消息服务 MNS

最佳实践

最佳实践

广播拉取消息模型

如何实现一对多拉取消息消费模型

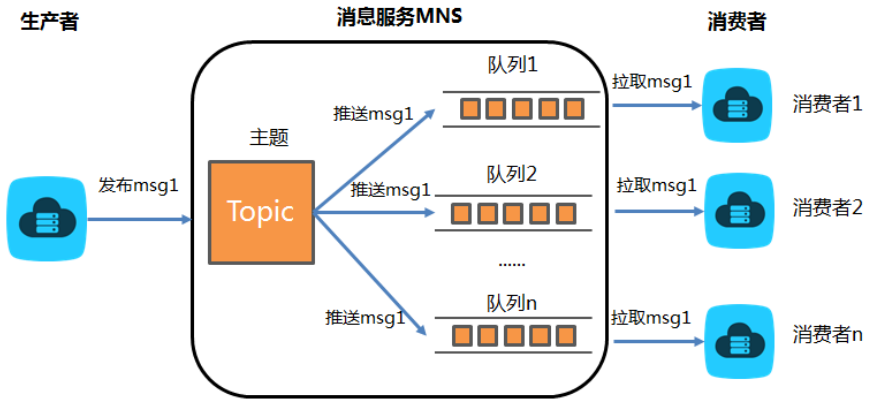
背景描述

阿里云消息服务MNS 已经提供队列（queue）和主题（topic）两种模型。其中队列提供的是一对多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，并且采用服务端主动**推送（Push）**模式。上面两种模型基本能满足我们大多数应用场景。

推送模式的好处是即时性能比较好，但是需要暴露客户端地址来接收服务端的消息推送。有些情况下，比如企业内网，我们无法暴露推送地址，希望改用拉取（Pull）的方式。虽然MNS不直接提供这种消费模型，但是我们可以结合主题和队列来实现一对多的拉取消息消费模型。具体方案如下：

解决方案

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1对多的广播消息，又不需要暴露消费者的地址；如下图所示：



接口说明

最新的Java SDK（1.1.5）中的CloudPullTopic 默认支持上述解决方案。其中MNSClient 提供下面两个接口来快速创建CloudPullTopic：

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

public CloudPullTopic createPullTopic(TopicMeta topicMeta, Vector<String> queueNameList)
```

其中，TopicMeta 是创建topic的meta 设置， queueNameList里指定topic消息推送的队列名列表；needCreateQueue表明queueNameList是否需要创建；queueMetaTemplate是创建queue需要的queue meta 参数设置；

Demo代码

```
CloudAccount account = new CloudAccount(accessKeyId, accessKeySecret, endpoint);
MNSClient client = account.getMNSClient();

// build consumer name list.
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{
//producer code:
// create pull topic which will send message to 3 queues for consumer.
```

```
String topicName = "demo-topic-for-pull";
TopicMeta topicMeta = new TopicMeta();
topicMeta.setTopicName(topicName);
CloudPullTopic pullTopic = client.createPullTopic(topicMeta, consumerNameList, true, queueMetaTemplate);

//publish message and consume message.
String messageBody = "broadcast message to all the consumers:hello the world.";
// if we sent raw message,then should use getMessageBodyAsRawString to parse the message body correctly.
TopicMessage tMessage = new RawTopicMessage();
tMessage.setBaseMessageBody(messageBody);
pullTopic.publishMessage(tMessage);

// consumer code:
//3 consumers receive the message.
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");
}

// delete the pullTopic.
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)
{
/*you can get more MNS service error code in following link.
https://help.aliyun.com/document_detail/mns/api_reference/error_code/error_code.html?spm=5176.docmns/api_reference/error_code/error_response
*/
```

```
se.printStackTrace();
}

client.close();
```

消息处理时长自适应

如何自适应消息处理时长

背景描述

阿里云MNS消息服务的规范中，每条Message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Message，那么消息就有可能被其他Worker接收到并处理。

Timeout计时的好处是：消息处理完之后需要显式地DeleteMessage，那么如果Worker进程Crash等情况发生，这条Message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

但是，在一些应用场景中，我们假设：

队列的VisibilityTimeout是6个小时。

A worker接收到了Message 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个小时之后才会被某个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，我们会希望：

队列的 VisibilityTimeout 比较短。比如是5分钟，这样的话，在发生了进程 Crash 之后，最多5分钟，之前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。

实现方案

对于这样的场景，我们提供一个BestPractice的C# Demo：（请见附件）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。

如果有任何问题，欢迎大家在论坛发帖，或者直接加入我们的官方MNS技术支持旺旺群（群号51222373）。

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，比如消息 register 的时候已经只剩下 0.1 秒的超时时间了，那么注册进来也来不及 ChangeVisibility 延长生命。所以，MessageMinimalLife 是为了确保 Message 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，比如某个 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 Message 做 ChangeVisibility 了，会放任 Message 的 Visibility 超时。

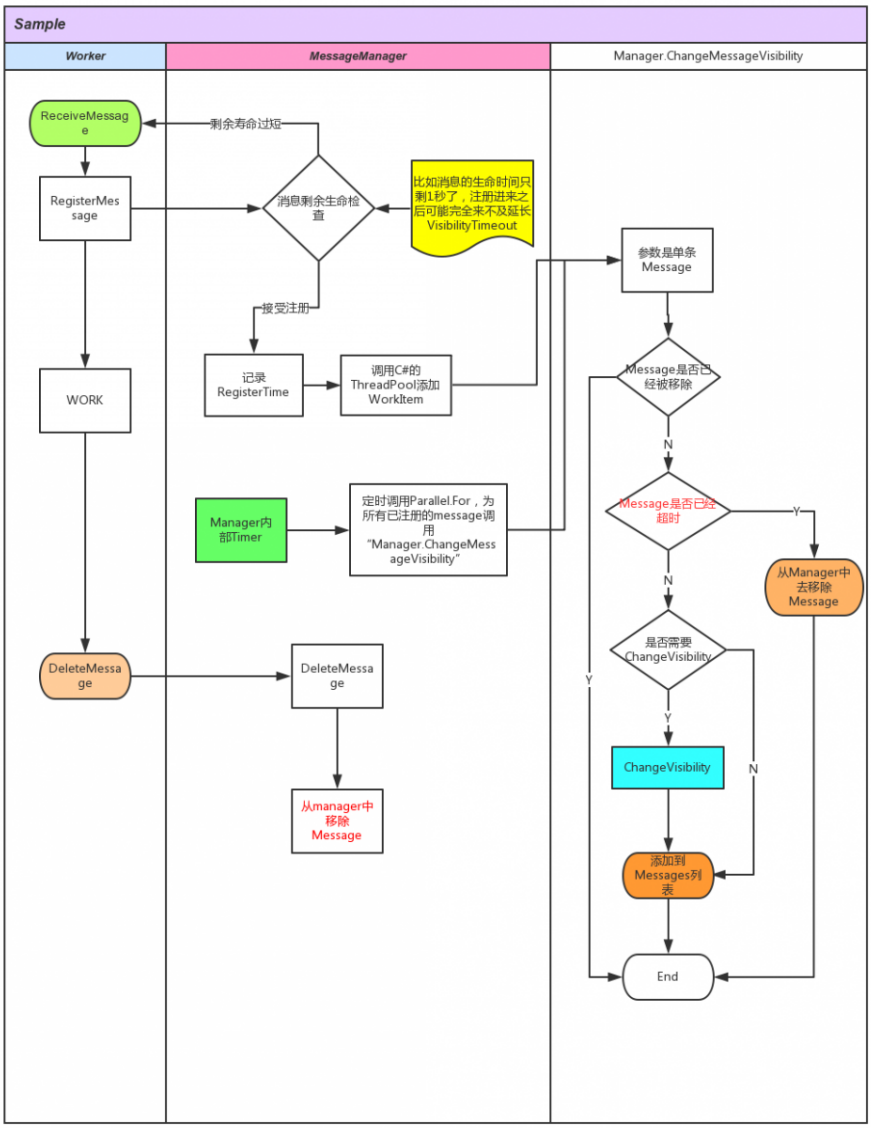
流程说明：

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：



示例代码下载：ChangeMessageVisibilitySample

长轮询

长轮询（LongPolling）

背景知识

MNS提供了LongPolling类型的ReceiveMessage的方法，只需要在ReceiveMessage的时候把WaitSecond设为一个1-30之间的数就可以了。使用LongPolling可以让Request一直挂在Server上，等到有Message的时候才返回，在保证了第一时间收到消息的同时也避免用户发送大量无效Request。LongPolling也是MNS的推荐用法。

LongPolling是需要挂HTTP层的长连接在Server上，而对于Server来说，HTTP层的长连接的资源是比较有限的。为了避免受到一些恶意攻击，所以MNS对单用户的LongPolling连接数是有限制的。

问题描述

有一些用户在单台机器上开了上百个线程同时访问MNS Server获取消息，遇到队列中没有消息的时候，单台机器上就挂了上百个LongPolling的Request。如果用户还同时使用了比较多的机器，那么这些用户就可能会需要同时发上千个LongPolling的请求。

这种情况下，用户在发LongPolling的Request的时候，就会比较容易遇到: MNS的Server直接返回“消息不存在”，而不是Request一直挂在Server端等待消息。

这会导致用户不能得到预期的LongPolling的效果。有一些用户是在一个While循环里面做不停的LongPolling请求而没有做一些异常处理，然后一夜醒来发现发出了极大量的请求。

解决方案

在开了上百个线程同时访问的情况下，如果队列里已经没有消息了，那么其实不需要上百个线程都同时挂LongPolling。只需要有1-N个线程挂LongPolling就足够了。挂LongPolling的线程在发现队列里有消息时，可以唤醒其他线程一起来取消息以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BestPractice。所有取消息的线程，都是new了一个MessageReceiver，然后使用receiver.receiveMessage来获取消息。

Receiver内部做了LongPolling的排他机制，只要有一个线程在做LongPolling，那么其他线程只需要Wait就可以了。

消息加密传输

背景描述

阿里云消息服务MNS提供公网http可访问的服务。对于包含敏感信息的消息，如何进一步提高从用户客户端程序到阿里云的服务之间的网络链路上的安全性？目前有两种解决方案： 1.MNS提供https的服务域名（计划中，预计4月中旬可用），用户选用https服务地址。 2.用户对传输的消息体进行加密，防止被窃取。

下面提供了如何对MNS消息进行加密传输的最佳实践。

解决方案

在消息发送前先消息进行加密，然后在发送；

在消息接收端先对消息进行解密，然后在消费；

示例代码（Java）：SecurityQueue.zip

1.SeurityQueue.java：提供putMessage，popMessage和deleteMessage3个接口，

putMesssage会在向服务器发消息前对messageBody按指定的key和加密算法加密。

popMessage 则在接收到服务端message后先按指定的方式解密然后，然后再返回解密后的消息体。

2.SecurityKeyGenerator.java 用于生成加解密需要的secretKey。

3.SecurityQueueDemo.java 提供了如何使用SeurityQueue的demo程序。

其他详见附件中的ReadMe。

注意事项

加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。

严格保序队列

问题背景

阿里云消息服务提供的队列（queue）主要特点是高可靠、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2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，也导致了消息服务队列无法像传统单机队列那样保证严格的消息FIFO特点，只能做到基本有序。

我们的队列如果同时有多个消息发送者（sender），由于并发和网络延迟不一等问题，消息的严格顺序本身就是失去了意义，因为在这种情况下，我们根本无法获知消息在多个sender上的实际发送顺序和消息到达服务器端的真实顺序。同样的，当有多个接收者并发接收消息时，其真正的处理顺序也是不可获知的。

因此，我们认为只有一个发送者（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我们能够感知和记录消息的真实发送顺序和消息的真实接收顺序。

解决方案

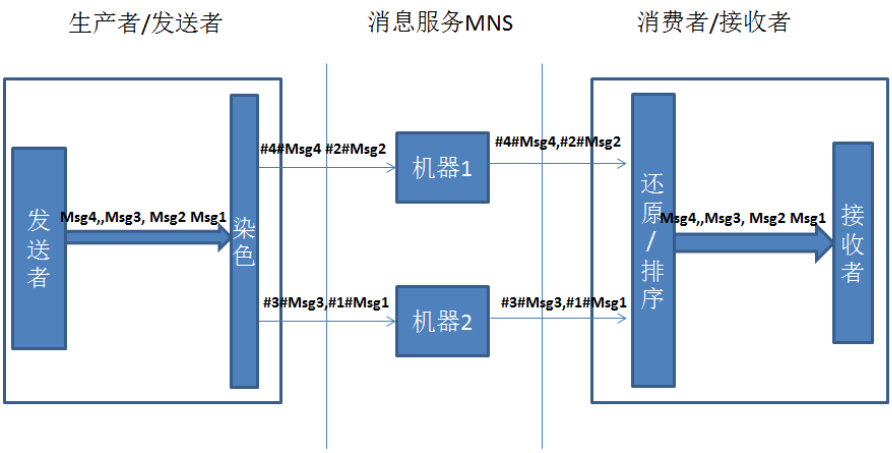
基于上述假设，同时为了满足部分用户对于消息消费顺序性的要求，我们设计了下面的方案，确保消息按照用户发送顺序被接收和消费。

主要步骤：

消息在发送端进行染色，加上SeqId（例如：#num#）

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receive的消息会有后台线程保证消息不会被重复消费。

为了避免因为发送者fail，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。(当然也可以存储到其他存储或数据库：例如OSS，OTS，RDS)



程序说明

附件提供了python版的方案实现（依赖MNS Python SDK）。其中，主要提供了OrderedQueueWrapper 类（oredered_queue.py文件），可以将普通的mns queue包装成有序queue。

OrderedQueueWrapper 提供两个方法：SendMessageInOrder()和ReceiveMessageInOrder()。send中对消息进行染色，receive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提供了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 中下列配置项
g_endpoint，g_accessKeyId，g_accessKeySecret，g_testQueueName

2.运行send_message_in_order.py

3.运行receive_message_in_order.py （可以不用等步骤2程序运行完成）发送程序会发送20条消息，接收程序会按顺序消费20条消息

```
orderedqueuewrapper receive message body is 0
orderedqueuewrapper receive message body is 1
orderedqueuewrapper receive message body is 2
orderedqueuewrapper receive message body is 3
orderedqueuewrapper receive message body is 4
orderedqueuewrapper receive message body is 5
orderedqueuewrapper receive message body is 6
orderedqueuewrapper receive message body is 7
orderedqueuewrapper receive message body is 8
orderedqueuewrapper receive message body is 9
orderedqueuewrapper receive message body is 10
orderedqueuewrapper receive message body is 11
orderedqueuewrapper receive message body is 12
orderedqueuewrapper receive message body is 13
orderedqueuewrapper receive message body is 14
orderedqueuewrapper receive message body is 15
orderedqueuewrapper receive message body is 16
orderedqueuewrapper receive message body is 17
orderedqueuewrapper receive message body is 18
orderedqueuewrapper receive message body is 19
Finish receive 20 messages
```

也可以运行oredered_queue.py (需配置endpoint 和AK)的测试case对比普通mns queue的区别：

运行命令：$python oredered_queue.py

非严格有序：（整体有序，部分相邻消息无序，同时侧面证明MNS 的单个queue同时有多个服务器在提供服务）

```
Test with common queue:
Send Message Succeed with commont queue. message body is :1
Send Message Succeed with commont queue. message body is :2
Send Message Succeed with commont queue. message body is :3
Send Message Succeed with commont queue. message body is :4
Send Message Succeed with commont queue. message body is :5
Send Message Succeed with commont queue. message body is :6
Send Message Succeed with commont queue. message body is :7
Send Message Succeed with commont queue. message body is :8
Send Message Succeed with commont queue. message body is :9
Send Message Succeed with commont queue. message body is :10
Receive Message Succeed with common queue! message body is 1
Receive Message Succeed with common queue! message body is 3
Receive Message Succeed with common queue! message body is 2
Receive Message Succeed with common queue! message body is 5
Receive Message Succeed with common queue! message body is 4
Receive Message Succeed with common queue! message body is 7
Receive Message Succeed with common queue! message body is 6
Receive Message Succeed with common queue! message body is 9
Receive Message Succeed with common queue! message body is 8
Receive Message Succeed with common queue! message body is 10
```

严格有序：

```
Test with OrderedQueueWrapper:
orderedqueuewrapper: send message body 1
orderedqueuewrapper: send message body 2
orderedqueuewrapper: send message body 3
orderedqueuewrapper: send message body 4
orderedqueuewrapper: send message body 5
orderedqueuewrapper: send message body 6
orderedqueuewrapper: send message body 7
orderedqueuewrapper: send message body 8
orderedqueuewrapper: send message body 9
orderedqueuewrapper: send message body 10
orderedqueuewrapper: receive message body is 1
orderedqueuewrapper: receive message body is 2
orderedqueuewrapper: receive message body is 3
orderedqueuewrapper: receive message body is 4
orderedqueuewrapper: receive message body is 5
orderedqueuewrapper: receive message body is 6
orderedqueuewrapper: receive message body is 7
orderedqueuewrapper: receive message body is 8
orderedqueuewrapper: receive message body is 9
orderedqueuewrapper: receive message body is 10
```

注意事项：

1.本帖主要目的是展示顺序消息的解决方案，本帖中的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，难免由瑕疵，欢迎回帖指正。

2.正常情况下，发送端和接收端的seqid应该和queue中的消息（染色）匹配，当出现删除queue重新创建等操作时，请注意磁盘文件中的seqid 是否和queue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。

3.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的乱序现象进行处理。

示例程序下载：

有序队列python示例代码

超大消息传输

问题背景

阿里云消息服务MNS的队列的消息大小最大限制是64K，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，就只能采用消息分片的方式。

那么如何能够基于MNS，又不做消息切片，传递大于64K的消息呢？解法是有的。

解决方案

1.生产者在往MNS 发送消息前，如果发现消息体大于64K，则先将消息体数据上传到OSS上；

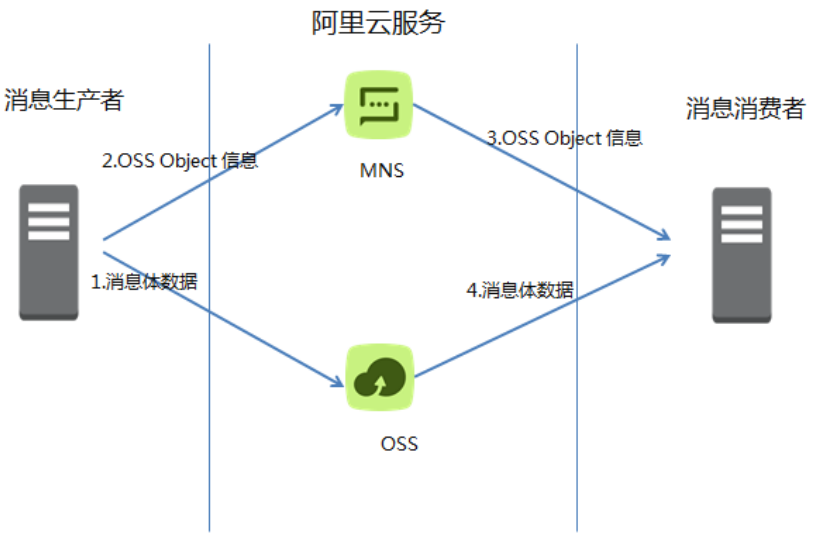
2.然后，生产者把数据对应的Objcet信息作为消息发送到MNS上；

3.消费者从MNS队列里读取消息，判断消息内容是否为OSS的Object信息；

4.如果消息内容是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；

5.对于大小小于64K的消息，仍然直接走MNS。

具体过程如下图所示：



程序实现

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

BigMessageSizeQueue提供的public方法如下：

```
public BigMessageSizeQueue(CloudQueue cq, OSSClient ossClient, String ossBucketName)
//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 Message putMessage(Message message) // 发送消息
public Message popMessage(int waitSeconds) // 接收消息
public void deleteMessage(String receiptHandle) //删除消息
public void setBigMessageSize(long bigMessageSize) //设置大消息的阈值（大于这个值的消息会走OSS），默认64K；
public void setNeedDeleteMessageObjectOnOSSFlag(boolean flag) // 设置是否需要删除OSS上的消息，默认yes；
```

具体使用示例代码请参考附件中Demo.java中的代码。

注意事项

1.大消息主要消息网络带宽，用该方案发送大size消息时，生产者和消费者的网络带宽可能会是瓶颈。

2.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。

事务消息

背景描述

有时候我们需要实现本地操作和消息发送的事务一致性功能。即：消息发送成功，则本地操作成功；反之，如果消息发送失败，本地操作失败（成功也需要rollback）。保证不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，消费端，我们也希望消息一定被成功处理一次，不会因为消息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

利用消息服务MNS的延迟消息功能来实现。

准备工作

创建两个队列：

1.事务消息队列

消息的有效期小于消息延迟时间。即如果生产者不主动修改（提交）消息可见时间，消息对消费者不可见；

2.操作日志队列

记录事务消息的操作记录信息。消息延迟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。

具体步骤

1.发送一条事务准备消息到事务消息队列；

2.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.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；

如下图：

异常分析：

生产者异常(例如：进程重启):

A.读取操作日志队列超时未确认日志

B.检查事务结果

C.如果检查得到事务已经成功，则提交消息（重复提交无副作用，同一句柄的消息只能成功提交一次）

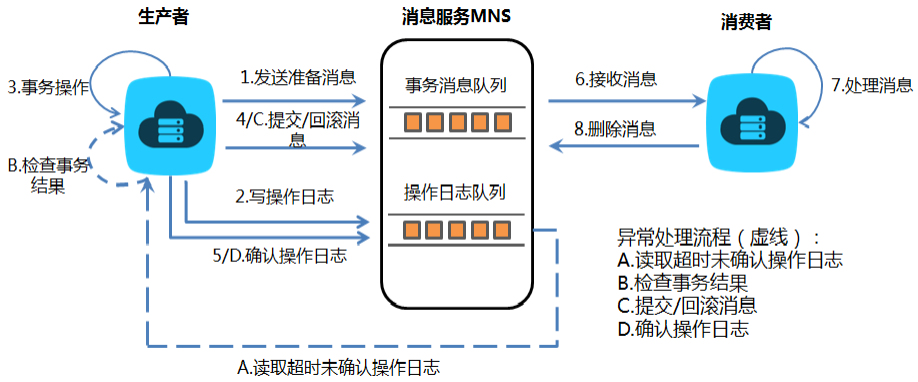
D.确认操作日志

消费者异常（例如：进程重启）：

消息服务提供至少保证消费一次的特性，只要步骤8不成功，消息在一段时间后可以继续可见，被当前消费者或者其他消费者处理。

消息服务不可达（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端，消息服务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，能保证只要生产者：操作成功，则消费者一定能够拿到消息并处理成功；或操作失败, 则消费者收不到消息的最终一致性。



代码实现：

MNS最新的Java SDK（1.1.5）中的TransactionQueue支持上述事务消息方案。使用者只需要在TransactionOperations和TransactionChecker 两个接口添加业务操作和检查逻辑，就可以方便的实现事务消息。

Demo 代码

```
public class TransactionMessageDemo{
```

```
public class MyTransactionChecker implements TransactionChecker
{
public boolean checkTransactionStatus(Message message)
{
boolean checkResult = false;
String messageHandler = message.getReceiptHandle();
try{
//TODO: check if the messageHandler related transaction is success.
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{
//TODO: do your local transaction according to the messageHandler and messageBody here.
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(); //this client need only initialize once

// create queue for transaction queue.
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);
```

```
// do transaction.
Message msg = new Message();
String messageBody = "TransactionMessageDemo";
msg.setMessageBody(messageBody);
transQueue.sendTransMessage(msg, transOperations);

// delete queue and close client if we won't use them.
transQueue.delete();

// close the client at the end.
client.close();
System.out.println("End TransactionMessageDemo");
}

}
```