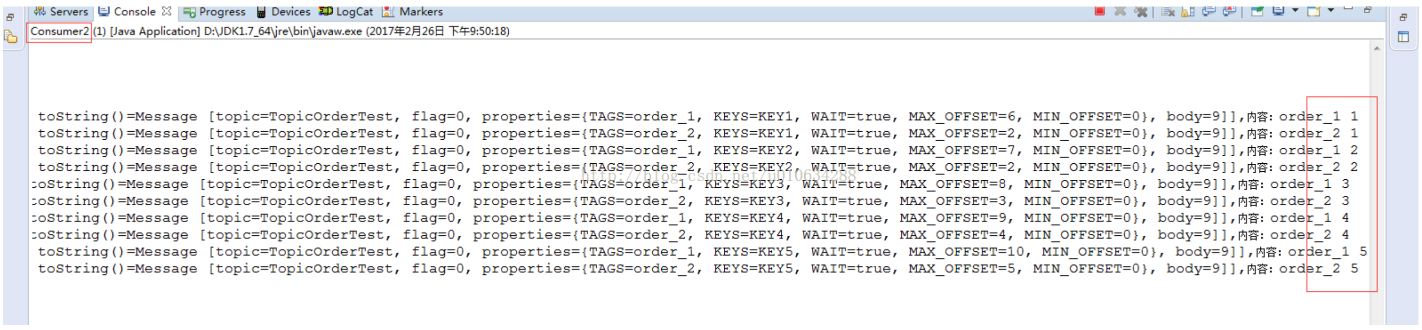
```java
/**
 * 顺序消息消费，带事务方式（应用可控制Offset什么时候提交）
 */
public class Consumer2 {

    public static void main(String[] args) throws MQClientException {
        DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("order_Consumer");
        consumer.setNamesrvAddr("192.168.100.145:9876;192.168.100.146:9876;192.168.100.149:9876;192.168.100.239:9876");

        /**
         * 设置Consumer第一次启动是从队列头部开始消费还是队列尾部开始消费<br>
         * 如果非第一次启动，那么按照上次消费的位置继续消费
         */
        consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_OFFSET);

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

        /**
         * 实现了MessageListenerOrderly表示一个队列只会被一个线程取到
         *，第二个线程无法访问这个队列
         */
        consumer.registerMessageListener(new MessageListenerOrderly() {
            AtomicLong consumeTimes = new AtomicLong(0);

            public ConsumeOrderlyStatus consumeMessage(List<MessageExt> msgs, ConsumeOrderlyContext context) {
                // 设置自动提交
                context.setAutoCommit(true);
                for (MessageExt msg : msgs) {
                    System.out.println(msg + ",内容：" + new String(msg.getBody()));
                }

                try {
                    TimeUnit.SECONDS.sleep(5L);
                } catch (InterruptedException e) {

                    e.printStackTrace();
                }
                ;

                return ConsumeOrderlyStatus.SUCCESS;
            }
        });

        consumer.start();

        System.out.println("Consumer2 Started.");
    }

}
```

先启动Consumer1和Consumer2，然后启动Producer，Producer会发送15条消息
Consumer1消费情况如图，都按照顺序执行了



```
Consumer1 (2) [Java Application] D:\JDK1.7_64\jre\bin\javaw.exe (2017年2月26日 下午9:51:07)

, toString()=Message [topic=TopicOrderTest, flag=0, properties={TAGS=order_3, KEYS=KEY1, WAIT=true, MAX_OFFSET=1, MIN_OFFSET=0}, body=9]],内容: order_3 1
, toString()=Message [topic=TopicOrderTest, flag=0, properties={TAGS=order_3, KEYS=KEY2, WAIT=true, MAX_OFFSET=2, MIN_OFFSET=0}, body=9]],内容: order_3 2
 toString()=Message [topic=TopicOrderTest, flag=0, properties={TAGS=order_3, KEYS=KEY3, WAIT=true, MAX_OFFSET=3, MIN_OFFSET=0}, body=9]],内容: order_3 3
 toString()=Message [topic=TopicOrderTest, flag=0, properties={TAGS=order_3, KEYS=KEY4, WAIT=true, MAX_OFFSET=4, MIN_OFFSET=0}, body=9]],内容: order_3 4
, toString()=Message [topic=TopicOrderTest, flag=0, properties={TAGS=order_3, KEYS=KEY5, WAIT=true, MAX_OFFSET=5, MIN_OFFSET=0}, body=9]],内容: order_3 5
```

Consumer2消费情况如图，都按照顺序执行了



```
Consumer2 (1) [Java Application] D:\JDK1.7_64\jre\bin\javaw.exe (2017年2月26日 下午9:50:18)

 toString()=Message [topic=TopicOrderTest, flag=0, properties={TAGS=order_1, KEYS=KEY1, WAIT=true, MAX_OFFSET=6, MIN_OFFSET=0}, body=9]],内容: order_1 1
 toString()=Message [topic=TopicOrderTest, flag=0, properties={TAGS=order_2, KEYS=KEY1, WAIT=true, MAX_OFFSET=2, MIN_OFFSET=0}, body=9]],内容: order_2 1
 toString()=Message [topic=TopicOrderTest, flag=0, properties={TAGS=order_1, KEYS=KEY2, WAIT=true, MAX_OFFSET=7, MIN_OFFSET=0}, body=9]],内容: order_1 2
 toString()=Message [topic=TopicOrderTest, flag=0, properties={TAGS=order_2, KEYS=KEY2, WAIT=true, MAX_OFFSET=2, MIN_OFFSET=0}, body=9]],内容: order_2 2
:oString()=Message [topic=TopicOrderTest, flag=0, properties={TAGS=order_1, KEYS=KEY3, WAIT=true, MAX_OFFSET=8, MIN_OFFSET=0}, body=9]],内容: order_1 3
:oString()=Message [topic=TopicOrderTest, flag=0, properties={TAGS=order_2, KEYS=KEY3, WAIT=true, MAX_OFFSET=3, MIN_OFFSET=0}, body=9]],内容: order_2 3
:oString()=Message [topic=TopicOrderTest, flag=0, properties={TAGS=order_1, KEYS=KEY4, WAIT=true, MAX_OFFSET=9, MIN_OFFSET=0}, body=9]],内容: order_1 4
:oString()=Message [topic=TopicOrderTest, flag=0, properties={TAGS=order_2, KEYS=KEY4, WAIT=true, MAX_OFFSET=4, MIN_OFFSET=0}, body=9]],内容: order_2 4
 toString()=Message [topic=TopicOrderTest, flag=0, properties={TAGS=order_1, KEYS=KEY5, WAIT=true, MAX_OFFSET=10, MIN_OFFSET=0}, body=9]],内容: order_1 5
 toString()=Message [topic=TopicOrderTest, flag=0, properties={TAGS=order_2, KEYS=KEY5, WAIT=true, MAX_OFFSET=5, MIN_OFFSET=0}, body=9]],内容: order_2 5
```

二、事务消费

这里说的主要是分布式事物。下面的例子的数据库分别安装在不同的节点上。

事物消费需要先说说什么是事务。比如说：我们跨行转账，从工商银行转到建设银行，也就是我从工商银行扣除1000元之后，我的建设银行也必须加1000元。这样才能保证数据的一致性。假如工商银行转1000元之后，建设银行的服务器突然宕机，那么我扣除了1000，但是并没有在建设银行给我加1000，就出现了数据的不一致。因此加1000和减1000才行，减1000和减1000必须一起成功，一起失败。

再比如，我们进行网购的时候，我们下单之后，订单提交成功，仓库商品的数量必须减一。但是订单可能是一个数据库，仓库数量可能又是在另个数据库里面。有可能订单提交成功之后，仓库数量服务器突然宕机。这样也出现了数据不一致的问题。

使用消息队列来解决分布式事物：

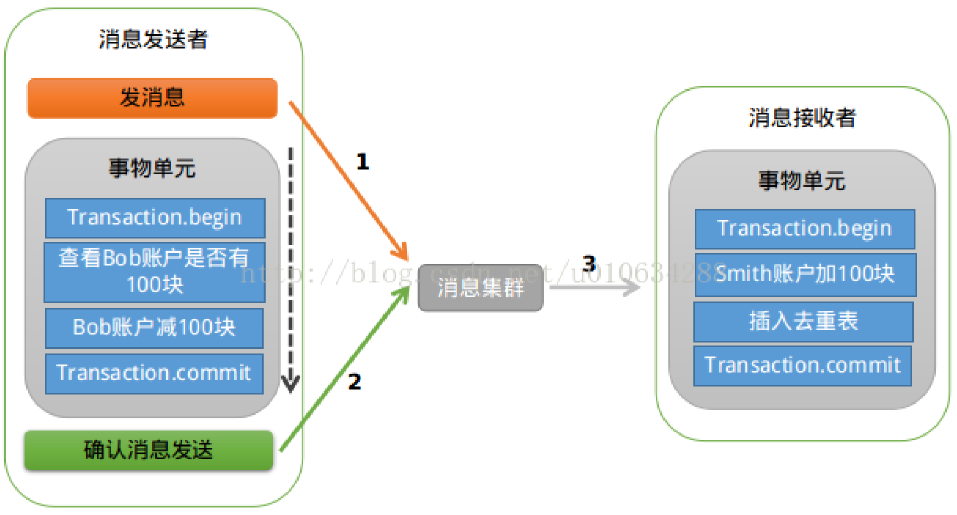
现在我们去外面饭店吃饭，很多时候都不会直接给了钱之后直接在付款的窗口递饭菜，而是付款之后他会给你一张小票，你拿着这个小票去出饭的窗口取饭。这里和我们的系统类似，提高了吞吐量。即使你到第二个窗口，师傅告诉你已经没饭了，你可以拿着这个凭证去退款，即使中途由于出了意外你无法到达窗口进行取饭，但是只要凭证还在，可以将钱退给你。这样就保证了数据的一致性。

如何保证凭证（消息）有2种方法：

1、在工商银行扣款的时候，余额表扣除1000，同时记录日志，而且这2个表是在同一个数据库实例中，可以使用本地事物解决。然后我们通知建设银行需要加1000给该用户，建设银行收到之后给我返回已经加了1000给用户的确认信息之后，我再标记日志表里面的日志为已经完成。

2、通过消息中间件

原文地址：http://www.jianshu.com/p/453c6e7ff81c



RocketMQ第一阶段发送Prepared消息时，会拿到消息的地址，第二阶段执行本地事物，第三阶段通过第一阶段拿到的地址去访问消息，并修改消息的状态。

细心的你可能又发现问题了，如果确认消息发送失败了怎么办？RocketMQ会定期扫描消息集群中的事物消息，如果发现了Prepared消息，它会向消息发送端(生产者)确认，Bob的钱到底是减了还是没减呢？如果减了是回滚还是继续发送确认消息呢？RocketMQ会根据发送端设置的策略来决定是回滚还是继续发送确认消息。这样就保证了消息发送与本地事务同时成功或同时失败。

例子：

Consumer.java

```java
package transaction;

import java.util.List;

import com.alibaba.rocketmq.client.consumer.DefaultMQPushConsumer;
import com.alibaba.rocketmq.client.consumer.listener.ConsumeConcurrentlyContext;
import com.alibaba.rocketmq.client.consumer.listener.ConsumeConcurrentlyStatus;
import com.alibaba.rocketmq.client.consumer.listener.MessageListenerConcurrently;
import com.alibaba.rocketmq.client.exception.MQClientException;
import com.alibaba.rocketmq.common.consumer.ConsumeFromWhere;
import com.alibaba.rocketmq.common.message.MessageExt;

/**
 * Consumer，订阅消息
 */
public class Consumer {

    public static void main(String[] args) throws InterruptedException, MQClientException {
        DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("transaction_Consumer");
        consumer.setNamesrvAddr("192.168.100.145:9876;192.168.100.146:9876;192.168.100.149:9876;192.168.100.239:9876");
        consumer.setConsumeMessageBatchMaxSize(10);
        /**
         * 设置Consumer第一次启动是从队列头部开始消费还是队列尾部开始消费<br>
         * 如果非第一次启动，那么按照上次消费的位置继续消费
         */
        consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_OFFSET);

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

        consumer.registerMessageListener(new MessageListenerConcurrently() {

            public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs, ConsumeConcurrentlyContext context) {


                try {
                    for (MessageExt msg : msgs) {
                        System.out.println(msg + ",内容：" + new String(msg.getBody()));
                    }

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

                    return ConsumeConcurrentlyStatus.RECONSUME_LATER;// 重试

                }

                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;// 成功
            }
        });

        consumer.start();

        System.out.println("transaction_Consumer Started.");
    }
}
```

Producer.java

```
package transaction;

import com.alibaba.rocketmq.client.exception.MQClientException;
import com.alibaba.rocketmq.client.producer.SendResult;
import com.alibaba.rocketmq.client.producer.TransactionCheckListener;
import com.alibaba.rocketmq.client.producer.TransactionMQProducer;
import com.alibaba.rocketmq.common.message.Message;

/**
 * 发送事务消息例子
 *
 */
public class Producer {
    public static void main(String[] args) throws MQClientException, InterruptedException {

        TransactionCheckListener transactionCheckListener = new TransactionCheckListenerImpl();
        TransactionMQProducer producer = new TransactionMQProducer("transaction_Producer");
        producer.setNamesrvAddr("192.168.100.145:9876;192.168.100.146:9876;192.168.100.149:9876;192.168.100.239:9876");
        // 事务回查最小并发数
        producer.setCheckThreadPoolMinSize(2);
        // 事务回查最大并发数
        producer.setCheckThreadPoolMaxSize(2);
        // 队列数
        producer.setCheckRequestHoldMax(2000);
        producer.setTransactionCheckListener(transactionCheckListener);
        producer.start();

        // String[] tags = new String[] { "TagA", "TagB", "TagC", "TagD", "TagE"
        // };
        TransactionExecuterImpl tranExecuter = new TransactionExecuterImpl();
        for (int i = 1; i <= 2; i++) {
            try {
                Message msg = new Message("TopicTransactionTest", "transaction" + i, "KEY" + i,
                        ("Hello RocketMQ " + i).getBytes());
                SendResult sendResult = producer.sendMessageInTransaction(msg, tranExecuter, null);
                System.out.println(sendResult);

                Thread.sleep(10);
            } catch (MQClientException e) {
                e.printStackTrace();
            }
        }
        for (int i = 0; i < 100000; i++) {
            Thread.sleep(1000);
        }

        producer.shutdown();

    }
}
```

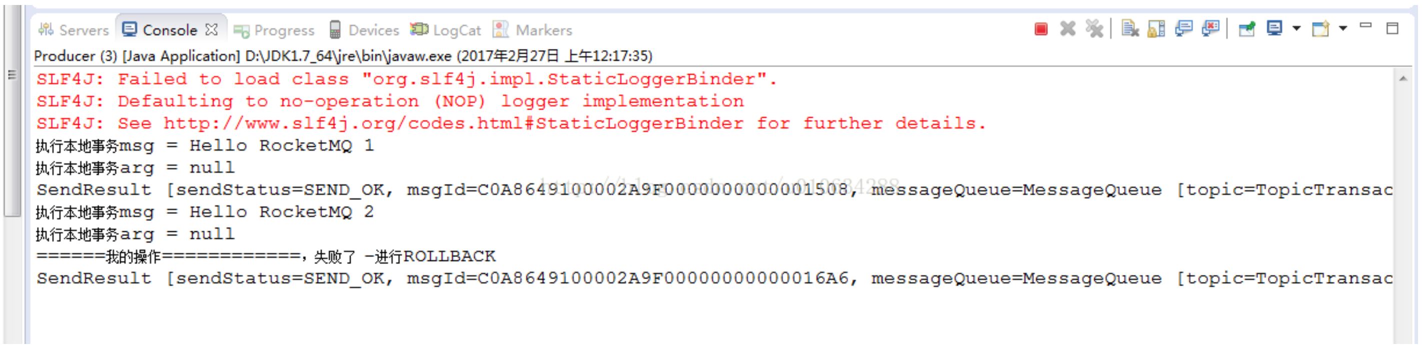
TransactionExecuterImpl .java --执行本地事务

```
package transaction;

import com.alibaba.rocketmq.client.producer.LocalTransactionExecuter;
import com.alibaba.rocketmq.client.producer.LocalTransactionState;
import com.alibaba.rocketmq.common.message.Message;

/**
 * 执行本地事务
 */
public class TransactionExecuterImpl implements LocalTransactionExecuter {
    // private AtomicInteger transactionIndex = new AtomicInteger(1);

    public LocalTransactionState executeLocalTransactionBranch(final Message msg, final Object arg) {

        System.out.println("执行本地事务msg = " + new String(msg.getBody()));

        System.out.println("执行本地事务arg = " + arg);

        String tags = msg.getTags();

        if (tags.equals("transaction2")) {
            System.out.println("======我的操作============，失败了  -进行ROLLBACK");
            return LocalTransactionState.ROLLBACK_MESSAGE;
        }
        return LocalTransactionState.COMMIT_MESSAGE;
        // return LocalTransactionState.UNKNOW;
    }
}
```

TransactionCheckListenerImpl--未决事务，服务器回查客户端(目前已经被阉割啦)

```
package transaction;

import com.alibaba.rocketmq.client.producer.LocalTransactionState;
import com.alibaba.rocketmq.client.producer.TransactionCheckListener;
import com.alibaba.rocketmq.common.message.MessageExt;

/**
 * 未决事务，服务器回查客户端
 */
public class TransactionCheckListenerImpl implements TransactionCheckListener {
    // private AtomicInteger transactionIndex = new AtomicInteger(0);

    //在这里，我们可以根据由MQ回传的key去数据库查询，这条数据到底是成功了还是失败了。
    public LocalTransactionState checkLocalTransactionState(MessageExt msg) {
        System.out.println("未决事务，服务器回查客户端msg =" + new String(msg.getBody().toString()));
        // return LocalTransactionState.ROLLBACK_MESSAGE;

        return LocalTransactionState.COMMIT_MESSAGE;

        // return LocalTransactionState.UNKNOW;
        }
}
```

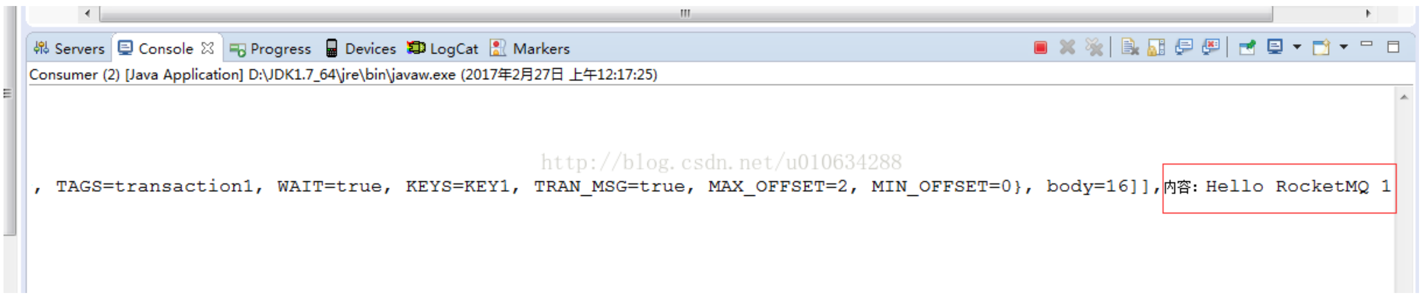
producer端：发送数据到MQ，并且处理本地事物。这里模拟了一个成功一个失败。Consumer只会接收到本地事物成功的数据。第二个数据失败了，不会被消费。



Producer (3) [Java Application] D:\JDK1.7_64\jre\bin\javaw.exe (2017年2月27日 上午12:17:35)

```
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
执行本地事务msg = Hello RocketMQ 1
执行本地事务arg = null
SendResult [sendStatus=SEND_OK, msgId=C0A8649100002A9F0000000000001508, messageQueue=MessageQueue [topic=TopicTransac
执行本地事务msg = Hello RocketMQ 2
执行本地事务arg = null
======我的操作============, 失败了 -进行ROLLBACK
SendResult [sendStatus=SEND_OK, msgId=C0A8649100002A9F00000000000016A6, messageQueue=MessageQueue [topic=TopicTransac
```

Consumer只会接收到一个，第二个数据不会被接收到



Consumer (2) [Java Application] D:\JDK1.7_64\jre\bin\javaw.exe (2017年2月27日 上午12:17:25)

```
, TAGS=transaction1, WAIT=true, KEYS=KEY1, TRAN_MSG=true, MAX_OFFSET=2, MIN_OFFSET=0}, body=16]],内容: Hello RocketMQ 1
```