ALIBABA CLOUD

Alibaba Cloud

消息服务MNS

最佳实践

文档版本：20201123

阿里云

法律声明

阿里云提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。

2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。

4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的“现状”、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。

5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含“阿里云”、“Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。

6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

格式	说明	样例
危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	危险 重置操作将丢失用户配置数据。
警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	警告 重启操作将导致业务中断，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	注意 权重设置为0，该服务器不会再接受新请求。
说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	说明 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	单击**设置**> **网络**> **设置网络类型**。
粗体	表示按键、菜单、页面名称等UI元素。	在**结果确认**页面，单击**确定**。
`Courier字体`	命令或代码。	执行 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *Instance_ID*
[] 或者 [a|b]	表示可选项，至多选择一个。	`ipconfig [-all|-t]`
{} 或者 {a|b}	表示必选项，至多选择一个。	`switch {active|stand}`

目录

1.广播拉取消息模型

本文介绍如何使用消息服务MNS实现一对多拉取消息消费模型，以满足一对多订阅、主动拉取的场景。

背景信息

消息服务MNS提供队列（Queue）和主题（Topic）两种模型，基本能满足大多数应用场景。

- 队列提供的是一对一的共享消息消费模型，采用客户端主动拉取（Pull）模式。
- 主题模型提供一对多的广播消息消费模型，采用服务端主动推送（Push）模式。

推送模式的好处是即时性能较好，但需暴露客户端地址来接收服务端的消息推送。有些情况下有的信息，例如企业内网，无法暴露推送地址，希望改用拉取（Pull）的方式。虽然消息服务MNS不直接提供这种消费模型，但可以结合主题和队列来实现一对多的拉取消息消费模型。

解决方案

通过创建订阅，让主题将消息先推送到队列，然后由消费者从队列拉取消息。这样既可以做到一对多的广播消息，又可避免暴露消费者的地址。

接口说明

最新的Java SDK（1.1.8）中的CloudPullTopic默认支持上述解决方案。其中MNSClient提供以下接口来快速创建CloudPullTopic；

```
public CloudPullTopic createPullTopic(TopicMeta topicMeta, Vector<String> queueNameList, boolean need
CreateQueue, QueueMeta queueMetaTemplate)
public CloudPullTopic createPullTopic(TopicMeta topicMeta, Vector<String> queueNameList)
```

参数说明如下：

- TopicMeta：创建主题的参数设置。
- QueueMeta：创建队列的参数设置。
- queueNameList：指定主题消息推送的队列名列表。
- needCreateQueue：queueNameList是否需要创建。
- queueMetaTemplate：创建队列需要的QueueMeta参数示例。

示例代码

广播拉取消息的示例代码如下：

```
CloudAccount account = new CloudAccount(accessKeyId, accessKeySecret, endpoint);
    MNSClient client = account.getMNSClient();
    // 创建消费者列表。
    Vector<String> consumerNameList = new Vector<String>();
    String consumerName1 = "consumer001";
    String consumerName2 = "consumer002";
    String consumerName3 = "consumer003";
    consumerNameList.add(consumerName1);
```

```
    consumerNameList.add(consumerName2);
    consumerNameList.add(consumerName3);
    QueueMeta queueMetaTemplate = new QueueMeta();
    queueMetaTemplate.setPollingWaitSeconds(30);
    try{
      // 创建主题。
      String topicName = "demo-topic-for-pull";
      TopicMeta topicMeta = new TopicMeta();
      topicMeta.setTopicName(topicName);
      CloudPullTopic pullTopic = client.createPullTopic(topicMeta, consumerNameList, true, queueMetaTem
plate);
      // 发布消息。
      String messageBody = "broadcast message to all the consumers:hello the world.";
      // 如果发送原始消息，使用getMessageBodyAsRawString解析消息体。
      TopicMessage tMessage = new RawTopicMessage();
      tMessage.setBaseMessageBody(messageBody);
      pullTopic.publishMessage(tMessage);
      // 接收消息。
      CloudQueue queueForConsumer1 = client.getQueueRef(consumerName1);
      CloudQueue queueForConsumer2 = client.getQueueRef(consumerName2);
      CloudQueue queueForConsumer3 = client.getQueueRef(consumerName3);
      Message consumer1Msg = queueForConsumer1.popMessage(30);
      if(consumer1Msg != null)
      {
        System.out.println("consumer1 receive message:" + consumer1Msg.getMessageBodyAsRawString());
      }else{
        System.out.println("the queue is empty");
      }
      Message consumer2Msg = queueForConsumer2.popMessage(30);
      if(consumer2Msg != null)
      {
        System.out.println("consumer2 receive message:" + consumer2Msg.getMessageBodyAsRawString());
      }else{
        System.out.println("the queue is empty");
      }
      Message consumer3Msg = queueForConsumer3.popMessage(30);
      if(consumer3Msg != null)
      {
        System.out.println("consumer3 receive message:" + consumer3Msg.getMessageBodyAsRawString());
      }else{
        System.out.println("the queue is empty");
```

```
            System.out.println("the queue is empty");
        }
        // 删除主题。
        pullTopic.delete();
    }catch(ClientException ce)
    {
        System.out.println("Something wrong with the network connection between client and MNS service."
            + "Please check your network and DNS availablity.");
        ce.printStackTrace();
    }
    catch(ServiceException se)
    {
        // 更多错误码信息，请参见错误码。
        se.printStackTrace();
    }
    client.close();
```

2.消息处理时长自适应

本文介绍如何使消息处理时长自适应。

背景信息

消息服务MNS的规范中，每条消息都有个默认的VisibilityTimeout，Worker在接收到消息后，Timeout就开始计时了。 如果Worker在Timeout时间内没能处理完消息，那么消息就有可能被其他Worker接收到并处理。

Timeout计时的好处在于消息处理完之后需要显式地DeleteMessage，那么如果Worker进程停止等情况发生，这条消息还有机会被其他Worker处理。

一些用户会将队列的默认VisibilityTimeout设置得比较长，以确保消息在被Worker处理完之前不会超时释放。

问题描述

队列的VisibilityTimeout是6个小时。

一个Worker接收到了消息M1，但是Worker在处理完消息之后，进程发生了Crash或者机器发生了重启。

那么M1这条消息至少在6个小时之后才会被另一个Worker接收到并处理。而自己写代码处理Failover的情况的话，程序又会变得比较复杂。

目标

在一些即时性要求比较高，并且又希望尽快响应每一条消息的场景下，用户会希望：

- 队列的VisibilityTimeout比较短，例如5分钟。在发生了进程Crash后，最多5分钟，未处理完的消息就会被某个Worker接收到并处理。
- Worker处理消息的过程中，耗时很有可能超过5分钟，那么消息在被处理的过程中不能超时。

解决方案

Best Practice的C# Demo可以实现这样的场景。Demo下载地址为Sample。具体的做法是，在Worker处理消息的过程中，为消息定期检查是否需要做ChangeVisibility，Worker处理完之后依然是主动deleteMessage。

遇到Demo使用问题，可提交工单处理。

下面是对于程序的几点说明：

- 运行前需要填写accessId、accessKey、EndPoint。
- 变量说明：
  - MessageMinimalLife是消息注册时必须有的最少的Life长度。需要这个的原因是，例如消息register的时候已经只剩下0.1秒的超时时间了，注册进来也来不及ChangeVisibility延长生命。所以，MessageMinimalLife是为了确保消息能活到被ChangeVisibility，用户可以自己根据业务压力来设置。
  - TimerInterval是Manager内部的Timer的Interval。只需要确保在Message到达MessageMinimalLife之前，Timer会被启动就足够了。时间可以设置得比较短（检查得就比较频繁）。
  - QueueMessageVisibilityTimeout是Sample里Message的默认超时时间，是Queue的属性。Sample里每次ChangeVisibility的时候，会把消息的VisibilityTimeout设置为QueueMessageVisibilityTimeout，所以它的值需要大于TimerInterval+ MessageMinimalLife，以确保消息不会超时。
  - MessageTimeout是Message在Manager里面的超时时间，例如某个Worker卡住了，消息在5个小时之后依然没有处理完毕（假设5个小时远远超出消息的正常处理时间），那么Manager就不会再为消息做ChangeVisibility了，会放任消息的Visibility超时。

- 流程说明：
  - Worker在ReceiveMessage之后，会先做RegisterMessage，然后处理Message，最后再调用Manager的deleteMessage。
  - Manager在消息第一次注册进来之后，调用ThreadPool调度一个ChangeVisibilityTask检查是否需要ChangeVisibility，并且把Message加到内部的messages列表中。
  - Manager内部的Timer，会定时调用Parallel启动 ChangeVisibilityTask检查messages列表里的所有message。
  - Manager.ChangeMessageVisibility (ChangeVisibilityTask)相关的具体事情，在流程图里有显示。流程图如下：

3.长轮询

本文介绍消息服务MNS中使用LongPolling的最佳实践。

LongPolling 长轮询

背景信息

- 消息服务MNS提供了LongPolling类型的接收消息的方法，在接收消息的时候把等待时间设置在1s~30s。使用LongPolling可以让请求一直在服务器上运行，等到有消息的时候才返回，保证第一时间收到消息的同时也避免您发送大量无效请求。LongPolling是消息服务MNS的推荐用法。

  LongPolling参数设置详情请参见ReceiveMessage。

- LongPolling需要保持HTTP层的长连接在服务器上，而对于服务器来说，HTTP层的长连接的资源是有限的。为了避免受到恶意攻击，消息服务MNS对单用户的每个Queue的连接数是有限制的。详情请参见使用限制。

问题描述

您在单台机器上打开上百个线程同时访问消息服务MNS服务器获取消息，队列中没有消息时，单台机器上发送了上百个LongPolling的请求。如果您同时使用了较多机器，那么可能同时发送上千个LongPolling的请求。

这时您再发LongPolling，消息服务MNS服务器就会返回：消息不存在。

有时您是在一个While循环里不停的做LongPolling请求，却没有做异常处理，最后发出了极大量的请求，不能达到使用LongPolling的预期效果。

解决方案

当您打开上百个线程同时访问消息服务MNS服务器时，如果队列里已经没有消息，就不需要上百个线程同时运行LongPolling。此时只需要打开1~N个线程运行LongPolling。当运行LongPolling的线程发现队列里有消息时，唤醒其他线程一起接收消息，达到快速响应的目的。

长轮询示例代码是一个使用MessageReceiver获取消息的最佳实践。所有获取消息的线程都新建了MessageReceiver，使用receiver.receiveMessage来获取消息。

MessageReceiver内部做了LongPolling的排他机制，只要有一个线程在做LongPolling，其它线程只需要等待。

4.消息加密传输

本文介绍如何进一步提高您与消息服务MNS服务之间的网络链路上的安全性。

背景信息

消息服务MNS提供在公网HTTPS上进行消息加密传输的服务。对于包含敏感信息的消息，进一步提高您与阿里云服务之间的网络链路上的安全性，目前有以下两种解决方案：

- 消息服务MNS提供HTTPS的服务域名，您可以选用HTTPS服务地址。
- 在客户端对传输的消息体进行加密，防止被窃取。

解决方案

以下是消息服务MNS对消息进行加密传输的解决方案。

1. 在消息发送端先对消息进行加密，然后再发送。
2. 在消息接收端先对消息进行解密，然后再消费。

示例代码：SecurityQueue.zip，其中：

- *SeurityQueue.java*提供putMessage、popMessage和deleteMessage接口。接口说明如下：
  - putMesssage在向服务器发消息前，对消息体按指定的Key和加密算法加密。
  - popMessage在接收到服务端消息后，先按指定的方式解密，然后再返回解密后的消息体。
- *SecurityKeyGenerator.java*用于生成加解密需要的secretKey。
- *SecurityQueueDemo.java*提供如何使用SeurityQueue的Demo程序。

详情请参见*ReadMe.txt*。

注意事项

- 加密消息和解密消息对性能会有一定影响。
- 请不要往加密队列里发送非加密的消息。

5.严格保序队列

本文介绍如何使用队列实现按顺序发送和消费消息。

背景信息

消息服务MNS提供的队列（Queue）主要的特点是高可靠、高可用、高并发。每个队列的数据都会被持久化三份到阿里云的飞天分布式平台。其中每个队列至少有两台服务器向外提供服务，同时每台服务器都支持高并发访问。这些分布式特性导致了消息服务MNS的队列无法像传统单机队列严格保证消息FIFO，只能做到基本有序。

队列如果同时有多个消息发送者（Sender），由于并发和网络延迟等问题，根本无法获知消息的实际发送顺序和消息到达服务器端的真实顺序。同理，当有多个接收者并发接收消息时，其真正的处理顺序也不可获知。

综上所述，只有一个发送者（一个进程，可以是多个线程）、一个接收者时，消息顺序才有意义，也只有在这种情况下才能感知和记录消息的真实发送和接收顺序。

解决方案

基于上述假设，同时为了满足部分用户对于消息消费顺序性的要求，设计了以下方案，确保消息按照用户发送顺序被接收和消费。

1. 消息在发送端进行染色，加上SeqId（例如：#num#）。
2. 消息在接收端进行还原，并根据SeqId排序后返回给上层，同时对于已经接收的消息会有后台线程保证消息不会被重复消费。
3. 为了避免因为发送者或者接收者fail导致SeqId丢失。SeqId会被持久存储到本地磁盘文件，或者其他存储和数据库，例如OSS、OTS或RDS。

注意事项

- 本文的主要目的是展示顺序消息的解决方案，示例代码未经过严格测试，不建议不加测试直接用于生产环境。同时程序仓促完成，难免有瑕疵，欢迎指正。
- 正常情况下，发送端和接收端的SeqId应该和队列中的消息（染色）匹配。当出现删除队列重新创建等操作时，请注意磁盘文件中的SeqId是否和队列中的真实情况相符，同时建议不要往染色的消息队列里发送非染色消息。
- 队列的消息有效期设置过短或者每条消息的实际处理结果都有可能会对消息有序性造成影响，在您的程序中需要对这些情况所导致的乱序现象进行处理。

示例代码

本文以Python版的方案实现为例，下载地址：有序队列Python示例代码（依赖MNS Python SDK）。其中，主要提供了OrderedQueueWrapper类（*ordered_queue.py*文件），可以将普通队列包装成有序队列。

OrderedQueueWrapper提供SendMessageInOrder()和ReceiveMessageInOrder()方法：

- 发送时对消息进行染色。
- 接收时还原消息，并且按顺序返回给接收者。

另外，*send_message_in_order.py*和*receive_message_in_order.py*提供发送者和接收者使用OrderedQueueWrapper的程序示例。

```
send_message_in_order.py:
  #init orderedQueue
  seqIdConfig = {"localFileName":"/tmp/mns_send_message_seq_id"} # 指定持久化发送SeqId的磁盘文件。
  seqIdPS = LocalDiskStorage(seqIdConfig)
  orderedQueue = OrderedQueueWrapper(myQueue, sendSeqIdPersistStorage = seqIdPS)
  orderedQueue.SendMessageInOrder(message)
receive_message_in_order.py:
  #init orderedQueue
  seqIdConfig = {"localFileName":"/tmp/mns_receive_message_seq_id"} # 指定持久化接收SeqId的磁盘文件。
  seqIdPS = LocalDiskStorage(seqIdConfig)
  orderedQueue = OrderedQueueWrapper(myQueue, receiveSeqIdPersistStorage = seqIdPS)
  recv_msg = orderedQueue.ReceiveMessageInOrder(wait_seconds)
```

运行方法

1. 配置*send_message_in_order.py*和*receive_message_in_order.py*中的配置项g_endpoint、g_accessKeyId、g_accessKeySecret以及g_testQueueName。
2. 运行*send_message_in_order.py*。

   ```
   python send_message_in_order.py
   ```

3. 运行*receive_message_in_order.py*。

   ```
   python receive_message_in_order.py
   ```

   发送程序会发送20条消息，接收程序会按顺序消费这20条消息。

运行*ordered_queue.py*的测试结果对比普通队列，运行命令如下：

```
python ordered_queue.py
```

> **说明** *ordered_queue.py*需配置Endpoint和AccessKey。

运行结果

运行结果如下：

- 非严格有序

  整体有序，部分相邻消息无序，说明单个队列有多个服务器在同时服务。

- 严格有序

6.超大消息传输

本文介绍如何基于消息服务MNS和OSS，不做消息切片来传递大于64 KB的消息。

背景信息

消息服务MNS的队列的消息大小最大限制是64 KB，这个限制基本能够满足在正常情况下消息作为控制流信息交换通道的需求。但是，在某些特殊场景下，消息数据比较大时，就只能采用消息切片的方式。

下面是不做消息切片，通过OSS来实现传递大于64 KB的消息的解决方案。

解决方案

1. 生产者在向消息服务MNS发送消息前，如果发现消息体大于64 KB，则先将消息体数据上传到OSS。
2. 生产者把数据对应的Objcet信息发送到消息服务MNS。
3. 消费者从消息服务MNS队列里读取消息，判断消息内容是否为OSS的Object信息。
4. 判断消息内容是OSS的Object信息，则从OSS下载对应的Object内容，并作为消息体返回给上层程序。

具体过程如下图所示。

示例代码

大消息示例代码提供了上述方案的一个Java语言版实现。主要功能都封装成BigMessageSizeQueue类。

BigMessageSizeQueue提供的public方法如下：

```
//构造函数，cq为普通的mnsqueue对象，ossClient和ossBucketName包含了大消息中转的oss region和bucket。
public BigMessageSizeQueue(CloudQueue cq, OSSClient ossClient, String ossBucketName)
// 发送消息。
public Message putMessage(Message message)
// 接收消息。
public Message popMessage(int waitSeconds)
// 删除消息。
public void deleteMessage(String receiptHandle)
//设置大消息的阈值（大于这个值的消息会走OSS），默认64 KB。
public void setBigMessageSize(long bigMessageSize)
// 设置是否需要删除OSS上的消息，默认yes。
public void setNeedDeleteMessageObjectOnOSSFlag(boolean flag)
```

具体使用示例代码请参见大消息示例代码中的Demo.java。

注意事项

- 大消息主要消费网络带宽，用该方案发送大消息时，生产者和消费者的网络带宽可能会是瓶颈。
- 大消息网络传输时间较长，受网络波动影响的概率更大，建议在上层做必要的重试。

7.事务消息

本文介绍如何使用消息服务MNS的延时消息功能来实现本地操作和消息发送的事务一致性。

背景信息

有些业务场景需要实现本地操作和消息发送的事务一致性，即消息发送成功，则本地操作成功；消息发送失败，则本地操作失败，避免出现操作成功但消息发送失败，或者操作失败但消息发送成功的情况。

另外，消息在消费端一定被成功处理一次，不会因为消费端程序崩溃而导致消息没有成功处理，进而需要人工重置消费进度。

解决方案

消息服务MNS的延时消息功能可以实现本地操作和消息发送的事务一致性。

准备工作

创建以下队列：

- 事务消息队列

  消息的有效期小于消息延时时间。即如果生产者不主动修改消息可见时间，消息对消费者不可见。

- 操作日志队列

  记录事务消息的操作记录信息。消息的延时时间为事务操作超时时间。日志队列中的消息确认（删除）后将对消费者不可见。

流程简介

流程如下图所示。

事务消息的操作步骤如下：

1. 生产者发送一条事务准备消息到事务消息队列。
2. 生产者写操作日志信息到操作日志队列，日志中包含步骤1消息的消息句柄。
3. 生产者执行本地事务操作。
4. 如果步骤3成功，提交消息（消息对消费者可见）；反之，回滚消息。
5. 确认步骤2中的操作日志（删除该日志消息）。
6. 步骤4后，消费者可以接收到事务消息。
7. 消费者处理消息。
8. 消费者确认删除消息。

异常分析

- 生产者异常（例如进程重启）
  i. 读取操作日志队列超时未确认日志。
  ii. 检查事务结果。
  iii. 如果检查得到事务已经成功，则提交消息（重复提交不会导致消息重复消费，同一句柄的消息只能成功提交一次）。
  iv. 确认操作日志。

- 消费者异常（例如进程重启）

  消息服务MNS保证消息至少消费一次，只要步骤8不成功，消息在一段时间后会继续成为可见状态，被当前消费者或者其他消费者处理。

- 消息服务MNS服务不可达（例如断网）

  消息发送和接收处理状态以及操作日志都在消息服务MNS服务端，消息服务MNS本身具备高可靠和高可用的特点，所以只要网络恢复，事务可以继续。如果生产者操作成功，消费者一定能够拿到消息并处理成功；如果生产者操作失败，消费者收不到消息。

示例代码

消息服务MNS最新的Java SDK（1.1.8）中的TransactionQueue支持上述事务消息方案。
在TransactionOperations和TransactionChecker两个接口添加业务操作和检查逻辑，您就可以方便地实现事务消息。

示例代码如下：

```
public class TransactionMessageDemo{
  public class MyTransactionChecker implements TransactionChecker
  {
    public boolean checkTransactionStatus(Message message)
    {
      boolean checkResult = false;
      String messageHandler = message.getReceiptHandle();
      try{
        //检查messageHandler事务是否成功。
        checkResult = true;
      }catch(Exception e)
      {
        checkResult = false;
      }
      return checkResult;
    }
  }
  public class MyTransactionOperations implements TransactionOperations
  {
    public boolean doTransaction(Message message)
    {
      boolean transactionResult = false;
      String messageHandler = message.getReceiptHandle();
      String messageBody = message.getMessageBody();
      try{
        //根据messageHandler和messageBody执行本地事务。
        transactionResult = true;
      }catch(Exception e)
```

```
        {
          transactionResult = false;
        }
        return transactionResult;
      }
    }
    public static void main(String[] args) {
      System.out.println("Start TransactionMessageDemo");
      String transQueueName = "transQueueName";
      String accessKeyId = ServiceSettings.getMNSAccessKeyId();
      String accessKeySecret = ServiceSettings.getMNSAccessKeySecret();
      String endpoint = ServiceSettings.getMNSAccountEndpoint();
      CloudAccount account = new CloudAccount(accessKeyId, accessKeySecret, endpoint);
      MNSClient client = account.getMNSClient(); //初始化客户端。
      //为事务队列创建队列。
      QueueMeta queueMeta = new QueueMeta();
      queueMeta.setQueueName(transQueueName);
      queueMeta.setPollingWaitSeconds(15);
      TransactionMessageDemo demo = new TransactionMessageDemo();
      TransactionChecker transChecker = demo.new MyTransactionChecker();
      TransactionOperations transOperations = demo.new MyTransactionOperations();
      TransactionQueue transQueue = client.createTransQueue(queueMeta, transChecker);
      //执行事务。
      Message msg = new Message();
      String messageBody = "TransactionMessageDemo";
      msg.setMessageBody(messageBody);
      transQueue.sendTransMessage(msg, transOperations);
      //如果不使用队列，请删除队列并关闭客户端。
      transQueue.delete();
      //关闭客户端。
      client.close();
      System.out.println("End TransactionMessageDemo");
    }
}
```

8.过滤消息

本文介绍消息服务MNS如何通过消息过滤标签把消息推送到不同的推送目标。

背景信息

通常情况下，在主题中创建订阅消息服务MNS可以把消息推送到订阅的推送目标。即消息和订阅没有设置消息过滤标签，所有消息都可以被成功推送到推送目标。

使用消息过滤标签功能时，消息的消息过滤标签和订阅的消息过滤标签一致，消息才能被成功推送到推送目标。即消息设置了消息过滤标签，订阅也设置了消息过滤标签，两个消息过滤标签一致，消息可以被成功推送到推送目标。

特殊情况下，当订阅没有设置消息过滤标签，无论消息是否设置了消息过滤标签，都可以被成功推送到推送目标。

应用场景

一些场景中需要根据消息内容把消息推送到不同的推送目标。为了达到这一功能，您可以创建多个主题，并为每个主题设置相应的推送目标。但是这样会增加额外的成本，并且增加了运维的复杂度。为了避免这种情况，消息服务MNS提供了消息过滤标签功能。您可以只创建一个主题，并在创建订阅时设置不同的消息过滤标签，结合消息的消息过滤标签消息服务MNS就可以把消息推送到不同的推送目标中。

消息过滤示例

下图介绍了不同消息过滤标签的消息经过订阅的消息过滤标签属性过滤后到达的目标队列。

在上图示例场景中，在主题Topic 1创建3个标签不同的订阅Subscription 1、Subscription 2和Subscription 3，这4个订阅的推送目标分别是Queue 1、Queue 2和Queue 3。

1. 消息的消息过滤标签和订阅的消息过滤标签一致。消息过滤过程如下：
   - 消息服务MNS将Message 1推送到队列Queue 1。
   - 消息服务MNS将Message 2推送到队列Queue 2。
2. 订阅没有消息过滤标签。消息过滤过程如下：
   - 消息服务MNS将Message 1推送到队列Queue 3。
   - 消息服务MNS将Message 2推送到队列Queue 3。
   - 消息服务MNS将Message 3推送到队列Queue 3。

9.实践问题

9.1. MNS是否支持长轮询?

本文介绍消息服务MNS是否支持长轮询的问题。

消息服务MNS支持长轮询。与传统的短轮询相比，长轮询只会在消息进入队列或长轮询超时时才返回响应。一旦消息可用，长轮询可立即以简单经济的方式从您的队列检索消息。详情请参见长轮询。

9.2. MNS是否提供对消息的先入先出（FIFO）访问?

本文介绍消息服务MNS能否提供消息先入先出访问的问题。

消息服务MNS消费消息时尽量做到先进先出，正是因为分布式消息队列的一些特性并不能保证您能按照消息的发送顺序消费消息，如果您的业务必须先进先出，建议在消息中加入序号信息以便消费消息后进行重新排序，详情请参见严格保序队列。