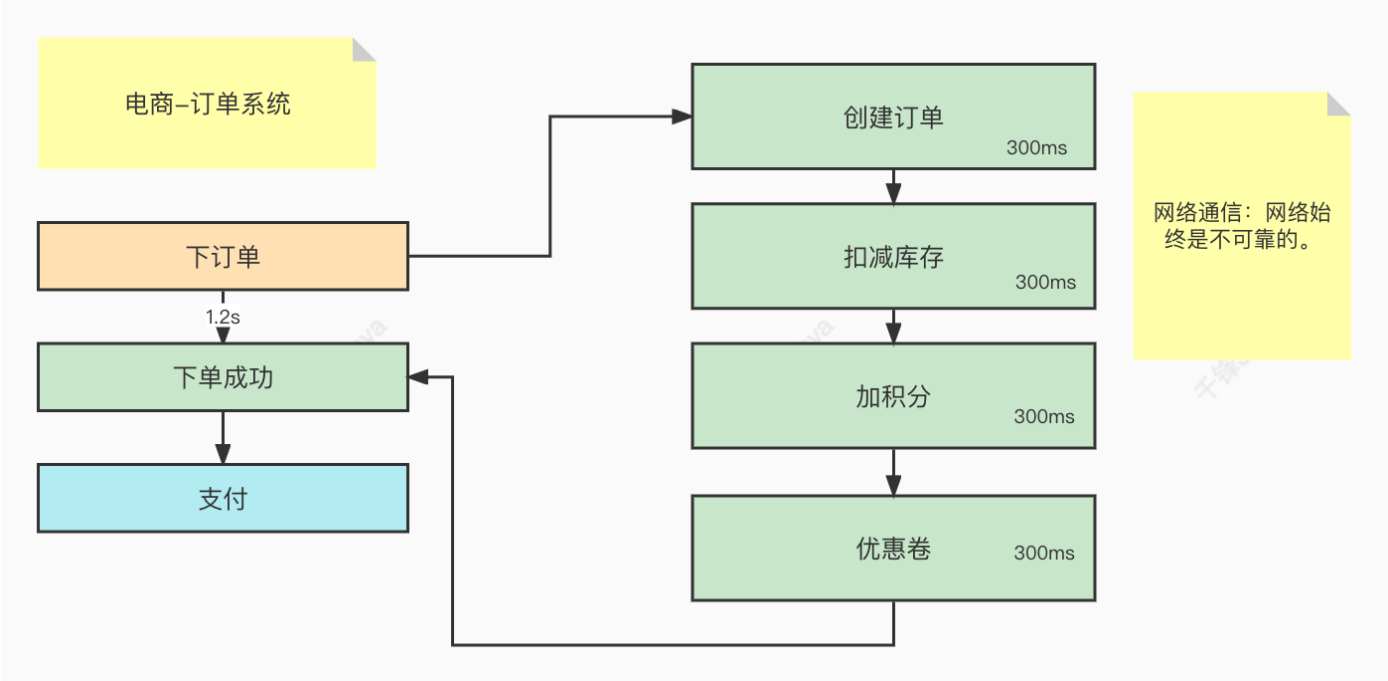
Author：千锋Java教研院

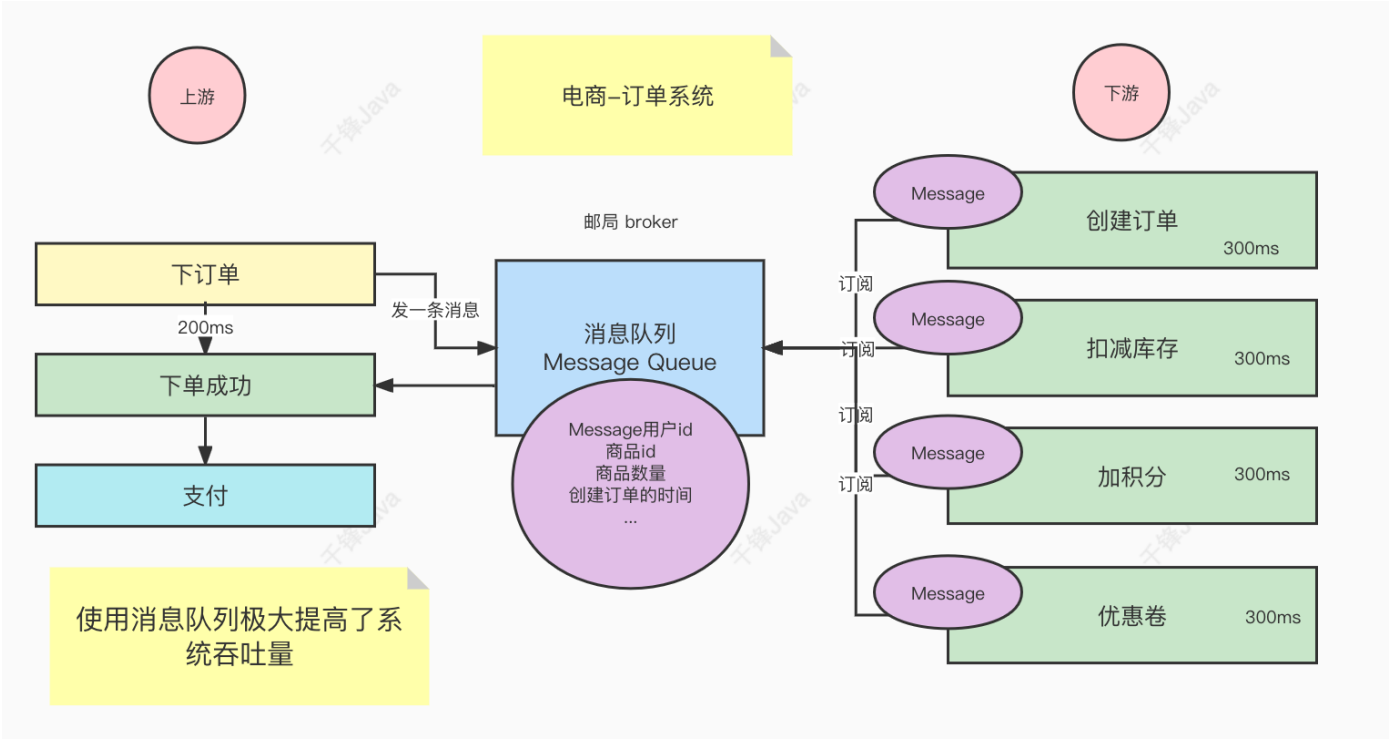
一、引言

Message Queue（消息 队列），从字面上理解：首先它是一个队列。FIFO先进先出的数据结构——队列。消息队列就是所谓的存放消息的队列。

消息队列解决的不是存放消息的队列的目的，解决的是通信问题。



比如以电商订单系统为例，如果各服务之间使用同步通信，不仅耗时较久，且过程中受到网络波动的影响，不能保证高成功率。因此，使用异步的通信方式对架构进行改造。



使用异步的通信方式对模块间的调用进行解耦，可以快速的提升系统的吞吐量。上游执行完消息的发送业务后立即获得结果，下游多个服务订阅到消息后各自消费。通过消息队列，屏蔽底层的通信协议，使得解藕和并行消费得以实现。

二、RocketMQ介绍

1.RocketMQ的由来

随着使用中队列和虚拟主题的增加，阿里巴巴团队使用的ActiveMQ IO 模块达到了瓶颈。为了尽力通过节流、断路器或降级来解决这个问题，但效果不佳。所以开始关注当时流行的消息传递解决方案Kafka。不幸的是，Kafka 无法满足要求，尤其是在低延迟和高可靠性方面。在这种情况下，决定发明一种新的消息传递引擎来处理更广泛的用例，从传统的发布/订阅场景到大容量实时零丢失交易系统。目前RocketMQ已经开源给Apache基金会。如今，已有 100 多家公司在其业务中使用开源版本的 RocketMQ。

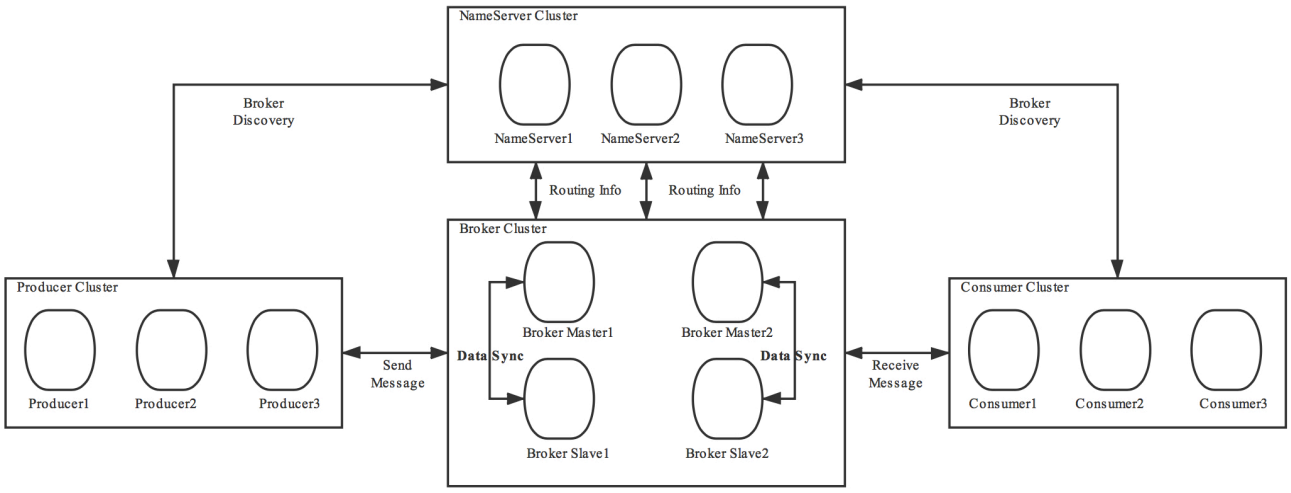
2.RocketMQ vs. ActiveMQ vs. Kafka

消息产品	客户端 SDK	协议和规范	订购信息	预定消息	批量消息	广播消息	消息过滤器	服务器触发的重新交付	消息存储	消息追溯	消息优先级	高可用性和故障转移	消息跟踪	配置	管理和运营工具
ActiveMQ	Java、.NET、C++ 等。	推送模型，支持 OpenWire、STOMP、AMQP、MQTT、JMS	Exclusive Consumer 或 Exclusive Queues 可以保证排序	支持的	不支持	支持的	支持的	不支持	使用 JDBC 和高性能日志支持非常快速的持久化，例如 levelDB、kahaDB	支持的	支持的	支持，取决于存储，如果使用 levelDB 则需要 ZooKeeper 服务器	不支持	默认配置为低级，用户需优化配置参数	支持的
Kafka	Java、Scala 等。	拉取模型，支持TCP	确保分区内消息的排序	不支持	支持，带有异步生产者	不支持	支持，可以使用Kafka Streams过滤消息	不支持	高性能文件存储	支持的偏移量指示	不支持	支持，需要 ZooKeeper 服务器	不支持	Kafka 使用键值对格式进行配置。这些值可以从文件或以编程方式提供。	支持，使用终端命令公开核心指标
RocketMQ	Java、C++、Go	拉取模型，支持 TCP、JMS、OpenMessaging	确保消息的严格排序，并且可以优雅地横向扩展	支持的	支持，具有同步模式以避免消息丢	支持的	支持的基于 SQL92 的属性过滤器表达式	支持的	高性能和低延迟的文件存储	支持的时间戳和偏移量两种表示	不支持	支持的主从模型，无需其他套件	支持的	开箱即用，用户只需注意一些配置	支持的、丰富的 Web 和终端命令以公开核心指标

失

三、RocketMQ的基本概念

1 技术架构

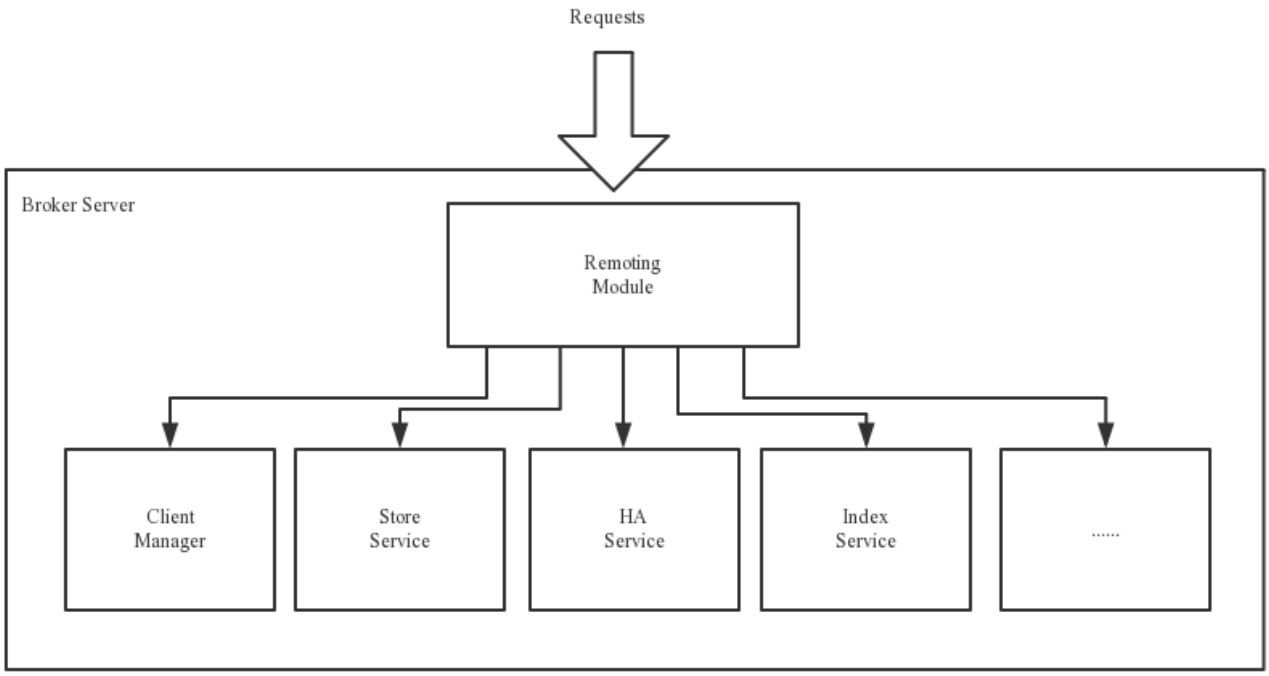


RocketMQ架构上主要分为四部分，如上图所示:

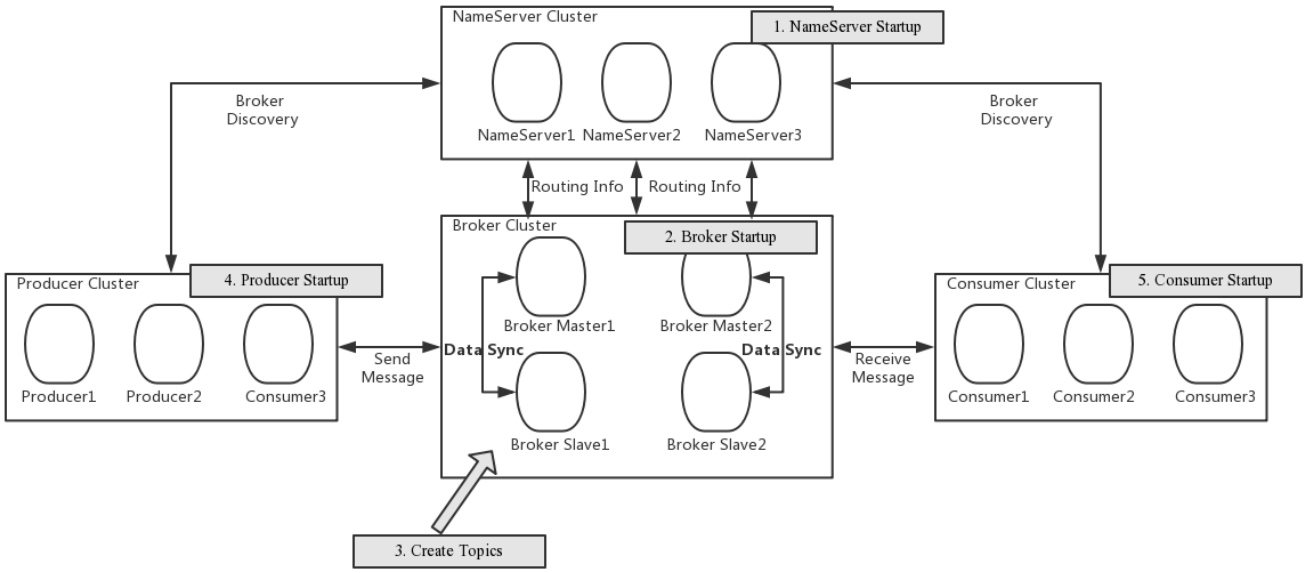
- Producer：消息发布的角色，支持分布式集群方式部署。Producer通过MQ的负载均衡模块选择相应的Broker集群队列进行消息投递，投递的过程支持快速失败并且低延迟。
- Consumer：消息消费的角色，支持分布式集群方式部署。支持以push推，pull拉两种模式对消息进行消费。同时也支持集群方式和广播方式的消费，它提供实时消息订阅机制，可以满足大多数用户的需求。
- NameServer：NameServer是一个非常简单的Topic路由注册中心，其角色类似Dubbo中的zookeeper，支持Broker的动态注册与发现。主要包括两个功能：Broker管理，NameServer接受Broker集群的注册信息并且保存下来作为路由信息的基本数据。然后提供心跳检测机制，检查Broker是否还存活；路由信息管理，每个NameServer将保存关于Broker集群的整个路由信息和用于客户端查询的队列信息。然后Producer和Conumser通过NameServer就可以知道整个Broker集群的路由信息，从而进行消息的投递和消费。NameServer通常也是集群的方

式部署，各实例间相互不进行信息通讯。Broker是向每一台NameServer注册自己的路由信息，所以每一个NameServer实例上面都保存一份完整的路由信息。当某个NameServer因某种原因下线了，Broker仍然可以向其它NameServer同步其路由信息，Producer,Consumer仍然可以动态感知Broker的路由的信息。

- BrokerServer：Broker主要负责消息的存储、投递和查询以及服务高可用保证，为了实现这些功能，Broker包含了以下几个重要子模块。
  - Remoting Module：整个Broker的实体，负责处理来自clients端的请求。
  - Client Manager：负责管理客户端(Producer/Consumer)和维护Consumer的Topic订阅信息
  - Store Service：提供方便简单的API接口处理消息存储到物理硬盘和查询功能。
  - HA Service：高可用服务，提供Master Broker 和 Slave Broker之间的数据同步功能。
  - Index Service：根据特定的Message key对投递到Broker的消息进行索引服务，以提供消息的快速查询。



2 部署架构



RocketMQ 网络部署特点

- NameServer是一个几乎无状态节点，可集群部署，节点之间无任何信息同步。
- Broker部署相对复杂，Broker分为Master与Slave，一个Master可以对应多个Slave，但是一个Slave只能对应一个Master，Master与Slave 的对应关系通过指定相同的BrokerName，不同的BrokerId 来定义，BrokerId为0表示Master，非0表示Slave。Master也可以部署多个。每个Broker与NameServer集群中的所有节点建立长连接，定时注册Topic信息到所有NameServer。 注意：当前RocketMQ版本在部署架构上支持一Master多Slave，但只有BrokerId=1的从服务器才会参与消息的读负载。
- Producer与NameServer集群中的其中一个节点（随机选择）建立长连接，定期从NameServer获取Topic路由信息，并向提供Topic 服务的Master建立长连接，且定时向Master发送心跳。Producer完全无状态，可集群部署。
- Consumer与NameServer集群中的其中一个节点（随机选择）建立长连接，定期从NameServer获取Topic路由信息，并向提供Topic服务的Master、Slave建立长连接，且定时向Master、Slave发送心跳。Consumer既可以从Master订阅消息，也可以从Slave订阅消息，消费者在向Master拉取消息时，Master服务器会根据拉取偏移量与最大偏移量的距离（判断是否读老消息，产生读I/O），以及从服务器是否可读等因素建议下一次是从Master还是Slave拉取。

结合部署架构图，描述集群工作流程：

- 启动NameServer，NameServer起来后监听端口，等待Broker、Producer、

Consumer连上来，相当于一个路由控制中心。
- Broker启动，跟所有的NameServer保持长连接，定时发送心跳包。心跳包中包含当前Broker信息(IP+端口等)以及存储所有Topic信息。注册成功后，NameServer集群中就有Topic跟Broker的映射关系。
- 收发消息前，先创建Topic，创建Topic时需要指定该Topic要存储在哪些Broker上，也可以在发送消息时自动创建Topic。
- Producer发送消息，启动时先跟NameServer集群中的其中一台建立长连接，并从NameServer中获取当前发送的Topic存在哪些Broker上，轮询从队列列表中选择一个队列，然后与队列所在的Broker建立长连接从而向Broker发消息。
- Consumer跟Producer类似，跟其中一台NameServer建立长连接，获取当前订阅Topic存在哪些Broker上，然后直接跟Broker建立连接通道，开始消费消息。

三、快速开始

1.下载RocketMQ

本教程使用的是RocketMQ4.7.1版本，建议使用该版本进行之后的demo训练。

- 运行版： **https://www.apache.org/dyn/closer.cgi?path=rocketmq/4.7.1/rocketmq-all-4.7.1-bin-release.zip**
- 源码： **https://www.apache.org/dyn/closer.cgi?path=rocketmq/4.7.1/rocketmq-all-4.7.1-source-release.zip**

2.安装RocketMQ

- 准备一台装有Linux系统的虚拟机。本教程使用的是Ubuntu16.04版本。
- 安装jdk，上传jdk-8u191安装包并解压缩在`/usr/local/java`目录下。
- 安装rocketmq，上传rocketmq安装包并使用unzip命令解压缩在`/usr/local/rocketmq`目录下。
- 配置jdk和rocketmq的环境变量

```
export JAVA_HOME=/usr/local/java/jdk1.8.0_191
export JRE_HOME=/usr/local/java/jdk1.8.0_191/jre
export ROCKETMQ_HOME=/usr/local/rocketmq/rocketmq-all-4.7.1-bin-release
export CLASSPATH=$CLASSPATH:$JAVA_HOME/lib:$JAVA_HOME/jre/lib
export PATH=$JAVA_HOME/bin:$JAVA_HOME/jre/bin:$ROCKETMQ_HOME/bin:$PATH:$HOME/bin
```

> 注意，RocketMQ的环境变量用来加载`ROCKETMQ_HOME/conf`下的配置文件，如果不配置则无法启动NameServer和Broker。

完成后执行命令，让环境变量生效

```
source /etc/profile
```

- 修改bin/runserver.sh文件，由于RocketMQ默认设置的JVM内存为4G，但虚拟机一般没有这么4G内存，因此调整为512mb。

```
JAVA_OPT="${JAVA_OPT} -server -Xms4g -Xmx4g -Xmn2g -XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=320m"
```

在runserver.sh文件中找到上面这段内容，改为下面的参数。

```
JAVA_OPT="${JAVA_OPT} -server -Xms512m -Xmx512m -Xmn256m -XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=320m"
```

3.启动NameServer

在上一章中介绍了RocketMQ的架构，启动RocketMQ服务需要先启动NameServer。

在bin目录内使用静默方式启动。

```
nohup ./mqnamesrv &
```

查看bin/nohup.out显示如下内容表示启动成功:

```
root@ubuntu:/usr/local/rocketmq/rocketmq-all-4.7.1-bin-release/bin# cat nohup.out
Java HotSpot(TM) 64-Bit Server VM warning: Using the DefNew young collector with the CMS collector is deprecated and will likely be removed in a future release
Java HotSpot(TM) 64-Bit Server VM warning: UseCMSCompactAtFullCollection is deprecated and will likely be removed in a future release.
The Name Server boot success. serializeType=JSON
```

4.启动Broker

- 修改broker的JVM参数配置，将默认8G内存修改为512m。修改`bin/runbroker.sh`文件

```
JAVA_OPT="${JAVA_OPT} -server -Xms512m -Xmx512m -Xmn256m"
```

- 在`conf/broker.conf`文件中加入如下配置，开启自动创建Topic功能

```
autoCreateTopicEnable=true
```

- 以静默方式启动broker

```
nohup ./mqbroker -n localhost:9876 &
```

- 查看`bin/nohup.out`日志，显示如下内容表示启动成功

```
The broker[ubuntu, 172.17.0.1:10911] boot success. serializeType=JSON
```

5.使用发送和接收消息验证MQ

- 配置nameserver的环境变量

在发送/接收消息之前，需要告诉客户端nameserver的位置。配置环境变量`NAMESRV_ADDR`：

```
export NAMESRV_ADDR=localhost:9876
```

- 使用bin/tools.sh工具验证消息的发送，默认会发1000条消息

```
./tools.sh org.apache.rocketmq.example.quickstart.Producer
```

发送的消息日志：

```
...
SendResult [sendStatus=SEND_OK,
msgId=FD154BA55A2B1008020C29FFFED6A0855CFC12A3A380885CB70A0235,
offsetMsgId=AC11000100002A9F000000000001F491,
messageQueue=MessageQueue [topic=TopicTest, brokerName=ubuntu,
queueId=0], queueOffset=141]
```

- 使用bin/tools.sh工具验证消息的接收

```
./tools.sh org.apache.rocketmq.example.quickstart.Consumer
```

看到接收到的消息：

```
...
ConsumeMessageThread_12 Receive New Messages: [MessageExt
[brokerName=ubuntu, queueId=1, storeSize=227, queueOffset=245,
sysFlag=0, bornTimestamp=1658892578234,
bornHost=/172.16.253.100:48524, storeTimestamp=1658892578235,
storeHost=/172.17.0.1:10911,
msgId=AC11000100002A9F0000000000036654, commitLogOffset=222804,
bodyCRC=683694034, reconsumeTimes=0, preparedTransactionOffset=0,
toString()=Message{topic='TopicTest', flag=0, properties=
{MIN_OFFSET=0, MAX_OFFSET=250, CONSUME_START_TIME=1658892813497,
UNIQ_KEY=FD154BA55A2B1008020C29FFFED6A0855CFC12A3A380885CB9BA03D6,
CLUSTER=DefaultCluster, WAIT=true, TAGS=TagA}, body=[72, 101, 108,
108, 111, 32, 82, 111, 99, 107, 101, 116, 77, 81, 32, 57, 56, 50],
transactionId='null'}]]
```

6.关闭服务器

- 关闭broker

```
./mqshutdown broker
```

- 关闭nameserver

```
./mqshutdown namesrv
```

四、搭建RocketMQ集群

1.RocketMQ集群模式

为了追求更好的性能，RocketMQ的最佳实践方式都是在集群模式下完成。RocketMQ官方提供了三种集群搭建方式。

- 2主2从异步通信方式

使用异步方式进行主从之间的数据复制，吞吐量大，但可能会丢消息。

使用`conf/2m-2s-async`文件夹内的配置文件做集群配置。

- 2主2从同步通信方式

使用同步方式进行主从之间的数据复制，保证消息安全投递，不会丢失，但影响吞吐量

使用`conf/2m-2s-sync`文件夹内的配置文件做集群配置。

- 2主无从方式

会存在单点故障，且读的性能没有前两种方式好。

使用`conf/2m-noslave`文件夹内的配置文件做集群配置。

- Dledger高可用集群

上述三种官方提供的集群没办法实现高可用，即在master节点挂掉后，slave节点没办法自动被选举为新的master，而需要人工实现。

RocketMQ在4.5版本之后引入了第三方的Dleger高可用集群。

2.搭建主从异步集群

1）准备三台Linux服务器

三台Linux服务器中nameserver和broker之间的关系如下：

服务器	服务器IP	NameServer	broker节点部署
服务器1	172.16.253.103	172.16.253.103:9876	
服务器2	172.16.253.101	172.16.253.101:9876	broker-a（master）,broker-b-s（slave）
服务器3	172.16.253.102	172.16.253.102:9876	broker-b（master）,broker-a-s（slave）

三台服务器都需要安装jdk和rocketmq，安装步骤参考上一章节。

2）启动三台nameserver

nameserver是一个轻量级的注册中心，broker把自己的信息注册到nameserver上。而且，nameserver是无状态的，直接启动即可。三台nameserver之间不需要通信，而是被请求方来关联三台nameserver的地址。

修改三台服务器的的runserver.sh文件

修改JVM内存默认的4g为512m。

```
JAVA_OPT="${JAVA_OPT} -server -Xms512m -Xmx512m -Xmn256m -
XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=320m"
```

在每台服务器的bin目录下执行如下命令：

- 服务器1:

```
nohup ./mqnamesrv -n 172.16.253.103:9876 &
```

- 服务器2:

```
nohup ./mqnamesrv -n 172.16.253.101:9876 &
```

- 服务器3:

```
nohup ./mqnamesrv -n 172.16.253.102:9876 &
```

3）配置broker

broker-a,broker-b-s这两台broker是配置在服务器2上，broker-b,broker-a-s这两台broker是配置在服务器3上。这两对主从节点在不同的服务器上，服务器1上没有部署broker。

需要修改每台broker的配置文件。注意，同一台服务器上的两个broker保存路径不能一样。

- broker-a的master节点

在服务器2上，进入到conf/2m-2s-async文件夹内，修改broker-a.properties文件。

```
# 所属集群名称
brokerClusterName=DefaultCluster
# broker名字
brokerName=broker-a
# broker所在服务器的ip
brokerIP1=172.16.253.101
# broker的id，0表示master，>0表示slave
brokerId=0
# 删除文件时间点，默认在凌晨4点
deleteWhen=04
# 文件保留时间为48小时
fileReservedTime=48
# broker的角色为master
brokerRole=ASYNC_MASTER
# 使用异步刷盘的方式
flushDiskType=ASYNC_FLUSH
# 名称服务器的地址列表
namesrvAddr=172.16.253.103:9876;172.16.253.101:9876;172.16.253.102:9876
# 在发送消息自动创建不存在的topic时，默认创建的队列数为4个
defaultTopicQueueNums=4
# 是否允许 Broker 自动创建Topic，建议线下开启，线上关闭
autoCreateTopicEnable=true
# 是否允许 Broker 自动创建订阅组，建议线下开启，线上关闭
autoCreateSubscriptionGroup=true
```

```
# broker对外服务的监听端口
listenPort=10911
# abort文件存储路径
abortFile=/usr/local/rocketmq/store/abort
# 消息存储路径
storePathRootDir=/usr/local/rocketmq/store
# commitLog存储路径
storePathCommitLog=/usr/local/rocketmq/store/commitlog
# 消费队列存储路径
storePathConsumeQueue=/usr/local/rocketmq/store/consumequeue
# 消息索引存储路径
storePathIndex=/usr/local/rocketmq/store/index
# checkpoint文件存储路径
storeCheckpoint=/usr/local/rocketmq/store/checkpoint
# 限制的消息大小
maxMessageSize=65536
# commitLog每个文件的大小默认1G
mapedFileSizeCommitLog=1073741824
# ConsumeQueue每个文件默认存30W条，根据业务情况调整
mapedFileSizeConsumeQueue=300000
```

- broker-a的slave节点

在服务器3上，进入到conf/2m-2s-async文件夹内，修改broker-a-s.properties文件。

```
brokerClusterName=DefaultCluster
brokerName=broker-a
brokerIP1=172.16.253.102
brokerId=1
deleteWhen=04
fileReservedTime=48
brokerRole=SLAVE
flushDiskType=ASYNC_FLUSH
namesrvAddr=172.16.253.103:9876;172.16.253.101:9876;172.16.253.102:9876
defaultTopicQueueNums=4
autoCreateTopicEnable=true
autoCreateSubscriptionGroup=true
```

```
listenPort=11011
abortFile=/usr/local/rocketmq/store-slave/abort
storePathRootDir=/usr/local/rocketmq/store-slave
storePathCommitLog=/usr/local/rocketmq/store-slave/commitlog
storePathConsumeQueue=/usr/local/rocketmq/store-slave/consumequeue
storePathIndex=/usr/local/rocketmq/store-slave/index
storeCheckpoint=/usr/local/rocketmq/store-slave/checkpoint
maxMessageSize=65536

```

- broker-b的master节点

在服务器3上，进入到conf/2m-2s-async文件夹内，修改broker-b.properties文件。

```
brokerClusterName=DefaultCluster
brokerName=broker-b
brokerIP1=172.16.253.102
brokerId=0
deleteWhen=04
fileReservedTime=48
brokerRole=ASYNC_MASTER
flushDiskType=ASYNC_FLUSH
namesrvAddr=172.16.253.103:9876;172.16.253.101:9876;172.16.253.102:9876
defaultTopicQueueNums=4
autoCreateTopicEnable=true
autoCreateSubscriptionGroup=true
listenPort=10911
abortFile=/usr/local/rocketmq/store/abort
storePathRootDir=/usr/local/rocketmq/store
storePathCommitLog=/usr/local/rocketmq/store/commitlog
storePathConsumeQueue=/usr/local/rocketmq/store/consumequeue
storePathIndex=/usr/local/rocketmq/store/index
storeCheckpoint=/usr/local/rocketmq/store/checkpoint
maxMessageSize=65536
```

- broker-b的slave节点

在服务器2上，进入到conf/2m-2s-async文件夹内，修改broker-b-s.properties文件。

```
brokerClusterName=DefaultCluster
brokerName=broker-b
brokerIP1=172.16.253.101
brokerId=1
deleteWhen=04
fileReservedTime=48
brokerRole=SLAVE
flushDiskType=ASYNC_FLUSH
namesrvAddr=172.16.253.103:9876;172.16.253.101:9876;172.16.253.102:9876
defaultTopicQueueNums=4
autoCreateTopicEnable=true
autoCreateSubscriptionGroup=true
listenPort=11011
abortFile=/usr/local/rocketmq/store-slave/abort
storePathRootDir=/usr/local/rocketmq/store-slave
storePathCommitLog=/usr/local/rocketmq/store-slave/commitlog
storePathConsumeQueue=/usr/local/rocketmq/store-slave/consumequeue
storePathIndex=/usr/local/rocketmq/store-slave/index
storeCheckpoint=/usr/local/rocketmq/store-slave/checkpoint
maxMessageSize=65536
```

- 修改服务器2和服务器3的runbroker.sh文件

修改JVM内存默认的8g为512m。

```
JAVA_OPT="${JAVA_OPT} -server -Xms512m -Xmx512m -Xmn256m"
```

4）启动broker

- 在服务器2中启动broker-a（master）和broker-b-s（slave）

```
nohup ./mqbroker -c ../conf/2m-2s-async/broker-a.properties &
nohup ./mqbroker -c ../conf/2m-2s-async/broker-b-s.properties &
```

- 在服务器3中启动broker-b（master）,broker-a-s（slave）

```
nohup ./mqbroker -c ../conf/2m-2s-async/broker-b.properties &
nohup ./mqbroker -c ../conf/2m-2s-async/broker-a-s.properties &
```

3.验证集群

使用RocketMQ提供的tools工具验证集群是否正常工作。

- 在服务器2上配置环境变量

  用于被tools中的生产者和消费者程序读取该变量。

```
export NAMESRV_ADDR='172.16.253.103:9876;172.16.253.101:9876;172.16.253.102:9876'
```

- 启动生产者

```
./tools.sh org.apache.rocketmq.example.quickstart.Producer
```

- 启动消费者

```
./tools.sh org.apache.rocketmq.example.quickstart.Consumer
```

4.mqadmin管理工具

RocketMQ提供了命令工具用于管理topic、broker、集群、消息等。比如可以使用mqadmin创建topic：

```
./mqadmin updateTopic -n 172.16.253.101:9876 -c DefaultCluster -t myTopic1
```

下面提供了mqadmin工具的各种命令。

- 创建topic：updateTopic

参数	是否必填	说明
-b	如果-c为空，则必填	broker 地址，表示topic 建在该broker
-c	如果-b为空，则必填	cluster 名称，表示topic 建在该集群（集群可通过clusterList 查询）
-h	否	打印帮助
-n	是	nameserve 服务地址列表，格式ip:port;ip:port;...
-p	否	指定新topic 的权限限制(W|R|WR)
-r	否	可读队列数（默认为8）
-w	否	可写队列数（默认为8）
-t	是	opic 名称（名称只能使用字符 ^[a-zA-Z0-9_-]+$ ）

- 删除Topic：deleteTopic

参数	是否必填	说明
-c	是	cluster 名称，表示删除某集群下的某个topic （集群可通过clusterList 查询）
-h	否	打印帮助
-n	是	nameserve 服务地址列表，格式ip:port;ip:port;...
-t	是	topic 名称（名称只能使用字符 ^[a-zA-Z0-9_-]+$ ）

- 创建（修订）订阅组：updateSubGroup

参数	是否必填	说明
-b	如果 –c 为空，则必填	broker 地址，表示订阅组建在该broker
-c	如果 –b 为空，则必填	cluster名称，表示topic 建在该集群（集群可通过clusterList查询）
-d	否	是否容许广播方式消费
-g	是	订阅组名
-i	否	从哪个broker 开始消费
-m	否	是否容许从队列的最小位置开始消费，默认会设置为false
-q	否	消费失败的消息放到一个重试队列，每个订阅组配置几个重试队列
-r	否	重试消费最大次数，超过则投递到死信队列，不再投递，并报警
-s	否	消费功能是否开启
-w	否	发现消息堆积后，将Consumer 的消费请求重定向到另外一台Slave 机器
-h	否	打印帮助
-n	是	nameserve 服务地址列表，格式ip:port;ip:port;...

- 删除订阅组配置：deleteSubGroup

参数	是否必填	说明
-b	如果–c 为空，则必填	broker 地址，表示订阅组建在该broker
-c	如果–b 为空，则必填	cluster 名称，表示topic建在该集群（集群可通过clusterList查询）
-g	是	订阅组名
-h	否	打印帮助
-n	是	nameserve 服务地址列表，格式ip:port;ip:port;...

- 更新Broker 配置文件：updateBrokerConfig

参数	是否必填	说明
-b	如果–c为空，则必填	broker 地址，表示订阅组建在该broker
-c	如果–b 为空，则必填	cluster名称，表示topic 建在该集群（集群可通过clusterList查询）
-k	是	key 值
-v	否	value 值
-h	否	打印帮助
-n	是	nameserve 服务地址列表，格式ip:port;ip:port;...

- 查看Topic 列表信息：topicList

参数	是否必填	说明
-h	否	打印帮助
-n	是	nameserve 服务地址列表，格式ip:port;ip:port;...

- 查看Topic 路由信息：topicRoute

参数	是否必填	说明
-t	是	topic 名称
-h	否	打印帮助
-n	是	nameserve 服务地址列表，格式ip:port;ip:port;...

- 查看Topic 统计信息：topicStats

参数	是否必填	说明
-t	是	topic 名称
-h	否	打印帮助
-n	是	nameserve 服务地址列表，格式ip:port;ip:port;...

- 查看Broker 统计信息：brokerStats

参数	是否必填	说明
-b	是	broker 地址
-h	否	打印帮助
-n	是	nameserve 服务地址列表，格式ip:port;ip:port;...

- 根据消息ID 查询消息：queryMsgById

参数	是否必填	说明
-i	是	消息id
-h	否	打印帮助
-n	是	nameserve 服务地址列表，格式ip:port;ip:port;...

- 根据消息Key 查询消息：queryMsgByKey

参数	是否必填	说明
-f	否	被查询消息的截止时间
-k	是	msgKey
-t	是	topic 名称
-h	否	打印帮助
-n	是	nameserve 服务地址列表，格式ip:port;ip:port;...

- 根据Offset 查询消息：queryMsgByOffset

参数	是否必填	说明
-b	是	Broker 名称，表示订阅组建在该broker（这里需要注意填写的是broker 的名称，不是broker 的地址，broker名称可以在clusterList 查到
-i	是	query 队列id
-o	是	offset 值
-t	是	topic 名称
-h	否	打印帮助
-n	是	nameserve 服务地址列表，格式ip:port;ip:port;...

- 查询Producer 的网络连接：producerConnection

参数	是否必填	说明
-g	是	生产者所属组名
-t	是	topic 名称
-h	否	打印帮助
-n	是	nameserve 服务地址列表，格式ip:port;ip:port;...

- 查询Consumer 的网络连接：consumerConnection

参数	是否必填	说明
-g	是	消费者所属组名
-h	否	打印帮助
-n	是	nameserve 服务地址列表，格式ip:port;ip:port;...

- 查看订阅组消费状态：consumerProgress

参数	是否必填	说明
-g	是	消费者所属组名
-h	否	打印帮助
-n	是	nameserve 服务地址列表，格式ip:port;ip:port;...

- 查看集群消息：clusterList

参数	是否必填	说明
-m	否	打印更多信息
-h	否	打印帮助
-n	是	nameserve 服务地址列表，格式ip:port;ip:port;...

- 添加（更新）KV 配置信息：updateKvConfig

参数	是否必填	说明
-k	是	key 值
-v	是	value 值
-s	是	Namespace 值
-h	否	打印帮助
-n	是	nameserve 服务地址列表，格式ip:port;ip:port;...

- 删除KV 配置信息：deleteKvConfig

参数	是否必填	说明
-k	是	key 值
-s	是	Namespace 值
-h	否	打印帮助
-n	是	nameserve 服务地址列表，格式ip:port;ip:port;...

- 添加（更新）Project group 配置信息：updateProjectGroup

参数	是否必填	说明
-p	是	project group 名
-i	否	服务器ip
-h	否	打印帮助
-n	是	nameserve 服务地址列表，格式ip:port;ip:port;...

- 删除Project group 配置信息：deleteProjectGroup

参数	是否必填	说明
-p	是	project group 名
-i	否	服务器ip
-h	否	打印帮助
-n	是	nameserve 服务地址列表，格式ip:port;ip:port;...

- 取得Project group 配置信息：getProjectGroup

参数	是否必填	说明
-p	是	project group 名
-i	否	服务器ip
-h	否	打印帮助
-n	是	nameserve 服务地址列表，格式ip:port;ip:port;...

- 设置消费进度：resetOffsetByTime

根据时间来设置消费进度，设置之前要关闭这个订阅组的所有consumer，设置完再启动，方可生效

参数	是否必填	说明
-f	否	通过时间戳强制回滚（true|false），默认为true
-s	是	时间戳
-g	是	消费者所属组名
-t	是	topic 名称
-h	否	打印帮助
-n	是	nameserve 服务地址列表，格式ip:port;ip:port;...

- 清除特定Broker权限：wipeWritePerm

参数	是否必填	说明
-b	是	broker 地址
-h	否	打印帮助
-n	是	nameserve 服务地址列表，格式ip:port;ip:port;...

- 获取Consumer消费进度：getConsumerStatus

该命令只打印当前与cluster 连接的consumer 的消费进度

参数	是否必填	说明
-g	是	消费者所属组名
-t	是	查询主题
-i	否	Consumer 客户端ip
-h	否	打印帮助
-n	是	nameserve 服务地址列表，格式ip:port;ip:port;...

5.安装可视化管理控制平台

RocketMQ没有提供可视化管理控制平台，可以使用第三方管理控制平台：**https://github.com/apache/rocketmq-externals/tree/rocketmq-console-1.0.0/rocketmq-console**

- 下载管理控制平台
- 解压缩在linux服务器上

可以安装在服务器1上

- 给服务器安装maven环境

```
1  apt install maven
```

- 修改`rocketmq-externals/rocketmq-externals-master/rocketmq-console/src/main/resources/application.properties`配置文件中的nameserver地址

```
rocketmq.config.namesrvAddr=172.16.253.103:9876;172.16.253.101:9876;172.16.253.102:9876
```

- 回到`rocketmq-externals/rocketmq-externals-master/rocketmq-console`路径下执行maven命令进行打包

```
mvn clean package -Dmaven.test.skip=true
```

- 运行jar包。进入到`rocketmq-externals/rocketmq-externals-master/rocketmq-console/target`目录内执行如下命令：

```
nohup java -jar rocketmq-console-ng-1.0.1.jar
```

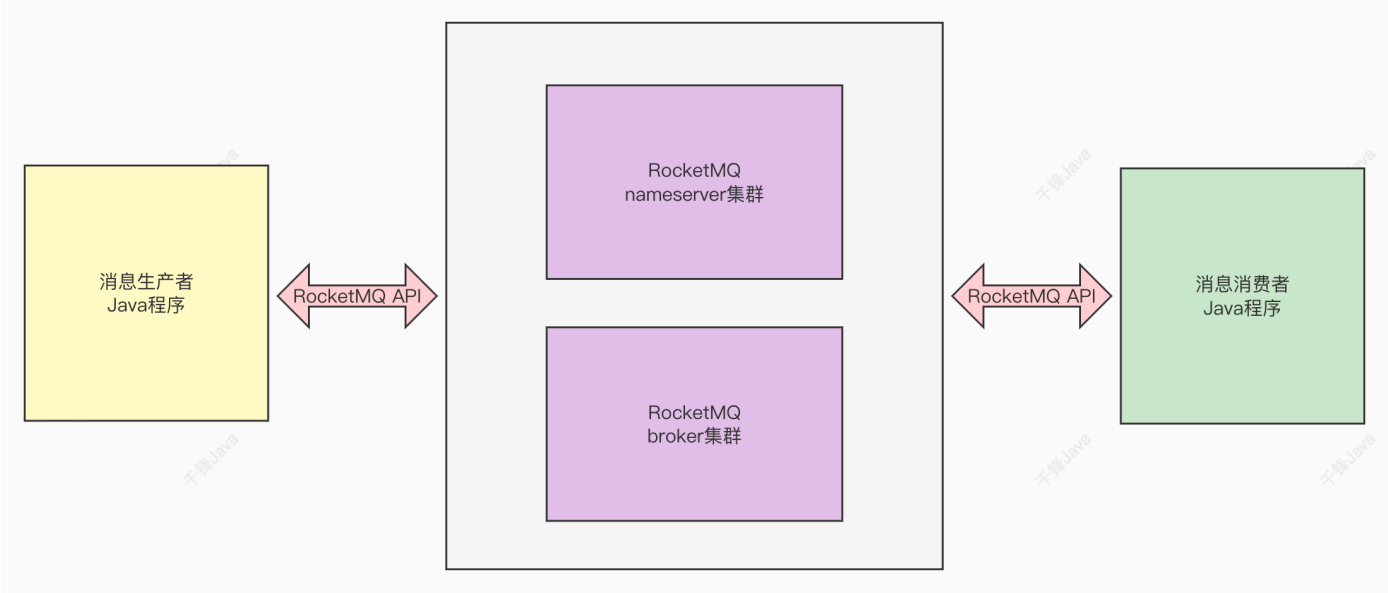
- 访问所在服务器的8080端口，查看集群界面，可以看到之前部署的集群

五、消息示例

在掌握RocketMQ的基本实现逻辑之后，接下来通过Java程序来学习RocketMQ的多种消息示例，它们拥有各自的应用场景。

1.构建Java基础环境

在maven项目中构建出RocketMQ消息示例的基础环境，即创建生产者程序和消费者程序。通过生产者和消费者了解RocketMQ操作消息的原生API。



- 引入依赖

```
    <dependencies>
        <dependency>
            <groupId>org.apache.rocketmq</groupId>
            <artifactId>rocketmq-client</artifactId>
            <version>4.7.1</version>
        </dependency>
    </dependencies>
```

- 编写生产者程序

```
package com.qf.producer.simple;

import org.apache.rocketmq.client.exception.MQBrokerException;
import org.apache.rocketmq.client.exception.MQClientException;
import org.apache.rocketmq.client.producer.DefaultMQProducer;
import org.apache.rocketmq.client.producer.SendResult;
import org.apache.rocketmq.common.message.Message;
import org.apache.rocketmq.remoting.common.RemotingHelper;
import org.apache.rocketmq.remoting.exception.RemotingException;

import java.io.UnsupportedEncodingException;

/**
 * @author Thor
```

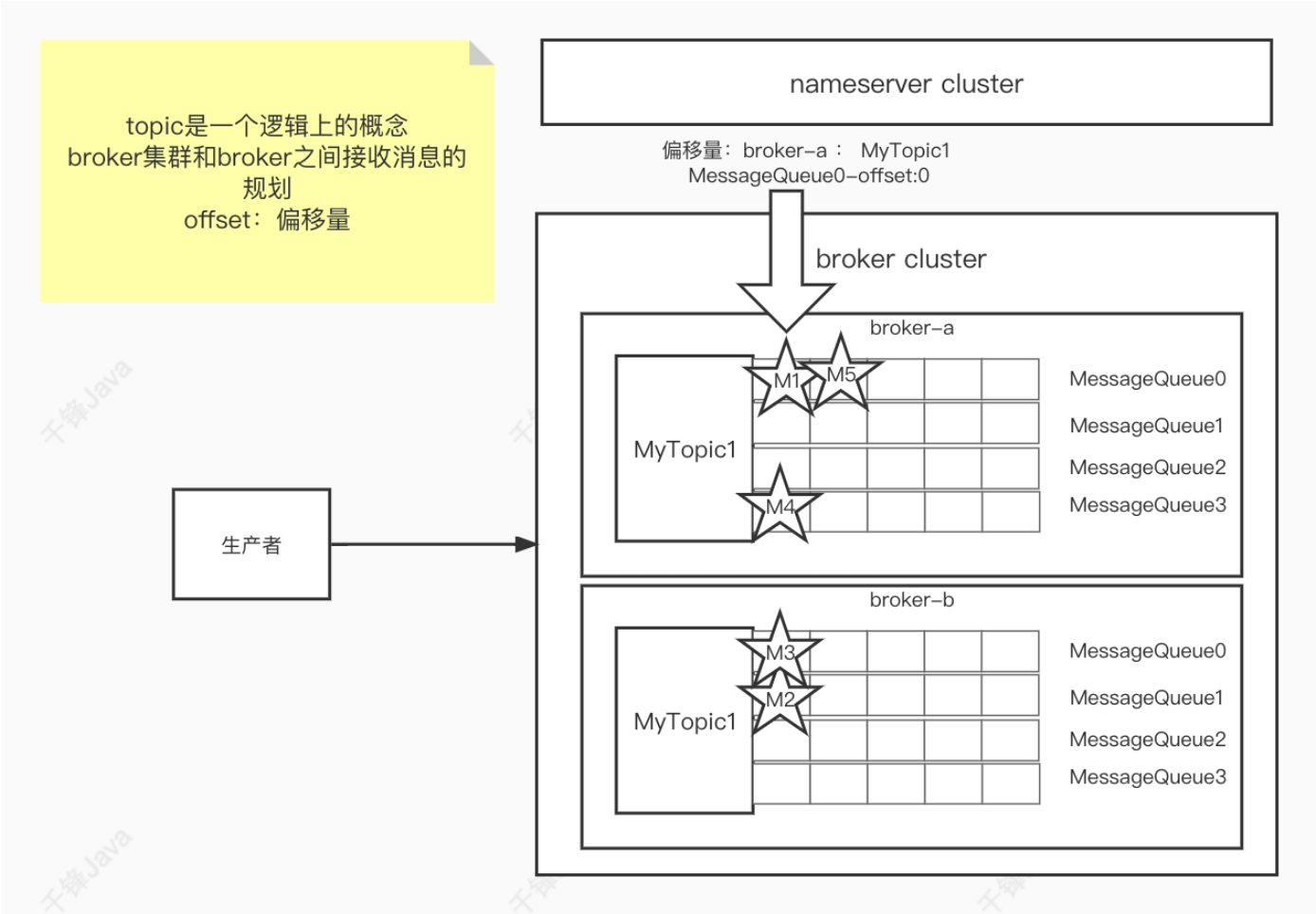
```
 * @公众号 Java架构栈
 */
public class SyncProducer {

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

        //Instantiate with a producer group name.
        DefaultMQProducer producer = new
                DefaultMQProducer("producerGroup1");
        // Specify name server addresses.
        producer.setNamesrvAddr("172.16.253.101:9876");
        //Launch the instance.
        producer.start();
        for (int i = 0; i < 100; i++) {
            //Create a message instance, specifying topic, tag and message body.
            Message msg = new Message("TopicTest" /* Topic */,
                    "TagA" /* Tag */,
                    ("Hello RocketMQ " +

 i).getBytes(RemotingHelper.DEFAULT_CHARSET) /* Message body */
            );
            //Call send message to deliver message to one of brokers.
            SendResult sendResult = producer.send(msg);
            System.out.printf("%s%n", sendResult);
        }
        //Shut down once the producer instance is not longer in use.
        producer.shutdown();
    }

}
```

- 编写消费者程序

```java
package com.qf.consumer;

import org.apache.rocketmq.client.consumer.DefaultMQPushConsumer;
import org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlyContext;
import org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlyStatus;
import org.apache.rocketmq.client.consumer.listener.MessageListenerConcurrently;
import org.apache.rocketmq.client.exception.MQClientException;
import org.apache.rocketmq.common.message.MessageExt;

import java.util.List;

/**
 * @author Thor
 * @公众号 Java架构栈
 */
public class MyConsumer {

    public static void main(String[] args) throws MQClientException {


        // Instantiate with specified consumer group name.
        DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("please_rename_unique_group_name");

        // Specify name server addresses.
        consumer.setNamesrvAddr("172.16.253.101:9876");

        // Subscribe one more more topics to consume.
        consumer.subscribe("TopicTest", "*");
```

```
29          // Register callback to execute on arrival of messages
    fetched from brokers.
30          consumer.registerMessageListener(new
    MessageListenerConcurrently() {
31
32              @Override
33              public ConsumeConcurrentlyStatus
    consumeMessage(List<MessageExt> msgs,
34
     ConsumeConcurrentlyContext context) {
35                  System.out.printf("%s Receive New Messages: %s
    %n", Thread.currentThread().getName(), msgs);
36                  return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
37              }
38          });
39
40          //Launch the consumer instance.
41          consumer.start();
42
43          System.out.printf("Consumer Started.%n");
44      }
45  }
```

- 启动消费者和生产者，验证消息的收发。



2.简单消息示例

简单消息分成三种：同步消息、异步消息、单向消息。

- 同步消息

生产者发送消息后，必须等待broker返回信息后才继续之后的业务逻辑，在broker返回信息之前，生产者阻塞等待。

```
1  package com.qf.producer.simple;
2
3  import org.apache.rocketmq.client.exception.MQBrokerException;
4  import org.apache.rocketmq.client.exception.MQClientException;
5  import org.apache.rocketmq.client.producer.DefaultMQProducer;
6  import org.apache.rocketmq.client.producer.SendResult;
7  import org.apache.rocketmq.common.message.Message;
8  import org.apache.rocketmq.remoting.common.RemotingHelper;
```

```
import org.apache.rocketmq.remoting.exception.RemotingException;

import java.io.UnsupportedEncodingException;

/**
 * @author Thor
 * @公众号 Java架构栈
 */
public class SyncProducer {

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

        //Instantiate with a producer group name.
        DefaultMQProducer producer = new
                DefaultMQProducer("producerGroup1");
        // Specify name server addresses.
        producer.setNamesrvAddr("172.16.253.101:9876");
        //Launch the instance.
        producer.start();
        for (int i = 0; i < 100; i++) {
            //Create a message instance, specifying topic, tag and message body.
            Message msg = new Message("TopicTest" /* Topic */,
                    "TagA" /* Tag */,
                    ("Hello RocketMQ " +

 i).getBytes(RemotingHelper.DEFAULT_CHARSET) /* Message body */
            );
            //Call send message to deliver message to one of brokers.
            SendResult sendResult = producer.send(msg);
            System.out.printf("%s%n", sendResult);
        }
        //Shut down once the producer instance is not longer in use.
```

```
        producer.shutdown();
    }

}

```

同步消息的应用场景：如重要通知消息、短信通知、短信营销系统等。

- 异步消息

生产者发完消息后，不需要等待broker的回信，可以直接执行之后的业务逻辑。生产者提供一个回调函数供broker调用，体现了异步的方式。

```
package com.qf.producer.simple;

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

import java.util.concurrent.CountDownLatch;
import java.util.concurrent.TimeUnit;

/**
 * 异步消息
 * @author Thor
 * @公众号 Java架构栈
 */
public class AsyncProducer {

    public static void main(String[] args) throws Exception {
        //Instantiate with a producer group name.
        DefaultMQProducer producer = new
DefaultMQProducer("please_rename_unique_group_name");
        // Specify name server addresses.
        producer.setNamesrvAddr("172.16.253.101:9876");
        //Launch the instance.
```

```
25          producer.start();
26          producer.setRetryTimesWhenSendAsyncFailed(0);
27
28          int messageCount = 100;
29          final CountDownLatch countDownLatch = new
    CountDownLatch(messageCount);
30          for (int i = 0; i < messageCount; i++) {
31              try {
32                  final int index = i;
33                  Message msg = new Message("Jodie_topic_1023",
34                          "TagA",
35                          "OrderID188",
36                          "Hello
    world".getBytes(RemotingHelper.DEFAULT_CHARSET));
37                  producer.send(msg, new SendCallback() {
38                      @Override
39                      public void onSuccess(SendResult sendResult) {
40                          countDownLatch.countDown();
41                          System.out.printf("%-10d OK %s %n", index,
    sendResult.getMsgId());
42                      }
43
44                      @Override
45                      public void onException(Throwable e) {
46                          countDownLatch.countDown();
47                          System.out.printf("%-10d Exception %s %n",
    index, e);
48                          e.printStackTrace();
49                      }
50                  });
51              } catch (Exception e) {
52                  e.printStackTrace();
53              }
54          }
55          countDownLatch.await(5, TimeUnit.SECONDS);
56          producer.shutdown();
57      }
```

```
58
59  }
60
```

异步传输一般用于响应时间敏感的业务场景。

- 单向消息

生产者发送完消息后不需要等待任何回复，直接进行之后的业务逻辑，单向传输用于需要中等可靠性的情况，例如日志收集。

```
1   package com.qf.producer.simple;
2
3   import org.apache.rocketmq.client.producer.DefaultMQProducer;
4   import org.apache.rocketmq.common.message.Message;
5   import org.apache.rocketmq.remoting.common.RemotingHelper;
6
7   /**
8    * 单向消息
9    * @author Thor
10   * @公众号 Java架构栈
11   */
12  public class OnewayProducer {
13
14      public static void main(String[] args) throws Exception {
15          //Instantiate with a producer group name.
16          DefaultMQProducer producer = new
    DefaultMQProducer("please_rename_unique_group_name");
17          // Specify name server addresses.
18          producer.setNamesrvAddr("172.16.253.101:9876");
19          //Launch the instance.
20          producer.start();
21          for (int i = 0; i < 100; i++) {
22              //Create a message instance, specifying topic, tag and
    message body.
23              Message msg = new Message("TopicTest" /* Topic */,
24                      "TagA" /* Tag */,
25                      ("Hello RocketMQ " +
```

```
26
     i).getBytes(RemotingHelper.DEFAULT_CHARSET) /* Message body */
27                 );
28                 //Call send message to deliver message to one of
    brokers.
29                 producer.sendOneway(msg);
30             }
31             //Wait for sending to complete
32             Thread.sleep(5000);
33             producer.shutdown();
34         }
35
36     }
37
```

3.顺序消息

顺序消息指的是消费者消费消息的顺序按照发送者发送消息的顺序执行。顺序消息分成两种：局部顺序和全局顺序。

- 局部顺序

局部消息指的是消费者消费某个topic的某个队列中的消息是顺序的。消费者使用MessageListenerOrderly类做消息监听，实现局部顺序。

```
1   package com.qf.producer.order;
2
3   import org.apache.rocketmq.client.consumer.DefaultMQPushConsumer;
4   import org.apache.rocketmq.client.consumer.listener.*;
5   import org.apache.rocketmq.client.exception.MQClientException;
6   import org.apache.rocketmq.common.consumer.ConsumeFromWhere;
7   import org.apache.rocketmq.common.message.MessageExt;
8
9   import java.util.List;
10  import java.util.concurrent.atomic.AtomicLong;
11
```

```
12  /**
13   * 顺序消息
14   * @author Thor
15   * @公众号 Java架构栈
16   */
17  public class OrderConsumer {
18      public static void main(String[] args) throws
    MQClientException {
19          DefaultMQPushConsumer consumer = new
    DefaultMQPushConsumer("example_group_name");
20
21
     consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_
    OFFSET);
22
23          consumer.subscribe("OrderTopicTest", "*");
24
25          consumer.registerMessageListener(new
    MessageListenerOrderly() {
26
27              @Override
28              public ConsumeOrderlyStatus
    consumeMessage(List<MessageExt> msgs,
29
    ConsumeOrderlyContext context) {
30                  context.setAutoCommit(true);
31                  for(MessageExt msg:msgs){
32                      System.out.println("消息内容: "+new
    String(msg.getBody()));
33                  }
34                  return ConsumeOrderlyStatus.SUCCESS;
35
36              }
37          });
38
39          consumer.start();
40
```

```
41          System.out.printf("Consumer Started.%n");
42      }
43  }
44
```

- 全局顺序

消费者消费全部消息都是顺序的，只能通过一个某个topic只有一个队列才能实现，这种应用场景较少，且性能较差。

- 乱序消费

消费者消费消息不需要关注消息的顺序。消费者使用MessageListenerConcurrently类做消息监听。

```
1   package com.qf.producer.order;
2
3   import org.apache.rocketmq.client.consumer.DefaultMQPushConsumer;
4   import org.apache.rocketmq.client.consumer.listener.*;
5   import org.apache.rocketmq.client.exception.MQClientException;
6   import org.apache.rocketmq.common.consumer.ConsumeFromWhere;
7   import org.apache.rocketmq.common.message.MessageExt;
8
9   import java.util.List;
10  import java.util.concurrent.atomic.AtomicLong;
11
12  /**
13   * 顺序消息
14   * @author Thor
15   * @公众号 Java架构栈
16   */
17  public class OrderConsumer {
18      public static void main(String[] args) throws
    MQClientException {
19          DefaultMQPushConsumer consumer = new
    DefaultMQPushConsumer("example_group_name");
```

```
20
21
     consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_
    OFFSET);
22
23          consumer.subscribe("OrderTopicTest", "*");
24
25          consumer.registerMessageListener(new
    MessageListenerConcurrently() {
26              @Override
27              public ConsumeConcurrentlyStatus
    consumeMessage(List<MessageExt> msgs, ConsumeConcurrentlyContext
    context) {
28                  for(MessageExt msg:msgs){
29                      System.out.println("消息内容: "+new
    String(msg.getBody()));
30                  }
31                  return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
32              }
33          });
34
35          consumer.start();
36
37          System.out.printf("Consumer Started.%n");
38      }
39  }
40
```

4.广播消息

广播是向主题（topic）的所有订阅者发送消息。订阅同一个topic的多个消费者，能全量收到生产者发送的所有消息。

- 消费者

```
1  package com.qf.producer.broadcast;
2
3  import org.apache.rocketmq.client.consumer.DefaultMQPushConsumer;
4  import
   org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlyCo
   ntext;
5  import
   org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlySt
   atus;
6  import
   org.apache.rocketmq.client.consumer.listener.MessageListenerConcur
   rently;
7  import org.apache.rocketmq.client.exception.MQClientException;
8  import org.apache.rocketmq.common.consumer.ConsumeFromWhere;
9  import org.apache.rocketmq.common.message.MessageExt;
10 import org.apache.rocketmq.common.protocol.heartbeat.MessageModel;
11
12 import java.util.List;
13
14 /**
15  * 广播消息
16  * @author Thor
17  * @公众号 Java架构栈
18  */
19 public class BroadcastConsumer {
20     public static void main(String[] args) throws Exception {
21         DefaultMQPushConsumer consumer = new
   DefaultMQPushConsumer("example_group_name");
22
23
    consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_
   OFFSET);
24
25         //set to broadcast mode
26         consumer.setMessageModel(MessageModel.BROADCASTING);
```

```
27
28          consumer.subscribe("TopicTest", "*");
29
30          consumer.registerMessageListener(new
    MessageListenerConcurrently() {
31
32              @Override
33              public ConsumeConcurrentlyStatus
    consumeMessage(List<MessageExt> msgs,
34
     ConsumeConcurrentlyContext context) {
35                  for(MessageExt msg:msgs){
36                      System.out.println("消息内容: "+new
    String(msg.getBody()));
37                  }
38                  return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
39              }
40          });
41
42          consumer.start();
43          System.out.printf("Broadcast Consumer Started.%n");
44      }
45  }
46
```

- 生产者

```
1   package com.qf.producer.broadcast;
2
3   import org.apache.rocketmq.client.exception.MQClientException;
4   import org.apache.rocketmq.client.producer.DefaultMQProducer;
5   import org.apache.rocketmq.client.producer.SendResult;
6   import org.apache.rocketmq.common.message.Message;
7   import org.apache.rocketmq.remoting.common.RemotingHelper;
8
9   /**
10   * @author Thor
```

```
 * @公众号 Java架构栈
 */
public class BroadcastProducer {

    public static void main(String[] args) throws Exception {
        DefaultMQProducer producer = new
DefaultMQProducer("ProducerGroupName");
        producer.start();

        for (int i = 0; i < 100; i++){
            Message msg = new Message("TopicTest",
                    "TagA",
                    "OrderID188",
                    ("Hello
world"+i).getBytes(RemotingHelper.DEFAULT_CHARSET));
            SendResult sendResult = producer.send(msg);
            System.out.printf("%s%n", sendResult);
        }
        producer.shutdown();
    }
}

```


5.延迟消息

延迟消息与普通消息的不同之处在于，它们要等到指定的时间之后才会被传递。
延迟消费而不是延迟发送

- 消息生产者

```
package com.qf.producer.scheduled;

import org.apache.rocketmq.client.exception.MQClientException;
import org.apache.rocketmq.client.producer.DefaultMQProducer;
import org.apache.rocketmq.common.message.Message;

```

```
6
7   /**
8    * @author Thor
9    * @公众号 Java架构栈
10   */
11  public class ScheduledProducer {
12      public static void main(String[] args) throws Exception {
13          // Instantiate a producer to send scheduled messages
14          DefaultMQProducer producer = new
    DefaultMQProducer("ExampleProducerGroup");
15          // Launch producer
16          producer.start();
17          int totalMessagesToSend = 100;
18          for (int i = 0; i < totalMessagesToSend; i++) {
19              Message message = new Message("TestTopic", ("Hello
    scheduled message " + i).getBytes());
20              // This message will be delivered to consumer 10
    seconds later.
21              message.setDelayTimeLevel(3);
22              // Send the message
23              producer.send(message);
24          }
25          // Shutdown producer after use.
26          producer.shutdown();
27      }
28  }
29
```

- 消息消费者

```
1   package com.qf.producer.scheduled;
2
3   import org.apache.rocketmq.client.consumer.DefaultMQPushConsumer;
4   import
    org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlyCo
    ntext;
```

```
5  import
   org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlySt
   atus;
6  import
   org.apache.rocketmq.client.consumer.listener.MessageListenerConcur
   rently;
7  import org.apache.rocketmq.client.exception.MQClientException;
8  import org.apache.rocketmq.common.message.MessageExt;
9
10 import java.util.List;
11
12 /**
13  * @author Thor
14  * @公众号 Java架构栈
15  */
16 public class ScheduledConsumer {
17     public static void main(String[] args) throws
   MQClientException {
18         // Instantiate message consumer
19         DefaultMQPushConsumer consumer = new
   DefaultMQPushConsumer("ExampleConsumer");
20         // Subscribe topics
21         consumer.subscribe("TestTopic", "*");
22         // Register message listener
23         consumer.registerMessageListener(new
   MessageListenerConcurrently() {
24             @Override
25             public ConsumeConcurrentlyStatus
   consumeMessage(List<MessageExt> messages,
   ConsumeConcurrentlyContext context) {
26                 for (MessageExt message : messages) {
27                     // Print approximate delay time period
28                     System.out.println("Receive message[msgId=" +
   message.getMsgId() + "] "
29                             + (System.currentTimeMillis() -
   message.getStoreTimestamp()) + "ms later");
30                 }
```

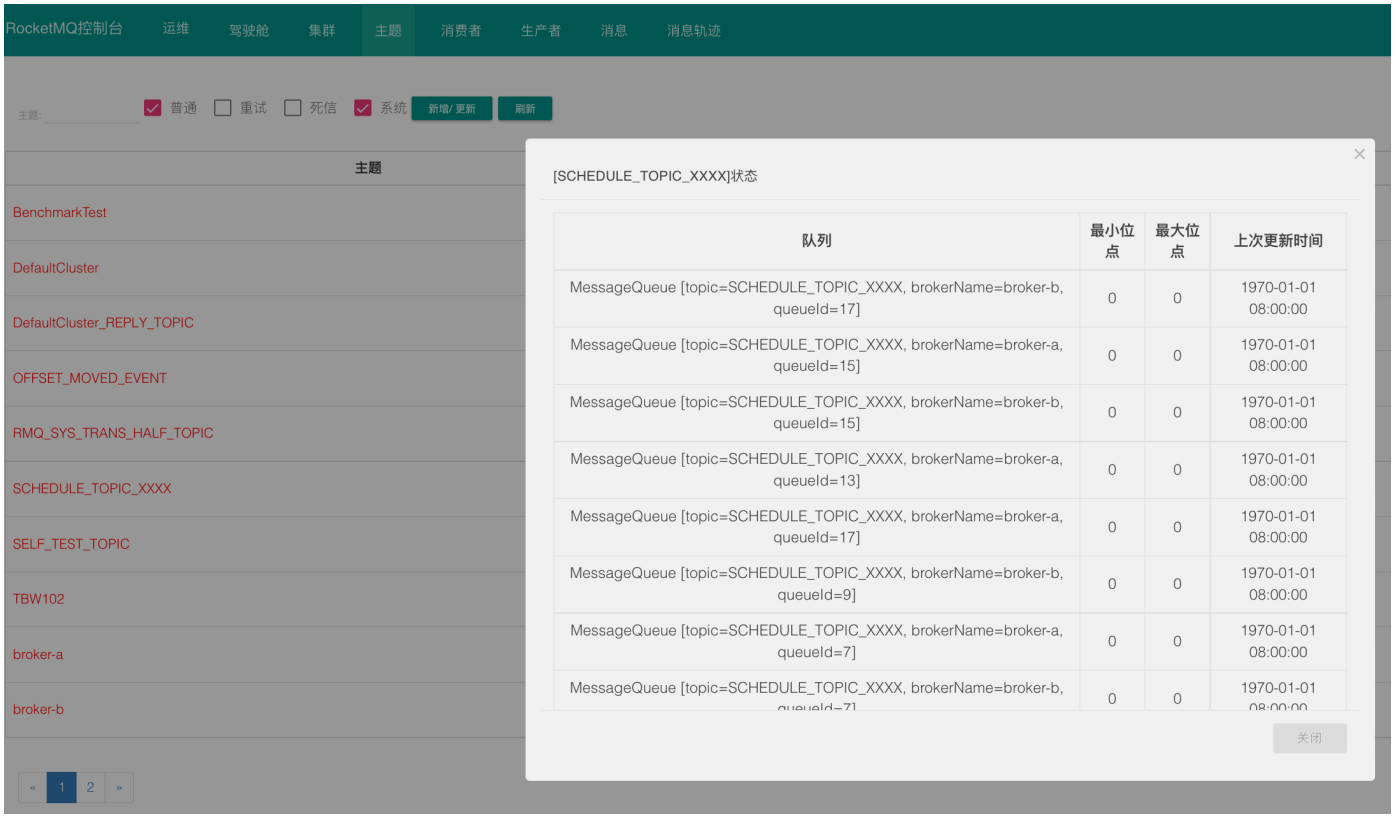
```
31                  return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
32              }
33          });
34          // Launch consumer
35          consumer.start();
36      }
37  }
```

- 延迟等级

RocketMQ设计了18个延迟等级，分别是：

```
1     1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m 30m 1h 2h
```

等级3对应的是10s。系统为这18个等级配置了18个topic，用于实现延迟队列的效果。



在商业版RocketMQ中，不仅可以设置延迟等级，还可以设置具体的延迟时间，但是在社区版RocketMQ中，只能设置延迟等级。

6.批量消息

批量发送消息提高了传递小消息的性能。

- 使用批量消息

```
1  package com.qf.producer.batch;
2
3  import org.apache.rocketmq.client.exception.MQClientException;
4  import org.apache.rocketmq.client.producer.DefaultMQProducer;
5  import org.apache.rocketmq.common.message.Message;
6
7  import java.util.ArrayList;
8  import java.util.List;
9
10 /**
11  * 批量消息
12  * @author Thor
13  * @公众号 Java架构栈
14  */
15 public class BatchProducer {
16
17     public static void main(String[] args) throws Exception {
18
19         DefaultMQProducer producer = new
   DefaultMQProducer("ProducerGroupName");
20         producer.start();
21         String topic = "BatchTest";
22         List<Message> messages = new ArrayList<>();
23         messages.add(new Message(topic, "TagA", "OrderID001",
   "Hello world 0".getBytes()));
24         messages.add(new Message(topic, "TagA", "OrderID002",
   "Hello world 1".getBytes()));
25         messages.add(new Message(topic, "TagA", "OrderID003",
   "Hello world 2".getBytes()));
26         producer.send(messages);
27         producer.shutdown();
```

```
28
29
30      }
31
32
33  }
34
```

- 超出限制的批量消息

官方建议批量消息的总大小不应超过1m，实际不应超过4m。如果超过4m的批量消息需要进行分批处理，同时设置broker的配置参数为4m（在broker的配置文件中修改：`maxMessageSize=4194304`）

```
1   package com.qf.producer.batch;
2
3   import org.apache.rocketmq.client.exception.MQClientException;
4   import org.apache.rocketmq.client.producer.DefaultMQProducer;
5   import org.apache.rocketmq.common.message.Message;
6
7   import java.util.ArrayList;
8   import java.util.List;
9
10  /**
11   * 大批量消息的处理
12   * @author Thor
13   * @公众号 Java架构栈
14   */
15  public class MaxBatchProducer {
16
17      public static void main(String[] args) throws Exception {
18
19          DefaultMQProducer producer = new
    DefaultMQProducer("BatchProducerGroupName");
20          producer.start();
21
22          //large batch
23          String topic = "BatchTest";
```

```
24          List<Message> messages = new ArrayList<>(100);
25          for (int i = 0; i < 100; i++) {
26              messages.add(new Message(topic, "Tag", "OrderID" + i,
    ("Hello world " + i).getBytes()));
27          }
28  //          producer.send(messages);
29
30          //split the large batch into small ones:
31          ListSplitter splitter = new ListSplitter(messages);
32          while (splitter.hasNext()) {
33              List<Message> listItem = splitter.next();
34              producer.send(listItem);
35          }
36          producer.shutdown();
37      }
38  }
39
```

ListSplitter

```
1   package com.qf.producer.batch;
2
3   import org.apache.rocketmq.common.message.Message;
4
5   import java.util.Iterator;
6   import java.util.List;
7   import java.util.Map;
8
9   /**
10   * 批量消息
11   * @author Thor
12   * @公众号 Java架构栈
13   */
14  public class ListSplitter implements Iterator<List<Message>> {
15      private int sizeLimit = 1000 * 1000;
16      private final List<Message> messages;
17      private int currIndex;
```

```
18
19      public ListSplitter(List<Message> messages) {
20          this.messages = messages;
21      }
22
23      @Override
24      public boolean hasNext() {
25          return currIndex < messages.size();
26      }
27
28      @Override
29      public List<Message> next() {
30          int nextIndex = currIndex;
31          int totalSize = 0;
32          for (; nextIndex < messages.size(); nextIndex++) {
33              Message message = messages.get(nextIndex);
34              int tmpSize = message.getTopic().length() +
    message.getBody().length;
35              Map<String, String> properties =
    message.getProperties();
36              for (Map.Entry<String, String> entry :
    properties.entrySet()) {
37                  tmpSize += entry.getKey().length() +
    entry.getValue().length();
38              }
39              tmpSize = tmpSize + 20; //for log overhead
40              if (tmpSize > sizeLimit) {
41                  //it is unexpected that single message exceeds the
    sizeLimit
42                  //here just let it go, otherwise it will block the
    splitting process
43                  if (nextIndex - currIndex == 0) {
44                      //if the next sublist has no element, add this
    one and then break, otherwise just break
45                      nextIndex++;
46                  }
47                  break;
```

```
48                 }
49                 if (tmpSize + totalSize > sizeLimit) {
50                     break;
51                 } else {
52                     totalSize += tmpSize;
53                 }
54 
55             }
56             List<Message> subList = messages.subList(currIndex,
    nextIndex);
57             currIndex = nextIndex;
58             return subList;
59         }
60     }
61 
```

- 使用限制

同一批次的消息应该具有：相同的主题、相同的 waitStoreMsgOK 并且不支持延迟消息和事务消息。

7.过滤消息

在大多数情况下，标签是一种简单而有用的设计，可以用来选择您想要的消息。

- tag过滤的生产者

```
1  package com.qf.rocketmq.filter;
2  
3  import org.apache.rocketmq.client.exception.MQClientException;
4  import org.apache.rocketmq.client.producer.DefaultMQProducer;
5  import org.apache.rocketmq.client.producer.SendResult;
6  import org.apache.rocketmq.common.message.Message;
7  import org.apache.rocketmq.remoting.common.RemotingHelper;
8  
9  /**
10  * @author Thor
```

```
 * @公众号 Java架构栈
 */
public class TagProducer {

    public static void main(String[] args) throws Exception {
        DefaultMQProducer producer = new
DefaultMQProducer("please_rename_unique_group_name");
        producer.start();

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

        for (int i = 0; i < 15; i++) {
            Message msg = new Message("TagFilterTest",
                    tags[i % tags.length],
                    "Hello
world".getBytes(RemotingHelper.DEFAULT_CHARSET));

            SendResult sendResult = producer.send(msg);
            System.out.printf("%s%n", sendResult);
        }

        producer.shutdown();
    }

}

```

- tag过滤的消费者

```
package com.qf.rocketmq.filter;

import org.apache.rocketmq.client.consumer.DefaultMQPushConsumer;
import
org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlyCo
ntext;
```

```
5  import
   org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlySt
   atus;
6  import
   org.apache.rocketmq.client.consumer.listener.MessageListenerConcur
   rently;
7  import org.apache.rocketmq.client.exception.MQClientException;
8  import org.apache.rocketmq.common.message.MessageExt;
9
10 import java.util.List;
11
12 /**
13  * @author Thor
14  * @公众号 Java架构栈
15  */
16 public class TagConsumer {
17     public static void main(String[] args) throws
   MQClientException {
18         DefaultMQPushConsumer consumer = new
   DefaultMQPushConsumer("please_rename_unique_group_name");
19
20         consumer.subscribe("TagFilterTest", "TagA || TagC");
21
22         consumer.registerMessageListener(new
   MessageListenerConcurrently() {
23
24             @Override
25             public ConsumeConcurrentlyStatus
   consumeMessage(List<MessageExt> msgs,
26
    ConsumeConcurrentlyContext context) {
27                 System.out.printf("%s Receive New Messages: %s
   %n", Thread.currentThread().getName(), msgs);
28                 return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
29             }
30         });
31
```

```
32          consumer.start();
33
34          System.out.printf("Consumer Started.%n");
35      }
36  }
37
```

消费者将收到包含 TAGA 或 TAGB 或 TAGC 的消息。但是限制是一条消息只能有一个标签，这可能不适用于复杂的场景。在这种情况下，您可以使用 SQL 表达式来过滤掉消息。

- 使用SQL过滤

SQL 功能可以通过您在发送消息时输入的属性进行一些计算。在 RocketMQ 定义的语法下，可以实现一些有趣的逻辑。这是一个例子：

```
------------
| message  |
|----------| a > 5 AND b = 'abc'
| a = 10   | --------------------> Gotten
| b = 'abc'|
| c = true |
------------
------------
| message  |
|----------| a > 5 AND b = 'abc'
| a = 1    | --------------------> Missed
| b = 'abc'|
| c = true |
------------
```

- 语法

RocketMQ 只定义了一些基本的语法来支持这个特性，也可以轻松扩展它。

```
1. 数值比较，如`>`, `>=`, `<`, `<=`, `BETWEEN`, `=`;
2. 字符比较，如`=`, `<>`, `IN`;
3. `IS NULL`或`IS NOT NULL`;
4. 逻辑`AND`, `OR`, `NOT`;
```

常量类型有：

```
1. 数字，如 123、3.1415;
2. 字符，如'abc'，必须用单引号;
3. `NULL`，特殊常数;
4. 布尔值，`TRUE`或`FALSE`;
```

使用注意：只有推模式的消费者可以使用SQL过滤。拉模式是用不了的。

- SQL过滤的生产者示例

```
package com.qf.rocketmq.filter;

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

/**
 * @author Thor
 * @公众号 Java架构栈
 */
public class SQLProducer {
    public static void main(String[] args) throws Exception {
        DefaultMQProducer producer = new
DefaultMQProducer("please_rename_unique_group_name");

        producer.start();

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

        for (int i = 0; i < 15; i++) {
```

```
            Message msg = new Message("SqlFilterTest",
                    tags[i % tags.length],
                    ("Hello RocketMQ " +
i).getBytes(RemotingHelper.DEFAULT_CHARSET)
            );
            // Set some properties.
            msg.putUserProperty("a", String.valueOf(i));

            SendResult sendResult = producer.send(msg);
            System.out.printf("%s%n", sendResult);
        }

        producer.shutdown();
    }
}

```

- SQL过滤的消费者示例

```
package com.qf.rocketmq.filter;

import org.apache.rocketmq.client.consumer.DefaultMQPushConsumer;
import org.apache.rocketmq.client.consumer.MessageSelector;
import
org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlyCo
ntext;
import
org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlySt
atus;
import
org.apache.rocketmq.client.consumer.listener.MessageListenerConcur
rently;
import org.apache.rocketmq.client.exception.MQClientException;
import org.apache.rocketmq.common.message.MessageExt;

import java.util.List;

```

```
13  /**
14   * @author Thor
15   * @公众号 Java架构栈
16   */
17  public class SQLConsumer {
18      public static void main(String[] args) throws
    MQClientException {
19
20          DefaultMQPushConsumer consumer = new
    DefaultMQPushConsumer("please_rename_unique_group_name");
21
22          // Don't forget to set enablePropertyFilter=true in broker
23          consumer.subscribe("SqlFilterTest",
24                  MessageSelector.bySql("(TAGS is not null and TAGS
    in ('TagA', 'TagB'))" +
25                          "and (a is not null and a between 0 and
    3)"));
26
27          consumer.registerMessageListener(new
    MessageListenerConcurrently() {
28
29              @Override
30              public ConsumeConcurrentlyStatus
    consumeMessage(List<MessageExt> msgs,
31
     ConsumeConcurrentlyContext context) {
32                  System.out.printf("%s Receive New Messages: %s
    %n", Thread.currentThread().getName(), msgs);
33                  return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
34              }
35          });
36
37          consumer.start();
38          System.out.printf("Consumer Started.%n");
39      }
40  }
41
```

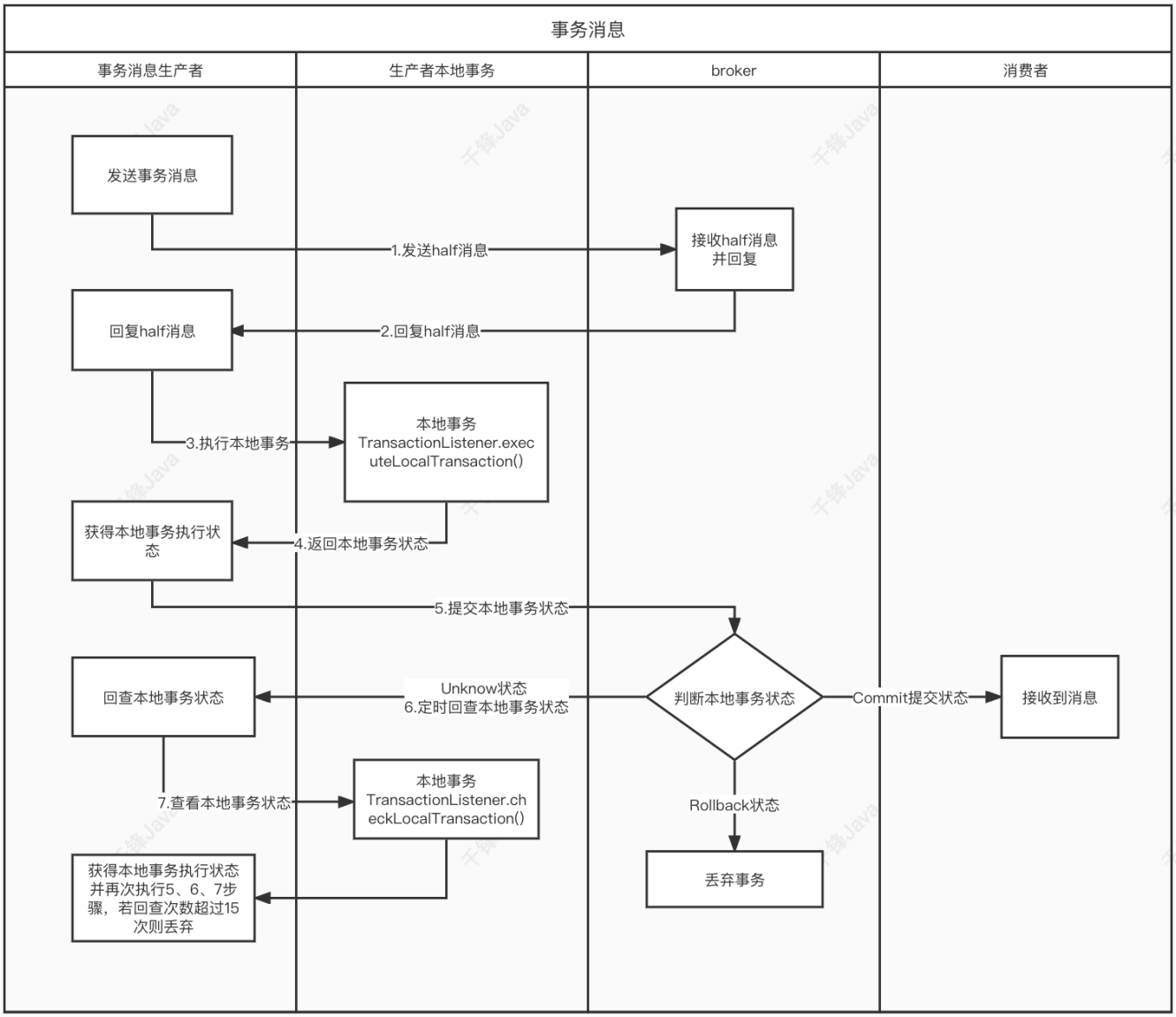
8.事务消息

- 事务消息的定义

它可以被认为是一个两阶段的提交消息实现，以确保分布式系统的最终一致性。事务性消息确保本地事务的执行和消息的发送可以原子地执行。

事务消息有三种状态：
a.TransactionStatus.CommitTransaction：提交事务，表示允许消费者消费该消息。
b.TransactionStatus.RollbackTransaction：回滚事务，表示该消息将被删除，不允许消费。
c.TransactionStatus.Unknown：中间状态，表示需要MQ回查才能确定状态。

- 事务消息的实现流程



- 生产者

```
1  package com.qf.rocketmq.transaction;
2
3  import org.apache.rocketmq.client.exception.MQClientException;
4  import org.apache.rocketmq.client.producer.SendResult;
5  import org.apache.rocketmq.client.producer.TransactionListener;
6  import org.apache.rocketmq.client.producer.TransactionMQProducer;
7  import org.apache.rocketmq.common.message.Message;
8  import org.apache.rocketmq.remoting.common.RemotingHelper;
9
10 import java.io.UnsupportedEncodingException;
11 import java.util.concurrent.*;
12
```

```
13  /**
14   * @author Thor
15   * @公众号 Java架构栈
16   */
17  public class TransactionProducer {
18      public static void main(String[] args) throws Exception {
19          TransactionListener transactionListener = new
    TransactionListenerImpl();
20          TransactionMQProducer producer = new
    TransactionMQProducer("please_rename_unique_group_name");
21          producer.setNamesrvAddr("172.16.253.101:9876");
22          ExecutorService executorService = new
    ThreadPoolExecutor(2, 5, 100, TimeUnit.SECONDS, new
    ArrayBlockingQueue<Runnable>(2000), new ThreadFactory() {
23              @Override
24              public Thread newThread(Runnable r) {
25                  Thread thread = new Thread(r);
26                  thread.setName("client-transaction-msg-check-
    thread");
27                  return thread;
28              }
29          });
30
31          producer.setExecutorService(executorService);
32          producer.setTransactionListener(transactionListener);
33          producer.start();
34
35          String[] tags = new String[] {"TagA", "TagB", "TagC",
    "TagD", "TagE"};
36          for (int i = 0; i < 10; i++) {
37              try {
38                  Message msg =
39                          new Message("TopicTest", tags[i %
    tags.length], "KEY" + i,
40                                  ("Hello RocketMQ " +
    i).getBytes(RemotingHelper.DEFAULT_CHARSET));
```

```
41                  SendResult sendResult =
    producer.sendMessageInTransaction(msg, null);
42                  System.out.printf("%s%n", sendResult);
43
44                  Thread.sleep(10);
45              } catch (MQClientException |
    UnsupportedEncodingException e) {
46                  e.printStackTrace();
47              }
48          }
49
50          for (int i = 0; i < 100000; i++) {
51              Thread.sleep(1000);
52          }
53          producer.shutdown();
54      }
55  }
56
```

- 本地事务处理-TransactionListener

```
1   package com.qf.rocketmq.transaction;
2
3   import org.apache.commons.lang3.StringUtils;
4   import org.apache.rocketmq.client.producer.LocalTransactionState;
5   import org.apache.rocketmq.client.producer.TransactionListener;
6   import org.apache.rocketmq.common.message.Message;
7   import org.apache.rocketmq.common.message.MessageExt;
8
9   /**
10   * @author Thor
11   * @公众号 Java架构栈
12   */
13  public class TransactionListenerImpl implements
    TransactionListener {
14      /**
```

```
15       * When send transactional prepare(half) message succeed, this
    method will be invoked to execute local transaction.
16       *
17       * @param msg Half(prepare) message
18       * @param arg Custom business parameter
19       * @return Transaction state
20       */
21      @Override
22      public LocalTransactionState executeLocalTransaction(Message
    msg, Object arg) {
23          String tags = msg.getTags();
24          if(StringUtils.contains(tags,"TagA")){
25              return LocalTransactionState.COMMIT_MESSAGE;
26          }else if(StringUtils.contains(tags,"TagB")){
27              return LocalTransactionState.ROLLBACK_MESSAGE;
28          }else{
29              return LocalTransactionState.UNKNOW;
30          }
31      }
32
33      /**
34       * When no response to prepare(half) message. broker will send
    check message to check the transaction status, and this
35       * method will be invoked to get local transaction status.
36       *
37       * @param msg Check message
38       * @return Transaction state
39       */
40      @Override
41      public LocalTransactionState checkLocalTransaction(MessageExt
    msg) {
42          String tags = msg.getTags();
43          if(StringUtils.contains(tags,"TagC")){
44              return LocalTransactionState.COMMIT_MESSAGE;
45          }else if(StringUtils.contains(tags,"TagD")){
46              return LocalTransactionState.ROLLBACK_MESSAGE;
47          }else{
```

```
48              return LocalTransactionState.UNKNOW;
49          }
50      }
51  }
52
```

- 消费者

```
1   package com.qf.rocketmq.transaction;
2
3   import org.apache.rocketmq.client.consumer.DefaultMQPushConsumer;
4   import
    org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlyCo
    ntext;
5   import
    org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlySt
    atus;
6   import
    org.apache.rocketmq.client.consumer.listener.MessageListenerConcur
    rently;
7   import org.apache.rocketmq.client.exception.MQClientException;
8   import org.apache.rocketmq.common.message.MessageExt;
9
10  import java.util.List;
11
12  /**
13   * @author Thor
14   * @公众号 Java架构栈
15   */
16  public class TransactionConsumer {
17
18      public static void main(String[] args) throws
    MQClientException {
19          //1.创建消费者对象
20          DefaultMQPushConsumer consumer = new
    DefaultMQPushConsumer("my-consumer-group1");
21          //2.指明nameserver的地址
```

```
22          consumer.setNamesrvAddr("172.16.253.101:9876");
23          //3.订阅主题:topic 和过滤消息用的tag表达式
24          consumer.subscribe("TopicTest","*");
25          //4.创建一个监听器，当broker把消息推过来时调用
26          consumer.registerMessageListener(new
    MessageListenerConcurrently() {
27              @Override
28              public ConsumeConcurrentlyStatus
    consumeMessage(List<MessageExt> msgs, ConsumeConcurrentlyContext
    context) {
29
30                  for (MessageExt msg : msgs) {
31  //                      System.out.println("收到的消息: "+new
    String(msg.getBody()));
32                      System.out.println("收到的消息: "+msg);
33                  }
34
35                  return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
36              }
37          });
38          //5.启动消费者
39          consumer.start();
40          System.out.println("消费者已启动");
41      }
42  }
43
```

- 使用限制
  - 事务性消息没有调度和批处理支持。
  - 为避免单条消息被检查次数过多，导致半队列消息堆积，我们默认将单条消息的检查次数限制为15次，但用户可以通过更改“transactionCheckMax”来更改此限制”参数在broker的配置中，如果一条消息的检查次数超过“transactionCheckMax”次，broker默认会丢弃这条消息，同时打印错误日志。用户可以通过重写“AbstractTransactionCheckListener”类来改变这种行为。
  - 事务消息将在一定时间后检查，该时间由代理配置中的参数

“transactionTimeout”确定。并且用户也可以在发送事务消息时通过设置用户属性“CHECK_IMMUNITY_TIME_IN_SECONDS”来改变这个限制，这个参数优先于“transactionMsgTimeout”参数。

- 一个事务性消息可能会被检查或消费不止一次。
- 提交给用户目标主题的消息reput可能会失败。目前，它取决于日志记录。高可用是由 RocketMQ 本身的高可用机制来保证的。如果要保证事务消息不丢失，保证事务完整性，推荐使用同步双写机制。
- 事务性消息的生产者 ID 不能与其他类型消息的生产者 ID 共享。与其他类型的消息不同，事务性消息允许向后查询。MQ 服务器通过其生产者 ID 查询客户端。

六、SpringBoot整合RocketMQ

Springboot提供了快捷操作RocketMQ的RocketMQTemplate对象。

1.引入依赖

注意依赖的版本需要和RocketMQ的版本相同。

```
        <dependency>
            <groupId>org.apache.rocketmq</groupId>
            <artifactId>rocketmq-spring-boot-starter</artifactId>
            <version>2.1.1</version>
        </dependency>
```

2.编写配置文件

```
# 应用名称
spring.application.name=my-boot-rocketmq-demo
# 应用服务 WEB 访问端口
server.port=8080
# nameserver地址
rocketmq.name-server=172.16.253.101:9876
# 配置生产者组
rocketmq.producer.group=my-producer-boot-group1
```

3.编写生产者发送普通消息

```
package com.qf.mybrdemo.producer;

import org.apache.rocketmq.spring.core.RocketMQTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

/**
 * @author Thor
 * @公众号 Java架构栈
 */
@Component
public class MyProducer {

    @Autowired
    private RocketMQTemplate rocketMQTemplate;

    /**
     * 发送消息
     * @param msg 消息内容
     * @param topic 目标主题
     */
    public void sendMessage(String msg,String topic){
        //将msg转换成Message对象并发送
        rocketMQTemplate.convertAndSend(topic,msg);
```

```
25      }
26  }
27
```

4.编写JUnit单元测试发送消息

```
1       @Test
2       void testSendMessage(){
3           String topic = "MyBootTopic";
4           String message = "hello spring boot rocketmq";
5           producer.sendMessage(message,topic);
6       }
```

5.创建消费者程序

```
1   package com.qf.mybootcdemo.consumer;
2
3   import
    org.apache.rocketmq.spring.annotation.RocketMQMessageListener;
4   import org.apache.rocketmq.spring.core.RocketMQListener;
5   import org.springframework.stereotype.Component;
6
7   /**
8    * @author Thor
9    * @公众号 Java架构栈
10   */
11  @Component
12  @RocketMQMessageListener(consumerGroup = "my-boot-consumer-group",
    topic = "MyBootTopic")
13  public class MyConsumer implements RocketMQListener<String> {
14      @Override
15      public void onMessage(String msg) {
16          System.out.println("收到的消息:" + msg);
17      }
```

```
18  }
19
```

6.发送事务消息

- 编写生产者方法

```
1      public void sendMessageInTransaction(String msg, String topic)
   throws Exception {
2          String[] tags = new String[]{"TagA", "TagB", "TagC",
   "TagD", "TagE"};
3          for (int i = 0; i < 10; i++) {
4              //注意该message为org.springframework.messaging.Message
5              Message<String> message =
   MessageBuilder.withPayload(msg).build();
6              //topic和tag整合在一起，以":"隔开
7              String destination = topic+":"+tags[i % tags.length];
8              //第一个destination为消息要发到的目的地，第二个destination为
   消协携带的业务数据
9              TransactionSendResult sendResult =
   rocketMQTemplate.sendMessageInTransaction(destination, message,
   destination);
10             System.out.println(sendResult);
11             Thread.sleep(10);
12         }
13
14     }
```

- 编写事务监听器类

```
1  package com.qf.mybrdemo.TransactionConfig;
2
3  import org.apache.commons.lang3.StringUtils;
4  import org.apache.rocketmq.client.producer.LocalTransactionState;
5  import
   org.apache.rocketmq.spring.annotation.RocketMQMessageListener;
```

```
import org.apache.rocketmq.spring.annotation.RocketMQTransactionListener;
import org.apache.rocketmq.spring.core.RocketMQLocalTransactionListener;
import org.apache.rocketmq.spring.core.RocketMQLocalTransactionState;
import org.apache.rocketmq.spring.support.RocketMQUtil;
import org.springframework.messaging.Message;
import org.springframework.messaging.converter.StringMessageConverter;

/**
 * @author Thor
 * @公众号 Java架构栈
 */
@RocketMQTransactionListener(rocketMQTemplateBeanName = "rocketMQTemplate")
public class MyTransactionListenerImpl implements RocketMQLocalTransactionListener {



    @Override
    public RocketMQLocalTransactionState executeLocalTransaction(Message msg, Object arg) {
        //获得业务参数中的数据
        String destination = (String) arg;
        //使用RocketMQUtil将spring的message转换成rocketmq的message
        org.apache.rocketmq.common.message.Message message = RocketMQUtil.convertToRocketMessage(
                new StringMessageConverter(),"utf-8",destination,msg);
        //获得消息中的业务数据tags
        String tags = message.getTags();
        if(StringUtils.contains(tags,"TagA")){
            //提交本地事务
            return RocketMQLocalTransactionState.COMMIT;
```

```
34          }else if(StringUtils.contains(tags,"TagB")){
35              //回滚
36              return RocketMQLocalTransactionState.ROLLBACK;
37          }else{
38              //中间状态
39              return RocketMQLocalTransactionState.UNKNOWN;
40          }
41
42      }
43
44      @Override
45      public RocketMQLocalTransactionState
    checkLocalTransaction(Message msg) {
46          return null;
47      }
48  }
49
```

7.编写单元测试发送事务消息

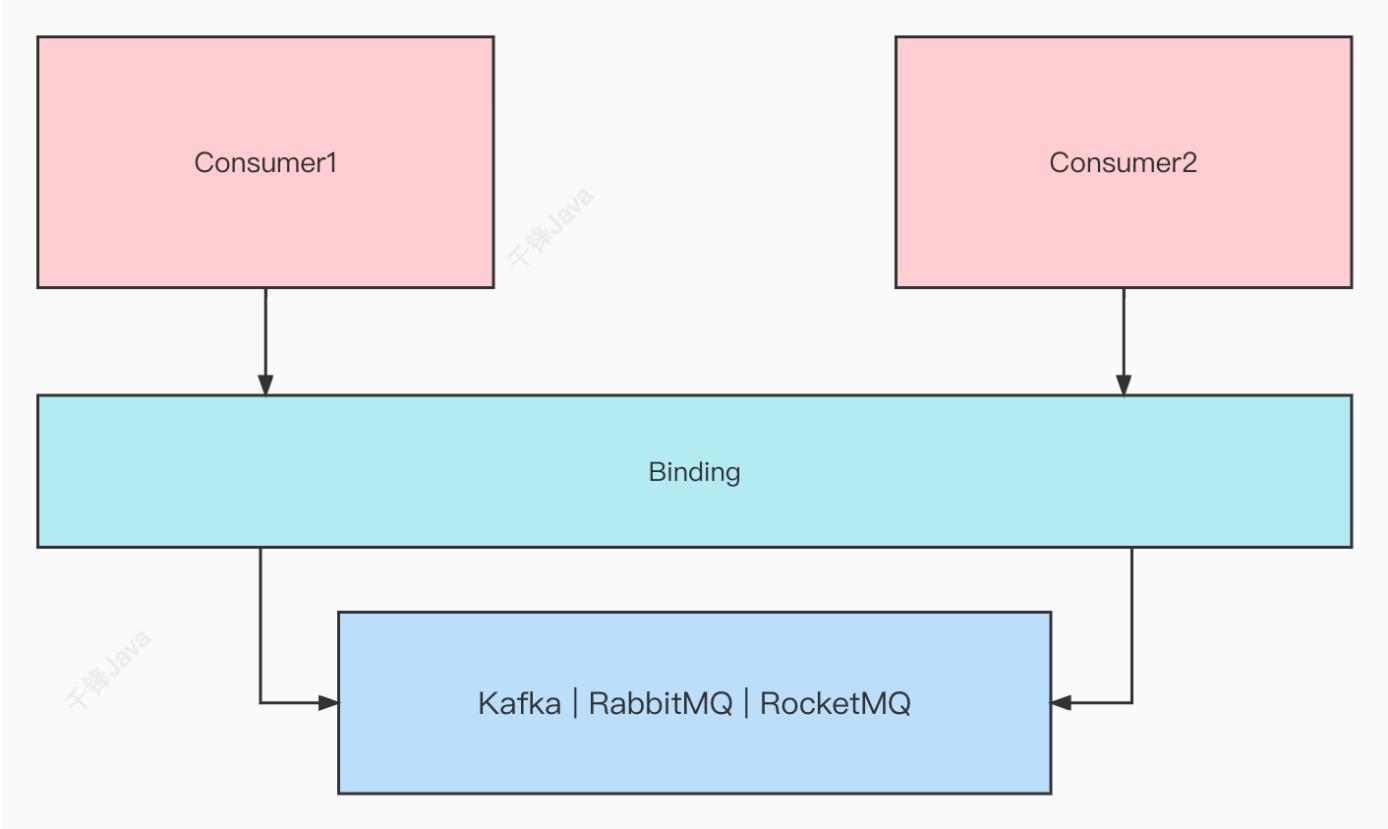
```
1       @Test
2       void testSendTransactionMessage() throws Exception {
3           String topic = "MyBootTopic";
4           String message = "hello transaction spring boot rocketmq";
5           producer.sendMessageInTransaction(message,topic);
6       }
```

七、Spring Cloud Stream整合RocketMQ

1.Spring Cloud Stream介绍

Spring Cloud Stream 是一个框架，用于构建与共享消息系统连接的高度可扩展的事件驱动微服务。

该框架提供了一个灵活的编程模型，该模型基于已经建立和熟悉的 Spring 习惯用法和最佳实践，包括对持久 pub/sub 语义、消费者组和有状态分区的支持。



Spring Cloud Stream 的核心构建块是：

- **Destination Binders**：负责提供与外部消息传递系统集成的组件。
- **Destination Bindings**：外部消息系统和最终用户提供的应用程序代码（生产者/消费者）之间的桥梁。
- **Message**：生产者和消费者用来与目标绑定器（以及通过外部消息系统的其他应用程序）进行通信的规范数据结构。

2.编写生产者

- 引入依赖

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

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.qf</groupId>
    <artifactId>my-s-rocketmq-demo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>my-s-rocketmq-demo</name>
    <description>Demo project for Spring Boot</description>

    <properties>
        <java.version>1.8</java.version>
        <project.build.sourceEncoding>UTF-
8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
        <spring-boot.version>2.3.7.RELEASE</spring-boot.version>
        <spring-cloud-alibaba.version>2.2.2.RELEASE</spring-
cloud-alibaba.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>com.alibaba.cloud</groupId>
            <artifactId>spring-cloud-starter-stream-
rocketmq</artifactId>
            <exclusions>
                <exclusion>
                    <groupId>org.apache.rocketmq</groupId>
                    <artifactId>rocketmq-client</artifactId>
                </exclusion>
                <exclusion>
```

```
                    <groupId>org.apache.rocketmq</groupId>
                    <artifactId>rocketmq-acl</artifactId>
                </exclusion>
            </exclusions>
        </dependency>

        <dependency>
            <groupId>org.apache.rocketmq</groupId>
            <artifactId>rocketmq-client</artifactId>
            <version>4.7.1</version>
        </dependency>
        <dependency>
            <groupId>org.apache.rocketmq</groupId>
            <artifactId>rocketmq-acl</artifactId>
            <version>4.7.1</version>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
            <exclusions>
                <exclusion>
                    <groupId>org.junit.vintage</groupId>
                    <artifactId>junit-vintage-engine</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
    </dependencies>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-dependencies</artifactId>
                <version>${spring-boot.version}</version>
                <type>pom</type>
```

```
70                  <scope>import</scope>
71              </dependency>
72              <dependency>
73                  <groupId>com.alibaba.cloud</groupId>
74                  <artifactId>spring-cloud-alibaba-
    dependencies</artifactId>
75                  <version>${spring-cloud-alibaba.version}
    </version>
76                  <type>pom</type>
77                  <scope>import</scope>
78              </dependency>
79          </dependencies>
80      </dependencyManagement>
81
82      <build>
83          <plugins>
84              <plugin>
85                  <groupId>org.apache.maven.plugins</groupId>
86                  <artifactId>maven-compiler-plugin</artifactId>
87                  <version>3.8.1</version>
88                  <configuration>
89                      <source>1.8</source>
90                      <target>1.8</target>
91                      <encoding>UTF-8</encoding>
92                  </configuration>
93              </plugin>
94              <plugin>
95                  <groupId>org.springframework.boot</groupId>
96                  <artifactId>spring-boot-maven-plugin</artifactId>
97                  <version>2.3.7.RELEASE</version>
98                  <configuration>
99
     <mainClass>com.qf.my.s.rocketmq.demo.MySRocketmqDemoApplication<
    /mainClass>
100                 </configuration>
101                 <executions>
102                     <execution>
```

```
                        <id>repackage</id>
                        <goals>
                            <goal>repackage</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>

</project>

```

注意，Rocket官方维护的spring-cloud-stream依赖中rocket用的版本为4.4，需要排除后加入4.7.1的依赖。

- 编写配置文件

```
# 应用名称
spring.application.name=my-s-rocketmq-demo
# 应用服务 WEB 访问端口
server.port=8080
# output 生产者
spring.cloud.stream.bindings.output.destination=TopicTest
# 配置rocketMQ
spring.cloud.stream.rocketmq.binder.name-server=172.16.253.101:9876
```

- 启动类上打上注解

```
package com.qf.my.s.rocketmq.demo;

import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.cloud.stream.messaging.Sink;
import org.springframework.cloud.stream.messaging.Source;

```

```
@EnableBinding(Source.class)
@SpringBootApplication
public class MySRocketmqDemoApplication {

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

}

```

其中`@EnableBinding(Source.class)`指向配置文件的output参数。

- 编写生产者程序

```
package com.qf.my.s.rocketmq.demo.producer;

import org.apache.rocketmq.common.message.MessageConst;
import org.springframework.cloud.stream.messaging.Source;
import org.springframework.messaging.Message;
import org.springframework.messaging.MessageHeaders;
import org.springframework.messaging.support.MessageBuilder;
import org.springframework.stereotype.Component;

import javax.annotation.Resource;
import java.util.HashMap;
import java.util.Map;

/**
 * @author Thor
 * @公众号 Java架构栈
 */
@Component
public class MyProducer {

    @Resource
    private Source source;
```

```
23
24      public void sendMessage(String msg){
25          Map<String, Object> headers = new HashMap<>();
26          headers.put(MessageConst.PROPERTY_TAGS,"TagA");
27          MessageHeaders messageHeaders = new
    MessageHeaders(headers);
28          Message<String> message =
    MessageBuilder.createMessage(msg, messageHeaders);
29          source.output().send(message);
30      }
31  }
```

- 编写单元测试发送消息

```
1   package com.qf.my.s.rocketmq.demo;
2
3   import com.qf.my.s.rocketmq.demo.producer.MyProducer;
4   import org.junit.jupiter.api.Test;
5   import org.springframework.beans.factory.annotation.Autowired;
6   import org.springframework.boot.test.context.SpringBootTest;
7
8   @SpringBootTest
9   class MySRocketmqDemoApplicationTests {
10
11      @Autowired
12      private MyProducer producer;
13
14      @Test
15      void testSendMessage(){
16          producer.sendMessage("hello spring cloud stream");
17      }
18  }
```

2.编写消费者

- 引入依赖

与生产者相同

- 编写配置文件

```
# 应用名称
spring.application.name=my-s-rocketmq-demo
# 应用服务 WEB 访问端口
server.port=8081
# input 消费者
spring.cloud.stream.bindings.input.destination=TopicTest
spring.cloud.stream.bindings.input.group=spring-cloud-strema-group
# 配置rocketMQ
spring.cloud.stream.rocketmq.binder.name-server=172.16.253.101:9876
```

- 启动类上打上注解

```
@EnableBinding(Sink.class)
@SpringBootApplication
public class MySpringCConsumerDemoApplication {

    public static void main(String[] args) {

 SpringApplication.run(MySpringCConsumerDemoApplication.class,
args);
    }
}
```

其中`@EnableBinding(Sink.class)`指向配置文件的input参数。

- 编写消费者程序

```
 package com.qf.my.spring.c.consumer.demo.listener;

 import org.springframework.cloud.stream.annotation.StreamListener;
 import org.springframework.cloud.stream.messaging.Sink;
 import org.springframework.stereotype.Component;

 /**
```

```
 * @author Thor
 * @公众号 Java架构栈
 */
@Component
public class MyConsumer {

    @StreamListener(Sink.INPUT)
    public void onMessage(String message){
        System.out.println("收到的消息: "+message);
    }
}

```

八、RocketMQ核心概念

1.消息模型（Message Model）

RocketMQ主要由 Producer、Broker、Consumer 三部分组成，其中Producer 负责生产消息，Consumer 负责消费消息，Broker 负责存储消息。Broker 在实际部署过程中对应一台服务器，每个 Broker 可以存储多个Topic的消息，每个Topic的消息也可以分片存储于不同的 Broker。Message Queue 用于存储消息的物理地址，每个Topic中的消息地址存储于多个 Message Queue 中。ConsumerGroup 由多个Consumer 实例构成。

2.消息生产者（Producer）

负责生产消息，一般由业务系统负责生产消息。一个消息生产者会把业务应用系统里产生的消息发送到broker服务器。RocketMQ提供多种发送方式，同步发送、异步发送、顺序发送、单向发送。同步和异步方式均需要Broker返回确认信息，单向发送不需要。

生产者组将多个生产者归为一组。用于保证生产者的高可用，比如在事务消息中回查本地事务状态，需要生产者具备高可用的特性，才能完成整个任务。

3.消息消费者（Consumer）

负责消费消息，一般是后台系统负责异步消费。一个消息消费者会从Broker服务器拉取消息、并将其提供给应用程序。从用户应用的角度而言提供了两种消费形式：拉取式消费、推动式消费。

消费者组将多个消息消费者归为一组，用于保证消费者的高可用和高性能。

4.主题（Topic）

表示一类消息的集合，每个主题包含若干条消息，每条消息只能属于一个主题，是RocketMQ进行消息订阅的基本单位。

5.代理服务器（Broker Server）

消息中转角色，负责存储消息、转发消息。代理服务器在RocketMQ系统中负责接收从生产者发送来的消息并存储、同时为消费者的拉取请求作准备。代理服务器也存储消息相关的元数据，包括消费者组、消费进度偏移和主题和队列消息等。

6.名字服务（Name Server）

名称服务充当路由消息的提供者。生产者或消费者能够通过名字服务查找各主题相应的Broker IP列表。多个Namesrv实例组成集群，但相互独立，没有信息交换。

7.拉取式消费（Pull Consumer）

Consumer消费的一种类型，应用通常主动调用Consumer的拉消息方法从Broker服务器拉消息、主动权由应用控制。一旦获取了批量消息，应用就会启动消费过程。

8.推动式消费（Push Consumer）

Consumer消费的一种类型，该模式下Broker收到数据后会主动推送给消费端，该消费模式一般实时性较高。

9.生产者组（Producer Group）

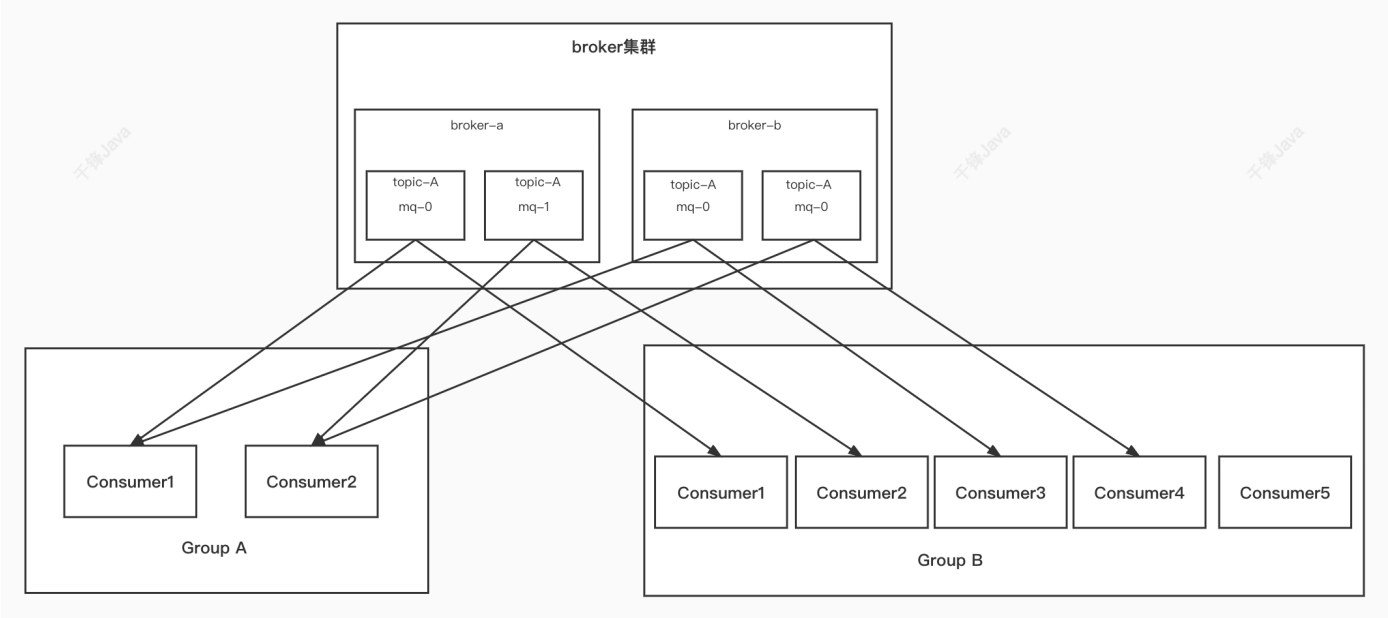
同一类Producer的集合，这类Producer发送同一类消息且发送逻辑一致。如果发送的是事务消息且原始生产者在发送之后崩溃，则Broker服务器会联系同一生产者组的其他生产者实例以提交或回溯消费。

10.消费者组（Consumer Group）

同一类Consumer的集合，这类Consumer通常消费同一类消息且消费逻辑一致。消费者组使得在消息消费方面，实现负载均衡和容错的目标变得非常容易。要注意的是，消费者组的消费者实例必须订阅完全相同的Topic。RocketMQ 支持两种消息模式：集群消费（Clustering）和广播消费（Broadcasting）。

11.集群消费（Clustering）

集群消费模式下,相同Consumer Group的每个Consumer实例平均分摊消息。



12.广播消费（Broadcasting）

广播消费模式下，相同Consumer Group的每个Consumer实例都接收全量的消息。

13.普通顺序消息（Normal Ordered Message）

普通顺序消费模式下，消费者通过同一个消费队列收到的消息是有顺序的，不同消息队列收到的消息则可能是无顺序的。

14.严格顺序消息（Strictly Ordered Message）

严格顺序消息模式下，消费者收到的所有消息均是有顺序的。

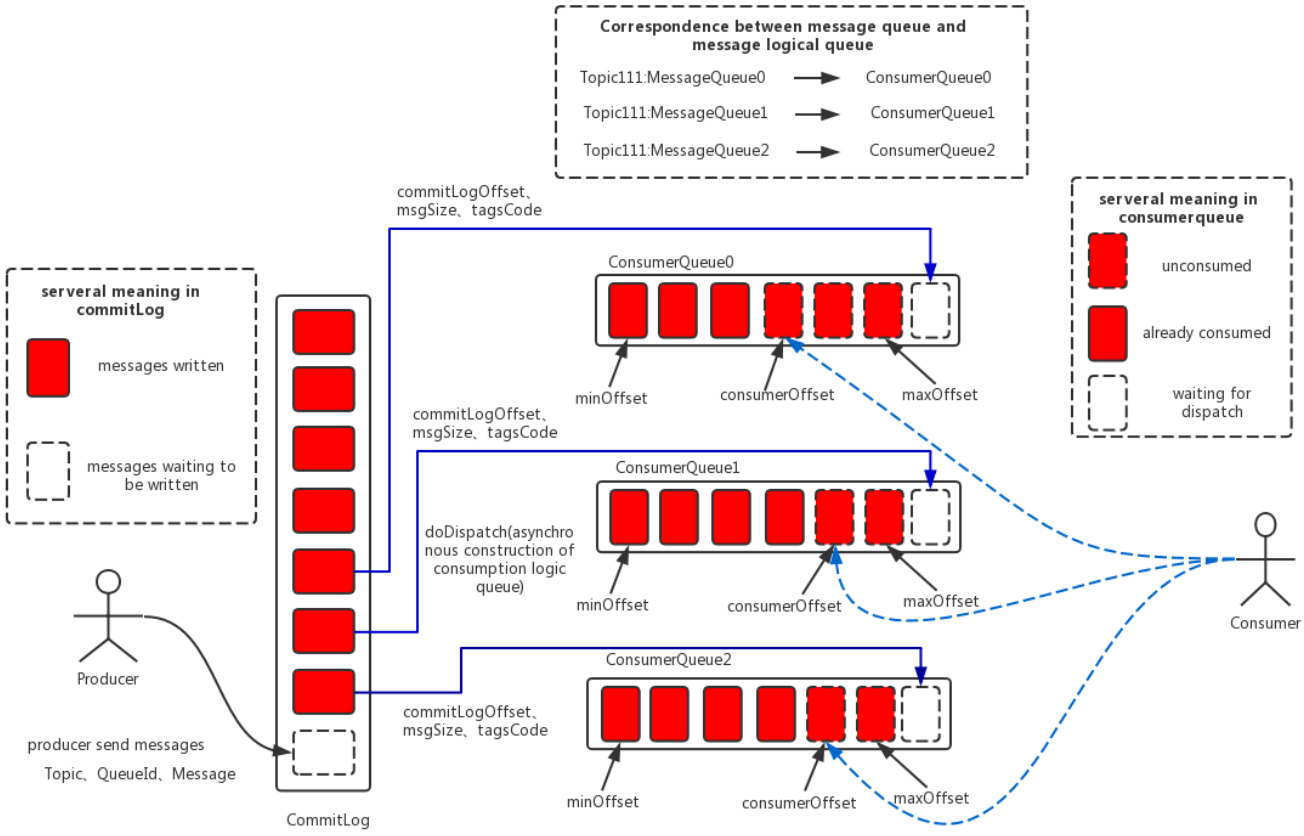
15.消息（Message）

消息系统所传输信息的物理载体，生产和消费数据的最小单位，每条消息必须属于一个主题。RocketMQ中每个消息拥有唯一的Message ID，且可以携带具有业务标识的Key。系统提供了通过Message ID和Key查询消息的功能。

16.标签（Tag）

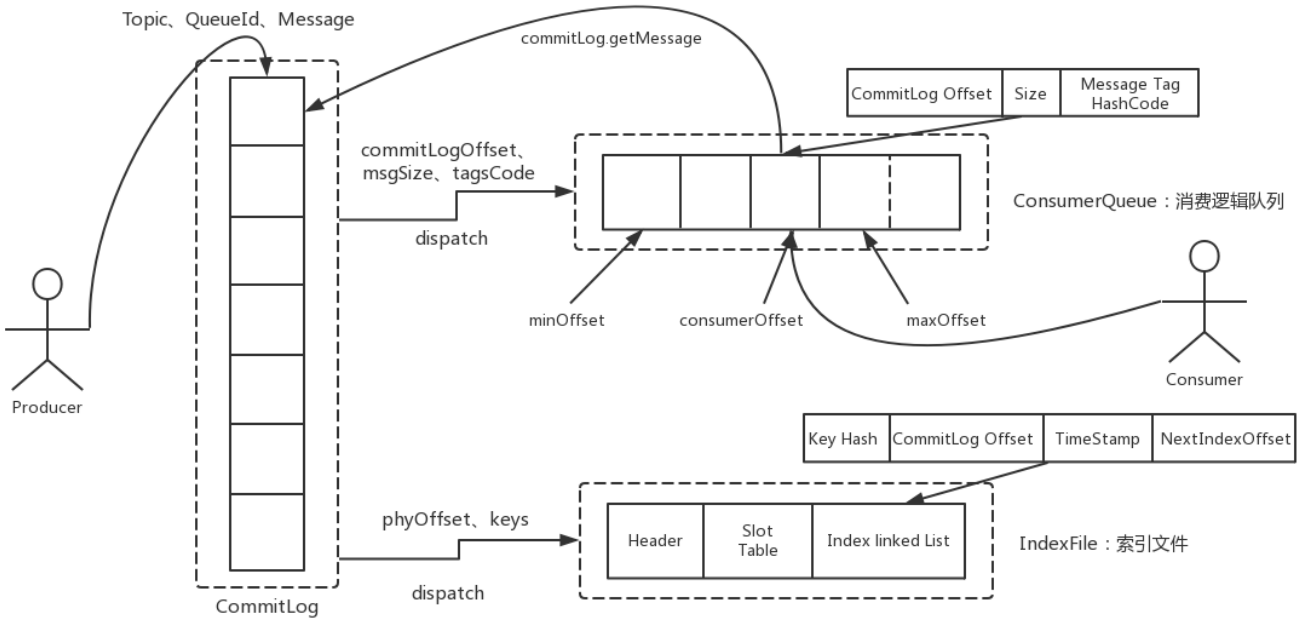
为消息设置的标志，用于同一主题下区分不同类型的消息。来自同一业务单元的消息，可以根据不同业务目的在同一主题下设置不同标签。标签能够有效地保持代码的清晰度和连贯性，并优化RocketMQ提供的查询系统。消费者可以根据Tag实现对不同子主题的不同消费逻辑，实现更好的扩展性。

九、消息存储机制



消息存储是RocketMQ中最为复杂和最为重要的一部分，本节将分别从RocketMQ的消息存储整体架构、PageCache与Mmap内存映射以及RocketMQ中两种不同的刷盘方式三方面来分别展开叙述。

1.消息存储整体架构



消息存储架构图中主要有下面三个跟消息存储相关的文件构成。

- CommitLog

消息主体以及元数据的存储主体，存储Producer端写入的消息主体内容,消息内容不是定长的。单个文件大小默认1G ，文件名长度为20位，左边补零，剩余为起始偏移量，比如00000000000000000000代表了第一个文件，起始偏移量为0，文件大小为1G=1073741824；当第一个文件写满了，第二个文件为00000000001073741824，起始偏移量为1073741824，以此类推。消息主要是顺序写入日志文件，当文件满了，写入下一个文件；

- ConsumeQueue

消息消费队列，引入的目的主要是提高消息消费的性能，由于RocketMQ是基于主题topic的订阅模式，消息消费是针对主题进行的，如果要遍历commitlog文件中根据topic检索消息是非常低效的。Consumer即可根据ConsumeQueue来查找待消费的消息。其中，ConsumeQueue（逻辑消费队列）作为消费消息的索引，保存了指定Topic下的队列消息在CommitLog中的起始物理偏移量offset，消息大小size和消息Tag的HashCode值。consumequeue文件可以看成是基于topic的commitlog索引文

件，故consumequeue文件夹的组织方式如下：topic/queue/file三层组织结构，具体存储路径为：$HOME/store/consumequeue/{topic}/{queueId}/{fileName}。同样consumequeue文件采取定长设计，每一个条目共20个字节，分别为8字节的commitlog物理偏移量、4字节的消息长度、8字节tag hashcode，单个文件由30W个条目组成，可以像数组一样随机访问每一个条目，每个ConsumeQueue文件大小约5.72M；

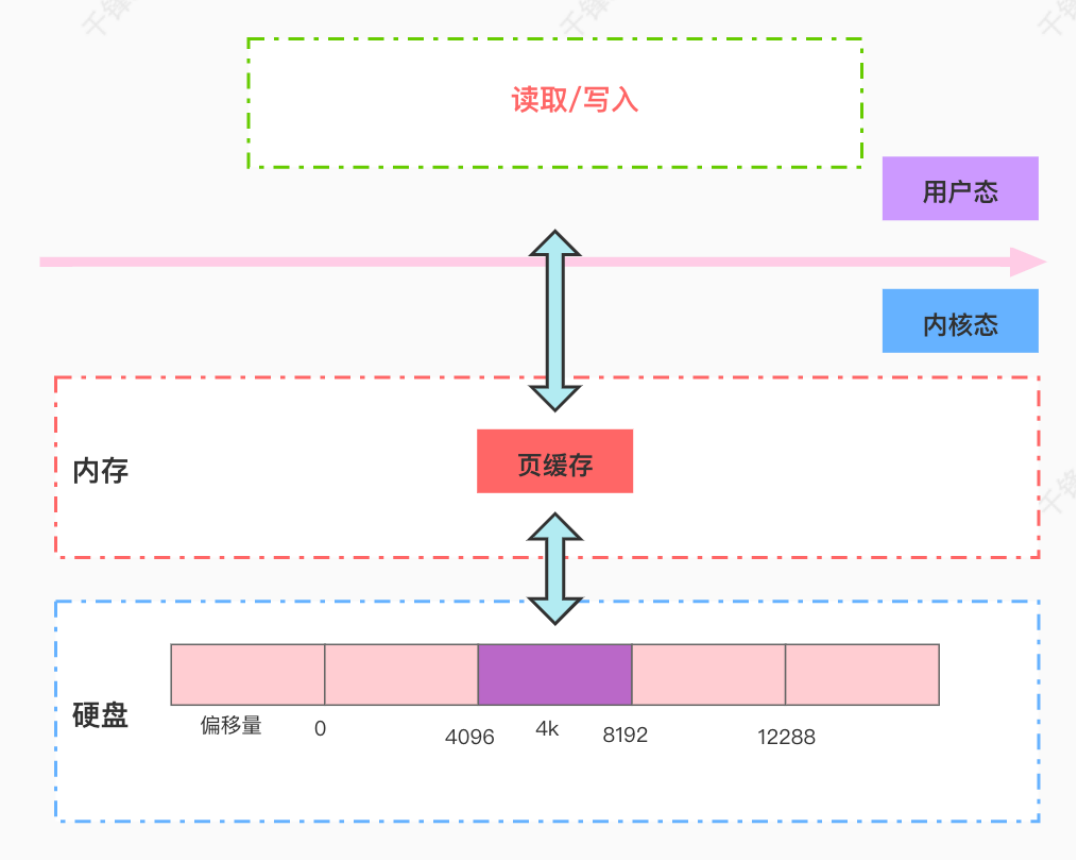
- IndexFile

IndexFile（索引文件）提供了一种可以通过key或时间区间来查询消息的方法。Index文件的存储位置是：$HOME \store\index${fileName}，文件名fileName是以创建时的时间戳命名的，固定的单个IndexFile文件大小约为400M，一个IndexFile可以保存 2000W个索引，IndexFile的底层存储设计为在文件系统中实现HashMap结构，故rocketmq的索引文件其底层实现为hash索引。

在上面的RocketMQ的消息存储整体架构图中可以看出，RocketMQ采用的是混合型的存储结构，即为Broker单个实例下所有的队列共用一个日志数据文件（即为CommitLog）来存储。RocketMQ的混合型存储结构(多个Topic的消息实体内容都存储于一个CommitLog中)针对Producer和Consumer分别采用了数据和索引部分相分离的存储结构，Producer发送消息至Broker端，然后Broker端使用同步或者异步的方式对消息刷盘持久化，保存至CommitLog中。只要消息被刷盘持久化至磁盘文件CommitLog中，那么Producer发送的消息就不会丢失。正因为如此，Consumer也就肯定有机会去消费这条消息。当无法拉取到消息后，可以等下一次消息拉取，同时服务端也支持长轮询模式，如果一个消息拉取请求未拉取到消息，Broker允许等待30s的时间，只要这段时间内有新消息到达，将直接返回给消费端。这里，RocketMQ的具体做法是，使用Broker端的后台服务线程—ReputMessageService不停地分发请求并异步构建ConsumeQueue（逻辑消费队列）和IndexFile（索引文件）数据。

2.页缓存与内存映射

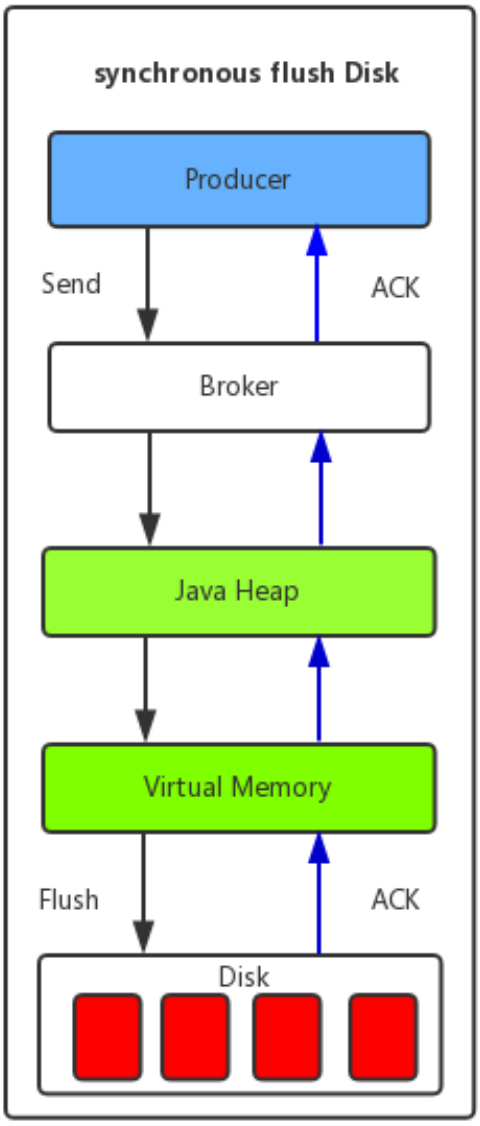
页缓存（PageCache)是OS对文件的缓存，用于加速对文件的读写。一般来说，程序对文件进行顺序读写的速度几乎接近于内存的读写速度，主要原因就是由于OS使用PageCache机制对读写访问操作进行了性能优化，将一部分的内存用作PageCache。对于数据的写入，OS会先写入至Cache内，随后通过异步的方式由pdflush内核线程将Cache内的数据刷盘至物理磁盘上。对于数据的读取，如果一次读取文件时出现未命中PageCache的情况，OS从物理磁盘上访问读取文件的同时，会顺序对其他相邻块的数据文件进行预读取。

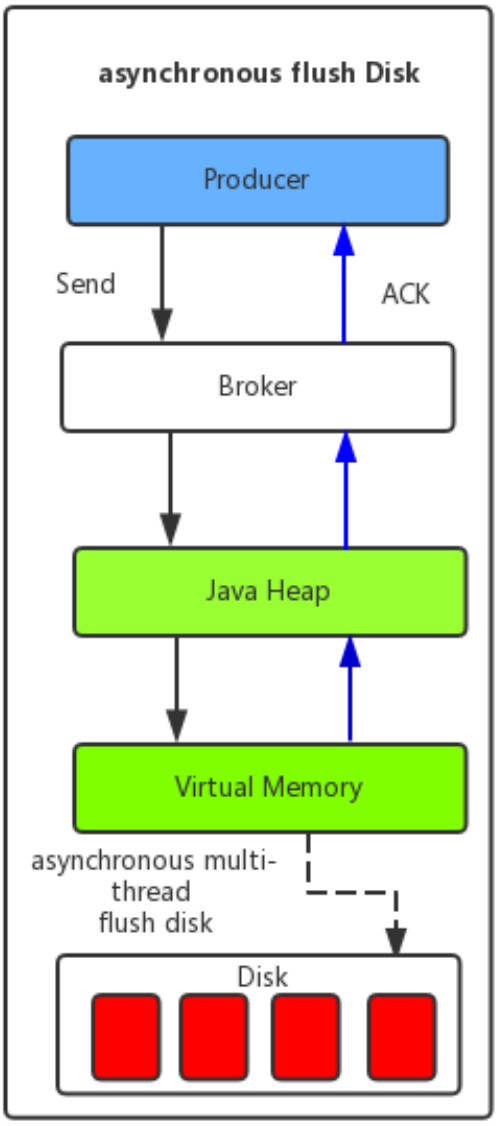


在RocketMQ中，ConsumeQueue逻辑消费队列存储的数据较少，并且是顺序读取，在page cache机制的预读取作用下，Consume Queue文件的读性能几乎接近读内存，即使在有消息堆积情况下也不会影响性能。而对于CommitLog消息存储的日志数据文件来说，读取消息内容时候会产生较多的随机访问读取，严重影响性能。如果选择合适的系统IO调度算法，比如设置调度算法为“Deadline”（此时块存储采用SSD的话），随机读的性能也会有所提升。

另外，RocketMQ主要通过MappedByteBuffer对文件进行读写操作。其中，利用了NIO中的FileChannel模型将磁盘上的物理文件直接映射到用户态的内存地址中（这种Mmap的方式减少了传统IO将磁盘文件数据在操作系统内核地址空间的缓冲区和用户应用程序地址空间的缓冲区之间来回进行拷贝的性能开销），将对文件的操作转化为直接对内存地址进行操作，从而极大地提高了文件的读写效率（正因为需要使用内存映射机制，故RocketMQ的文件存储都使用定长结构来存储，方便一次将整个文件映射至内存）。

3.消息刷盘





- 同步刷盘

如上图所示，只有在消息真正持久化至磁盘后RocketMQ的Broker端才会真正返回给Producer端一个成功的ACK响应。同步刷盘对MQ消息可靠性来说是一种不错的保障，但是性能上会有较大影响，一般适用于金融业务应用该模式较多。

- 异步刷盘

能够充分利用OS的PageCache的优势，只要消息写入PageCache即可将成功的ACK返回给Producer端。消息刷盘采用后台异步线程提交的方式进行，降低了读写延迟，提高了MQ的性能和吞吐量。

十、集群核心概念

1.消息主从复制

RocketMQ官方提供了三种集群搭建方式。

- 2主2从异步通信方式

使用异步方式进行主从之间的数据复制，吞吐量大，但可能会丢消息。

使用`conf/2m-2s-async`文件夹内的配置文件做集群配置。

- 2主2从同步通信方式

使用同步方式进行主从之间的数据复制，保证消息安全投递，不会丢失，但影响吞吐量

使用`conf/2m-2s-sync`文件夹内的配置文件做集群配置。

- 2主无从方式

不存在复制消息，会存在单点故障，且读的性能没有前两种方式好。

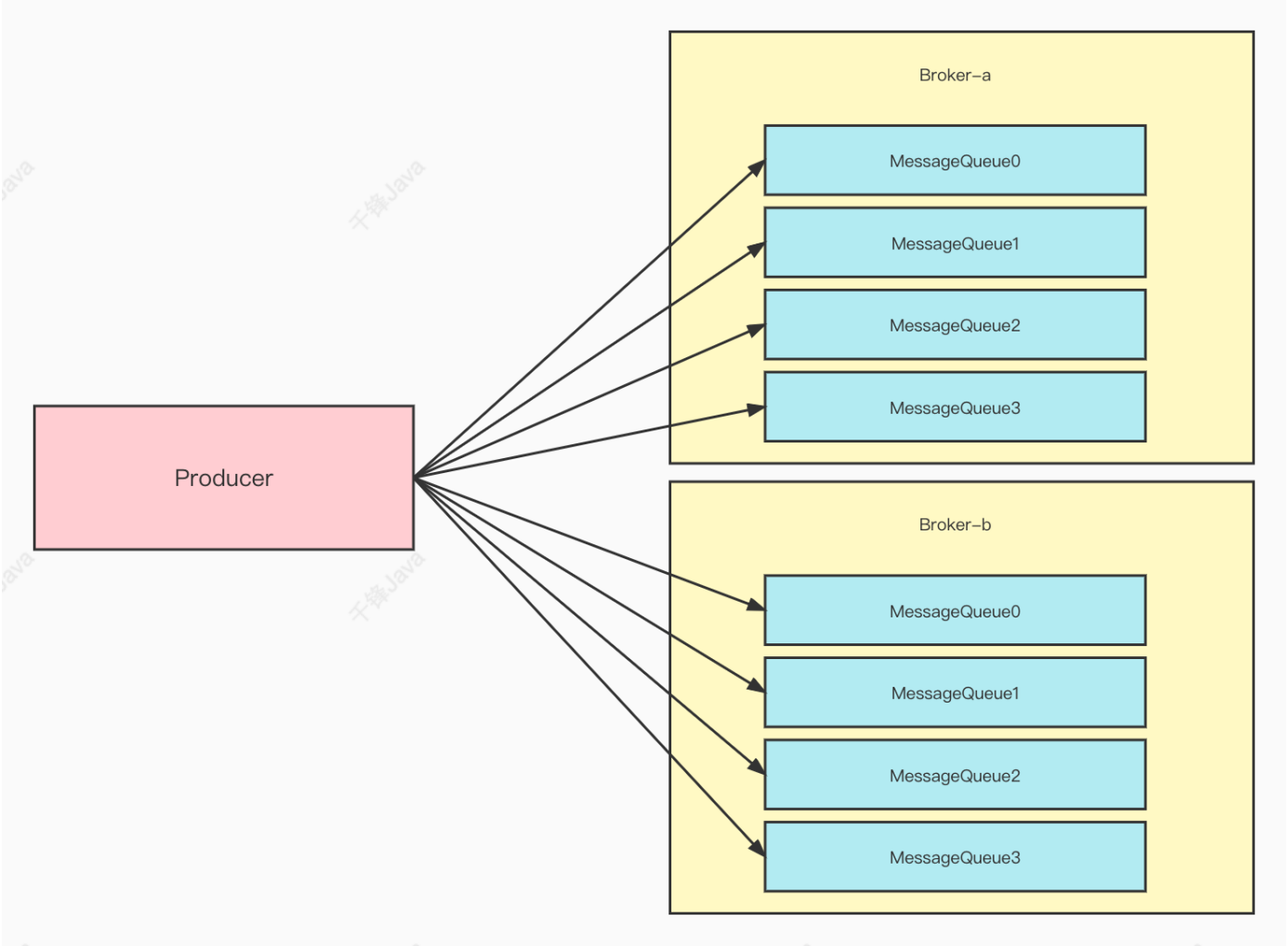
使用`conf/2m-noslave`文件夹内的配置文件做集群配置。

2.负载均衡

RocketMQ中的负载均衡都在Client端完成，具体来说的话，主要可以分为Producer端发送消息时候的负载均衡和Consumer端订阅消息的负载均衡。

- Producer的负载均衡

Producer端在发送消息的时候，会先根据Topic找到指定的TopicPublishInfo，在获取了TopicPublishInfo路由信息后，RocketMQ的客户端在默认方式下selectOneMessageQueue()方法会从TopicPublishInfo中的messageQueueList中选择一个队列（MessageQueue）进行发送消息。具体的容错策略均在MQFaultStrategy这个类中定义。这里有一个sendLatencyFaultEnable开关变量，如果开启，在随机递增取模的基础上，再过滤掉not available的Broker代理。所谓的"latencyFaultTolerance"，是指对之前失败的，按一定的时间做退避。例如，如果上次请求的latency超过550Lms，就退避3000Lms；超过1000L，就退避60000L；如果关闭，采用随机递增取模的方式选择一个队列（MessageQueue）来发送消息，latencyFaultTolerance机制是实现消息发送高可用的核心关键所在。



- Consumer的负载均衡

在RocketMQ中，Consumer端的两种消费模式（Push/Pull）都是基于拉模式来获取消息的，而在Push模式只是对pull模式的一种封装，其本质实现为消息拉取线程在从服务器拉取到一批消息后，然后提交到消息消费线程池后，又“马不停蹄”的继续向服务器再次尝试拉取消息。如果未拉取到消息，则延迟一下又继续拉取。在两种基于拉模式的消费方式（Push/Pull）中，均需要Consumer端在知道从Broker端的哪一个消息队列—队列中去获取消息。因此，有必要在Consumer端来做负载均衡，即Broker端中多个MessageQueue分配给同一个ConsumerGroup中的哪些Consumer消费。

Consumer的负责均衡可以通过consumer的api进行设置：

```
consumer.setAllocateMessageQueueStrategy(new
AllocateMessageQueueAveragelyByCircle());
```

AllocateMessageQueueStrategy接口的实现类表达了不同的负载均衡策略：

a.AllocateMachineRoomNearby :基于机房近侧优先级的代理分配策略。可以指定实际的分配策略。如果任何使用者在机房中活动，则部署在同一台机器中的代理的消息队列应仅分配给这些使用者。否则，这些消息队列可以与所有消费者共享，因为没有活着的消费者可以垄断它们

b.AllocateMessageQueueAveragely:平均哈希队列算法

c.AllocateMessageQueueAveragelyByCircle:循环平均哈希队列算法

d.AllocateMessageQueueByConfig:不分配，通过指定MessageQueue列表来消费

e.AllocateMessageQueueByMachineRoom:机房哈希队列算法，如支付宝逻辑机房

f.AllocateMessageQueueConsistentHash:一致哈希队列算法，带有虚拟节点的一致性哈希环。

注意，在MessageQueue和Consumer之间一旦发生对应关系的改变，就会触发rebalance，进行重新分配。

3.消息重试

非广播模式下，Consumer消费消息失败后，要提供一种重试机制，令消息再消费一次。Consumer消费消息失败通常可以认为有以下几种情况：

- 由于消息本身的原因，例如反序列化失败，消息数据本身无法处理（例如话费充值，当前消息的手机号被注销，无法充值）等。这种错误通常需要跳过这条消息，再消费其它消息，而这条失败的消息即使立刻重试消费，99%也不成功，所以最好提供一种定时重试机制，即过10秒后再重试。
- 由于依赖的下游应用服务不可用，例如db连接不可用，外系统网络不可达等。遇到这种错误，即使跳过当前失败的消息，消费其他消息同样也会报错。这种情况建议应用sleep 30s，再消费下一条消息，这样可以减轻Broker重试消息的压力。

在代码层面，如果消费者返回的是以下三种情况，则消息会重试消费

```
        consumer.registerMessageListener(new
MessageListenerConcurrently() {
            @Override
            public ConsumeConcurrentlyStatus
consumeMessage(List<MessageExt> msgs, ConsumeConcurrentlyContext
context) {
                for (MessageExt msg : msgs) {
                    System.out.println("收到的消息: "+msg);
                }
                return null;
                //return
ConsumeConcurrentlyStatus.RECONSUME_LATER;
                //抛出异常
            }
        });
```

消费者返回null，或者返回`ConsumeConcurrentlyStatus.RECONSUME_LATER`，或者抛出异常，都会触发重试。

关于重试次数

RocketMQ会为每个消费组都设置一个Topic名称为“%RETRY%+consumerGroup”的重试队列（这里需要注意的是，这个Topic的重试队列是针对消费组，而不是针对每个Topic设置的），用于暂时保存因为各种异常而导致Consumer端无法消费的消息。考虑到异常恢复起来需要一些时间，会为重试队列设置多个重试级别，每个重试级别都有与之对应的重新投递延时，重试次数越多投递延时就越大。RocketMQ对于重试消息的处理是先保存至Topic名称为“SCHEDULE_TOPIC_XXXX”的延迟队列中，后台定时任务按照对应的时间进行Delay后重新保存至“%RETRY%+consumerGroup”的重试队列中。

与延迟队列的设置相同，消息默认会重试16次，每次重试的时间间隔如下：

```
  10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m 30m 1h 2h
```

重试超过指定次数的消息，将会进入到死信队列中`%DLQ%my-consumer-group1`。

4.死信队列

死信队列用于处理无法被正常消费的消息。当一条消息初次消费失败，消息队列会自动进行消息重试；达到最大重试次数后，若消费依然失败，则表明消费者在正常情况下无法正确地消费该消息，此时，消息队列 不会立刻将消息丢弃，而是将其发送到该消费者对应的特殊队列中。

RocketMQ将这种正常情况下无法被消费的消息称为死信消息（Dead-Letter Message），将存储死信消息的特殊队列称为死信队列（Dead-Letter Queue）。在RocketMQ中，可以通过使用console控制台对死信队列中的消息进行重发来使得消费者实例再次进行消费。

死信队列具备以下特点：

- RocketMQ会自动为需要死信队列的ConsumerGroup创建死信队列。
- 死信队列与ConsumerGroup对应，死信队列中包含该ConsumerGroup所有相关topic的死信消息。
- 死信队列中消息的有效期与正常消息相同，默认48小时。
- 若要消费死信队列中的消息，需在控制台将死信队列的权限设置为6，即可读可写。

5.幂等消息

幂等性：多次操作造成的结果是一致的。对于非幂等的操作，幂等性如何保证?

1）在请求方式中的幂等性的体现

- get：多次get 结果是一致的
- post：添加，非幂等
- put：修改：幂等，根据id修改
- delete：根据id删除，幂等

对于非幂等的请求，我们在业务里要做幂等性保证。

2）在消息队列中的幂等性体现

消息队列中，很可能一条消息被冗余部署的多个消费者收到，对于非幂等的操作，比如用户的注册，就需要做幂等性保证，否则消息将会被重复消费。可以将情况概括为以下几种：

- 生产者重复发送：由于网络抖动，导致生产者没有收到broker的ack而再次重发消息，实际上broker收到了多条重复的消息，造成消息重复
- 消费者重复消费：由于网络抖动，消费者没有返回ack给broker，导致消费者重试消费。
- rebalance时的重复消费：由于网络抖动，在rebalance重分配时也可能出现消费者重复消费某条消息。

3）如何保证幂等性消费

- mysql 插入业务id作为主键，主键是唯一的，所以一次只能插入一条
- 使用redis或zk的分布式锁（主流的方案）

十一、RocketMQ最佳实践

1.保证消息顺序消费

1）为什么要保证消息有序

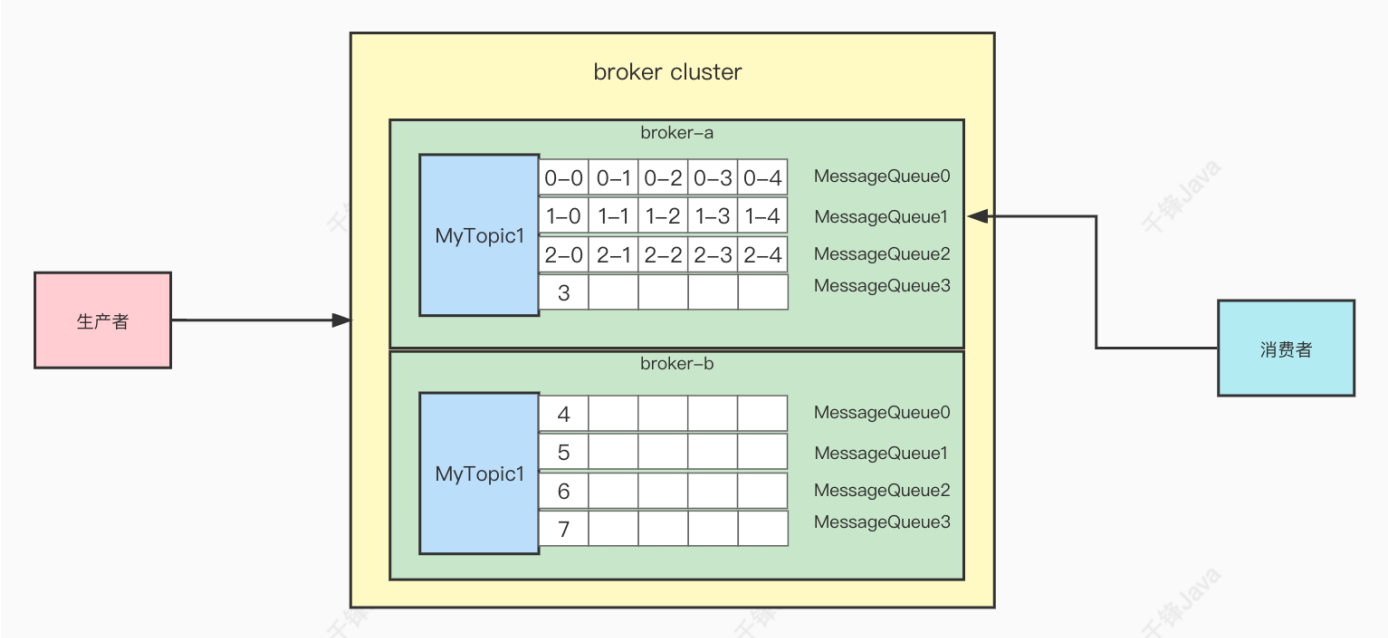
比如有这么一个物联网的应用场景，IOT中的设备在初始化时需要按顺序接收这样的消息：

- 设置设备名称
- 设置设备的网络
- 重启设备使配置生效

如果这个顺序颠倒了，可能就没有办法让设备的配置生效，因为只有重启设备才能让配置生效，但重启的消息却在设置设备消息之前被消费。

2）如何保证消息顺序消费

- 全局有序：消费的所有消息都严格按照发送消息的顺序进行消费
- 局部有序：消费的部分消息按照发送消息的顺序进行消费



2.快速处理积压消息

在rocketmq中，如果消费者消费速度过慢，而生产者生产消息的速度又远超于消费者消费消息的速度，那么就会造成大量消息积压在mq中。

1）如何查看消息积压的情况

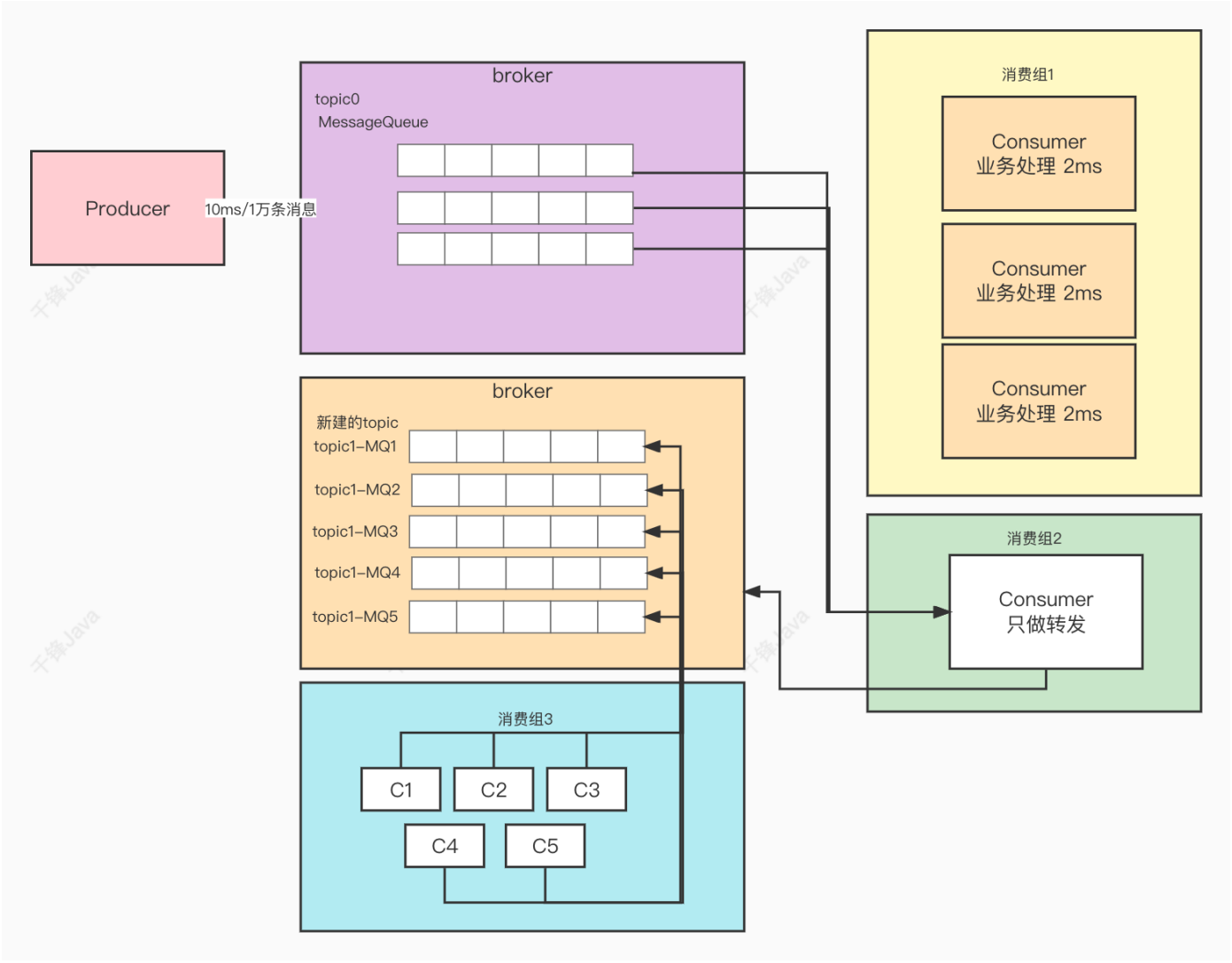
在console控制台中可以查看

主题	TopicTest	延迟	810	最后消费时间	2022-07-29 11:06:00

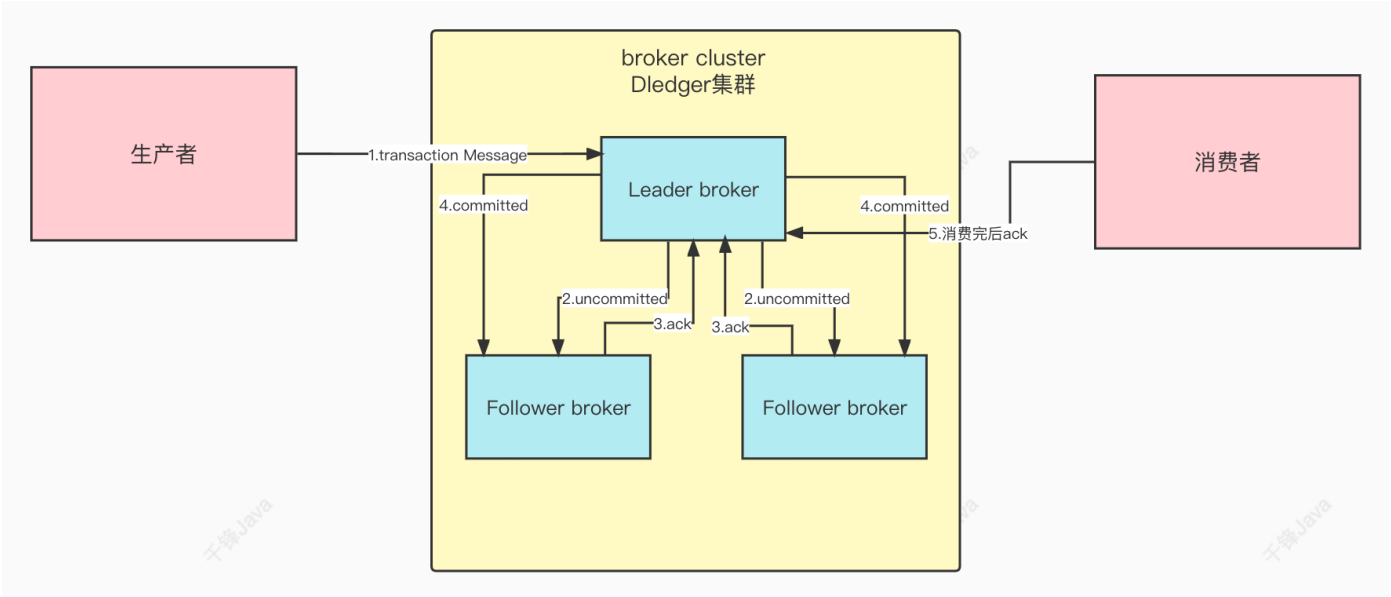
broker	queue	consumerClient	brokerOffset	consumerOffset	diffTotal	lastTimestamp
broker-a	0		241	125	116	2022-07-29 11:06:00
broker-a	1		243	125	118	2022-07-29 11:06:00
broker-a	2		220	125	95	2022-07-29 11:06:00
broker-a	3		222	125	97	2022-07-29 11:06:00
broker-b	0		221	125	96	2022-07-29 11:06:00
broker-b	1		220	125	95	2022-07-29 11:06:00
broker-b	2		222	125	97	2022-07-29 11:06:00
broker-b	3		221	125	96	2022-07-29 11:06:00

2）如何解决消息积压

- 在这个消费者中，使用多线程，充分利用机器的性能进行消费消息。
- 通过业务的架构设计，提升业务层面消费的性能。
- 创建一个消费者，该消费者在RocketMQ上另建一个主题，该消费者将poll下来的消息，不进行消费，直接转发到新建的主题上。新建的主题配上多个MessageQueue，多个MessageQueue再配上多个消费者。此时，新的主题的多个分区的多个消费者就开始一起消费了。


broker
topic0
MessageQueue
Producer
10ms/1万条消息
消费组1
Consumer
业务处理 2ms
Consumer
业务处理 2ms
Consumer
业务处理 2ms
broker
新建的topic
topic1-MQ1
topic1-MQ2
topic1-MQ3
topic1-MQ4
topic1-MQ5
消费组2
Consumer
只做转发
消费组3
C1
C2
C3
C4
C5

3.保证消息可靠性投递


broker cluster
Dledger集群
生产者
1.transaction Message
Leader broker
4.committed
4.committed
5.消费完后ack
消费者
2.uncommitted
3.ack
2.uncommitted
3.ack
Follower broker
Follower broker

保证消息可靠性投递，目的是消息不丢失，可以顺利抵达消费者并被消费。要想实现可靠性投递，需要完成以下几个部分。

1）生产者发送事务消息

参考第五章事务消息章节的内容

2）broker集群使用Dledger高可用集群

dledger集群的数据同步由两阶段完成

- 第一阶段：同步消息到follower，消息状态是uncommitted。follower在收到消息以后，返回一个ack给leader，leader自己也会返回ack给自己。leader在收到集群中的半数以上的ack后开始进入到第二阶段。
- 第二阶段：leader发送committed命令，集群中的所有的broker把消息写入到日志文件中，此时该消息才表示接收完毕。

3）保证消费者的同步消费

消费者使用同步的方式，在消费完后返回ack。

4）使用基于缓存中间件的MQ降级方案

当MQ整个服务不可用时，为了防止服务雪崩，消息可以暂存于缓存中间件中，比如redis。待MQ恢复后，将redis中的数据重新刷进MQ中。

千锋教育Java教研院 关注公众号【Java架构栈】下载所有课程代码课件及工具 让技术回归本该有的纯静!