

消息队列 MQ

消息队列 RocketMQ

消息队列 RocketMQ

特色消息类型

定时消息和延时消息

本文档主要介绍 MQ 定时消息和延时消息的概念、适用场景以及使用过程中的注意事项。

概念介绍

- 定时消息：Producer 将消息发送到 MQ 服务端，但并不期望这条消息立马投递，而是推迟到在当前时间点之后的某一个时间投递到 Consumer 进行消费，该消息即定时消息。
- 延时消息：Producer 将消息发送到 MQ 服务端，但并不期望这条消息立马投递，而是延迟一定时间后才投递到 Consumer 进行消费，该消息即延时消息。

定时消息与延迟消息在代码配置上存在一些差异，但是最终达到的效果相同：消息在发送到 MQ 服务端后并不会立马投递，而是根据消息中的属性延迟固定时间后才投递给消费者。

适用场景

定时/延时消息适用于如下一些场景：

- 消息生产和消费有时间窗口要求：比如在电商交易中超时未支付关闭订单的场景，在订单创建时会发送一条 MQ 延时消息，这条消息将会在30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的订单是否已完成支付。如支付未完成，则关闭订单，如已完成支付则忽略。
- 通过消息触发一些定时任务，比如在某一固定时间点向用户发送提醒消息。

使用方式

定时消息、延时消息的使用在代码编写上存在略微的区别：

- 发送**定时消息**需要明确指定消息发送时间点之后的某一时间点作为消息投递的时间点。
- 发送**延时消息**时需要设定一个延时时间长度，消息将从当前发送时间点开始延迟固定时间之后才开始

投递。

注意事项

1. 定时/延时消息 `msg.setStartDeliverTime` 的参数需要设置成当前时间戳之后的某个时刻（单位毫秒），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。
2. 定时/延时消息 `msg.setStartDeliverTime` 的参数可设置40天内的任何时刻（单位毫秒），超过40天消息发送将失败。
3. `StartDeliverTime` 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、延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。
4. 由于客户端和服务端可能存在时间差，定时消息/延时消息的投递也可能与客户端设置的时间存在偏差。
5. 设置定时、延时消息的投递时间后，依然受 3 天的消息保存时长限制。例如，设置定时消息 5 天后才能被消费，如果第 5 天后一直未被消费，那么这条消息将在第8天被删除。
6. 除 TCP 协议接入的 Java 语言支持延时消息，其他方式都不支持延时消息。

示例代码

发送定时消息和延时消息的示例代码请参考以下文档：

TCP 协议接入：

Java

- 发送定时消息
- 发送延时消息

C++

- 发送定时消息

.NET

- 发送定时消息

HTTP 协议接入：

- 发送定时消息

顺序消息

本文主要介绍 MQ 顺序的概念、适用场景以及使用过程中的注意事项。

SDK 支持

请使用 Java SDK 1.2.7 及以上版本。C++/.NET SDK 即将支持。

示例代码请参考以下文档：

- TCP Java 收发送顺序消息
- HTTP 发送顺序消息

概念介绍

顺序消息（FIFO 消息）是 MQ 提供的一种严格按照顺序进行发布和消费的消息类型。顺序消息由两个部分组成：顺序发布和顺序消费。

- **顺序发布**：对于指定的一个 Topic，客户端将按照一定的先后顺序进行发送消息。
- **顺序消费**：对于指定的一个 Topic，按照一定的先后顺序进行接收消息，即先发送的消息一定会先被客户端接收到。

顺序消息类型分为两种：全局顺序和分区顺序。以下分别对 2 种顺序消息类型进行介绍。

全局顺序

对于指定的一个 Topic，所有消息按照严格的先入先出（FIFO）的顺序进行发布和消费。



适用场景：

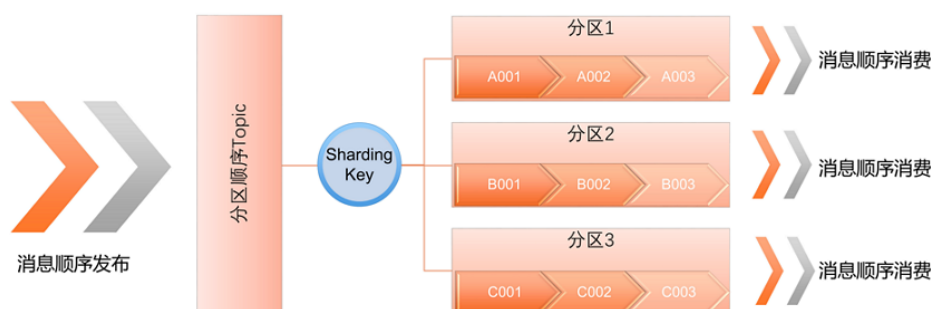
- 性能要求不高，所有的消息严格按照 FIFO 原则进行消息发布和消费的场景。

举例说明：

- 【例一】证券处理中，以人民币兑换美元为 Topic，在价格相同的情况下，先出价者优先处理，则可以通过全局顺序的方式按照 FIFO 的方式进行发布和订阅。

分区顺序

对于指定的一个 Topic，所有消息根据 sharding key 进行区块分区。同一个分区内的消息按照严格的 FIFO 顺序进行发布和消费。Sharding key 是顺序消息中用来区分不同分区的关键字段，和普通消息的 Key 是完全不同的概念。



适用场景：

- 性能要求高，以 sharding key 作为分区字段，在同一个区块中严格的按照 FIFO 原则进行消息发布和消费的场景。

举例说明：

【例一】用户注册需要发送发验证码，以用户 ID 作为 sharding key，那么同一个用户发送的消息都会按照先后顺序来发布和订阅。

【例二】电商的订单创建，以订单 ID 作为 sharding key，那么同一个订单相关的创建订单消息、订单支付消息、订单退款消息、订单物流消息都会按照先后顺序来发布和订阅。

阿里巴巴集团内部电商系统均使用此种分区顺序消息，既保证业务的顺序，同时又能保证业务的高性能。

全局顺序与分区顺序对比

在控制台创建顺序消息使用的 Topic，各种类型 Topic 对比如下。

消息类型对比

Topic 类型	支持事务消息	支持定时消息	性能
无序消息	是	是	最高
分区顺序	否	否	高
全局顺序	否	否	一般

发送方式对比

消息类型	支持可靠同步发送	支持可靠异步发送	支持 Oneway 发送
无序消息	是	是	是
分区顺序	是	否	否
全局顺序	是	否	否

注意事项

- 顺序消息暂不支持广播模式。
- 同一个 Producer ID 或者 Consumer ID 只能对应一种类型的 Topic，即不能同时用于顺序消息和无序消息的收发。
- 顺序消息不支持异步发送方式，否则将无法严格保证顺序。
- 对于全局顺序消息，建议创建实例个数 ≥ 2 。同时运行多个实例的作用是为了防止工作实例意外退出时，业务中断。当工作实例退出时，其他实例可以立即接手工作，不会导致业务中断，实际同时工作的只会有一个实例。
- 对于分区顺序，建议创建实例个数 ≥ 2 并且 $\leq$ 分区总数，当工作实例退出时，其他实例可以立即接手工作并重新进行负载均衡，不会导致业务中断，每个实例平均分配分区数。

事务消息

MQ 事务消息

本文档主要介绍 MQ 事务的概念、适用场景以及使用过程中的注意事项。

概念介绍

- 事务消息：MQ 提供类似 X/Open XA 的分布事务功能，通过 MQ 事务消息能达到分布式事务的最终一致。
- 半消息：暂不能投递的消息，发送方已经将消息成功发送到了 MQ 服务端，但是服务端未收到生产者对该消息的二次确认，此时该消息被标记成“暂不能投递”状态，处于该种状态下的消息即半消息。
- 消息回查：由于网络闪断、生产者应用重启等原因，导致某条事务消息的二次确认丢失，MQ 服务端通过扫描发现某条消息长期处于“半消息”时，需要主动向消息生产者询问该消息的最终状态（Commit 或是 Rollback），该过程即消息回查。

适用场景

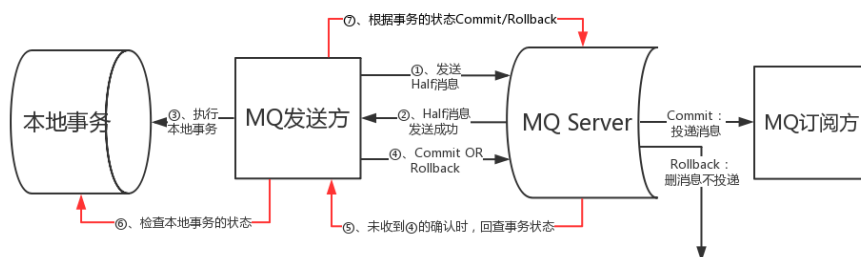
MQ 事务消息适用于如下场景：

- 帮助用户实现类似 X/Open XA 的分布事务功能，通过 MQ 事务消息能达到分布式事务的最终一致。

使用方式

交互流程

MQ 事务消息交互流程如下所示：



其中：

1. 发送方向 MQ 服务端发送消息；
2. MQ Server 将消息持久化成功之后，向发送方 ACK 确认消息已经发送成功，此时消息为半消息。
3. 发送方开始执行本地事务逻辑。
4. 发送方根据本地事务执行结果向 MQ Server 提交二次确认（Commit 或是 Rollback），MQ Server 收到 Commit 状态则将半消息标记为可投递，订阅方最终将收到该消息；MQ Server 收到 Rollback 状态则删除半消息，订阅方将不会接受该消息。
5. 在断网或者是应用重启的特殊情况下，上述步骤4提交的二次确认最终未到达 MQ Server，经过固定时间后 MQ Server 将对该消息发起消息回查。
6. 发送方收到消息回查后，需要检查对应消息的本地事务执行的最终结果。
7. 发送方根据检查得到的本地事务的最终状态再次提交二次确认，MQ Server 仍按照步骤4对半消息进行操作。

事务消息发送对应步骤1、2、3、4，事务消息回查对应步骤5、6、7。

示例代码

- TCP Java 发送事务消息
- TCP C++ 发送事务消息
- TCP .NET 发送事务消息

注意事项

事务消息的 Producer ID 不能与其他类型消息的 Producer ID 共用。

通过 `ONFactory.createTransactionProducer` 创建事务消息的 Producer 时必须指定 `LocalTransactionChecker` 的实现类，处理异常情况下事务消息的回查。

事务消息发送完成本地事务后，可在 `execute` 方法中返回如下三种状态：

- `TransactionStatus.CommitTransaction` 提交事务，允许订阅方消费该消息。
- `TransactionStatus.RollbackTransaction` 回滚事务，消息将被丢弃不允许消费。
- `TransactionStatus.Unknow` 暂时无法判断状态，期待固定时间以后 MQ Server 向发送方

进行消息回查。

可通过如下方式给每条消息设定第一次消息回查的最快时间：

```
Message message = new Message();  
// 在消息属性中添加第一次消息回查的最快时间，单位秒，如下设置实际第一次回查时间为 120 ~ 125 秒之间  
message.putUserProperties(PropertyKeyConst.CheckImmunityTimeInSeconds,"120");  
// 以上方式只确定事务消息的第一次回查的最快时间，实际回查时间向后浮动0~5秒；如第一次回查后事务仍未提交，后续每隔5秒回查一次。
```

Demo 工程

Demo 环境准备

本 Demo 主要目的在于帮助初次接触 MQ 的工程师，一步一步搭建 MQ 测试工程。Demo 程序以 Java 为例，包括普通消息、事务消息、定时消息的测试代码，以及相关 Spring 的配置示例。

安装 IDE

您可以使用 IDEA 或者 Eclipse。本文以 IDEA 为例。

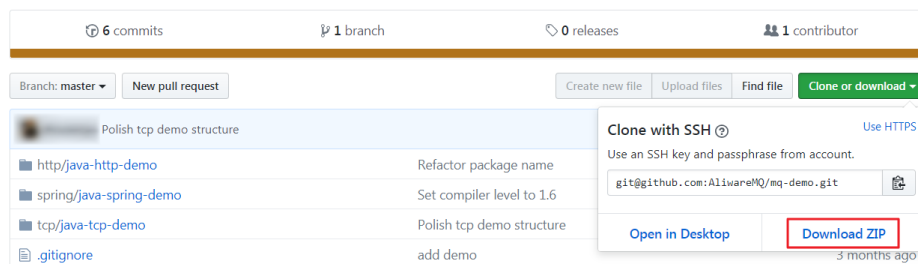
请在 <https://www.jetbrains.com/idea/> 下载 IDEA Ultimate 版本，并参考 IDEA 说明进行安装。

下载 MQ Demo 工程

在 <https://github.com/AlibabaMQ/mq-demo> 下载 MQ Demo 工程到本地：

Demo for AliwareMQ <https://www.aliyun.com/product/ons>

aliware



下载完成后解压即可看到本地新增了 Aliware-MQ-demo-master 文件夹。

Demo 工程配置

Demo 工程设置包含以下几个步骤。

MQ Demo 工程文件导入 IDEA

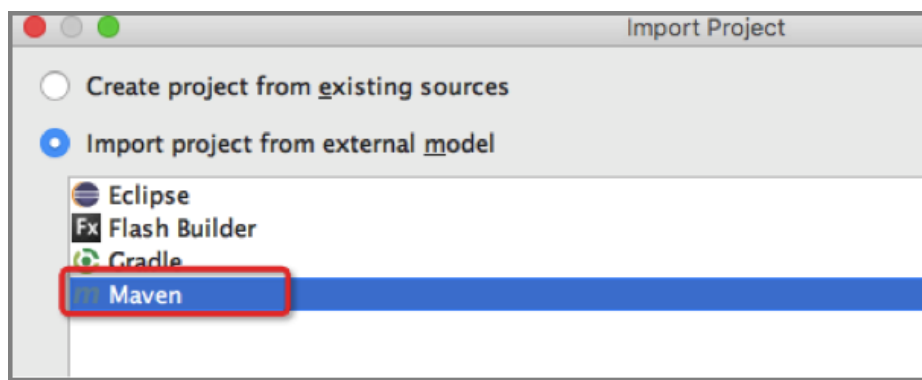
注意：如果本地未安装 JDK，请先下载安装。

1. 双击 IDEA 图标打开 IDEA。

选择 **Import Project**，选择 **mq-demo-master** 文件夹。



选择 **Import** 类型。



默认单击 **Next**，直到导入完成。Demo 工程需要加载依赖的 JAR 包，因此导入过程需要等待2-3分钟。

创建资源

请在 MQ 控制台创建 Topic、Producer ID(PID)、Consumer ID (CID)。

进入 MQ 产品首页：<https://www.aliyun.com/product/ons/>。

单击**立即开通**，进入管理控制台。

选择**公网**域（默认），单击右侧**创建 Topic**，输入 Topic 名称（尽量个性化避免重复）、选择消息类型、输入描述，然后单击**确定**。

在 Topic 管理列表中单击**创建生产者**创建 Producer ID。

在 Topic 管理列表中单击**创建消费者**创建 Consumer ID。

更多详细信息，请参见步骤二：创建资源。

配置 MQ Demo

需要配置3个文件：MqConfig 类，producer.xml，consumer.xml。

配置 MqConfig 类。具体说明如下：

- public static final String TOPIC = "刚才创建的 Topic";
- public static final String PRODUCER_ID = "刚才创建的 PID";
- public static final String CONSUMER_ID = "刚才创建的 CID";
- public static final String ACCESS_KEY = "您的阿里云账号的 AK";
- public static final String SECRET_KEY = "您的阿里云账号的 SK";

AK SK 获取：请登录阿里云账号，点击 AccessKeys。AK 即 Access Key ID，SK 即 Access Key Secret。

说明：主账号创建 Topic 后，如果在 MQ 控制台的 Topic 管理列表里对子账号进行了授权，那么也可以使用 RAM 子账号的 AK SK。

配置 producer.xml。



```
<bean id="producer" class="com.aliyun.openservices.ons.api.bean.ProducerBean"
init-method="start" destroy-method="shutdown">
  <property name="properties">
    <map>
      <entry key="ProducerId" value="XXX" /> <!-- PID, 请替换 -->
      <entry key="AccessKey" value="XXX" /> <!-- ACCESS_KEY, 请替换 -->
      <entry key="SecretKey" value="XXX" /> <!-- SECRET_KEY, 请替换 -->
      <entry key="ONSAddr" value="http://onsaddr-internet.aliyun.com/rocketmq/nsaddr4client-internet"/>
    </map>
  </property>
</bean>
```

配置 consumer.xml。

```
<bean id="consumer" class="com.aliyun.openservices.ons.api.bean.ConsumerBean"
  init-method="start" destroy-method="shutdown">
  <property name="properties">
    <map>
      <entry key="ConsumerId" value="XXX"/> <!-- CID, 请替换 -->
      <entry key="AccessKey" value="XXX"/> <!-- ACCESS_KEY, 请替换 -->
      <entry key="SecretKey" value="XXX"/> <!-- SECRET_KEY, 请替换 -->
      <entry key="ONSAddr" value="http://onsaddr-internet.aliyun.com/rocketmq/nsaddr4client-internet"/>
    </map>
  </property>

  <property name="subscriptionTable">
    <map>
      <entry value-ref="messageListener">
        <key>
          <bean class="com.aliyun.openservices.ons.api.bean.Subscription">
            <property name="topic" value="XXX"/> <!-- Topic, 请替换 -->
            <property name="expression" value="*" /> <!-- MessageType名: 多个messageType 用 "|" 分割 -->
          </bean>
        </key>
      </entry>
    </map>
  </property>
</bean>
```

运行 Demo

以 Main 方式启动收发消息

运行 SimpleMQProducer 类发送消息。

登录 MQ 控制台，在左侧菜单栏选择**消息查询**>**按 Topic 查询**，选择 Topic 名称进行查询。可以看见消息已经发送至 Topic。

运行 SimpleMQConsumer 类接收消息。可以看到消息被接收打印的日志。因为有初始化，所以需要等待几秒，在生产环境中不会经常初始化。

从 MQ 控制台进入**消费者管理**>**消费者状态**，在**查询消费者状态**页面可以看到启动的消费端已经在线，并且订阅关系一致。

以 Spring 方式启动收发消息

1. 运行 MQProducer4Spring 类发送消息。
2. 运行 MQConsumer4Spring 类接收消息。

查看结果跟上面过程类似。

发送事务消息

运行 SimpleTransactionProducer 类发送消息。

说明：LocalTransactionCheckerImpl 类为本地事务 check 接口类。用于校验事务。详情请参考收发事务消息。

收发顺序消息

运行 SimpleOrderConsumer 类接收消息。

运行 SimpleOrderProducer 类发送消息。

说明：一种按照顺序进行发布和消费的方式。详情请参考收发顺序消息。

发送定时（延时）消息

运行 MQTimerProducer 类发送消息。延时3秒后投递。

说明：MQ 也可以指定一个精确的投递时间，最长定时时间为40天。具体请参考收发定时消息。

HTTP、MQTT 物联接入

HTTP 接入请参考 MQ HTTP 接入。

MQTT 接入请参考 MQTT 接入。

常见使用方式说明

集群消费和广播消费

本文档主要介绍 MQ 集群消费和广播消费的基本概念，适用场景以及注意事项。

基本概念

MQ 是基于发布订阅模型的消息系统。在 MQ 消息系统中消息的订阅方订阅关注的 Topic，以获取并消费消息。由于订阅方应用一般是分布式系统，以集群方式部署有多台机器。因此 MQ 约定以下概念。

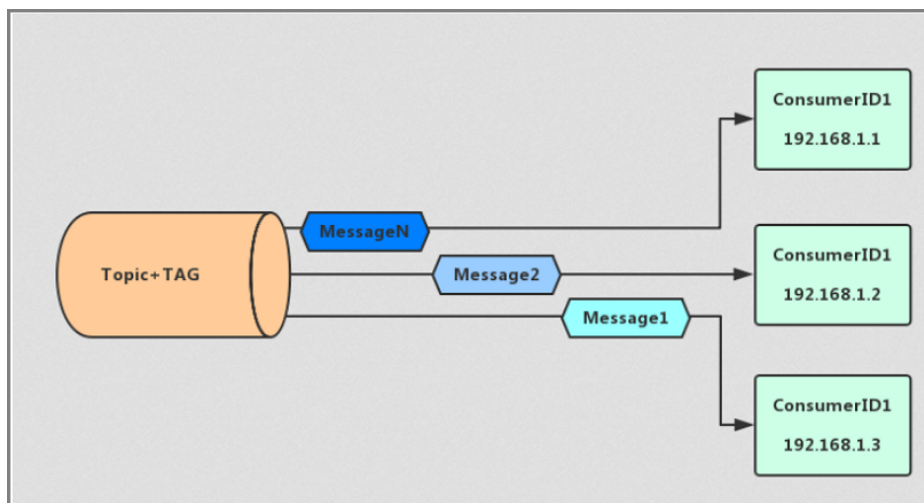
集群：MQ 约定使用相同 Consumer ID 的订阅者属于同一个集群，同一个集群下的订阅者消费逻辑必须完全一致（包括 Tag 的使用），这些订阅者在逻辑上可以认为是一个消费节点。

集群消费：当使用集群消费模式时，MQ 认为任意一条消息只需要被集群内的任意一个消费者处理即可。

广播消费：当使用广播消费模式时，MQ 会将每条消息推送给集群内所有注册过的客户端，保证消息至少被每台机器消费一次。

场景对比

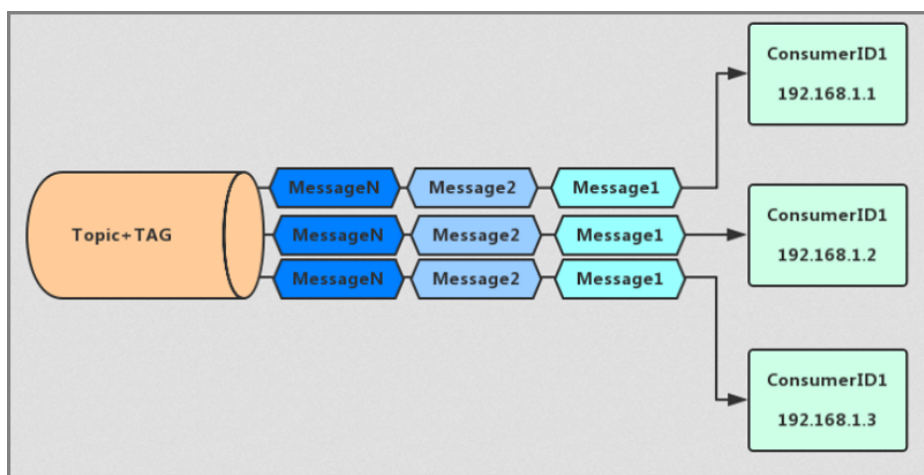
集群消费模式：



适用场景&注意事项

- 消费端集群化部署，每条消息只需要被处理一次。
- 由于消费进度在服务端维护，可靠性更高。
- 集群消费模式下，每一条消息都只会被分发到一台机器上处理，如果需要被集群下的每一台机器都处理，请使用广播模式。
- 集群消费模式下，不保证消息的每一次失败重投等逻辑都能路由到同一台机器上，因此处理消息时不应该做任何确定性假设。

广播消费模式：



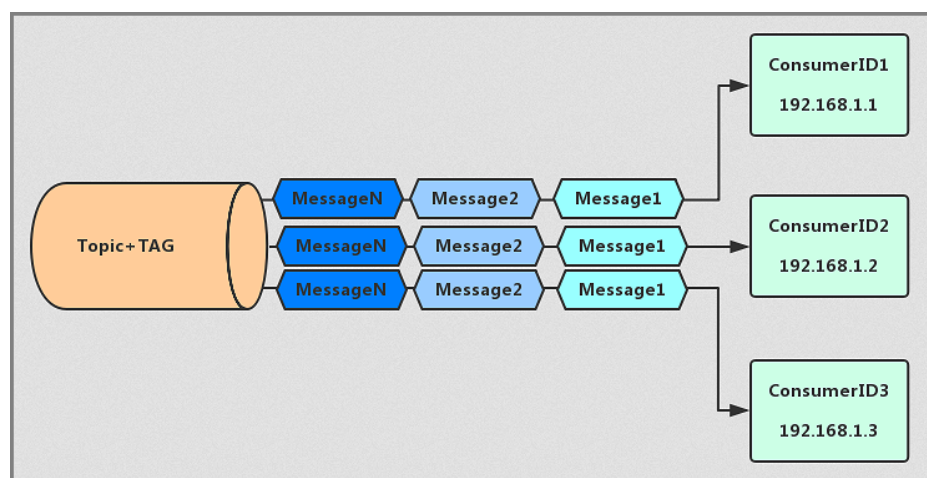
适用场景&注意事项

- 顺序消息暂不支持广播消费模式。

- 每条消息都需要被相同逻辑的多台机器处理。
- 消费进度在客户端维护，出现重复的概率稍大于集群模式。
- 广播模式下，MQ 保证每条消息至少被每台客户端消费一次，但是并不会对消费失败的消息进行失败重投，因此业务方需要关注消费失败的情况。
- 广播模式下，第一次启动时默认从最新消息消费，客户端的消费进度是被持久化在客户端本地的隐藏文件中，因此不建议删除该隐藏文件，否则会丢失部分消息。
- 广播模式下，每条消息都会被大量的客户端重复处理，因此推荐尽可能使用集群模式。
- 目前仅 Java 客户端支持广播模式。
- 广播模式下服务端不维护消费进度，所以 MQ 控制台不支持消息堆积查询和堆积报警功能。

使用集群模式模拟广播：

如果业务需要使用广播模式，也可以创建多个 Consumer ID，用于订阅同一个 Topic。



适用场景&注意事项

- 每条消息都需要被多台机器处理，每台机器的逻辑可以相同也可以不一样。
- 消费进度在服务端维护，可靠性高于广播模式。
- 一个云账户所能创建的 Consumer ID 数量是有限制的，具体可以咨询售后技术支持。
- 对于一个 Consumer ID 来说，可以部署一个消费端实例，也可以部署多个消费端实例。当部署多个消费端实例时，实例之间又组成了集群模式（共同分担消费消息）。假设 Consumer ID1 部署了三个消费者实例 C1，C2，C3，那么这三个实例将共同分担服务器发送给 Consumer ID1 的消息。同时，实例之间订阅关系必须保持一致。

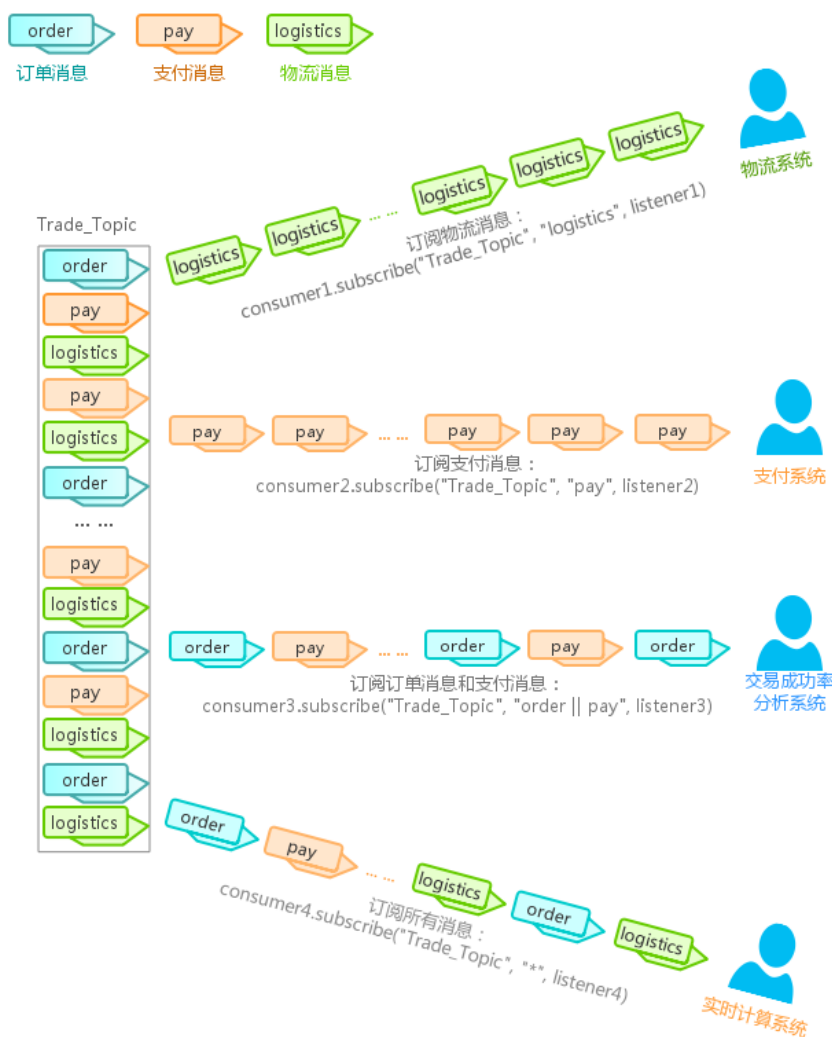
消息过滤

本文描述 MQ 消费者如何根据 Tag 在 MQ 服务端完成消息过滤。

Tag，即消息标签、消息类型，用来区分某个 MQ 的 Topic 下的消息分类。MQ 允许消费者按照 Tag 对消息

进行过滤，确保消费者最终只消费到他关心的消息类型。

以下图电商交易场景为例，从客户下单到收到商品这一过程会生产一系列消息，比如订单创建消息（order）、支付消息（pay）、物流消息（logistics）。这些消息会发送到 Topic 为 Trade_Topic 的队列中，被各个不同的系统所接收，比如支付系统、物流系统、交易成功率分析系统、实时计算系统等。其中，物流系统只需接收物流类型的消息（logistics），而实时计算系统需要接收所有和交易相关（order、pay、logistics）的消息。



说明：针对消息归类，您可以选择创建多个 Topic，或者在同一个 Topic 下创建多个 Tag。但通常情况下，不同的 Topic 之间的消息没有必然的联系，而 Tag 则用来区分同一个 Topic 下相互关联的消息，比如全集和子集的关系，流程先后的关系。

参考示例

发送消息

发送消息时，每条消息必须指明消息类型 Tag：

```
Message msg = new Message("MQ_TOPIC", "TagA", "Hello MQ".getBytes());
```

消费方式-1

消费者如需订阅某 Topic 下所有类型的消息，Tag 用符号 * 表示：

```
consumer.subscribe("MQ_TOPIC", "*", new MessageListener() {  
    public Action consume(Message message, ConsumeContext context) {  
        System.out.println(message.getMsgID());  
        return Action.CommitMessage;  
    }  
});
```

消费方式-2

消费者如需订阅某 Topic 下某一种类型的消息，请明确标明 Tag：

```
consumer.subscribe("MQ_TOPIC", "TagA", new MessageListener() {  
    public Action consume(Message message, ConsumeContext context) {  
        System.out.println(message.getMsgID());  
        return Action.CommitMessage;  
    }  
});
```

消费方式-3

消费者如需订阅某 Topic 下多种类型的消息，请在多个 Tag 之间用 || 分隔：

```
consumer.subscribe("MQ_TOPIC", "TagA||TagB", new MessageListener() {  
    public Action consume(Message message, ConsumeContext context) {  
        System.out.println(message.getMsgID());  
        return Action.CommitMessage;  
    }  
});
```

消费方式-4 (错误示例)

同一个消费者多次订阅某 Topic 下的不同 Tag，后者会覆盖前者：

```
//如下错误代码中，consumer 只能接收到 MQ_TOPIC 下 TagB 的消息，而不能接收 TagA 的消息。  
consumer.subscribe("MQ_TOPIC", "TagA", new MessageListener() {  
    public Action consume(Message message, ConsumeContext context) {  
        System.out.println(message.getMsgID());  
        return Action.CommitMessage;  
    }  
});  
consumer.subscribe("MQ_TOPIC", "TagB", new MessageListener() {  
    public Action consume(Message message, ConsumeContext context) {  
        System.out.println(message.getMsgID());  
        return Action.CommitMessage;  
    }  
});
```



```
});
```

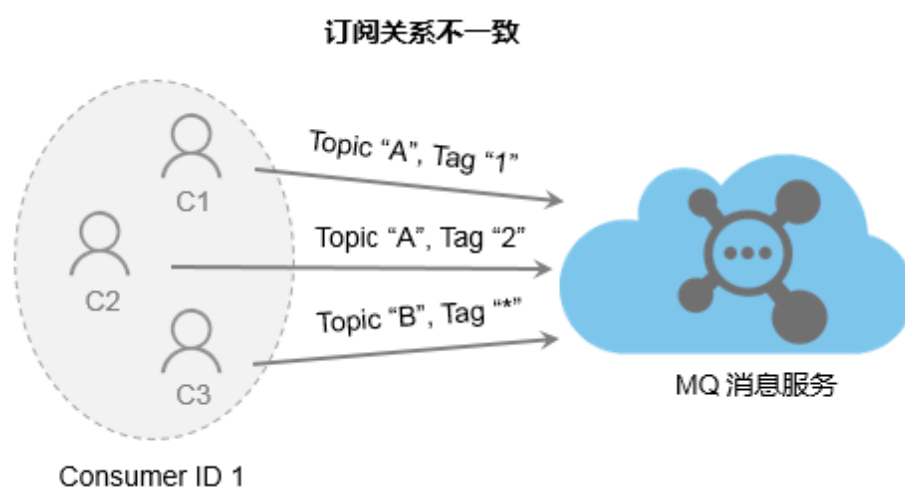
订阅关系一致

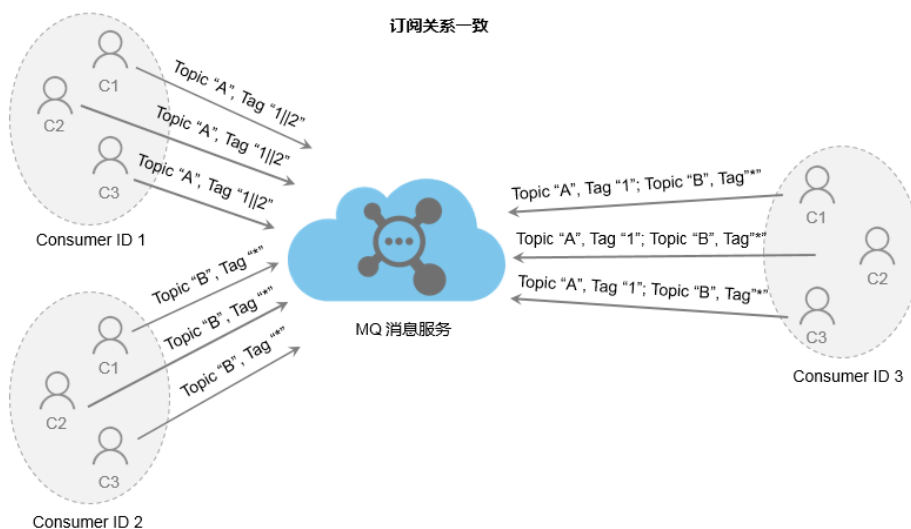
MQ 里的一个 Consumer ID 代表一个 Consumer 实例群组。对于大多数分布式应用来说，一个 Consumer ID 下通常会挂载多个 Consumer 实例。订阅关系一致指的是同一个 Consumer ID 下所有 Consumer 实例的处理逻辑必须完全一致。一旦订阅关系不一致，消息消费的逻辑就会混乱，甚至导致消息丢失。

由于 MQ 的订阅关系主要由 Topic+Tag 共同组成，因此，保持订阅关系一致意味着同一个 Consumer ID 下所有的实例需在以下两方面均保持一致：

订阅的 Topic 必须一致；

订阅的 Topic 中的 Tag 必须一致。





如上图所示，一个 Consumer ID 也可以订阅多个 Topic，但是这个 Consumer ID 里的多个消费者实例的订阅关系一定要保持一致。

下文给出了订阅关系不一致的错误代码示例。

【例一】以下例子中，同一个 Consumer ID 下的两个实例订阅的 Topic 不一致。

Consumer 实例 1-1：

```
Properties properties = new Properties();
properties.put(PropertyKeyConst.ConsumerId, "CID_jodie_test_1");
Consumer consumer = ONSFactory.createConsumer(properties);
consumer.subscribe("jodie_test_A", "*", new MessageListener() {
    public Action consume(Message message, ConsumeContext context) {
        System.out.println(message.getMsgID());
        return Action.CommitMessage;
    }
});
```

Consumer 实例1-2：

```
Properties properties = new Properties();
properties.put(PropertyKeyConst.ConsumerId, " CID_jodie_test_1");
Consumer consumer = ONSFactory.createConsumer(properties);
consumer.subscribe("jodie_test_B ", "*", new MessageListener() {
    public Action consume(Message message, ConsumeContext context) {
        System.out.println(message.getMsgID());
        return Action.CommitMessage;
    }
});
```

【例二】以下例子中，同一个 Consumer ID 下订阅 Topic 的 Tag 不一致。Consumer 实例2-1 订阅了 TagA，而 Consumer 实例2-2 未指定 Tag。

Consumer 实例2-1：

```
Properties properties = new Properties();
properties.put(PropertyKeyConst.ConsumerId, "CID_jodie_test_2");
Consumer consumer = ONSFactory.createConsumer(properties);
consumer.subscribe("jodie_test_A", "TagA", new MessageListener() {
    public Action consume(Message message, ConsumeContext context) {
        System.out.println(message.getMsgID());
        return Action.CommitMessage;
    }
});
```

Consumer 实例2-2 :

```
Properties properties = new Properties();
properties.put(PropertyKeyConst.ConsumerId, " CID_jodie_test_2");
Consumer consumer = ONSFactory.createConsumer(properties);
consumer.subscribe("jodie_test_A ", "*", new MessageListener() {
    public Action consume(Message message, ConsumeContext context) {
        System.out.println(message.getMsgID());
        return Action.CommitMessage;
    }
});
```

【例三】此例中，错误的原因有两个：

1. 同一个 Consumer ID 下订阅 Topic 个数不一致。
2. 同一个 Consumer ID 下订阅 Topic 的 Tag 不一致。

Consumer 实例3-1 :

```
Properties properties = new Properties();
properties.put(PropertyKeyConst.ConsumerId, "CID_jodie_test_3");
Consumer consumer = ONSFactory.createConsumer(properties);
consumer.subscribe("jodie_test_A", "TagA", new MessageListener() {
    public Action consume(Message message, ConsumeContext context) {
        System.out.println(message.getMsgID());
        return Action.CommitMessage;
    }
});
consumer.subscribe("jodie_test_B", "TagB", new MessageListener() {
    public Action consume(Message message, ConsumeContext context) {
        System.out.println(message.getMsgID());
        return Action.CommitMessage;
    }
});
```

Consumer 实例3-2 :

```
Properties properties = new Properties();
properties.put(PropertyKeyConst.ConsumerId, " CID_jodie_test_3");
Consumer consumer = ONSFactory.createConsumer(properties);
consumer.subscribe("jodie_test_A ", "TagB", new MessageListener() {
```

```
public Action consume(Message message, ConsumeContext context) {
    System.out.println(message.getMsgID());
    return Action.CommitMessage;
}
});
```

消息重试

MQ 消费者的消费逻辑失败时，可以通过设置返回状态达到消息重试的结果。

MQ 消息重试只针对集群消费方式生效；广播方式不提供失败重试特性，即消费失败后，失败消息不再重试，继续消费新的消息。

重试次数

MQ 默认允许每条消息最多重试 16 次，每次重试的间隔时间如下：

第几次重试	每次重试间隔时间	第几次重试	每次重试间隔时间
1	10 秒	9	7 分钟
2	30 秒	10	8 分钟
3	1 分钟	11	9 分钟
4	2 分钟	12	10 分钟
5	3 分钟	13	20 分钟
6	4 分钟	14	30 分钟
7	5 分钟	15	1 小时
8	6 分钟	16	2 小时

如果消息重试 16 次后仍然失败，消息将不再投递。如果严格按照上述重试时间间隔计算，某条消息在一直消费失败的前提下，将会在接下来的 4 小时 46 分钟之内进行 16 次重试，超过这个时间范围消息将不再重试投递。

注意：一条消息无论重试多少次，这些重试消息的 Message ID 不会改变。

配置方式

消费失败后，重试配置方式

集群消费方式下，消息消费失败后期望消息重试，需要在消息监听器接口的实现中明确进行配置（三种方式任

选一种)：

- 返回 Action.ReconsumeLater (推荐)
- 返回 Null
- 抛出异常

代码示例如下：

```
public class MessageListenerImpl implements MessageListener {

    @Override
    public Action consume(Message message, ConsumeContext context) {
        //方法3：消息处理逻辑抛出异常，消息将重试
        doConsumeMessage(message);
        //方式1：返回 Action.ReconsumeLater，消息将重试
        return Action.ReconsumeLater;
        //方式2：返回 null，消息将重试
        return null;
        //方式3：直接抛出异常，消息将重试
        throw new RuntimeException("Consumer Message exception");
    }
}
```

消费失败后，不重试配置方式

集群消费方式下，消息失败后期望消息不重试，需要捕获消费逻辑中可能抛出的异常，最终返回 Action.CommitMessage，此后这条消息将不会再重试。代码示例如下：

```
public class MessageListenerImpl implements MessageListener {

    @Override
    public Action consume(Message message, ConsumeContext context) {
        try {
            doConsumeMessage(message);
        } catch (Throwable e) {
            //捕获消费逻辑中的所有异常，并返回 Action.CommitMessage;
            return Action.CommitMessage;
        }
        //消息处理正常，直接返回 Action.CommitMessage;
        return Action.CommitMessage;
    }
}
```

自定义消息最大重试次数

自定义 MQ 客户端日志配置，请升级 TCP Java SDK 版本到1.2.2及以上。

MQ 允许 Consumer 启动的时候设置最大重试次数，重试时间间隔将按照如下策略：

- 最大重试次数小于等于16次，则重试时间间隔同上表描述。

- 最大重试次数大于16次，超过16次的重试时间间隔均为每次2小时。

配置方式如下：

```
Properties properties = new Properties();
//配置对应 Consumer ID 的最大消息重试次数为20 次
properties.put(PropertyKeyConst.MaxReconsumeTimes,"20");
Consumer consumer =ONSFactory.createConsumer(properties);
```

注意：

- 消息最大重试次数的设置对相同 Consumer ID 下的所有 Consumer 实例有效。
- 如果只对相同 Consumer ID 下两个 Consumer 实例中的其中一个设置了 MaxReconsumeTimes，那么该配置对两个 Consumer 实例均生效。
- 配置采用覆盖的方式生效，即最后启动的 Consumer 实例会覆盖之前的启动实例的配置。

获取消息重试次数

消费者收到消息后，可按照如下方式获取消息的重试次数：

```
public class MessageListenerImpl implements MessageListener {

    @Override
    public Action consume(Message message, ConsumeContext context) {
        //获取消息的重试次数
        System.out.println(message.getReconsumeTimes());
        return Action.CommitMessage;
    }
}
```

消费幂等

本文档主要强调 MQ 消费者在接收到消息以后根据业务上的唯一 Key 对消息做幂等处理的必要性。

消费幂等的必要性

在互联网应用中，尤其在网络不稳定的情况下，MQ 的消息有可能会出现重复，这个重复简单可以概括为以下两种情况：

发送时消息重复【消息 Message ID 不同】：

MQ Producer 发送消息场景下，消息已成功发送到服务端并完成持久化，此时网络闪断或者客户端宕机导致服务端应答给客户端失败。如果此时 MQ Producer 意识到消息发送失败并尝试再次发送消

息，MQ 消费者后续会收到两条内容相同但是 Message ID 不同的消息。

投递时消息重复【消息 Message ID 相同】；

MQ Consumer 消费消息场景下，消息已投递到消费者并完成业务处理，当客户端给服务端反馈应答的时候网络闪断。为了保证消息至少被消费一次，MQ 服务端将在网络恢复后再次尝试投递之前已被处理过的消息，MQ 消费者后续会收到两条内容相同并且 Message ID 也相同的消息。

处理建议

基于上述第一种原因，内容相同的消息 Message ID 可能会不同，真正安全的幂等处理，不建议以 Message ID 作为处理依据。最好的方式是以业务唯一标识作为幂等处理的关键依据，而业务的唯一标识可以通过消息 Key 进行设置：

```
Message message = new Message();
message.setKey("ORDERID_100");
SendResult sendResult = producer.send(message);
```

订阅方收到消息时可以根据消息的 Key 进行幂等处理：

```
consumer.subscribe("ons_test", "*", new MessageListener() {
    public Action consume(Message message, ConsumeContext context) {
        String key = message.getKey()
        // 根据业务唯一标识的 key 做幂等处理
    }
});
```

TCP 接入

TCP 接入说明

TCP 协议多语言 SDK 支持

目前 TCP 协议接入方式提供 Java，C/C++，.NET 的 SDK。如果需要使用其他语言，建议您使用 MQ Kafka 消息接入。

TCP 协议接入域名

环境说明	接入点
公共云内网接入（阿里云经典网络/VPC）： 华东1、华东2、华北1、华北2、华南1、香港	http://onsaddr-internal.aliyun.com:8080/rocketmq/nsaddr4client-internal
公共云公网接入	http://onsaddr-internet.aliyun.com/rocketmq/nsaddr4client-internet
公共云 Region：亚太东南1(新加坡)	http://ap-southeastaddr-internal.aliyun.com:8080/rocketmq/nsaddr4client-internal
公共云 Region：亚太东南3(吉隆坡)	http://ons-ap-southeast-3-internal.aliyun.com:8080/rocketmq/nsaddr4client-internal
公共云 Region：亚太东北 1 (东京)	http://ons-ap-northeast-1-internal.aliyun.com:8080/rocketmq/nsaddr4client-internal
金融云 Region：华东1	http://jbponsaddr-internal.aliyun.com:8080/rocketmq/nsaddr4client-internal
金融云 Region：华东2、华南1	http://mq4finance-sz.addr.aliyun.com:8080/rocketmq/nsaddr4client-internal

Java SDK

Java SDK 使用说明

消息队列（MQ）提供 Java SDK 实现消息发布与消息订阅。本文将详细介绍各接口的相关参数以及各接口的使用说明。

说明：关于 TCP 接入点域名，请参见TCP 接入说明。

消息收发代码示例：

1. 普通消息收发

- 2. 顺序消息收发
- 3. 定时消息收发
- 4. 延时消息收发
- 5. 事务消息收发

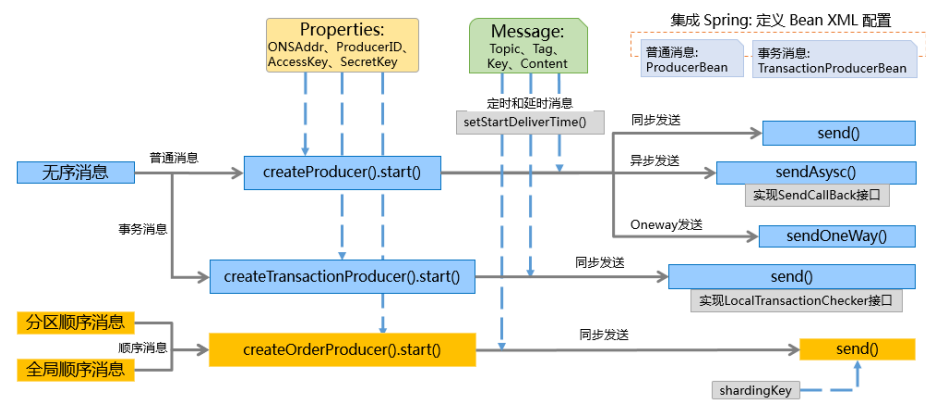
通用参数说明

参数名	参数说明
ONSAAddr	设置 MQ TCP 协议接入点，参考上面表格（推荐）
NAMESRV_ADDR	设置 Name Server 列表（不推荐），与 ONSAddr 二选一
AccessKey	您在阿里云账号管理控制台中创建的 AccessKey，用于身份认证
SecretKey	您在阿里云账号管理控制台中创建的 SecretKey，用于身份认证
OnsChannel	用户渠道，默认为：ALIYUN，聚石塔用户为：CLOUD

发送消息参数说明

参数名	参数说明
ProducerId	您在控制台创建的 Producer ID
SendMsgTimeoutMillis	设置消息发送的超时时间，单位（毫秒），默认：3000
CheckImmunityTimeInSeconds（事务消息）	设置事务消息第一次回查的最快时间，单位（秒）
shardingKey（顺序消息）	顺序消息中用来计算不同分区的值

Java SDK - 发布消息接口以及相关参数说明

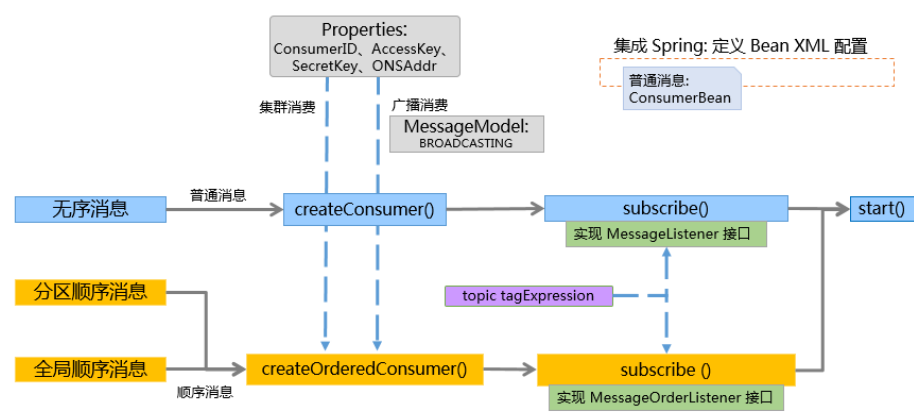


订阅消息参数说明

参数名	参数说明
-----	------

ConsumerId	您在 MQ 控制台上创建的 Consumer ID
MessageModel	设置 Consumer 实例的消费模式，默认为集群消费（值：CLUSTERING）；广播消费（BROADCASTING）
ConsumeThreadNums	设置 Consumer 实例的消费线程数，默认：64
MaxReconsumeTimes	设置消息消费失败的最大重试次数，默认：16
ConsumeTimeout	设置每条消息消费的最大超时时间，超过设置时间则被视为消费失败，等下次重新投递再次消费。每个业务需要设置一个合理的值，单位（分钟）。默认：15
suspendTimeMillis（顺序消息）	只适用于顺序消息，设置消息消费失败的重试间隔时间

Java SDK - 订阅消息接口以及相关参数说明



Java SDK 环境准备

运行本节描述的 Java 代码之前，请按以下说明准备好环境。

通过下面两种方式可以引入依赖(任选一种)：

- Maven 方式引入依赖：

```
<dependency>
  <groupId>com.aliyun.openservices</groupId>
  <artifactId>ons-client</artifactId>
  <version>1.7.0.Final</version>
</dependency>
```

下载依赖 JAR 包：

下载链接

代码里涉及到的 Topic，Producer ID，Consumer ID，需要到 MQ 控制台上创建。Message Tag 可以完全由应用自定义，具体创建过程可参考 [创建资源](#)。

以 TCP 方式接入使用 MQ 服务的应用程序需要部署在同一个地域的 ECS 上。如果需要跨地域使用，请用 HTTP 方式接入。

日志配置

本文档主要介绍 MQ 客户端日志的正常打印方式，MQ 客户端日志格式解析以及如何自定义 MQ 客户端日志配置。

打印 MQ 客户端日志

MQ 客户端日志在问题定位排查中扮演着非常重要的角色，通过日志记录客户端运行过程中的异常，能够帮助尽可能真实的还原某个时间点的异常场景，最终达到快速定位、修复 Bug 的目的。

TCP Java SDK 打印 MQ 客户端日志

MQ 的 TCP Java SDK 基于 SLF4J 接口编程，客户端日志的打印依赖用户在配置文件中指定的日志实现。目前支持 log4j（暂不支持 log4j2）、logback，可在 pom.xml 或者 lib 中添加对应的日志实现依赖即可：

方式一：依赖 log4j 作为日志实现

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

方式二：依赖 logback 作为日志实现

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

注意：应用中同时依赖 log4j 和 logback 的日志实现会造成日志冲突导致客户端日志打印混乱。确保应用只依赖其中一个日志实现，是正确打印 MQ 客户端日志的前提条件，建议通过 `mvn clean dependency:tree | grep log` 命令排查。

MQ 客户端日志默认配置

在应用中添加了唯一的日志实现后启动 MQ 客户端，MQ 客户端将会按照如下的配置生成日志文件：

- 日志保存路径：/{user.home}/logs/ons.log，其中{user.home}是指启动当前 Java 进程的用户的根目录
- 单个日志文件大小：64MB
- 保存历史日志文件的最大个数：10个
- 日志级别：INFO

自定义日志配置

MQ 客户端支持用户自定义日志保存路径、日志级别以及保存历史日志文件的最大个数；考虑到日志传输以及阅读的便利性，暂不允许自定义单个日志文件大小，仍保持默认64MB；可通过配置保存历史日志文件的最大个数来自定义日志文件保存的时间范围。

各个参数配置说明如下：

- 日志保存路径：请确保应用进程有对该路径写的权限，否则日志不会打印。
- 保存历史日志文件的最大个数：支持1到100之前的数值，超出范围或者格式错误默认保存10个。
- 日志级别：支持 ERROR、WARN、INFO、DEBUG 中任何一种，不匹配默认 INFO。

TCP Java SDK 自定义 MQ 客户端日志

自定义 MQ 客户端日志配置，请升级 TCP Java SDK 版本到1.2.5及以上。

在 TCP Java SDK 中自定义 MQ 客户端日志配置，请设置如下系统参数：

- `ons.client.logRoot`：日志保存路径
- `ons.client.logFileMaxIndex`：保存历史日志文件的最大个数

- ons.client.logLevel : 日志级别

举例说明，可在启动脚本中或者 IDE 的 VM options 中添加如下系统参数：

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

Spring 集成

本文介绍如何在 Spring 框架下用 MQ 收发消息。主要包括三部分内容：普通消息生产者和 Spring 集成，事务消息生产者和 Spring 集成，消息消费者与 Spring 集成。

请确保同一个 Consumer ID 下所有 Consumer 实例的订阅关系保持一致，具体请参考订阅关系一致文档。

Spring 框架下支持的配置参数和 TCP Java 一致，具体可以参考Java SDK 使用说明。

生产者与 Spring 集成

在 producer.xml 中定义生产者 Bean 等信息。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="producer" class="com.aliyun.openservices.ons.api.bean.ProducerBean" init-method="start"
destroy-method="shutdown">
        <!-- Spring 接入方式支持 Java SDK 支持的所有配置项 -->
        <property name="properties" > <!--生产者配置信息-->
            <props>
                <prop key="ProducerId">PID_DEMO</prop> <!--请替换 XXX-->
                <prop key="AccessKey">XXX</prop>
                <prop key="SecretKey">XXX</prop>
            </props>
        </property>
    </bean>

</beans>
```

通过已经与 Spring 集成好的生产者生产消息。

```
package demo;

import com.aliyun.openservices.ons.api.Message;
```

```

import com.aliyun.openservices.ons.api.Producer;
import com.aliyun.openservices.ons.api.SendResult;
import com.aliyun.openservices.ons.api.exception.ONSClientException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class ProduceWithSpring {
    public static void main(String[] args) {
        /**
         * 生产者 Bean 配置在 producer.xml 中,可通过 ApplicationContext 获取或者直接注入到其他类(比如具体的
         的 Controller)中.
         */
        ApplicationContext context = new ClassPathXmlApplicationContext("producer.xml");

        Producer producer = (Producer) context.getBean("producer");
        //循环发送消息
        for (int i = 0; i < 100; i++) {
            Message msg = new Message( //
                // Message 所属的 Topic
                "TopicTestMQ",
                // Message Tag 可理解为 Gmail 中的标签, 对消息进行再归类, 方便 Consumer 指定过滤条件在
                MQ 服务器过滤
                "TagA",
                // Message Body 可以是任何二进制形式的数据, MQ 不做任何干预
                // 需要 Producer 与 Consumer 协商好一致的序列化和反序列化方式
                "Hello MQ".getBytes());
            // 设置代表消息的业务关键属性, 请尽可能全局唯一
            // 以方便您在无法正常收到消息情况下, 可通过 MQ 控制台查询消息并补发
            // 注意: 不设置也不会影响消息正常收发
            msg.setKey("ORDERID_100");
            // 发送消息, 只要不抛异常就是成功
            try {
                SendResult sendResult = producer.send(msg);
                assert sendResult != null;
                System.out.println("send success: " + sendResult.getMessageId());
            } catch (ONSClientException e) {
                System.out.println("发送失败");
            }
        }
    }
}

```

事务消息生产者与 Spring 集成

有关事务消息的概念, 请参见收发事务消息。

首先需要实现一个 LocalTransactionChecker, 如下所示。一个消息生产者只能有一个 LocalTransactionChecker。

```
package demo;
```

```
import com.aliyun.openservices.ons.api.Message;
import com.aliyun.openservices.ons.api.transaction.LocalTransactionChecker;
import com.aliyun.openservices.ons.api.transaction.TransactionStatus;

public class DemoLocalTransactionChecker implements LocalTransactionChecker {
    public TransactionStatus check(Message msg) {
        System.out.println("开始回查本地事务状态");
        return TransactionStatus.CommitTransaction; //根据本地事务状态检查结果返回不同的 TransactionStatus
    }
}
```

其次，在 transactionProducer.xml 中定义事务消息生产者 Bean 等信息。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="localTransactionChecker" class="demo.DemoLocalTransactionChecker"></bean>

    <bean id="transactionProducer"
class="com.aliyun.openservices.ons.api.bean.TransactionProducerBean" init-method="start" destroy-
method="shutdown">
        <property name="properties" > <!--事务消息生产者配置信息-->
            <props>
                <prop key="ProducerId">PID_DEMO</prop> <!--请替换 XXX-->
                <prop key="AccessKey">AKDEMO</prop>
                <prop key="SecretKey">SKDEMO</prop>
            </props>
        </property>
        <property name="localTransactionChecker" ref="localTransactionChecker"></property>
    </bean>

</beans>
```

通过已经与 Spring 集成好的生产者生产事务消息。

```
package demo;

import com.aliyun.openservices.ons.api.Message;
import com.aliyun.openservices.ons.api.SendResult;
import com.aliyun.openservices.ons.api.transaction.LocalTransactionExecuter;
import com.aliyun.openservices.ons.api.transaction.TransactionProducer;
import com.aliyun.openservices.ons.api.transaction.TransactionStatus;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class ProduceTransMsgWithSpring {

    public static void main(String[] args) {
        /**
         * 事务消息生产者 Bean 配置在 transactionProducer.xml 中,可通过 ApplicationContext 获取或者直接注
```

```

入到其他类(比如具体的 Controller)中.
    * 请结合例子"发送事务消息"
    */
    ApplicationContext context = new ClassPathXmlApplicationContext("transactionProducer.xml");

    TransactionProducer transactionProducer = (TransactionProducer)
context.getBean("transactionProducer");

    Message msg = new Message("XXX", "TagA", "Hello MQ transaction===".getBytes());

    SendResult sendResult = transactionProducer.send(msg, new LocalTransactionExecuter() {
        @Override
        public TransactionStatus execute(Message msg, Object arg) {
            System.out.println("执行本地事务");
            return TransactionStatus.CommitTransaction; //根据本地事务执行结果来返回不同的
TransactionStatus
        }
    }, null);
    }
}

```

消费者与 Spring 集成

创建 MessageListener，如下所示。

```

package demo;

import com.aliyun.openservices.ons.api.Action;
import com.aliyun.openservices.ons.api.ConsumeContext;
import com.aliyun.openservices.ons.api.Message;
import com.aliyun.openservices.ons.api.MessageListener;

public class DemoMessageListener implements MessageListener {

    public Action consume(Message message, ConsumeContext context) {

        System.out.println("Receive: " + message.getMsgID());
        try {
            //do something..
            return Action.CommitMessage;
        } catch (Exception e) {
            //消费失败
            return Action.ReconsumeLater;
        }
    }
}

```

在 consumer.xml 中定义消费者 Bean 等信息。

```
<?xml version="1.0" encoding="UTF-8"?>
```



```

<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="msgListener" class="demo.DemoMessageListener"></bean> <!--Listener 配置-->
  <!-- 多 CID 订阅同一个 Topic，可以创建多个 ConsumerBean-->
  <bean id="consumer" class="com.aliyun.openservices.ons.api.bean.ConsumerBean" init-
method="start" destroy-method="shutdown">
    <property name="properties" > <!--消费者配置信息-->
      <props>
        <prop key="ConsumerId">CID_DEMO</prop> <!--请替换 XXX-->
        <prop key="AccessKey">AKDEMO</prop>
        <prop key="SecretKey">SKDEMO</prop>
        <!--将消费者线程数固定为50个
        <prop key="ConsumeThreadNums">50</prop>
        -->
      </props>
    </property>
    <property name="subscriptionTable">
      <map>
        <entry value-ref="msgListener">
          <key>
            <bean class="com.aliyun.openservices.ons.api.bean.Subscription">
              <property name="topic" value="TopicTestMQ"/>
              <property name="expression" value=""/> <!--expression 即 Tag，可以设置成具体的
Tag，如 taga||tagb||tagc，也可设置成*。*仅代表订阅所有 Tag，不支持通配-->
            </bean>
          </key>
        </entry>
        <!--更多的订阅添加 entry 节点即可，如下所示-->
        <entry value-ref="msgListener">
          <key>
            <bean class="com.aliyun.openservices.ons.api.bean.Subscription">
              <property name="topic" value="TopicTestMQ-Other"/> <!--订阅另外一个 Topic -->
              <property name="expression" value="taga||tagb"/> <!-- 订阅多个 Tag -->
            </bean>
          </key>
        </entry>
      </map>
    </property>
  </bean>

</beans>

```

运行已经与 Spring 集成好的消费者，如下所示。

```

package demo;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class ConsumeWithSpring {
    public static void main(String[] args) {

```

```
/**
 * 消费者 Bean 配置在 consumer.xml 中，可通过 ApplicationContext 获取或者直接注入到其他类(比如具
体的 Controller)中
 */
ApplicationContext context = new ClassPathXmlApplicationContext("consumer.xml");
System.out.println("Consumer Started");
}
}
```

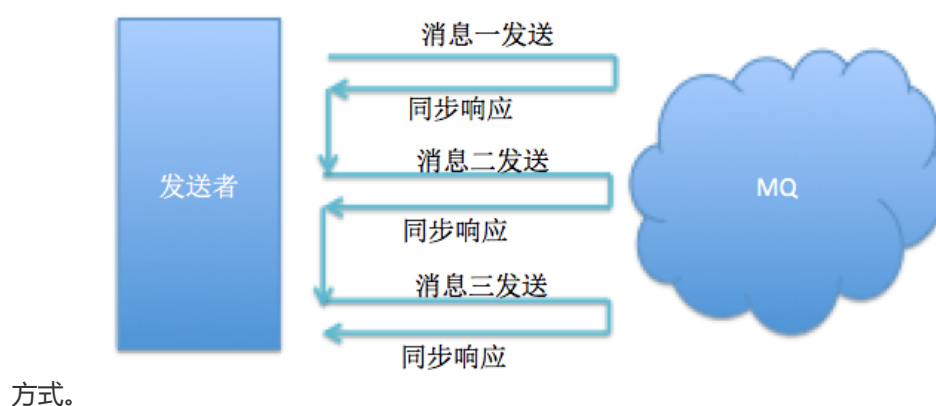
发送消息（三种方式）

MQ 发送消息有三种实现方式：可靠同步发送、可靠异步发送、单向(Oneway)发送。本文介绍了每种实现的原
理、使用场景以及三种实现的异同，同时提供了代码示例以供参考。

注意：顺序消息只支持可靠同步发送。

可靠同步发送

原理：同步发送是指消息发送方发出数据后，会在收到接收方发回响应之后才发下一个数据包的通讯



应用场景：此种方式应用场景非常广泛，例如重要通知邮件、报名短信通知、营销短信系统等。

可靠异步发送

原理：异步发送是指发送方发出数据后，不等接收方发回响应，接着发送下个数据包的通讯方式。

MQ 的异步发送，需要用户实现异步发送回调接口（SendCallback），在执行消息的异步发送时，应用不需要等待服务器响应即可直接返回，通过回调接口接收服务器响应，并对服务器的响应结果进行处理。



应用场景：异步发送一般用于链路耗时较长，对 RT 响应时间较为敏感的业务场景，例如用户视频上传后通知启动转码服务，转码完成后通知推送转码结果等。

单向（Oneway）发送

原理：单向（Oneway）发送特点为只负责发送消息，不等待服务器回应且没有回调函数触发，即只发送请求不等待应答。此方式发送消息的过程耗时非常短，一般在微秒级别。



应用场景：适用于某些耗时非常短，但对可靠性要求并不高的场景，例如日志收集。

下表概括了三者的特点和主要区别。

发送方式	发送 TPS	发送结果反馈	可靠性
同步发送	快	有	不丢失
异步发送	快	有	不丢失
单向发送	最快	无	可能丢失

示例代码

TCP 接入点域名，请前往查看。

同步发送

```

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

import java.util.Properties;

public class ProducerTest {
    public static void main(String[] args) {
        Properties properties = new Properties();
        //您在控制台创建的 Producer ID
        properties.put(PropertyKeyConst.ProducerId, "XXX");
        // AccessKey 阿里云身份验证，在阿里云服务器管理控制台创建
        properties.put(PropertyKeyConst.AccessKey, "XXX");
        // SecretKey 阿里云身份验证，在阿里云服务器管理控制台创建
        properties.put(PropertyKeyConst.SecretKey, "XXX");
        //设置发送超时时间，单位毫秒
        properties.setProperty(PropertyKeyConst.SendMsgTimeoutMillis, "3000");
        // 设置 TCP 接入域名（此处以公共云生产环境为例）
        properties.put(PropertyKeyConst.ONSDAddr,
            "http://onsaddr-internal.aliyun.com:8080/rocketmq/nsaddr4client-internal");

        Producer producer = ONSFactory.createProducer(properties);
        // 在发送消息前，必须调用 start 方法来启动 Producer，只需调用一次即可
        producer.start();

        //循环发送消息
        for (int i = 0; i < 100; i++){
            Message msg = new Message( //
                // Message 所属的 Topic
                "TopicTestMQ",
                // Message Tag 可理解为 Gmail 中的标签，对消息进行再归类，方便 Consumer 指定过滤条件在 MQ 服务器过
                // 滤
                "TagA",
                // Message Body 可以是任何二进制形式的数据，MQ 不做任何干预，
                // 需要 Producer 与 Consumer 协商好一致的序列化和反序列化方式
                "Hello MQ".getBytes());
            // 设置代表消息的业务关键属性，请尽可能全局唯一。
            // 以方便您在无法正常收到消息情况下，可通过阿里云服务器管理控制台查询消息并补发
            // 注意：不设置也不会影响消息正常收发
            msg.setKey("ORDERID_" + i);

            try {
                SendResult sendResult = producer.send(msg);
                // 同步发送消息，只要不抛异常就是成功
                if (sendResult != null) {
                    System.out.println(new Date() + " Send mq message success. Topic is:" + msg.getTopic() + " msgId is: "
                        + sendResult.getMessageId());
                }
            }
            catch (Exception e) {
                // 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(new Date() + " Send mq message failed. Topic is:" + msg.getTopic());
        e.printStackTrace();
    }
}

// 在应用退出前，销毁 Producer 对象
// 注意：如果不销毁也没有问题
producer.shutdown();
}
}

```

异步发送

```

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

import java.util.Properties;

public static void main(String[] args) {
    Properties properties = new Properties();
    // AccessKey 阿里云身份验证，在阿里云服务器管理控制台创建
    properties.put(PropertyKeyConst.AccessKey, "DEMO_AK");
    // SecretKey 阿里云身份验证，在阿里云服务器管理控制台创建
    properties.put(PropertyKeyConst.SecretKey, "DEMO_SK");
    //您在控制台创建的 Producer ID
    properties.put(PropertyKeyConst.ProducerId, "DEMO_PID");
    //设置发送超时时间，单位毫秒
    properties.setProperty(PropertyKeyConst.SendMsgTimeoutMillis, "3000");
    // 设置 TCP 接入域名（此处以公共云生产环境为例）
    properties.put(PropertyKeyConst.ONSDAddr,
        "http://onsaddr-internal.aliyun.com:8080/rocketmq/nsaddr4client-internal");

    Producer producer = ONSFactory.createProducer(properties);
    // 在发送消息前，必须调用 start 方法来启动 Producer，只需调用一次即可。
    producer.start();

    Message msg = new Message(
        // Message 所属的 Topic
        "TopicTestMQ",
        // Message Tag,可理解为 Gmail 中的标签，对消息进行再归类，方便 Consumer 指定过滤条件在 MQ 服务器过
        // 滤
        "TagA",
        // Message Body，任何二进制形式的数据，MQ 不做任何干预，需要 Producer 与 Consumer 协商好一致的序列化
        // 化和反序列化方式
        "Hello MQ".getBytes());

    // 设置代表消息的业务关键属性，请尽可能全局唯一。以方便您在无法正常收到消息情况下，可通过 MQ 控制台查询消息并补发。
    // 注意：不设置也不会影响消息正常收发
    msg.setKey("ORDERID_100");
}

```

```

// 异步发送消息, 发送结果通过 callback 返回给客户端。
producer.sendAsync(msg, new SendCallback() {
    @Override
    public void onSuccess(final SendResult sendResult) {
        // 消费发送成功
        System.out.println("send message success. topic=" + sendResult.getTopic() + ", msgId=" +
sendResult.getMessageId());
    }

    @Override
    public void onException(OnExceptionContext context) {
        // 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. topic=" + context.getTopic() + ", msgId=" +
context.getMessageId());
    }
});

// 在 callback 返回之前即可取得 msgId。
System.out.println("send message async. topic=" + msg.getTopic() + ", msgId=" + msg.getMsgID());

// 在应用退出前, 销毁 Producer 对象。注意: 如果不销毁也没有问题
producer.shutdown();
}

```

单向(Oneway)发送

```

import com.aliyun.openservices.ons.api.Message;
import com.aliyun.openservices.ons.api.Producer;
import com.aliyun.openservices.ons.api.ONSFactory;
import com.aliyun.openservices.ons.api.PropertyKeyConst;

import java.util.Properties;

public static void main(String[] args) {
    Properties properties = new Properties();
    // AccessKey 阿里云身份验证, 在阿里云服务器管理控制台创建
    properties.put(PropertyKeyConst.AccessKey, "DEMO_AK");
    // SecretKey 阿里云身份验证, 在阿里云服务器管理控制台创建
    properties.put(PropertyKeyConst.SecretKey, "DEMO_SK");
    //您在控制台创建的 Producer ID
    properties.put(PropertyKeyConst.ProducerId, "DEMO_PID");
    //设置发送超时时间, 单位毫秒
    properties.setProperty(PropertyKeyConst.SendMsgTimeoutMillis, "3000");
    // 设置 TCP 接入域名 (此处以公共云生产环境为例)
    properties.put(PropertyKeyConst.ONSAAddr,
"http://onsaddr-internal.aliyun.com:8080/rocketmq/nsaddr4client-internal");

    Producer producer = ONSFactory.createProducer(properties);
    // 在发送消息前, 必须调用 start 方法来启动 Producer, 只需调用一次即可。
    producer.start();
    //循环发送消息
    for (int i = 0; i < 100; i++){
        Message msg = new Message(
            // Message 所属的 Topic

```

```

        "TopicTestMQ",
        // Message Tag,
        // 可理解为 Gmail 中的标签，对消息进行再归类，方便 Consumer 指定过滤条件在 MQ 服务器过滤
        "TagA",
        // Message Body
        // 任何二进制形式的数据，MQ 不做任何干预，需要 Producer 与 Consumer 协商好一致的序列化和反序列化
方式
        "Hello MQ".getBytes());

// 设置代表消息的业务关键属性，请尽可能全局唯一。
// 以方便您在无法正常收到消息情况下，可通过阿里云服务器管理控制台查询消息并补发。
// 注意：不设置也不会影响消息正常收发
msg.setKey("ORDERID_" + i);

// oneway 方式发送消息，因为没有请求应答处理，一旦出现消息发送失败会因为没有进行重试而导致数据丢失，若
数据不可丢，建议选用可靠同步或可靠异步发送方式。
producer.sendOneway(msg);
}

// 在应用退出前，销毁 Producer 对象
// 注意：如果不销毁也没有问题
producer.shutdown();
}

```

发送消息（多线程）

MQ 的消费者和生产者客户端对象是线程安全的，可以在多个线程之间共享使用。

您可以在服务器上（或者多台服务器）部署多个生产端和消费端实例，也可以在生产端和消费端采用多线程发送或接收消息，从而提高生产端和消费端的消息发送或接收 TPS。请避免为每个线程创建一个客户端实例。

说明：关于 TCP 接入点域名，请参见TCP 接入说明。

在多线程之间共享 Producer 的示例代码如下：

```

import com.aliyun.openservices.ons.api.Message;
import com.aliyun.openservices.ons.api.Producer;
import com.aliyun.openservices.ons.api.ONSFactory;
import com.aliyun.openservices.ons.api.PropertyKeyConst;

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

public class SharedProducer {
    public static void main(String[] args) {
        // producer 实例配置初始化
        Properties properties = new Properties();
        //您在控制台创建的 Producer ID
    }
}

```

```

properties.put(PropertyKeyConst.ProducerId, "XXX");
// AccessKey 阿里云身份验证, 在阿里云服务器管理控制台创建
properties.put(PropertyKeyConst.AccessKey, "XXX");
// SecretKey 阿里云身份验证, 在阿里云服务器管理控制台创建
properties.put(PropertyKeyConst.SecretKey, "XXX");
//设置发送超时时间, 单位毫秒
properties.setProperty(PropertyKeyConst.SendMsgTimeoutMillis, "3000");
// 设置 TCP 接入域名 (此处以公共云生产环境为例)
properties.put(PropertyKeyConst.ONSAAddr,
    "http://onsaddr-internal.aliyun.com:8080/rocketmq/nsaddr4client-internal");
final Producer producer = ONSFactory.createProducer(properties);
// 在发送消息前, 必须调用 start 方法来启动 Producer, 只需调用一次即可
producer.start();

//创建的 Producer 和 Consumer 对象为线程安全的, 可以在多线程间进行共享, 避免每个线程创建一个实例。

final Message msg = new Message( //
    // Message 所属的 Topic
    "TopicTestMQ",
    // Message Tag 可理解为 Gmail 中的标签, 对消息进行再归类, 方便 Consumer 指定过滤条件在 MQ 服务器过滤
    "TagA",
    // Message Body 可以是任何二进制形式的数据, MQ 不做任何干预,
    // 需要 Producer 与 Consumer 协商好一致的序列化和反序列化方式
    "Hello MQ".getBytes());

//在 thread 和 anotherThread 中共享 producer 对象, 并发地发送消息至 MQ。
Thread thread = new Thread(new Runnable() {
    @Override
    public void run() {
        try {
            SendResult sendResult = producer.send(message);
            // 同步发送消息, 只要不抛异常就是成功
            if (sendResult != null) {
                System.out.println(new Date() + " Send mq message success. Topic is:" + MqConfig.TOPIC + " msgId
is: " + sendResult.getMessageId());
            }
        } catch (Exception e) {
            // 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(new Date() + " Send mq message failed. Topic is:" + MqConfig.TOPIC);
            e.printStackTrace();
        }
    }
});
thread.start();

Thread anotherThread = new Thread(new Runnable() {
    @Override
    public void run() {
        try {
            SendResult sendResult = producer.send(message);
            // 同步发送消息, 只要不抛异常就是成功
            if (sendResult != null) {
                System.out.println(new Date() + " Send mq message success. Topic is:" + MqConfig.TOPIC + " msgId
is: " + sendResult.getMessageId());
            }
        }
    }
});
anotherThread.start();

```



```
        } catch (Exception e) {
            // 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(new Date() + " Send mq message failed. Topic is:" + MqConfig.TOPIC);
            e.printStackTrace();
        }
    }
});
anotherThread.start();

// producer 实例若不再使用时，可将 producer 关闭，进行资源释放
// producer.shutdown();
}
}
```

订阅消息

本文介绍如何通过 MQ SDK 使用 Java 语言进行消息订阅。

说明：

- 请确保同一个 Consumer ID 下所有 Consumer 实例的订阅关系保持一致，详见订阅关系一致。
- 关于 TCP 接入点域名，请参见TCP 接入说明。

订阅方式

MQ 支持以下两种订阅方式：

集群订阅：同一个 Consumer ID 所标识的所有 Consumer 平均分摊消费消息。例如某个 Topic 有 9 条消息，一个 Consumer ID 有 3 个 Consumer 实例，那么在集群消费模式下每个实例平均分摊，只消费其中的 3 条消息。

```
// 集群订阅方式设置（不设置的情况下，默认为集群订阅方式）
properties.put(PropertyKeyConst.MessageModel, PropertyValueConst.CLUSTERING);
```

广播订阅：同一个 Consumer ID 所标识的所有 Consumer 都会各自消费某条消息一次。例如某个 Topic 有 9 条消息，一个 Consumer ID 有 3 个 Consumer 实例，那么在广播消费模式下每个实例都会各自消费 9 条消息。

```
// 广播订阅方式设置
properties.put(PropertyKeyConst.MessageModel, PropertyValueConst.BROADCASTING);
```

示例代码

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

import java.util.Properties;

public class ConsumerTest {
    public static void main(String[] args) {
        Properties properties = new Properties();
        // 您在控制台创建的 Consumer ID
        properties.put(PropertyKeyConst.ConsumerId, "XXX");
        // AccessKey 阿里云身份验证，在阿里云服务器管理控制台创建
        properties.put(PropertyKeyConst.AccessKey, "XXX");
        // SecretKey 阿里云身份验证，在阿里云服务器管理控制台创建
        properties.put(PropertyKeyConst.SecretKey, "XXX");
        // 设置 TCP 接入域名（此处以公共云生产环境为例）
        properties.put(PropertyKeyConst.ONSDAddr,
            "http://onsaddr-internal.aliyun.com:8080/rocketmq/nsaddr4client-internal");
        // 集群订阅方式（默认）
        // properties.put(PropertyKeyConst.MessageModel, PropertyValueConst.CLUSTERING);
        // 广播订阅方式
        // properties.put(PropertyKeyConst.MessageModel, PropertyValueConst.BROADCASTING);

        Consumer consumer = ONSFactory.createConsumer(properties);
        consumer.subscribe("TopicTestMQ", "TagA||TagB", new MessageListener() { //订阅多个 Tag
            public Action consume(Message message, ConsumeContext context) {
                System.out.println("Receive: " + message);
                return Action.CommitMessage;
            }
        });

        //订阅另外一个 Topic
        consumer.subscribe("TopicTestMQ-Other", "**", new MessageListener() { //订阅全部 Tag
            public Action consume(Message message, ConsumeContext context) {
                System.out.println("Receive: " + message);
                return Action.CommitMessage;
            }
        });

        consumer.start();
        System.out.println("Consumer Started");
    }
}
```

注意：

- 在广播消费模式下，控制台无法设置消息堆积报警，也无法进行消息堆积查询。因此，您也可以创建多个 Consumer ID 来达到广播模式的效果，请参见集群消费和广播消费中《使用集群模式模拟广播》一节。
- 关于 MQ 消费端流控的最佳实践，请参见MQ 客户端流控设计。

收发顺序消息

发送和订阅顺序消息，请使用 Java SDK 1.2.7 及以上版本。

顺序消息是 MQ 提供的一种按照顺序进行发布消费的消息类型，适用由于需要严格按照先进先出的原则进行消息发布和消费的场景，详见顺序消息。

全局顺序消息和分区顺序消息的收发方式基本一样，具体请参考以下示例代码。

说明：关于 TCP 接入点域名，请参见TCP 接入说明。

发送顺序消息

发送消息的示例代码如下：

```
package com.aliyun.openservices.ons.example.order;

import com.aliyun.openservices.ons.api.Message;
import com.aliyun.openservices.ons.api.ONSType;
import com.aliyun.openservices.ons.api.PropertyKeyConst;
import com.aliyun.openservices.ons.api.SendResult;
import com.aliyun.openservices.ons.api.order.OrderProducer;

import java.util.Properties;

public class ProducerClient {

    public static void main(String[] args) {
        Properties properties = new Properties();
        // 您在控制台创建的 Producer ID
        properties.put(PropertyKeyConst.ProducerId, "XXX");
        // AccessKey 阿里云身份验证，在阿里云服务器管理控制台创建
        properties.put(PropertyKeyConst.AccessKey, "XXX");
        // SecretKey 阿里云身份验证，在阿里云服务器管理控制台创建
        properties.put(PropertyKeyConst.SecretKey, "XXX");
        // 设置 TCP 接入域名（此处以公共云生产环境为例）
        properties.put(PropertyKeyConst.ONSType,
            "http://onsaddr-internal.aliyun.com:8080/rocketmq/nsaddr4client-internal");
        OrderProducer producer = ONSType.createOrderProducer(properties);
        // 在发送消息前，必须调用 start 方法来启动 Producer，只需调用一次即可。
```

```

producer.start();
for (int i = 0; i < 1000; i++) {
    String orderId = "biz_" + i % 10;
    Message msg = new Message(//
        // Message 所属的 Topic
        "Order_global_topic",
        // Message Tag, 可理解为 Gmail 中的标签, 对消息进行再归类, 方便 Consumer 指定过滤条件在 MQ 服务器
过滤
        "TagA",
        // Message Body 可以是任何二进制形式的数据, MQ 不做任何干预, 需要 Producer 与 Consumer 协商好一
一致的序列化和反序列化方式
        "send order global msg".getBytes()
    );
    // 设置代表消息的业务关键属性, 请尽可能全局唯一。
    // 以方便您在无法正常收到消息情况下, 可通过 MQ 控制台查询消息并补发。
    // 注意: 不设置也不会影响消息正常收发
    msg.setKey(orderId);
    // 分区顺序消息中区分不同分区的关键字段, sharding key 于普通消息的 key 是完全不同的概念。
    // 全局顺序消息, 该字段可以设置为任意非空字符串。
    String shardingKey = String.valueOf(orderId);
    try {
        SendResult sendResult = producer.send(msg, shardingKey);
        // 发送消息, 只要不抛异常就是成功
        if (sendResult != null) {
            System.out.println(new Date() + " Send mq message success. Topic is:" + msg.getTopic() + " msgId is: "
+ sendResult.getMessageId());
        }
    }
    catch (Exception e) {
        // 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(new Date() + " Send mq message failed. Topic is:" + msg.getTopic());
        e.printStackTrace();
    }
}
// 在应用退出前, 销毁 Producer 对象
// 注意: 如果不销毁也没有问题
producer.shutdown();
}
}

```

订阅顺序消息

订阅顺序消息的示例代码如下：

```

package com.aliyun.openservices.ons.example.order;

import com.aliyun.openservices.ons.api.Message;
import com.aliyun.openservices.ons.api.ONSFactory;
import com.aliyun.openservices.ons.api.PropertyKeyConst;
import com.aliyun.openservices.ons.api.order.ConsumeOrderContext;
import com.aliyun.openservices.ons.api.order.MessageOrderListener;
import com.aliyun.openservices.ons.api.order.OrderAction;
import com.aliyun.openservices.ons.api.order.OrderConsumer;

```

```

import java.util.Properties;

public class ConsumerClient {

    public static void main(String[] args) {
        Properties properties = new Properties();
        // 您在控制台创建的 Consumer ID
        properties.put(PropertyKeyConst.ConsumerId, "XXX");
        // AccessKey 阿里云身份验证，在阿里云服务器管理控制台创建
        properties.put(PropertyKeyConst.AccessKey, "XXX");
        // SecretKey 阿里云身份验证，在阿里云服务器管理控制台创建
        properties.put(PropertyKeyConst.SecretKey, "XXX");
        // 设置 TCP 接入域名（此处以公共云生产环境为例）
        properties.put(PropertyKeyConst.ONSDAddr,
            "http://onsaddr-internal.aliyun.com:8080/rocketmq/nsaddr4client-internal");
        // 顺序消息消费失败进行重试前的等待时间 单位(毫秒)
        properties.put(PropertyKeyConst.SuspendTimeMillis, "100");
        // 消息消费失败时的最大重试次数
        properties.put(PropertyKeyConst.MaxReconsumeTimes, "20");

        // 在订阅消息前，必须调用 start 方法来启动 Consumer，只需调用一次即可。
        OrderConsumer consumer = ONSFactory.createOrderedConsumer(properties);

        consumer.subscribe(
            // Message 所属的 Topic
            "Jodie_Order_Topic",
            // 订阅指定 Topic 下的 Tags：
            // 1. * 表示订阅所有消息
            // 2. TagA || TagB || TagC 表示订阅 TagA 或 TagB 或 TagC 的消息
            "*",
            new MessageOrderListener() {
                /**
                 * 1. 消息消费处理失败或者处理出现异常，返回 OrderAction.Suspend<br>
                 * 2. 消息处理成功，返回与返回 OrderAction.Success
                 */
                @Override
                public OrderAction consume(Message message, ConsumeOrderContext context) {
                    System.out.println(message);
                    return OrderAction.Success;
                }
            }
        );

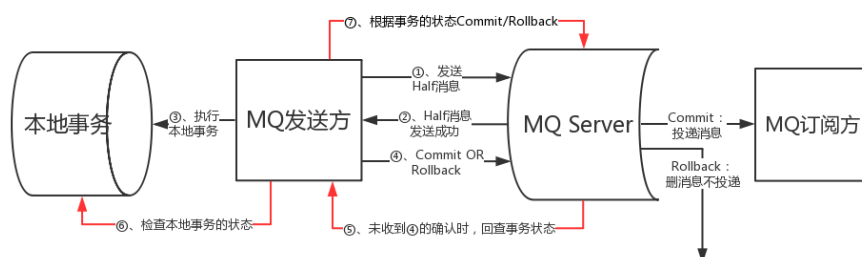
        consumer.start();
    }
}

```

收发事务消息

交互流程

MQ 的事务消息交互流程如下图所示：



说明：关于 TCP 接入点域名，请参见 TCP 接入说明。

发送事务消息

发送事务消息包含以下两个步骤：

发送半消息及执行本地事务。示例代码如下：

```
package com.alibaba.webx.TryHsf.app1;

import com.aliyun.openservices.ons.api.Message;
import com.aliyun.openservices.ons.api.PropertyKeyConst;
import com.aliyun.openservices.ons.api.SendResult;
import com.aliyun.openservices.ons.api.transaction.LocalTransactionExecuter;
import com.aliyun.openservices.ons.api.transaction.TransactionProducer;
import com.aliyun.openservices.ons.api.transaction.TransactionStatus;
import java.util.Properties;
import java.util.concurrent.TimeUnit;

public class TransactionProducerClient {
    private final static Logger log = ClientLogger.getLog(); // 用户需要设置自己的 log, 记录日志便于排查问题

    public static void main(String[] args) throws InterruptedException {
        final BusinessService businessService = new BusinessService(); // 本地业务 Service
        Properties properties = new Properties();
        // 您在控制台创建的 Producer ID。注意：事务消息的 Producer ID 不能与其他类型消息的 Producer ID 共用
        properties.put(PropertyKeyConst.ProducerId, "");
        // 阿里云身份验证，在阿里云服务器管理控制台创建
        properties.put(PropertyKeyConst.AccessKey, "");
        // 阿里云身份验证，在阿里云服务器管理控制台创建
        properties.put(PropertyKeyConst.SecretKey, "");
        // 设置 TCP 接入域名（此处以公共云生产环境为例）
        properties.put(PropertyKeyConst.ONSAddr,
```

```

"http://onsaddr-internal.aliyun.com:8080/rocketmq/nsaddr4client-internal");

TransactionProducer producer = ONSFactory.createTransactionProducer(properties,
    new LocalTransactionCheckerImpl());
producer.start();
Message msg = new Message("Topic", "TagA", "Hello MQ transaction===".getBytes());
try {
    SendResult sendResult = producer.send(msg, new LocalTransactionExecuter() {
        @Override
        public TransactionStatus execute(Message msg, Object arg) {
            // 消息 ID(有可能消息体一样, 但消息 ID 不一样, 当前消息 ID 在控制台无法查询)
            String msgId = msg.getMsgID();
            // 消息体内容进行 crc32, 也可以使用其它的如 MD5
            long crc32Id = HashUtil.crc32Code(msg.getBody());
            // 消息 ID 和 crc32id 主要是用来防止消息重复
            // 如果业务本身是幂等的, 可以忽略, 否则需要利用 msgId 或 crc32Id 来做幂等
            // 如果要求消息绝对不重复, 推荐做法是对消息体 body 使用 crc32或 md5来防止重复消息
            Object businessServiceArgs = new Object();
            TransactionStatus transactionStatus = TransactionStatus.Unknow;
            try {
                boolean isCommit =
                    businessService.execbusinessService(businessServiceArgs);
                if (isCommit) {
                    // 本地事务成功、提交消息
                    transactionStatus = TransactionStatus.CommitTransaction;
                } else {
                    // 本地事务失败、回滚消息
                    transactionStatus = TransactionStatus.RollbackTransaction;
                }
            } catch (Exception e) {
                log.error("Message Id:{}, msgId, e);
            }
            System.out.println(msg.getMsgID());
            log.warn("Message Id:{}transactionStatus:{}, msgId, transactionStatus.name());
            return transactionStatus;
        }
    }, null);
} catch (Exception e) {
    // 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(new Date() + " Send mq message failed. Topic is:" + msg.getTopic());
    e.printStackTrace();
}
// demo example 防止进程退出(实际使用不需要这样)
TimeUnit.MILLISECONDS.sleep(Integer.MAX_VALUE);
}
}

```

提交事务消息状态

当本地事务执行完成（执行成功或执行失败），需要通知服务器当前消息的事务状态。通知方式有以下两种：

- 执行本地事务完成后提交
- 执行本地事务一直没提交状态，等待服务器回查消息的事务状态

事务状态有以下三种：

- TransactionStatus.CommitTransaction 提交事务，允许订阅方消费该消息。
- TransactionStatus.RollbackTransaction 回滚事务，消息将被丢弃不允许消费。
- TransactionStatus.Unknow 无法判断状态，期待 MQ Broker 向发送方再次询问该消息对应的本地事务的状态。

```
public class LocalTransactionCheckerImpl implements LocalTransactionChecker {
    private final static Logger log = ClientLogger.getLog();
    final BusinessService businessService = new BusinessService();

    @Override
    public TransactionStatus check(Message msg) {
        //消息 ID(有可能消息体一样，但消息 ID 不一样, 当前消息属于 Half 消息，所以消息 ID 在控制台无法查询)
        String msgId = msg.getMsgID();
        //消息体内容进行 crc32, 也可以使用其它的方法如 MD5
        long crc32Id = HashUtil.crc32Code(msg.getBody());
        //消息 ID、消息本 crc32Id 主要是用来防止消息重复
        //如果业务本身是幂等的, 可以忽略, 否则需要利用 msgId 或 crc32Id 来做幂等
        //如果要求消息绝对不重复, 推荐做法是对消息体使用 crc32或 md5来防止重复消息.
        //业务自己的参数对象, 这里只是一个示例, 实际需要用户根据情况来处理
        Object businessServiceArgs = new Object();
        TransactionStatus transactionStatus = TransactionStatus.Unknow;
        try {
            boolean isCommit = businessService.checkbusinessService(businessServiceArgs);
            if (isCommit) {
                //本地事务已成功、提交消息
                transactionStatus = TransactionStatus.CommitTransaction;
            } else {
                //本地事务已失败、回滚消息
                transactionStatus = TransactionStatus.RollbackTransaction;
            }
        } catch (Exception e) {
            log.error("Message Id:{}, msgId, e);
        }
        log.warn("Message Id:{}transactionStatus:{}, msgId, transactionStatus.name());
        return transactionStatus;
    }
}
```

工具类

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

事务回查机制说明

发送事务消息为什么必须要实现回查 Check 机制？

当步骤（1）中 Half 消息发送完成，但本地事务返回状态为 `TransactionStatus.Unknow`，或者应用退出导致本地事务未提交任何状态时，从 MQ Broker 的角度看，这条 Half 状态的消息的状态是未知的。因此 MQ Broker 会定期要求发送方能 Check 该 Half 状态消息，并上报其最终状态。

Check 被回调时，业务逻辑都需要做些什么？

MQ 事务消息的 check 方法里面，应该写一些检查事务一致性的逻辑。MQ 发送事务消息时需要实现 `LocalTransactionChecker` 接口，用来处理 MQ Broker 主动发起的本地事务状态回查请求；因此事务消息的 Check 方法中，需要完成两件事情：

(1) 检查该 Half 消息对应的本地事务的状态（committed or rollback）；

(2) 向 MQ Broker 提交该 Half 消息本地事务的状态。

订阅事务消息

事务消息的订阅与普通消息订阅一致，详见[订阅消息](#)。

收发延时消息

延时消息用于指定消息发送到 MQ 服务器端后，延时一段时间才被投递到客户端进行消费（例如 3 秒后才被消费），适用于解决一些消息生产和消费有时间窗口要求的场景，或者通过消息触发延迟任务的场景，类似于延迟队列。

关于延时消息的概念介绍及使用过程中的注意事项，请参见[延时消息](#)。

说明：关于 TCP 接入点域名，请参见[TCP 接入说明](#)。

发送延时消息

发送延时消息的示例代码如下：

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

import com.aliyun.openservices.ons.api.SendResult;
```

```

import java.util.Properties;

public class ProducerDelayTest {
    public static void main(String[] args) {
        Properties properties = new Properties();
        // 您在控制台创建的 Producer ID
        properties.put(PropertyKeyConst.ProducerId, "XXX");
        // AccessKey 阿里云身份验证，在阿里云服务器管理控制台创建
        properties.put(PropertyKeyConst.AccessKey, "XXX");
        // SecretKey 阿里云身份验证，在阿里云服务器管理控制台创建
        properties.put(PropertyKeyConst.SecretKey, "XXX");
        // 设置 TCP 接入域名（此处以公共云生产环境为例）
        properties.put(PropertyKeyConst.ONSDAddr,
            "http://onsaddr-internal.aliyun.com:8080/rocketmq/nsaddr4client-internal");

        Producer producer = ONSFactory.createProducer(properties);
        // 在发送消息前，必须调用 start 方法来启动 Producer，只需调用一次即可。
        producer.start();
        Message msg = new Message( //
            // 您在控制台创建的 Topic
            "Topic",
            // Message Tag, 可理解为 Gmail 中的标签，对消息进行再归类，方便 Consumer 指定过滤条件在 MQ 服务器过滤
            "tag",
            // Message Body 可以是任何二进制形式的数据，MQ 不做任何干预，需要 Producer 与 Consumer 协商好一致的序列化和
            // 反序列化方式
            "Hello MQ".getBytes());
        // 设置代表消息的业务关键属性，请尽可能全局唯一。
        // 以方便您在无法正常收到消息情况下，可通过 MQ 控制台查询消息并补发。
        // 注意：不设置也不会影响消息正常收发
        msg.setKey("ORDERID_100");
        try {
            // 延时消息，单位毫秒（ms），在指定延迟时间（当前时间之后）进行投递，例如消息在 3 秒后投递
            long delayTime = System.currentTimeMillis() + 3000;

            // 设置消息需要被投递的时间
            msg.setStartDeliverTime(delayTime);

            SendResult sendResult = producer.send(msg);
            // 同步发送消息，只要不抛异常就是成功
            if (sendResult != null) {
                System.out.println(new Date() + " Send mq message success. Topic is:" + msg.getTopic() + " msgId is: " +
                    sendResult.getMessageId());
            }
        } catch (Exception e) {
            // 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(new Date() + " Send mq message failed. Topic is:" + msg.getTopic());
            e.printStackTrace();
        }
        // 在应用退出前，销毁 Producer 对象<br>
        // 注意：如果不销毁也没有问题
        producer.shutdown();
    }
}

```

订阅延时消息

延时消息的订阅与普通消息订阅一致，详见[订阅消息](#)。

收发定时消息

定时消息可以做到在指定时间戳之后才可被消费者消费，用于解决一些消息生产和消费有时间窗口要求的场景，或者通过消息触发定时任务的场景。

关于定时消息的概念介绍及使用过程中的注意事项，请参见[定时消息](#)。

说明：关于 TCP 接入点域名，请参见[TCP 接入说明](#)。

发送定时消息

发送定时消息的示例代码如下：

```
import com.aliyun.openservices.ons.api.Message;
import com.aliyun.openservices.ons.api.ONSTFactory;
import com.aliyun.openservices.ons.api.Producer;
import com.aliyun.openservices.ons.api.PropertyKeyConst;

import com.aliyun.openservices.ons.api.SendResult;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Properties;

public class ProducerDelayTest {
    public static void main(String[] args) {
        Properties properties = new Properties();
        //您在 MQ 控制台创建的 Producer ID
        properties.put(PropertyKeyConst.ProducerId, "XXX");
        // 阿里云身份验证，在阿里云服务器管理控制台创建
        properties.put(PropertyKeyConst.AccessKey, "XXX");
        // 阿里云身份验证，在阿里云服务器管理控制台创建
        properties.put(PropertyKeyConst.SecretKey, "XXX");

        // 设置 TCP 接入域名（此处以公共云生产环境为例）
        properties.put(PropertyKeyConst.ONSTAddr,
            "http://onsaddr-internal.aliyun.com:8080/rocketmq/nsaddr4client-internal");
        Producer producer = ONSTFactory.createProducer(properties);
        // 在发送消息前，必须调用 start 方法来启动 Producer，只需调用一次即可。
        producer.start();
        Message msg = new Message( //
            // Message 所属的 Topic
            "Topic",
            // Message Tag 可理解为 Gmail 中的标签，对消息进行再归类，方便 Consumer 指定过滤条件在 MQ 服务器过滤
            "tag",
            // Message Body 可以是任何二进制形式的数据，MQ 不做任何干预，需要 Producer 与 Consumer 协商好一致的序列化和
```

```
反序列化方式
"Hello MQ".getBytes());
// 设置代表消息的业务关键属性，请尽可能全局唯一
// 以方便您在无法正常收到消息情况下，可通过 MQ 控制台查询消息并补发。
// 注意：不设置也不会影响消息正常收发
msg.setKey("ORDERID_100");

try {
// 定时消息，单位毫秒（ms），在指定时间戳（当前时间之后）进行投递，例如 2016-03-07 16:21:00 投递
long timeStamp = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").parse("2016-03-07 16:21:00").getTime();

msg.setStartDeliverTime(timeStamp);
// 发送消息，只要不抛异常就是成功
SendResult sendResult = producer.send(msg);
System.out.println("Message Id:" + sendResult.getMessageId());
}
catch (Exception e) {
// 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(new Date() + " Send mq message failed. Topic is:" + msg.getTopic());
e.printStackTrace();
}

// 在应用退出前，销毁 Producer 对象
// 注意：如果不销毁也没有问题
producer.shutdown();
}
}
```

订阅定时消息

定时消息的订阅方式与普通消息一致，详见订阅消息。

C/C++ SDK

C_C++ SDK 环境准备

用 C++ SDK 方式接入 MQ，需要完成以下准备工作。

注意：

代码里涉及到的 Topic，Producer ID，Consumer ID，需要到 MQ 控制台上创建。Message Tag 可以完全由应用自定义，具体创建过程参见步骤二：创建资源。

使用 MQ 服务的应用程序需要部署在阿里云 ECS 上。

下载 SDK

CPP 支持 Windows 和 Linux 两个跨平台的 SDK，而且接口完全一致。下载依赖 SDK 包链接：

Linux CPP 下载链接

Windows CPP 下载链接

下载完成后进行解压，会有如下目录结构：

- example/
- include/
- lib/
- SDK_GUIDE.pdf
- changelog

上面的目录和文件的作用如下：

demo: 包含了一个创建好的 Windows C++ 演示 Demo。

example：包含了普通消息发送、Oneway 消息发送、顺序消息发送、普通消息消费、顺序消息消费等例子，Linux 下还包含了 Makefile 用于 example 的编译和管理。

include：用户自己编写的程序需要 include 的头文件。

lib：Linux SDK 子目录如下，分别是 64 的静态库和动态库。

```
lib-boost-share/  
  libonsclient4cpp.so  
lib-boost-static/  
  libonsclient4cpp.a
```

Windows SDK 子目录如下，是 64 位系统下 SDK 的 dll 库。如果没有安装 Visual Studio 2015 环境下，需要拷贝安装vc_redist.x64。这是 Visual C++ 2015 的运行环境。

```
64/  
vc_redist.x64
```

SDK_GUIDE.pdf：SDK 环境准备文档和 FAQ。

changelog：新版本发布解决的问题和引入的新特性列表。

Linux C++ SDK 使用

自2016.12.02开始，Linux CPP 版本依赖了高性能 boost 库(1.62.0版本)，不仅降低了 CPU 资源占用率，而且提高了运行效率。目前主要依赖了 boost_system，boost_thread，boost_chrono，boost_filesystem 四个库。我们提供了静态库和动态库两种解决方案：

静态解决方案

MQ 库文件在 lib/lib-boost-static 目录下，boost 库静态链接到 libonsclient4cpp.a 中。对于没有依赖 boost 库的业务方，可以直接选用静态库方案。静态库方案中，相应的boost库已经链接到 libonsclient4cpp.a，编译时只需要链接 libonsclient4cpp.a 即可，无需执行其他操作。使用方式如下：

```
cd aliyun-mq-linux-cpp-sdk //下载的SDK解压后的路径
cd example //进入demo目录,修改demo文件，填入自己创建的 Topic，key相关的信息
g++ -static -I ../include -L ../lib/lib-boost-static ProducerExampleForEx.cpp -lonsclient4cpp -lpthread -ldl
```

注意：完全的静态链接请确保机器上安装了 libstdc++，pthread 等相关的静态库，默认安装的 libstdc++ 是没有安装静态库的，所以需要通过 yum 或者 > apt-get 来安装相关的静态库。此外使用如上方式会出现一些警告信息如下：

```
warning: Using 'gethostbyaddr' in statically linked applications requires at runtime the shared libraries from the
glibc version used for linking
```

建议最佳的方式，不要使用完全的静态链接，而是只静态链接 lonsclient4cpp，其他库动态链接即可。使用方式如下：

```
g++ -I ../include -L ../lib/lib-boost-static ProducerExampleForEx.cpp -Wl,-dn -lonsclient4cpp -Wl,-dy -lpthread -ldl
```

动态解决方案

MQ 库文件在 lib/lib-boost-share 目录下，需要业务方生成可执行文件时链接 boost 动态库和 libonsclient4cpp.so。对于业务方已经依赖了 boost 库，需要选择动态库方案的情况，对 boost 库的依赖需要做如下工作：

下载 boost 1.62.0 版本：

boost 1.62.0

解压 boost 1.62.0:

```
tar --bzip2 -xf /path/to/boost_1_62_0.tar.bz2
```

安装 boost 1.62.0 版本：

- 1) cd path/to/boost_1_62_0
- 2) 配置 boost : ./bootstrap.sh
- 3) 编译 boost: ./b2 link=shared runtime-link=shared
- 4) 安装 boost: ./b2 install

执行 `ldconfig -v|grep libboost`。如果有相关的输出表明 boost 动态库在动态库搜索路径中。

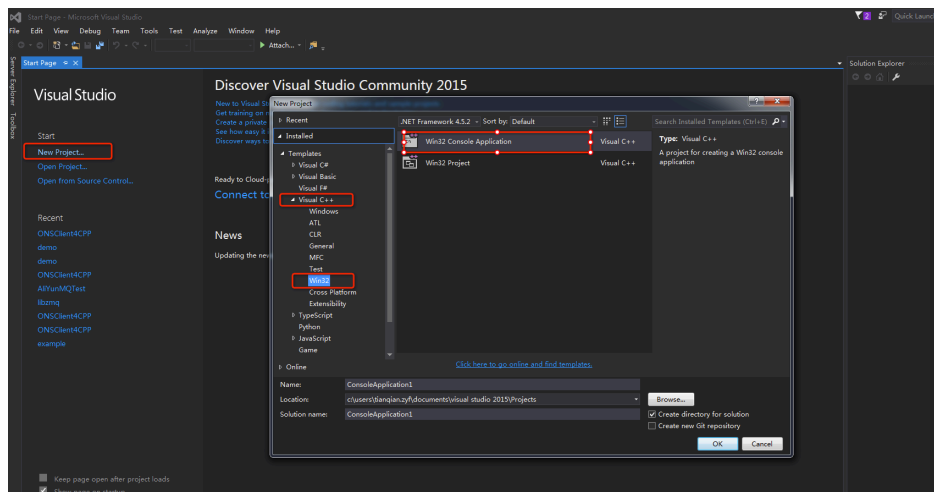
生成可执行文件时，需要链接 boost 动态库和 MQ 动态库。方法如下：

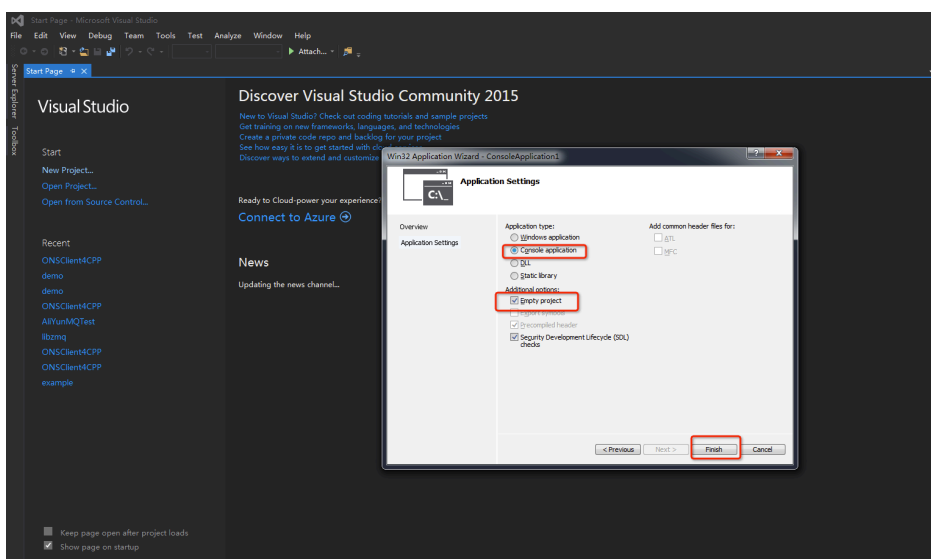
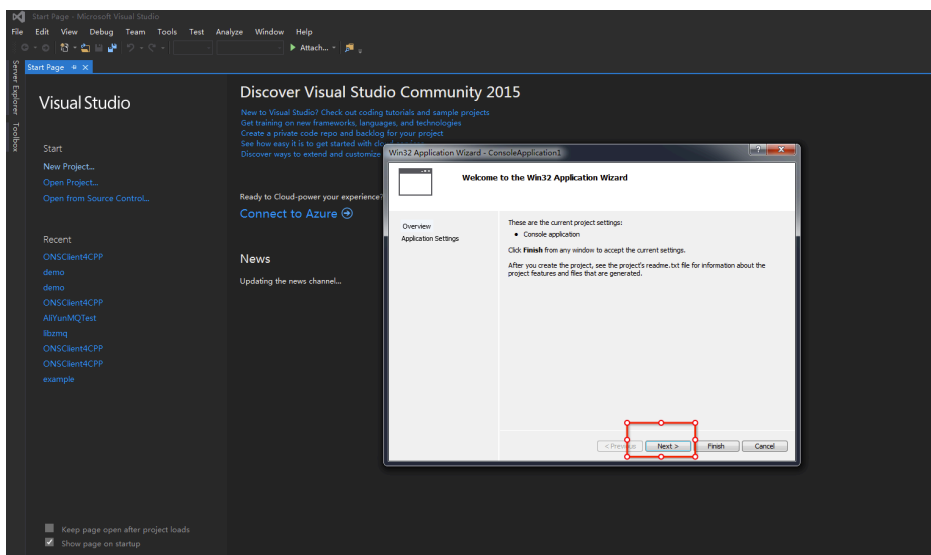
```
cd aliyun-mq-linux-cpp-sdk //下载的 SDK 解压后的路径
cd example //进入 demo 目录，修改 demo 文件，填入自己在 MQ 控制台创建的 Topic，key 相关的信息
g++ -Wall -Wno-deprecated -L ../lib/lib-boost-share/ -I ../include/ ProducerExampleForEx.cpp -lonsclient4cpp -lboost_system -lboost_thread -lboost_chrono -lboost_filesystem -lpthread
export LD_LIBRARY_PATH="../lib/lib-boost-share/" //添加动态载入的搜索路径
./a.out //运行程序
```

Windows C++ SDK 使用

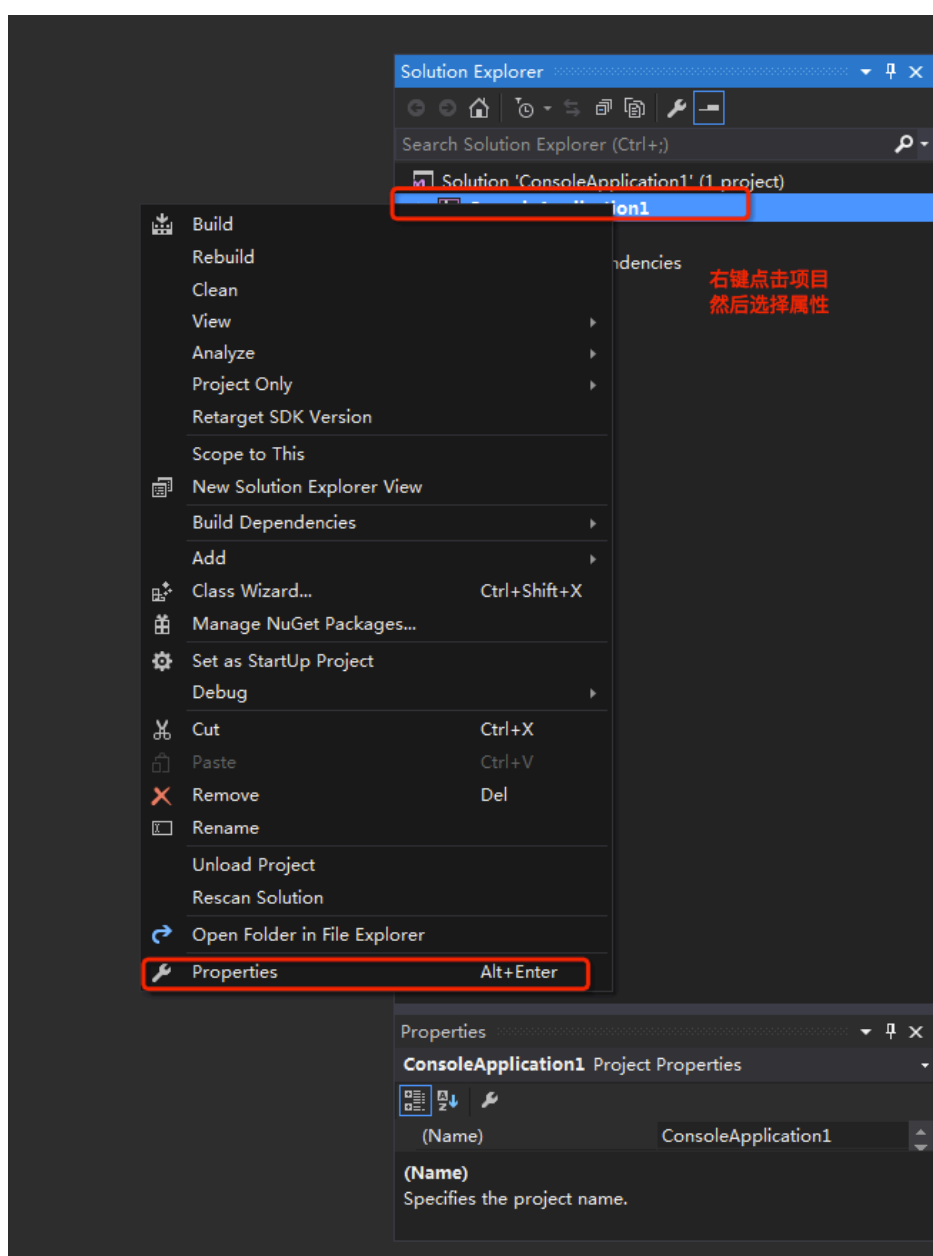
Visual Studio 2015 环境下使用 C++ SDK

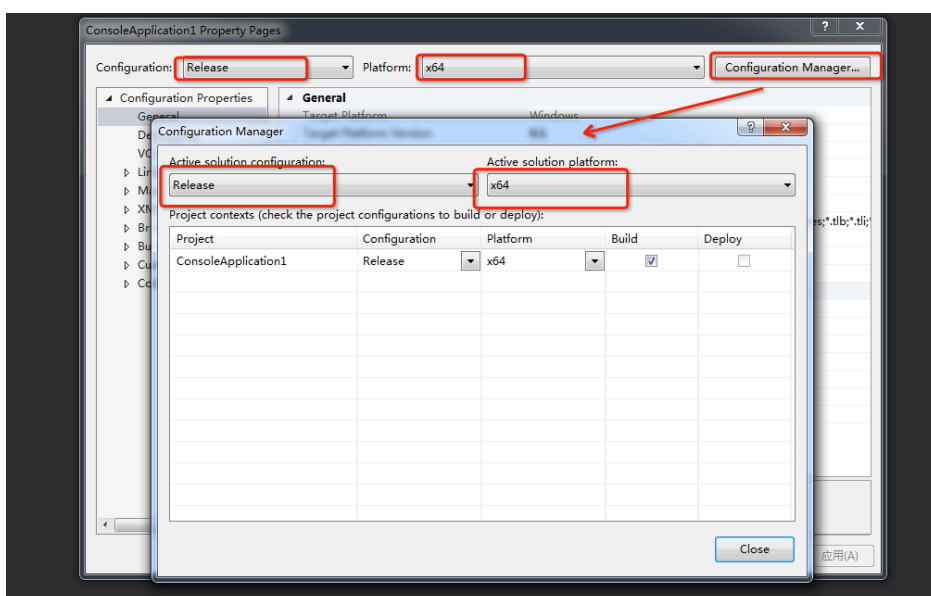
使用 Visual Studio 2015 创建自己的项目。



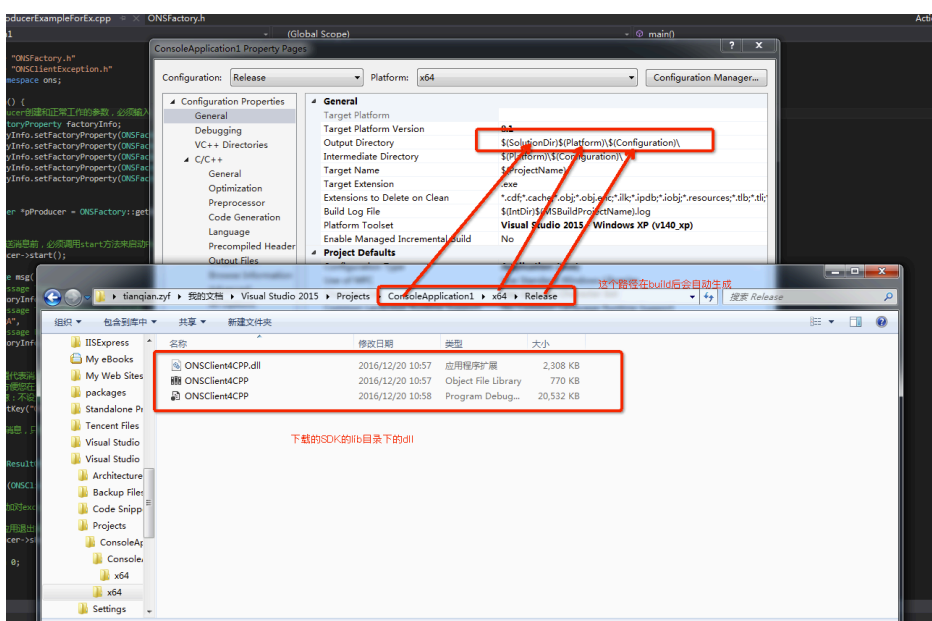


右键单击项目，选择属性>配置管理器，设置活动解决方案配置为 **release**，设置活动解决方案平台为 **x64**。

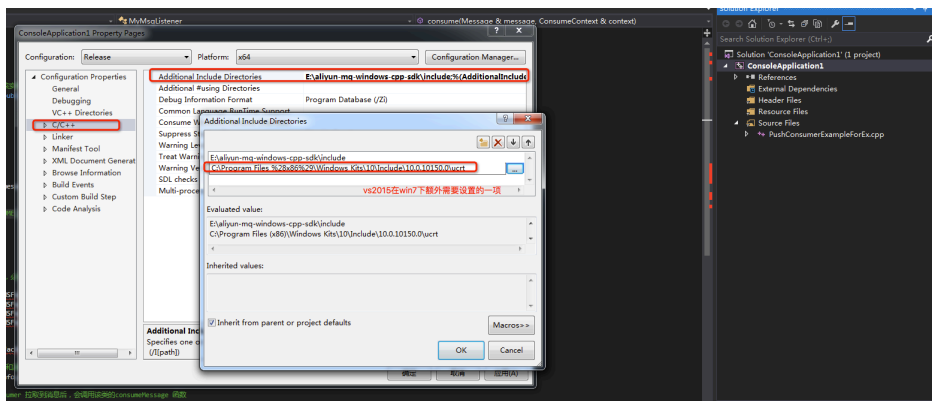
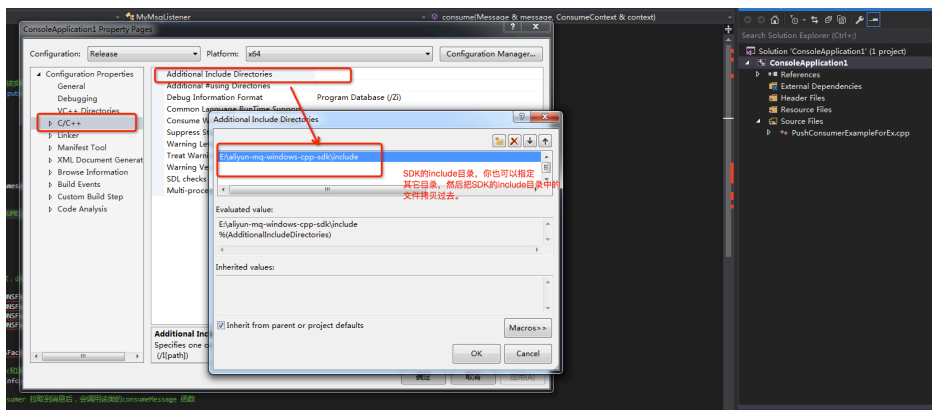




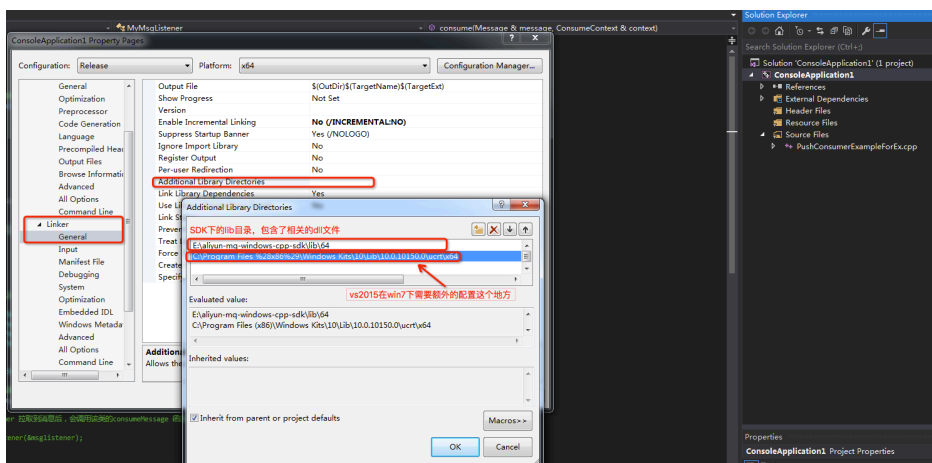
右键单击项目，依次选择属性>配置属性>常规>输出目录：/A。按照活动解决方案平台的设置，拷贝 64 位 lib 目录下的所有文件到输出目录 /A。



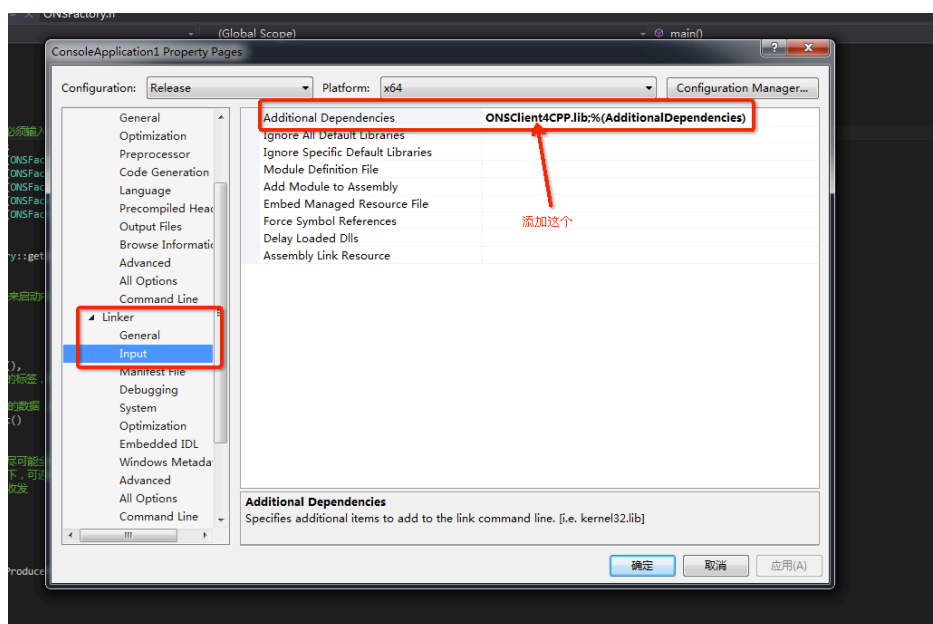
右键单击项目，依次选择属性>配置属性>C/C++-常规>附加包含目录：/B。拷贝 include 目录下的头文件到包含目录: /B。



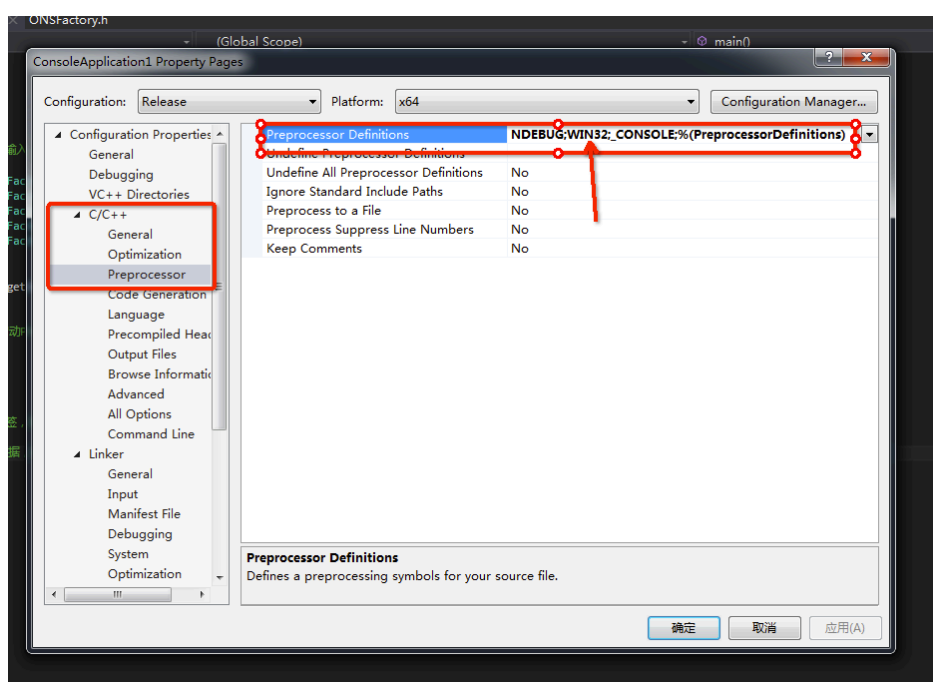
右键单击项目，依次选择属性>配置属性>链接>常规>附加库目录：/A。



右键单击项目，依次选择属性>配置属性>链接>输入>附加依赖项：ONSClnt4CPP.lib。



右键单击项目，依次选择属性>配置属性>>C/C++-常规>>预处理器定义: 添加 WIN32宏。

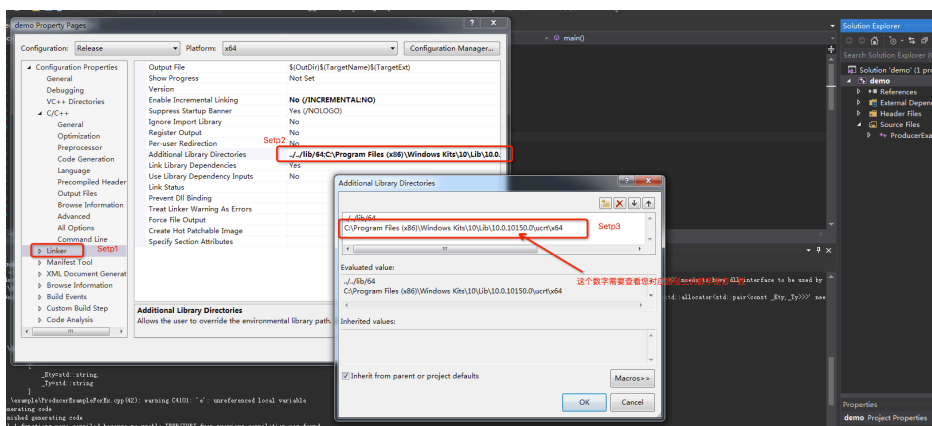
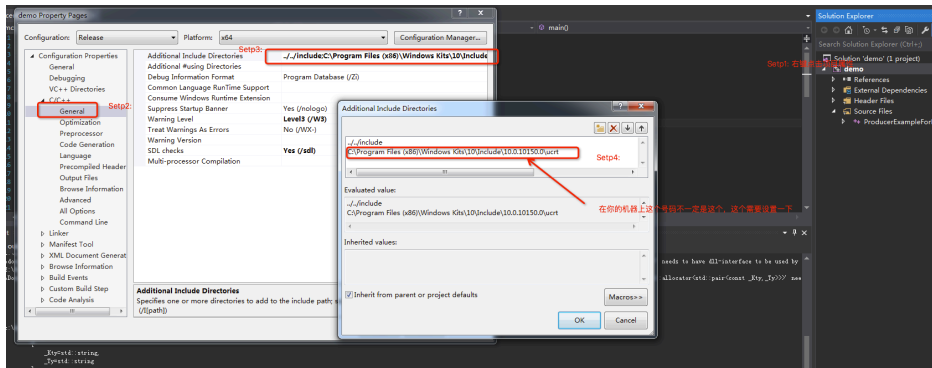
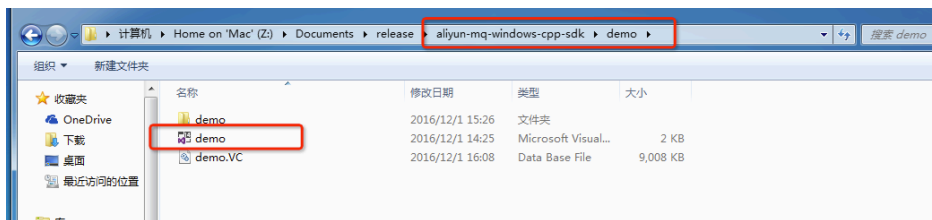


非 Visual Studio 2015 环境下使用 C++ SDK

首先需要按照 Visual Studio 2015 的环境来配置，配置过程同上。

安装 vc_redist.x64。这是 Visual C++ 2015 的运行时环境。

注意: 如果不想进行繁琐的设置，您也可以使用设置好的 SDK Demo，下载 SDK 后进行解压，然后进入 Demo 目录，使用 VS2015 打开工程。



到此为止就设置好编译环境了。点击 Build，即可编译出可执行的程序，然后拷贝 dll 到可执行程序的目录下即可运行，或者拷贝 dll 到系统目录下。

收发普通消息

发送普通消息

请参考以下示例代码进行消息发送。

```
#include "ONSFactory.h"
#include "ONSClientException.h"
```

```

using namespace ons;

int main()
{

    //创建 producer 和发送消息所必需的信息;
    ONSFactoryProperty factoryInfo;
    factoryInfo.setFactoryProperty(ONSFactoryProperty::ProducerId, "XXX");//您在控制台创建的 Producer ID
    factoryInfo.setFactoryProperty(ONSFactoryProperty::PublishTopics,"XXX" );// 消息内容
    factoryInfo.setFactoryProperty(ONSFactoryProperty::MsgContent, "XXX");//消息内容
    factoryInfo.setFactoryProperty(ONSFactoryProperty::AccessKey, "XXX");//AccessKey 阿里云身份验证，在阿里云服务器管理控制台创建
    factoryInfo.setFactoryProperty(ONSFactoryProperty::SecretKey, "XXX" );//SecretKey 阿里云身份验证，在阿里云服务器管理控制台创建

    //create producer;
    Producer *pProducer = ONSFactory::getInstance()->createProducer(factoryInfo);

    //在发送消息前，必须调用 start 方法来启动 Producer，只需调用一次即可;
    pProducer->start();

    Message msg(
        //Message Topic
        factoryInfo.getPublishTopics(),
        //Message Tag,可理解为 Gmail 中的标签，对消息进行再归类，方便 Consumer 指定过滤条件在 MQ 服务器过滤
        "TagA",
        //Message Body,不能为空，MQ 不做任何干预，需要 Producer 与 Consumer 协商好一致的序列化和反序列化方式
        factoryInfo.getMessageContent()
    );

    // 设置代表消息的业务关键属性，请尽可能全局唯一
    // 以方便您在无法正常收到消息情况下，可通过 MQ 控制台查询消息并补发
    // 注意：不设置也不会影响消息正常收发
    msg.setKey("ORDERID_100");

    //发送消息，只要不抛出异常，就代表发送成功
    try
    {
        SendResultONS sendResult = pProducer->send(msg);
    }
    catch(ONSClientException & e)
    {
        //自定义处理 exception 的细节
    }
    // 在应用退出前，必须销毁 Producer 对象，否则会导致内存泄露等问题
    pProducer->shutdown();

    return 0;
}

```

订阅普通消息

关于订阅普通消息的说明和示例代码，详见[订阅消息](#)。

收发顺序消息

发送顺序消息

发送顺序消息的示例代码如下：

```
#include "ONSFactory.h"
#include "ONSClientException.h"
#include <iostream>
using namespace ons;

int main()
{
    //producer创建和正常工作的参数，必须输入
    ONSFactoryProperty factoryInfo;
    factoryInfo.setFactoryProperty(ONSFactoryProperty::ProducerId, "PID_XXXXXXXXXX");//在ONS控制台申请的producerId
    factoryInfo.setFactoryProperty(ONSFactoryProperty::PublishTopics, "XXXXXXXXXXXX" );// 在ONS 控制台申请的msg topic
    factoryInfo.setFactoryProperty(ONSFactoryProperty::MsgContent, "input msg content");//消息内容
    factoryInfo.setFactoryProperty(ONSFactoryProperty::AccessKey, "XXXXXXXXXX");//ONS AccessKey
    factoryInfo.setFactoryProperty(ONSFactoryProperty::SecretKey, "XXXXXXXXXXXXXXXXXXXX" );// ONS SecretKey
    // 示例为公共云环境, 其它环境参考:
    https://help.aliyun.com/document_detail/55448.html?spm=a2c4g.11186623.6.579.mu1gZX
    factoryInfo.setFactoryProperty(ONSFactoryProperty::ONSAddr, "http://onsaddr-
internet.aliyun.com/rocketmq/nsaddr4client-internet")

    //创建producer
    OrderProducer *pProducer = ONSFactory::getInstance()->createOrderProducer(factoryInfo);

    //在发送消息前，必须调用start方法来启动Producer，只需调用一次即可。
    pProducer->start();

    Message msg(
        //Message Topic
        factoryInfo.getPublishTopics(),
        //Message Tag,可理解为Gmail中的标签，对消息进行再归类，方便Consumer指定过滤条件在ONS服务器过滤
        "TagA",
        //Message Body，任何二进制形式的数据，ONS不做任何干预，需要Producer与Consumer协商好一致的序列化和反序列化
        方式
        factoryInfo.getMessageContent()
    );

    // 设置代表消息的业务属性，请尽可能全局唯一。
    // 以方便您在无法正常收到消息情况下，可通过ONS Console查询消息并补发。
    // 注意：不设置也不会影响消息正常收发
    msg.setKey("ORDERID_100");
    // 分区顺序消息中区分不同分区的关键字段
```

```

// 如果是全局顺序消息，该字段可以设置为任意非空字符串。
std::string shardingKey = "abc"; //带有同一 sharding key 的消息会按照顺序发送
try
{
    //发送消息，只要不抛异常就是成功
    SendResultONS sendResult = pProducer->send(msg, shardingKey);
    std::cout << "send success" << std::endl;
}
catch(ONSClientException & e)
{
    //添加对exception的处理
}
// 在应用退出前，必须销毁Producer对象，否则会导致内存泄露等问题
pProducer->shutdown();

return 0;
}

```

接收顺序消息

接收顺序消息的示例代码如下：

```

#include "ONSFactory.h"
using namespace std;
using namespace ons;

//创建消费消息的实例
//pushConsumer拉取到消息后，会主动调用该实例的consumeMessage 函数
class ONSCLIENT_API MyMsgListener : public MessageOrderListener
{
public:
    MyMsgListener()
    {
    }
    virtual ~MyMsgListener()
    {
    }

    virtual OrderAction consume(Message &message, ConsumeOrderContext &context)
    {
        //根据业务需求，消费消息
        return Success; //CONSUME_SUCCESS;
    }
};

int main(int argc, char* argv[])
{
    //OrderConsumer创建和工作需要的参数，必须输入
    ONSFactoryProperty factoryInfo;
    factoryInfo.setFactoryProperty(ONSFactoryProperty::ConsumerId, ""); //在ONS控制台申请的consumerId
    factoryInfo.setFactoryProperty(ONSFactoryProperty::PublishTopics, "" ); // 在ONS 控制台申请的msg topic
    factoryInfo.setFactoryProperty(ONSFactoryProperty::AccessKey, ""); // ONS AccessKey
    factoryInfo.setFactoryProperty(ONSFactoryProperty::SecretKey, ""); //ONS SecretKey
}

```



```

factoryInfo.setFactoryProperty(ONSFactoryProperty::SecretKey, "xxxxxxxxxxxxxxxxxxxx" );// ONS SecretKey
// 示例为公共云环境, 其它环境参考:
https://help.aliyun.com/document_detail/55448.html?spm=a2c4g.11186623.6.579.mu1gZX
factoryInfo.setFactoryProperty(ONSFactoryProperty::ONSAddr, "http://onsaddr-
internet.aliyun.com/rocketmq/nsaddr4client-internet");

// 创建orderConsumer
OrderConsumer* orderConsumer = ONSFactory::getInstance()->createOrderConsumer(factoryInfo);
MyMsgListener msglistener;
//指定orderConsumer 订阅的消息topic和消息tag
orderConsumer->subscribe(factoryInfo.getPublishTopics(), "*",&msglistener );

// 注册消息监听的处理实例, orderConsumer 拉取到消息后, 会调用该类的consumeMessage 函数

//启动orderConsumer
orderConsumer->start();

for(volatile int i = 0; i < 1000000000; ++i) {
//wait
}

//销毁orderConsumer, 在应用退出前, 必须销毁Consumer 对象, 否则会导致内存泄露等问题
orderConsumer->shutdown();

return 0;
}

```

收发定时消息

目前支持的域包括公网、华东1、华北2、华东2、华南1。

定时消息可以做到在指定时间之后才可被消费者消费, 用于解决一些消息生产和消费有时间窗口要求的场景, 或者通过消息触发定时任务的场景, 类似于延迟队列。

发送定时消息

发送定时消息的代码示例如下：

```

#include "ONSFactory.h"
#include "ONSClientException.h"
using namespace ons;
int main()
{

    //创建 producer 和发送消息所必需的信息;

```

```

ONSFactoryProperty factoryInfo;
factoryInfo.setFactoryProperty(ONSFactoryProperty::ProducerId, "XXX");//您在 MQ 控制台创建的 producer
factoryInfo.setFactoryProperty(ONSFactoryProperty::PublishTopics,"XXX" );//您在 MQ 控制台创建的 topic
factoryInfo.setFactoryProperty(ONSFactoryProperty::MsgContent, "xxx");//msg content
factoryInfo.setFactoryProperty(ONSFactoryProperty::AccessKey, "xxx");//阿里云身份验证，在阿里云服务器管理控制台创建
factoryInfo.setFactoryProperty(ONSFactoryProperty::SecretKey, "xxx" );//阿里云身份验证，在阿里云服务器管理控制台创建

//create producer;
Producer *pProducer = ONSFactory::getInstance()->createProducer(factoryInfo);

//在发送消息前，必须调用 start 方法来启动 Producer，只需调用一次即可;
pProducer->start();

Message msg(
    //Message Topic
    factoryInfo.getPublishTopics(),
    //Message Tag,可理解为 Gmail 中的标签，对消息进行再归类，方便 Consumer 指定过滤条件在 MQ 服务器过滤
    "TagA",
    //Message Body, 不能为空，MQ 不做任何干预，需要 Producer 与 Consumer 协商好一致的序列化和反序列化方式
    factoryInfo.getMessageContent()
);

// 设置代表消息的业务关键属性，请尽可能全局唯一。
// 以方便您在无法正常收到消息情况下，可通过 MQ 控制台查询消息并补发。
// 注意：不设置也不会影响消息正常收发
msg.setKey("ORDERID_100");

// deliver time 单位 ms，指定一个时刻，在这个时刻之后才能被消费，这个例子表示3s 后才能被消费
long deliverTime = 获取系统当前时间(ms) + 3000;
msg.setStartDeliverTime(deliverTime);

//发送消息，只要不抛出异常，就代表发送成功
try
{
    SendResultONS sendResult = pProducer->send(msg);
}
catch(ONSClientException & e)
{
    //自定义处理 exception 的细节
}

// 在应用退出前，必须销毁 Producer 对象，否则会导致内存泄露等问题
pProducer->shutdown();

return 0;
}

```

订阅定时消息

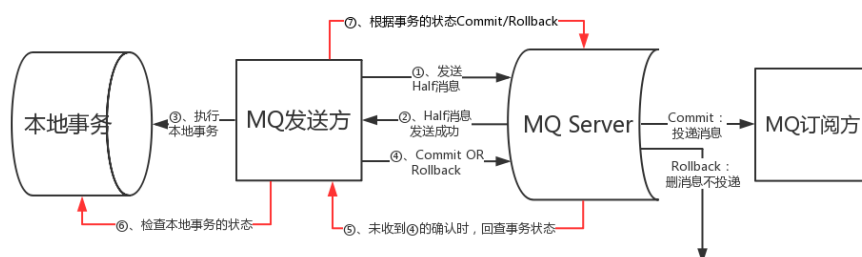
关于订阅定时消息的说明和示例代码，详见[订阅消息](#)。

收发事务消息

目前支持的域包括公网、华东1、华北2、华东2、华南1。

交互流程

MQ 事务消息的交互流程如下图所示：



发送事务消息

发送事务消息包含以下两个步骤：

1. 发送半消息（Half Message）及执行本地事务。示例代码如下：

```
#include "ONFactory.h"
#include "ONClientException.h"
using namespace ons;

class MyLocalTransactionExecuter : LocalTransactionExecuter
{
    MyLocalTransactionExecuter()
    {
    }

    ~MyLocalTransactionExecuter()
    {
    }

    virtual TransactionStatus execute(Message &value)
    {
        // 消息 ID(有可能消息体一样，但消息 id 不一样, 当前消息 ID 在 console 控制不可能查询)
        string msgId = value.getMsgID();
        // 消息体内容进行 crc32, 也可以使用其它的如 MD5
        // 消息 ID 和 crc32id 主要是用来防止消息重复
        // 如果业务本身是幂等的, 可以忽略, 否则需要利用 msgId 或 crc32Id 来做幂等
        // 如果要求消息绝对不重复, 推荐做法是对消息体 body 使用 crc32或 md5来防止重复消息.
        TransactionStatus transactionStatus = Unknow;
        try {
```

```

        boolean isCommit = 本地事务执行结果;
        if (isCommit) {
            // 本地事务成功、提交消息
            transactionStatus = CommitTransaction;
        } else {
            // 本地事务失败、回滚消息
            transactionStatus = RollbackTransaction;
        }
    } catch (...) {
        //exception handle
    }
    return transactionStatus ;
}
}

int main(int argc, char* argv[])
{
    ONSFactoryProperty factoryInfo;
    factoryInfo.setFactoryProperty(ONSFactoryProperty::ProducerId, "XXX");//您在控制台创建的 Producer ID
    factoryInfo.setFactoryProperty(ONSFactoryProperty::PublishTopics,"XXX" );//输入您在控制台创建的 Topic
    factoryInfo.setFactoryProperty(ONSFactoryProperty::MsgContent, "XXX");//msg content
    factoryInfo.setFactoryProperty(ONSFactoryProperty::AccessKey, "xxxxxxx");//阿里云身份验证，在阿里云服务器
    管理控制台创建
    factoryInfo.setFactoryProperty(ONSFactoryProperty::SecretKey, "xxxxxxxxxxxxxxxxxxxx");//阿里云身份验证，在阿
    里云服务器管理控制台创建

    //创建 producer，MQ 不负责 pChecker 的释放，需要业务方自行释放资源
    MyLocalTransactionChecker *pChecker = new MyLocalTransactionChecker();
    g_producer = ONSFactory::getInstance()->createTransactionProducer(factoryInfo,pChecker);

    //在发送消息前，必须调用 start 方法来启动 Producer，只需调用一次即可;
    pProducer->start();

    Message msg(
        //Message Topic
        factoryInfo.getPublishTopics(),
        //Message Tag,可理解为 Gmail 中的标签，对消息进行再归类，方便 Consumer 指定过滤条件在 ONS 服务器过滤
        "TagA",
        //Message Body,不能为空，MQ 不做任何干预，需要 Producer 与 Consumer 协商好一致的序列化和反序列化方式
        factoryInfo.getMessageContent()
    );

    // 设置代表消息的业务关键属性，请尽可能全局唯一。
    // 以方便您在无法正常收到消息情况下，可通过 MQ Console 查询消息并补发。
    // 注意：不设置也不会影响消息正常收发
    msg.setKey("ORDERID_100");

    //发送消息，只要不抛出异常，就代表发送成功
    try
    {
        //MQ 不负责 pExecuter 的释放，需要业务方自行释放资源
        MyLocalTransactionExecuter pExecuter = new MyLocalTransactionExecuter();
        SendResultONS sendResult = pProducer->send(msg,pExecuter);
    }
    catch(ONSClientException & e)
    {

```

```

        //自定义处理 exception 的细节
    }
    // 在应用退出前，必须销毁 Producer 对象，否则会导致内存泄露等问题
    pProducer->shutdown();

    return 0;

}

```

2.提交事务消息状态

当本地事务执行完成（执行成功或执行失败），需要通知服务器当前消息的事务状态。通知方式有以下两种：

- 执行本地事务完成后提交；
- 执行本地事务一直没提交状态，等待服务器回查消息的事务状态。

事务状态有以下三种：

- TransactionStatus.CommitTransaction 提交事务，允许订阅方消费该消息；
- TransactionStatus.RollbackTransaction 回滚事务，消息将被丢弃不允许消费；

TransactionStatus.Unknow 无法判断状态，期待 MQ Broker 向发送方再次询问该消息对应的本地事务的状态。

```

class MyLocalTransactionChecker : LocalTransactionChecker
{
    MyLocalTransactionChecker()
    {
    }

    ~MyLocalTransactionChecker()
    {
    }

    virtual TransactionStatus check(Message &value)
    {
        // 消息 ID(有可能消息体一样，但消息 id 不一样, 当前消息 ID 在 console 控制不可能查询)
        string msgId = value.getMsgID();
        // 消息体内容进行 crc32, 也可以使用其它的如 MD5
        // 消息 ID 和 crc32id 主要是用来防止消息重复
        // 如果业务本身是幂等的, 可以忽略, 否则需要利用 msgId 或 crc32id 来做幂等
        // 如果要求消息绝对不重复, 推荐做法是对消息体 body 使用 crc32或 md5来防止重复消息.
        TransactionStatus transactionStatus = Unknow;
        try {
            boolean isCommit = 本地事务执行结果;
            if (isCommit) {
                // 本地事务成功、提交消息
                transactionStatus = CommitTransaction;
            } else {
                // 本地事务失败、回滚消息
                transactionStatus = RollbackTransaction;
            }
        } catch(...) {
            //exception error
        }
    }
}

```

```
    }  
    return transactionStatus ;  
  }  
}
```

事务回查机制说明

1、发送事务消息为什么必须要实现 check 机制？

当步骤1发送半消息完成，但本地事务返回状态为 `TransactionStatus.Unknow` 时，亦或是应用退出导致本地事务未提交任何状态时，从 MQ Broker 的角度看，这条半状态的消息的状态是未知的，因此 MQ Broker 会定期要求发送方能 check 该半状态消息，并上报其最终状态。

2、Check 被回调时，业务逻辑都需要做些什么？

MQ 事务消息的 check 方法里面，应该写一些检查事务一致性的逻辑。MQ 发送事务消息时需要实现 `LocalTransactionChecker` 接口，用来处理 MQ Broker 主动发起的本地事务状态回查请求；因此在事务消息的 check 方法中，需要完成两件事情：

- (1) 检查该半消息对应的本地事务的状态（committed or rollback）；
- (2) 向 MQ Broker 提交该半消息本地事务的状态。

3、本地事务的不同状态对 Half 消息的影响？

`TransactionStatus.CommitTransaction` 提交事务，允许订阅方消费该消息。

`TransactionStatus.RollbackTransaction` 回滚事务，消息将被丢弃不允许消费。

`TransactionStatus.Unknow` 无法判断状态，期待 MQ Broker 向发送方再次询问该消息对应的本地事务的状态。

具体代码详见 `MyLocalTransactionChecker` 的实现。

订阅事务消息

关于订阅普通消息的说明和示例代码，详见[订阅消息](#)。

订阅消息

本文介绍如何通过 MQ SDK 使用 C/C++ 语言进行消息订阅。

说明：

- 请确保同一个 Consumer ID 下所有 Consumer 实例的订阅关系保持一致，详见订阅关系一致。
- 关于 TCP 接入点域名，请参见TCP 接入说明。

订阅方式

MQ 支持以下两种订阅方式：

集群订阅：同一个 Consumer ID 所标识的所有 Consumer 平均分摊消费消息。例如某个 Topic 有 9 条消息，一个 Consumer ID 有 3 个 Consumer 实例，那么在集群消费模式下每个实例平均分摊，只消费其中的 3 条消息。

```
// 集群订阅方式设置（不设置的情况下，默认为集群订阅方式）
factoryInfo.setFactoryProperty(ONSFactoryProperty:: MessageModel, ONSFactoryProperty::CLUSTERING);
```

广播订阅：同一个 Consumer ID 所标识的所有 Consumer 都会各自消费某条消息一次。例如某个 Topic 有 9 条消息，一个 Consumer ID 有 3 个 Consumer 实例，那么在广播消费模式下每个实例都会各自消费 9 条消息。

```
// 广播订阅方式设置
factoryInfo.setFactoryProperty(ONSFactoryProperty:: MessageModel,
ONSFactoryProperty::BROADCASTING);
```

示例代码

```
#include "ONSFactory.h"
using namespace ons;

// MyMsgListener：创建消费消息的实例
//pushConsumer 拉取到消息后，会主动调用该实例的 consumer 函数
class MyMsgListener : public MessageListener
{
public:
    MyMsgListener()
    {
    }

    virtual ~MyMsgListener()
    {
    }

    virtual Action consume(Message &message, ConsumeContext &context)
```

```

    {
        //自定义消息处理细节
        return CommitMessage; //CONSUME_SUCCESS;
    }
};

int main(int argc, char* argv[])
{
    //pushConsumer 创建和工作需要的参数，必须输入
    ONSFactoryProperty factoryInfo;
    factoryInfo.setFactoryProperty(ONSFactoryProperty::ConsumerId, "XXX");//您在 MQ 控制台创建的 consumerId
    factoryInfo.setFactoryProperty(ONSFactoryProperty::PublishTopics,"XXX" );//您在 MQ 控制台创建的 msg topic
    factoryInfo.setFactoryProperty(ONSFactoryProperty::AccessKey, "XXX");//阿里云身份验证，在阿里云服务器管理控制台创建
    factoryInfo.setFactoryProperty(ONSFactoryProperty::SecretKey, "XXX");//阿里云身份验证，在阿里云服务器管理控制台创建
    // 集群订阅方式 (默认)
    // factoryInfo.setFactoryProperty(ONSFactoryProperty:: MessageModel, ONSFactoryProperty::CLUSTERING);
    // 广播订阅方式
    // factoryInfo.setFactoryProperty(ONSFactoryProperty:: MessageModel, ONSFactoryProperty::BROADCASTING);

    //create pushConsumer
    PushConsumer* pushConsumer = ONSFactory::getInstance()->createPushConsumer(factoryInfo);

    //指定 pushConsumer 订阅的消息 topic 和 tag, 注册消息回调函数
    MyMsgListener msglistener;
    pushConsumer->subscribe(factoryInfo.getPublishTopics(), "*",&msglistener );

    //start pushConsumer
    pushConsumer->start();

    //NOTE:直到不再接收消息，才能调用 shutdown；调用 shutdown 之后，consumer 退出，不能接收到任何消息

    //销毁 pushConsumer, 在应用退出前，必须销毁 Consumer 对象，否则会导致内存泄露等问题
    pushConsumer->shutdown();
    return 0;
}

```

.NET SDK

.NET SDK 环境准备

用 .NET SDK 方式接入 MQ，需要完成以下准备工作。

注意：

代码里涉及到的 Topic，Producer ID，Consumer ID，需要到 MQ 控制台上创建。Message Tag 可以完全由应用自定义，具体创建过程可参考 [创建资源](#)。

使用 MQ 服务的应用程序需要部署在阿里云 ECS 上。

下载 SDK

Windows .NET 版本:

我们提供的 .NET 版本是基于 MQ CPP 版本的托管封装，这样能保证 .NET 完全不依赖于 Windows .NET 公共库，内部采用 C++ 多线程并发处理，保证 .NET 版本的高效稳定。

在使用 VS 开发 .NET 的应用程序和类库时，默认的目标平台为 “Any CPU”，即运行时可根据 CPU 类型自动选择 X86 或 X64。拥有这样的能力是因为 .NET 编译后的程序集是基于 IL 的。在运行时，CLR 才会将其 JIT 发射为 X86 或 X64 的机器码。而 C 或 C++ 编译生成的 DLL 就是机器码。所以，其平台的决策是在编译时决定的。通过编译选项的设置，我们将 C/C++ 项目编译为 X86 的 32 位 dll 或者 X64 的 64 位 dll，因此我们提供了包含 VS2015 编译的 release64 位版本 DLL。其他 VS 版本也可以使用。

旧版 Windows .NET SDK 下载

注意：

- 基于托管封装的 SDK 存在诸多问题，并且无法稳定在 ASP.NET 上正常工作，因此在 2016/12/29 号开始推出新版本的 SDK。
- 新版 SDK 基于 C# PINVOKE 调用底层的 dll，利用开源软件 SWIG 生成 PINVOKE 封装代码。新版 SDK 相比于托管版本的 SDK 更稳定，部署安装更简单。
- 托管版本的 SDK 不再维护，只提供最后一次的稳定版本。

新用户或者不考虑升级成本的老用户请下载新版 SDK

新版 Windows .NET SDK 下载

下载完成后进行解压，会有如下目录结构:

- example/
- lib/
- demo/
- interface/
- SDK_GUIDE.pdf
- changelog

上面的目录和文件的作用如下:

example：包含了普通消息发送、Oneway 消息发送、顺序消息发送、普通消息消费、顺序消息消费等例子。

lib：底层的 C++ DLL 相关文件，以及 **Virtual C++ 2015** 运行时环境安装包。

```
64/  
  NSClient4CPP.lib  
  ONSClnt4CPP.dll  
  ONSClnt4CPP.pdb  
  vc_redist.x64.exe
```

SDK_GUIDE.pdf：SDK 环境准备文档和 FAQ。

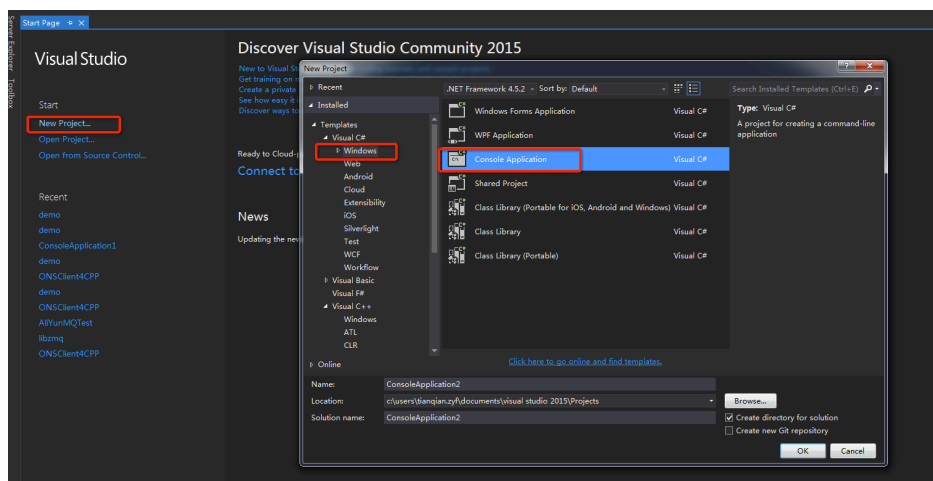
changelog：新版本发布解决的问题和引入的新特性列表。

interface: 封装 PINVOKE 调用的代码，需要包含到用户项目代码中。

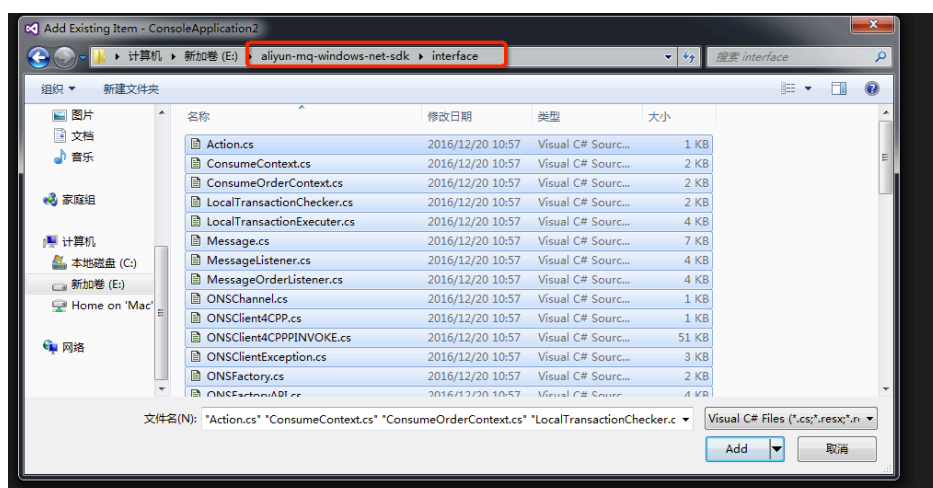
NET SDK 配置说明

Visual Studio 2015 使用 .NET SDK 配置说明

使用 Visual Studio 2015 创建自己的项目。

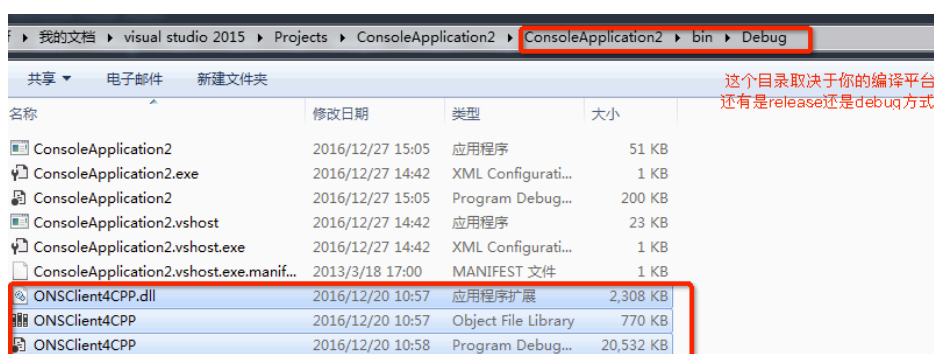


右键单击项目选择**添加>现存在项** 将下载的 SDK 中的 **interface** 目录下的所有文件都添加进去。



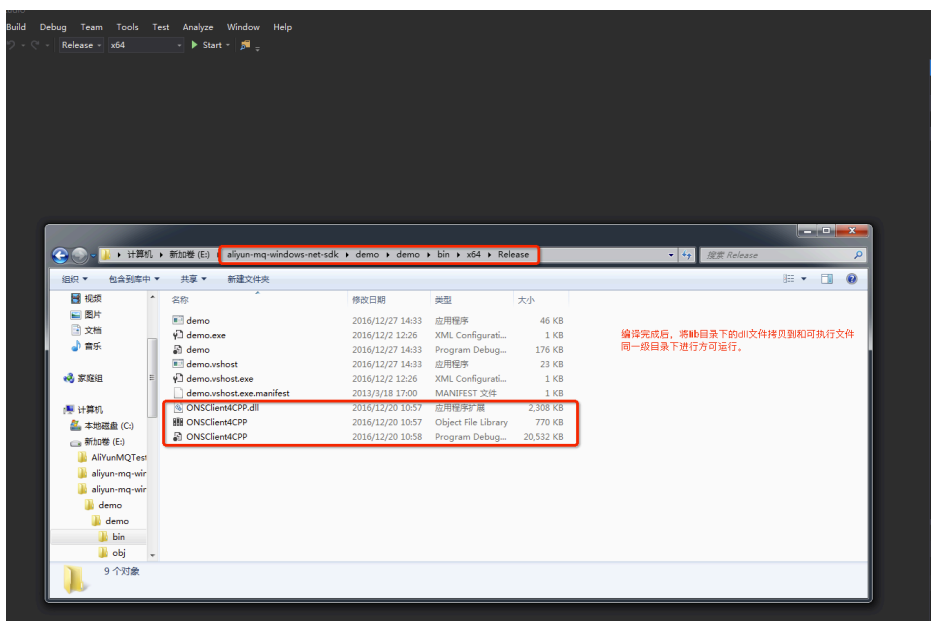
右键单击项目选择属性，选择配置管理器。设置活动解决方案配置为 **release**；设置活动解决方案平台为 **x64**。

编写测试程序，然后进行编译，最后将 SDK 下的 dll 放到和可执行文件同一级目录下，或者系统目录下即可运行。



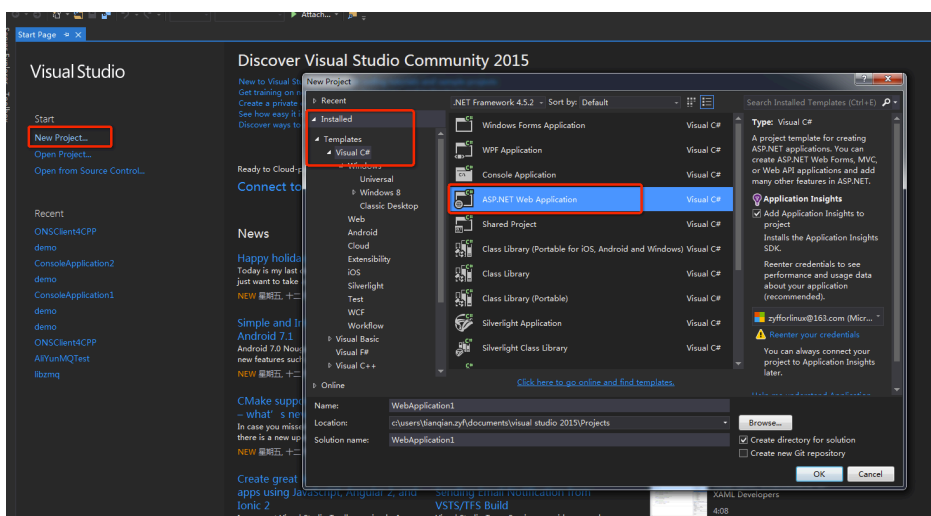
说明：

SDK 提供了设置好的 Demo，直接打开工程进行编译即可。运行的时候将相关的 DLL 文件拷贝到和可执行文件同级目录下，如下图：

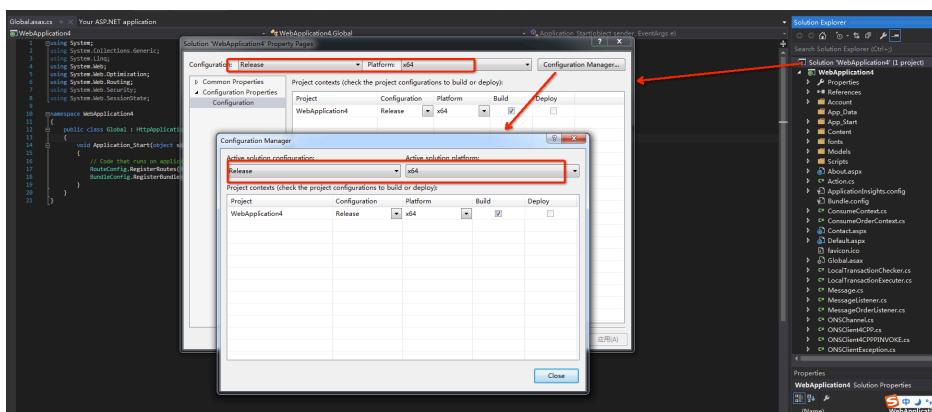


Visual Studio 2015 配置 ASP.NET 使用 MQ SDK

使用 VS2015 创建一个 ASP.NET 的 Web Forms 项目。



右键单击项目选择属性，选择配置管理器。设置活动解决方案配置为 release；设置活动解决方案平台为 x64。



右键单击项目选择**添加>现存在项**，将下载的 SDK 中的 interface 目录下的所有文件都添加进去。

请参考上文中配置普通的 .net 项目的步骤 2。

在 **Global.asax.cs** 文件中添加启动和关闭 SDK 的代码。

注意：

建议将 SDK 的代码封装成一个单例类，这样可以避免因为作用域的问题被垃圾回收器回收。SDK 的 example 目录下提供了一个 Example.cs，实现了一个简单的单例实现。您需要把 Example.cs 包含到自己的项目中才能使用。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Optimization;
using System.Web.Routing;
using System.Web.Security;
using System.Web.SessionState;
using ons; // 这个命名空间是 SDK 所在的命名空间
using test; // 这是一个简单封装 SDK 的类所在命名空间，见 SDK 的 example 目录下的 Example.cs
namespace WebApplication4
{
    public class Global : HttpApplication
    {
        void Application_Start(object sender, EventArgs e)
        {
            // Code that runs on application startup
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);
            try
            {
                // 启动 SDK 的代码,下面是简单封装 SDK 后的代码
                OnscSharp.CreateProducer();
                OnscSharp.StartProducer();
            }
            catch (Exception ex)
            {
            }
        }
    }
}
```

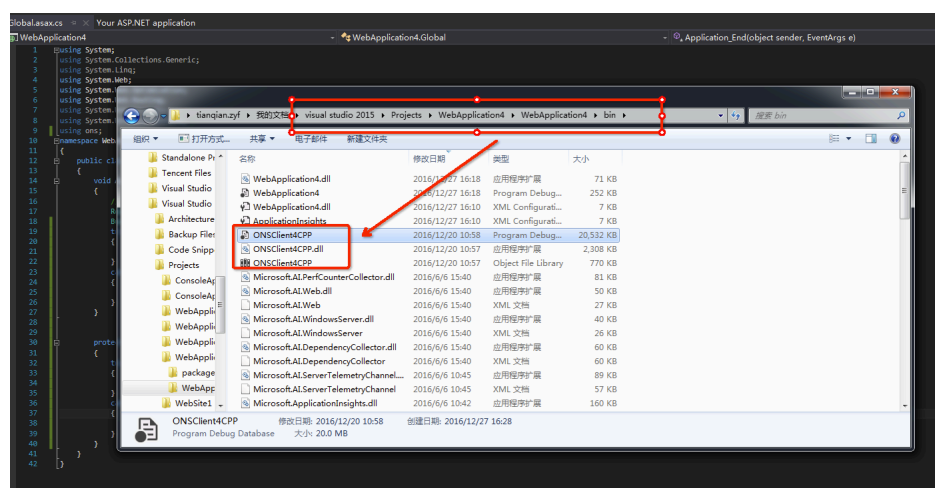
```

        }
    }
    protected void Application_End(object sender, EventArgs e)
    {
        try
        {
            // 关闭 SDK 的代码
            OnscSharp.ShutdownProducer();
        }
        catch (Exception ex)
        {
            // 处理异常
        }
    }
}
}

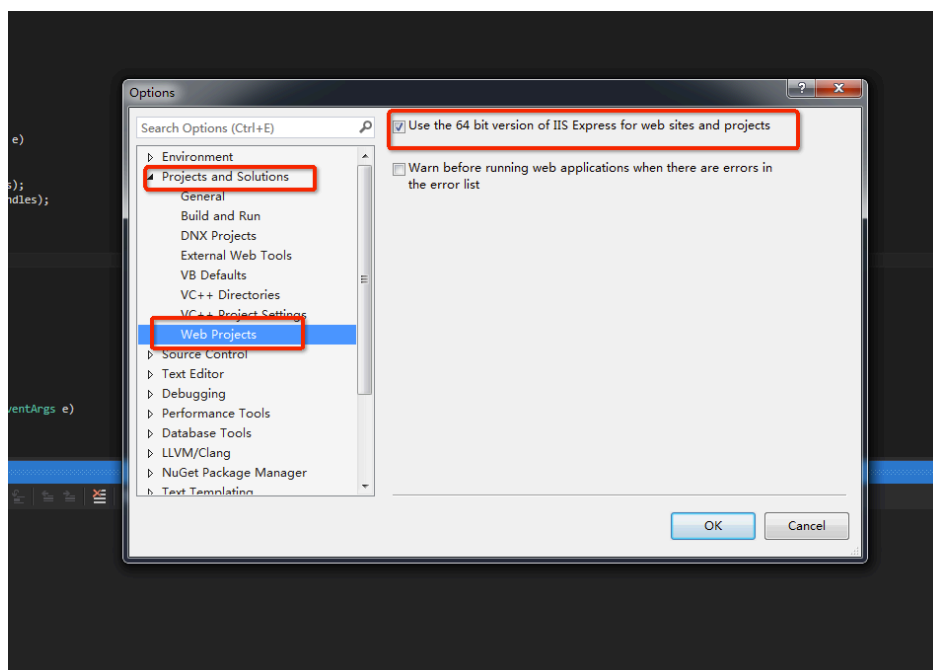
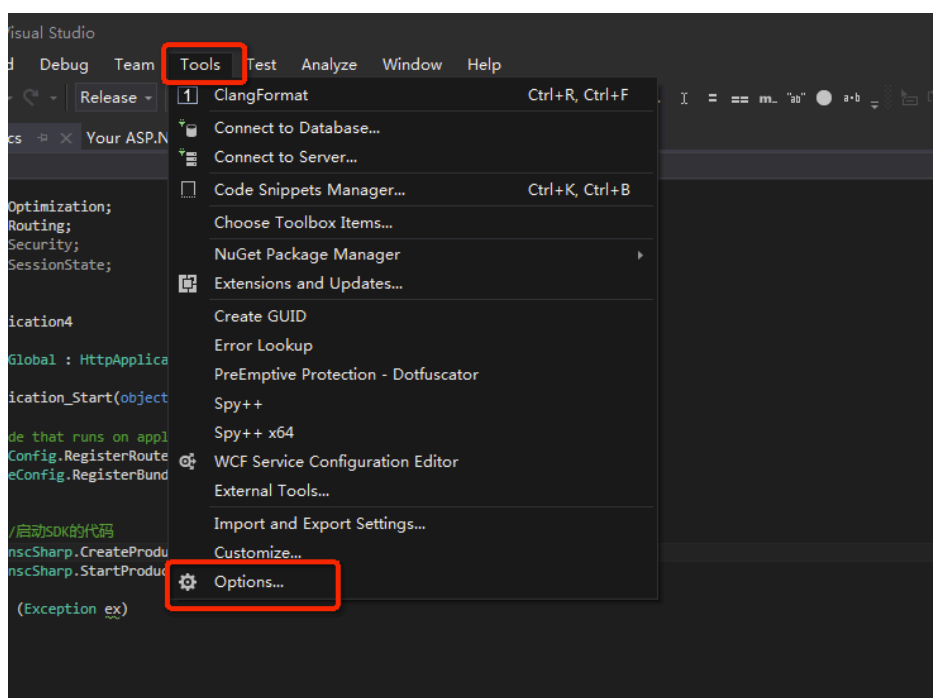
```

编写测试程序，然后进行编译。

将 SDK 下的 dll 放到和可执行文件同一级目录下，或者系统目录下即可运行。



依次点击**工具>选项>项目和解决方案>Web 项目**，然后勾选对网站和项目使用 **IIS Express** 的 64 位版。



收发普通消息

发送普通消息

您可以运行以下代码进行消息发送。请按说明正确设置相关参数。

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.InteropServices;
using ons;

namespace ons
{
    class onscsharp
    {
        static void Main(string[] args)
        {
            //Producer 创建和正常工作的参数，必须输入
            ONSFactoryProperty factoryInfo = new ONSFactoryProperty();
            factoryInfo.setFactoryProperty(factoryInfo.ProducerId, "PID_xxxx");//您在 MQ 控制台创建的 Producer ID
            factoryInfo.setFactoryProperty(factoryInfo.PublishTopics, "xxx");//您在 MQ 控制台创建的 Topic
            factoryInfo.setFactoryProperty(factoryInfo.MsgContent, "xxx");//msg content
            factoryInfo.setFactoryProperty(factoryInfo.AccessKey, "xxx");//AccessKey 阿里云身份验证，在阿里云服务器管理
控制台创建
            factoryInfo.setFactoryProperty(factoryInfo.SecretKey, "xxx");//SecretKey 阿里云身份验证，在阿里云服务器管
理控制台创建

            //创建 producer
            ONSFactory onsfactory = new ONSFactory();
            Producer pProducer = onsfactory.getInstance().createProducer(factoryInfo);

            //在发送消息前，必须调用 start 方法来启动 Producer，只需调用一次即可
            pProducer.start();

            Message msg = new Message(
                //Message Topic
                factoryInfo.getPublishTopics(),
                //Message Tag
                "TagA",
                //Message Body
                factoryInfo.getMessageContent()
            );

            // 设置代表消息的业务关键属性，请尽可能全局唯一。
            // 以方便您在无法正常收到消息情况下，可通过 MQ 控制台查询消息并补发。
            // 注意：不设置也不会影响消息正常收发
            msg.setKey("ORDERID_100");

            //发送消息，只要不抛出异常，就代表发送成功
            try
            {
                SendResultONS sendResult = pProducer.send(msg);
            }
            catch(ONSClientException e)
            {
                //发送失败处理
            }

            // 在应用退出前，必须销毁 Producer 对象，否则会导致内存泄露等问题

```



```
pProducer.shutdown();  
  
    }  
}  
}
```

订阅普通消息

关于订阅普通消息的说明和示例代码，详见[订阅消息](#)。

收发顺序消息

发送顺序消息

发送顺序消息的示例代码如下：

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
using System.Threading;  
using ons;  
  
namespace test  
{  
    public class OnscSharp  
    {  
        private static OrderProducer _orderproducer;  
        private static string Ons_Topic = "xxxxxxxxxxx";  
        private static string Ons_ProducerID = "xxxxxxxxxxx";  
        private static string Ons_AccessKey = "xxxxxxxxxxx";  
        private static string Ons_SecretKey = "xxxxxxxxxxxxxxxx";  
        private static string Ons_ConsumerId = "xxxxxxxxxxx";  
  
        public static void SendOrderMessage(string msgBody, String tag = "RegisterLog", String key = "test")  
        {  
            Message msg = new Message(Ons_Topic, tag, msgBody);  
            msg.setKey(Guid.NewGuid().ToString());  
            try  
            {  
                SendResultONS sendResult = _orderproducer.send(msg, shardingKey);  
                Console.WriteLine("send success {0}", sendResult.getMessageId());  
            }  
            catch (Exception ex)  
            {  

```

```

    Console.WriteLine("send failure{0}", ex.ToString());
}
}

public static void StartOrderProducer()
{
    _orderproducer.start();
}

public static void ShutdownOrderProducer()
{
    _orderproducer.shutdown();
}

private static ONSFactoryProperty getFactoryProperty()
{
    ONSFactoryProperty factoryInfo = new ONSFactoryProperty();
    factoryInfo.setFactoryProperty(ONSFactoryProperty.AccessKey, Ons_AccessKey);
    factoryInfo.setFactoryProperty(ONSFactoryProperty.SecretKey, Ons_SecretKey);
    factoryInfo.setFactoryProperty(ONSFactoryProperty.ConsumerId, Ons_ConsumerId);
    factoryInfo.setFactoryProperty(ONSFactoryProperty.ProducerId, Ons_ProducerID);
    factoryInfo.setFactoryProperty(ONSFactoryProperty.PublishTopics, Ons_Topic);
    return factoryInfo;
}

public static void CreateOrderProducer()
{
    _orderproducer = ONSFactory.getInstance().createOrderProducer(getFactoryProperty());
}
}

class OrderProducerForEx
{
    static void Main(string[] args)
    {
        OnscSharp.CreateOrderProducer();
        OnscSharp.StartOrderProducer();
        System.DateTime beforDt = System.DateTime.Now;
        for (int i = 0; i < 1000; ++i)
        {
            OnscSharp.SendOrderMessage("Test", "Tag", "key");
        }
        System.DateTime endDt = System.DateTime.Now;
        System.TimeSpan ts = endDt.Subtract(beforDt);
        Console.WriteLine("per message:{0}ms.", ts.TotalMilliseconds / 10000);
        OnscSharp.ShutdownOrderProducer();
    }
}
}

```

接收顺序消息

接收顺序消息的示例代码如下：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Threading;
using ons;

namespace test
{
    public class MyMsgOrderListener : MessageOrderListener
    {
        public MyMsgOrderListener()
        {

        }

        ~MyMsgOrderListener()
        {

        }

        public override ons.OrderAction consume(Message value, ConsumeOrderContext context)
        {
            Byte[] text = Encoding.Default.GetBytes(value.getBody());
            Console.WriteLine(Encoding.UTF8.GetString(text));
            return ons.OrderAction.Success;
        }
    }

    public class OnscSharp
    {
        private static OrderConsumer _orderconsumer;
        private static MyMsgOrderListener _order_listen;
        private static string Ons_Topic = "xxxxxxxxxx";
        private static string Ons_ProducerID = "xxxxxxxxxx";
        private static string Ons_AccessKey = "xxxxxxxxxx";
        private static string Ons_SecretKey = "xxxxxxxxxxxxxxxx";
        private static string Ons_ConsumerId = "xxxxxxxxxx";

        public static void StartOrderConsumer()
        {
            _order_listen = new MyMsgOrderListener();
            _orderconsumer.subscribe(Ons_Topic, "*", _order_listen);
            _orderconsumer.start();
        }

        public static void shutdownOrderConsumer()
        {
            _orderconsumer.shutdown();
        }
    }
}
```

```
private static ONSFactoryProperty getFactoryProperty()
{
    ONSFactoryProperty factoryInfo = new ONSFactoryProperty();
    factoryInfo.setFactoryProperty(ONSFactoryProperty.AccessKey, Ons_AccessKey);
    factoryInfo.setFactoryProperty(ONSFactoryProperty.SecretKey, Ons_SecretKey);
    factoryInfo.setFactoryProperty(ONSFactoryProperty.ConsumerId, Ons_ConsumerId);
    factoryInfo.setFactoryProperty(ONSFactoryProperty.ProducerId, Ons_ProducerID);
    factoryInfo.setFactoryProperty(ONSFactoryProperty.PublishTopics, Ons_Topic);
    factoryInfo.setFactoryProperty(ONSFactoryProperty.LogPath, "E://log");
    return factoryInfo;
}

public static void CreateOrderConsumer()
{
    _orderconsumer = ONSFactory.getInstance().createOrderConsumer(getFactoryProperty());
}
}

class OrderConsumerForEx
{
    static void Main(string[] args)
    {
        OnscSharp.CreateOrderConsumer();
        OnscSharp.StartOrderConsumer();
        Thread.Sleep(1000 * 1000);
        OnscSharp.shutdownOrderConsumer();
    }
}
}
```

收发定时消息

目前支持的域包括公网、华东1、华北2、华东2、华南1。

定时消息可以做到在指定时间之后才可被消费者消费，用于解决一些消息生产和消费有时间窗口要求的场景，或者通过消息触发定时任务的场景，类似于延迟队列。

发送定时消息

发送定时消息的代码示例如下：

```
using System;
```

```

using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.InteropServices;
using ons;

namespace ons
{
    class onscsharp
    {
        static void Main(string[] args)
        {
            //producer 创建和正常工作的参数，必须输入
            ONSFactoryProperty factoryInfo = new ONSFactoryProperty();
            factoryInfo.setFactoryProperty(factoryInfo.ProducerId, "XXX ");//您在 MQ 控制台创建的 Producer ID
            factoryInfo.setFactoryProperty(factoryInfo.PublishTopics, "XXX");//您在 MQ 控制台创建的 Topic
            factoryInfo.setFactoryProperty(factoryInfo.MsgContent, "XXX");//消息内容
            factoryInfo.setFactoryProperty(factoryInfo.AccessKey, "XXX");//AccessKey 阿里云身份验证，在阿里云服务器管
            理控制台创建
            factoryInfo.setFactoryProperty(factoryInfo.SecretKey, "XXX");//SecretKey 阿里云身份验证，在阿里云服务器管理
            控制台创建

            //创建 producer
            ONSFactory onsfactory = new ONSFactory();
            Producer pProducer = onsfactory.getInstance().createProducer(factoryInfo);

            //在发送消息前，必须调用 start 方法来启动 Producer，只需调用一次即可
            pProducer.start();

            Message msg = new Message(
                //Message Topic
                factoryInfo.getPublishTopics(),
                //Message Tag
                "TagA",
                //Message Body
                factoryInfo.getMessageContent()
            );

            // 设置代表消息的业务关键属性，请尽可能全局唯一。
            // 以方便您在无法正常收到消息情况下，可通过 MQ Console 查询消息并补发。
            // 注意：不设置也不会影响消息正常收发
            msg.setKey("ORDERID_100");

            // deliver time 单位 ms，指定一个时刻，在这个时刻之后才能被消费，这个例子表示3s 后才能被消费
            long deliverTime = 获取系统当前时间(ms) + 3000;
            msg.setStartDeliverTime(deliverTime);

            //发送消息，只要不抛出异常，就代表发送成功
            try
            {
                SendResultONS sendResult = pProducer.send(msg);
            }
            catch(ONSClientException e)
            {
                //发送失败处理
            }
        }
    }
}

```

```
// 在应用退出前，必须销毁 Producer 对象，否则会导致内存泄露等问题
pProducer.shutdown();

    }
}
}
```

订阅定时消息

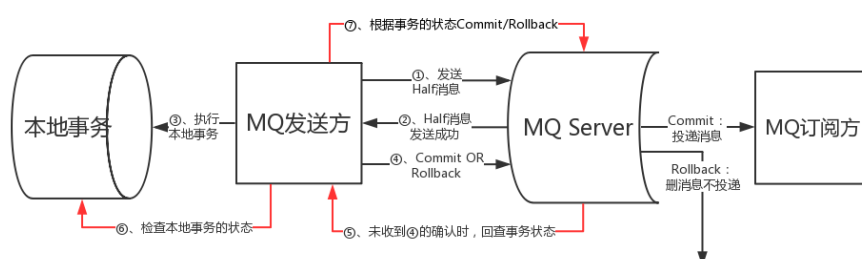
关于订阅普通消息的说明和示例代码，详见[订阅消息](#)。

收发事务消息

目前支持的域包括公网、华东1、华北2、华东2、华南1。

交互流程

MQ 事务消息的交互流程如下图所示：



发送事务消息

发送事务消息包含以下两个步骤：

发送半消息（Half Message）及执行本地事务。示例代码如下：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.InteropServices;
```

```

using ons;

namespace ons
{
    public class MyLocalTransactionExecuter : LocalTransactionExecuter
    {
        public MyLocalTransactionExecuter()
        {
        }

        ~MyLocalTransactionExecuter()
        {
        }

        public override TransactionStatus execute(Message value)
        {
            Console.WriteLine("execute topic: {0}, tag:{1}, key:{2}, msgId:{3},msgbody:{4}, userProperty:{5}",
                value.getTopic(), value.getTag(), value.getKey(), value.getMsgID(), value.getBody(),
                value.getUserProperty("VincentNoUser"));

            // 消息 ID(有可能消息体一样, 但消息 ID 不一样, 当前消息 ID 在控制台控制不可能查询)
            string msgId = value.getMsgID();
            // 消息体内容进行 crc32, 也可以使用其它的如 MD5
            // 消息 ID 和 crc32id 主要是用来防止消息重复
            // 如果业务本身是幂等的, 可以忽略, 否则需要利用 msgId 或 crc32Id 来做幂等
            // 如果要求消息绝对不重复, 推荐做法是对消息体 body 使用 crc32或 md5来防止重复消息.
            TransactionStatus transactionStatus = TransactionStatus.Unknow;
            try {
                boolean isCommit = 本地事务执行结果;
                if (isCommit) {
                    // 本地事务成功、提交消息
                    transactionStatus = TransactionStatus.CommitTransaction;
                } else {
                    // 本地事务失败、回滚消息
                    transactionStatus = TransactionStatus.RollbackTransaction;
                }
            } catch (Exception e) {
                //exception handle
            }
            return transactionStatus ;
        }
    }
}

class onscsharp
{
    static void Main(string[] args)
    {
        ONSFactoryProperty factoryInfo = new ONSFactoryProperty();
        factoryInfo.setFactoryProperty(factoryInfo.ProducerId, ""); //您在 MQ 控制台创建的 Producer ID
        factoryInfo.setFactoryProperty(factoryInfo.PublishTopics, ""); //您在 MQ 控制台创建的 Topic
        factoryInfo.setFactoryProperty(factoryInfo.MsgContent, ""); //message body
        factoryInfo.setFactoryProperty(factoryInfo.AccessKey, ""); //AccessKey 阿里云身份验证, 在阿里云服务器
        管理控制台创建
        factoryInfo.setFactoryProperty(factoryInfo.SecretKey, ""); //SecretKey 阿里云身份验证, 在阿里云服务器
        管理控制台创建

        //create transaction producer
    }
}

```

```

    ONSFactory onsfactory = new ONSFactory();
    LocalTransactionChecker myChecker = new MyLocalTransactionChecker();
    TransactionProducer pProducer = onsfactory.getInstance().createTransactionProducer(factoryInfo,ref
myChecker);

    //在发送消息前，必须调用 start 方法来启动 Producer，只需调用一次即可,启动之后可以多线程并发发送消息
    pProducer.start();

    Message msg = new Message(
        //Message Topic
        factoryInfo.getPublishTopics(),
        //Message Tag
        "TagA",
        //Message Body
        factoryInfo.getMessageContent()
    );

    // 设置代表消息的业务关键属性，请尽可能全局唯一。
    // 以方便您在无法正常收到消息情况下，可通过 MQ K 控制台查询消息并补发。
    // 注意：不设置也不会影响消息正常收发
    msg.setKey("ORDERID_100");

    //发送消息，只要不抛出异常，就代表发送成功
    try
    {
        LocalTransactionExecuter myExecuter = new MyLocalTransactionExecuter();
        SendResultONS sendResult = pProducer.send(msg, ref myExecuter);
    }
    catch(ONSClientException e)
    {
        Console.WriteLine("\nexception of sendmsg:{0}",e.what() );
    }

    // 在应用退出前，必须销毁 Producer 对象，否则会导致内存泄露等问题；
    // shutdown 之后不能重新 start 此 producer
    pProducer.shutdown();
}
}
}

```

提交事务消息状态

当本地事务执行完成（执行成功或执行失败），需要通知服务器当前消息的事务状态。通知方式有以下两种：

- 执行本地事务完成后提交；
- 执行本地事务一直没提交状态，等待服务器回查消息的事务状态。

事务状态有以下三种：

- TransactionStatus.CommitTransaction 提交事务，允许订阅方消费该消息；
- TransactionStatus.RollbackTransaction 回滚事务，消息将被丢弃不允许消费；
- TransactionStatus.Unknow 无法判断状态，期待 MQ Broker 向发送方再次询问该消息对应的本地事务的状态。


```

public class MyLocalTransactionChecker : LocalTransactionChecker
{
    public MyLocalTransactionChecker()
    {
    }

    ~MyLocalTransactionChecker()
    {
    }

    public override TransactionStatus check(Message value)
    {
        Console.WriteLine("check topic: {0}, tag:{1}, key:{2}, msgId:{3},msgbody:{4}, userProperty:{5}",
            value.getTopic(), value.getTag(), value.getKey(), value.getMsgID(), value.getBody(),
            value.getUserProperty("VincentNoUser"));
        // 消息 ID(有可能消息体一样, 但消息 ID 不一样, 当前消息 ID 在控制台控制不可能查询)
        string msgId = value.getMsgID();
        // 消息体内容进行 crc32, 也可以使用其它的如 MD5
        // 消息 ID 和 crc32id 主要是用来防止消息重复
        // 如果业务本身是幂等的, 可以忽略, 否则需要利用 msgId 或 crc32Id 来做幂等
        // 如果要求消息绝对不重复, 推荐做法是对消息体 body 使用 crc32或 md5来防止重复消息.
        TransactionStatus transactionStatus = TransactionStatus.Unknow;
        try {
            boolean isCommit = 本地事务执行结果;
            if (isCommit) {
                // 本地事务成功、提交消息
                transactionStatus = TransactionStatus.CommitTransaction;
            } else {
                // 本地事务失败、回滚消息
                transactionStatus = TransactionStatus.RollbackTransaction;
            }
        } catch (Exception e) {
            //exception handle
        }
        return transactionStatus ;
    }
}

```

事务回查机制说明

1、发送事务消息为什么必须要实现 check 机制？

当步骤 1 半消息发送完成，但本地事务返回状态为 TransactionStatus.Unknow 时，亦或是应用退出导致本地事务未提交任何状态时，从 MQ Broker 的角度看，这条半状态的消息的状态是未知的，因此 MQ Broker 会定期要求发送方能 check 该半状态消息，并上报其最终状态。

2、Check 被回调时，业务逻辑都需要做些什么？

MQ 事务消息的 check 方法里面，应该写一些检查事务一致性的逻辑。MQ 发送事务消息时需要实现 LocalTransactionChecker 接口，用来处理 MQ Broker 主动发起的本地事务状态回查请求；因此在事务消息的 check 方法中，需要完成两件事情：

（1）检查该半消息对应的本地事务的状态（committed or rollback）；

(2) 向 MQ Broker 提交该半消息本地事务的状态。

3、本地事务的不同状态对半消息的影响？

TransactionStatus.CommitTransaction 提交事务，允许订阅方消费该消息；

TransactionStatus.RollbackTransaction 回滚事务，消息将被丢弃不允许消费；

TransactionStatus.Unknow 无法判断状态，期待 MQ Broker 向发送方再次询问该消息对应的本地事务的状态。

具体代码详见 MyLocalTransactionChecker 的实现。

订阅事务消息

关于订阅普通消息的说明和示例代码，详见[订阅消息](#)。

订阅消息

本文介绍如何通过 MQ SDK 使用.NET 语言进行消息订阅。

说明：

- 请确保同一个 Consumer ID 下所有 Consumer 实例的订阅关系保持一致，详见[订阅关系一致](#)。
- TCP 接入点域名，请参见[TCP 接入说明](#)。

订阅方式

MQ 支持以下两种订阅方式：

集群订阅：同一个 Consumer ID 所标识的所有 Consumer 平均分摊消费消息。例如某个 Topic 有 9 条消息，一个 Consumer ID 有 3 个 Consumer 实例，那么在集群消费模式下每个实例平均分摊，只消费其中的 3 条消息。

```
// 集群订阅方式设置（不设置的情况下，默认为集群订阅方式）  
factoryInfo.setFactoryProperty(ONSFactoryProperty.MessageModel, ONSFactoryProperty.CLUSTERING);
```

广播订阅：同一个 Consumer ID 所标识的所有 Consumer 都会各自消费某条消息一次。例如某个

Topic 有 9 条消息，一个 Consumer ID 有 3 个 Consumer 实例，那么在广播消费模式下每个实例都会各自消费 9 条消息。

```
// 广播订阅方式设置
factoryInfo.setFactoryProperty(ONSFactoryProperty.MessageModel,
ONSFactoryProperty.BROADCASTING);
```

示例代码

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.InteropServices;
using ons;

namespace ons
{
    //pushConsumer 拉取到消息后，会主动调用该实例的 consumer 函数
    public class MyMsgListener : MessageListener
    {
        public MyMsgListener()
        {
        }

        ~MyMsgListener()
        {
        }

        public override Action consume(Message value, ConsumeContext context)
        {
            // Message 包含了消费到的消息，通过 getBody 接口可以拿到消息体
            Console.WriteLine(value.getBody());
            /*
             * 所有中文编码相关问题都在 SDK 压缩包包含的文档里做了说明，请仔细阅读
             */
            return ons.Action.CommitMessage;
        }
    }

    class onsharp
    {
        static void Main(string[] args)
        {
            //pushConsumer 创建和工作需要的参数，必须输入
            ONSFactoryProperty factoryInfo = new ONSFactoryProperty();
            factoryInfo.setFactoryProperty(factoryInfo.ConsumerId, "XXX");//您在 MQ 控制台创建的 Consumer ID
            factoryInfo.setFactoryProperty(factoryInfo.PublishTopics, "XXX");//您在 MQ 控制台创建的 Topic
            factoryInfo.setFactoryProperty(factoryInfo.AccessKey, "xx");//AccessKey 阿里云身份验证，在阿里云服务器管理控制台创建
        }
    }
}
```

```

        factoryInfo.setFactoryProperty(factoryInfo.SecretKey, "xxx");//SecretKey 阿里云身份验证，在阿里云服务器管理
控制台创建
        // 集群订阅方式 (默认)
        // factoryInfo.setFactoryProperty(ONSFactoryProperty.MessageModel, ONSFactoryProperty.CLUSTERING);
        // 广播订阅方式
        // factoryInfo.setFactoryProperty(ONSFactoryProperty.MessageModel,
ONSFactoryProperty.BROADCASTING);

        //创建 consumer 用于消费消息
        ONSFactory onsfactory = new ONSFactory();
        PushConsumer pConsumer = onsfactory.getInstance().createPushConsumer(factoryInfo);

        //给 consumer 注册消费回调，一旦有消息就会触发回调函数
        MessageListener msgListener = new MyMsgListener();
        pConsumer.subscribe(factoryInfo.getPublishTopics(), "*", msgListener);

        //启动 consumer
        pConsumer.start();
        //consumer 启动后，会异步拉取消息，拉取到消息后，回调 MyMsgListener 实例的 consumer 函数，将消息体通
过参数传递给 consumer

        // 这里可以继续做业务相关的逻辑处理，确定消费完成后，调用 shutdown 函数，释放资源。应用退出的时候也需要
调用 shutdown 函数。
        pConsumer.shutdown();
    }
}
}

```

HTTP 下线通告

通告：感谢大家对消息队列 HTTP 协议的支持。考虑到与消息服务（MNS）功能上的重复性，避免客户在选型上产生困惑，消息队列 HTTP 协议将于 2018.04.01 起逐步下线。因此，我们建议您从今起：

- HTTP 协议相关需求可前往 [消息服务 MNS](#)；
- MQ TCP 多语言客户端相关需求可前往 [消息队列 Kafka](#)，目前已支持包括 Java、C++、GO、NodeJs、PHP、Python、logStash、filebeat 以及 springcloud 等；
- 如有其它疑问，您可以通过 [提交工单](#) 进行反馈。