

1. MQ介绍

##1.1 为什么要用MQ

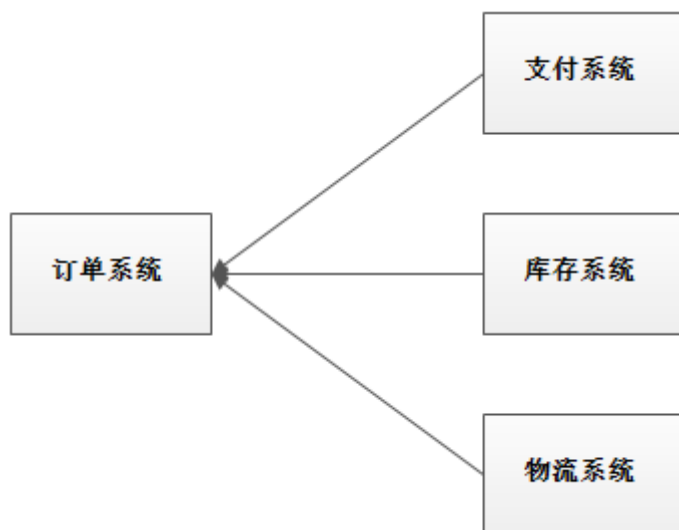
消息队列是一种“先进先出”的数据结构



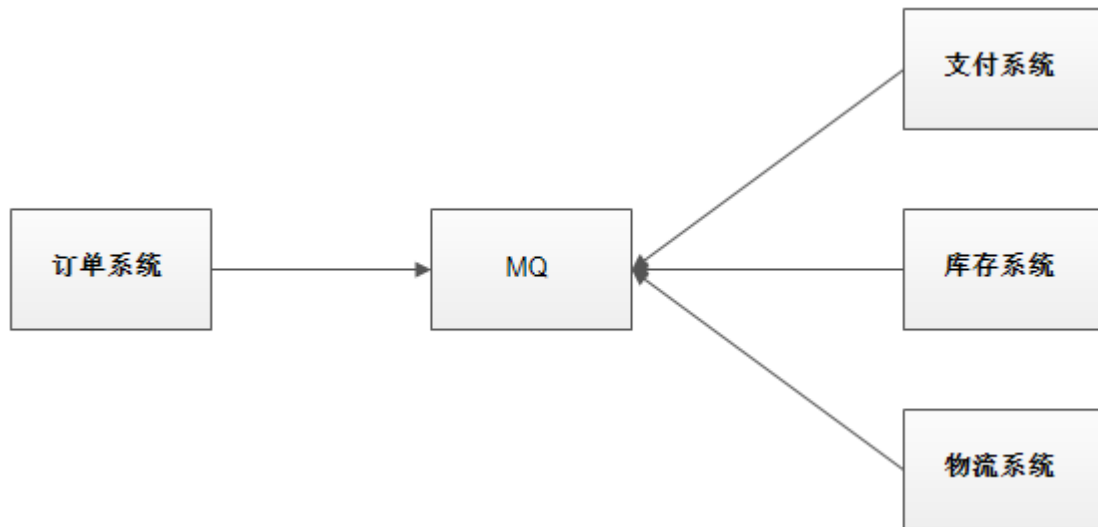
其应用场景主要包含以下3个方面

- 应用解耦

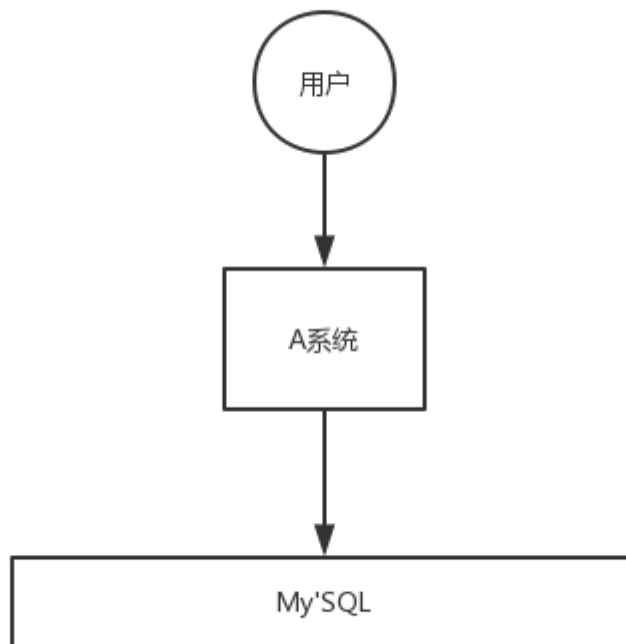
系统的耦合性越高，容错性就越低。以电商应用为例，用户创建订单后，如果耦合调用库存系统、物流系统、支付系统，任何一个子系统出了故障或者因为升级等原因暂时不可用，都会造成下单操作异常，影响用户使用体验。



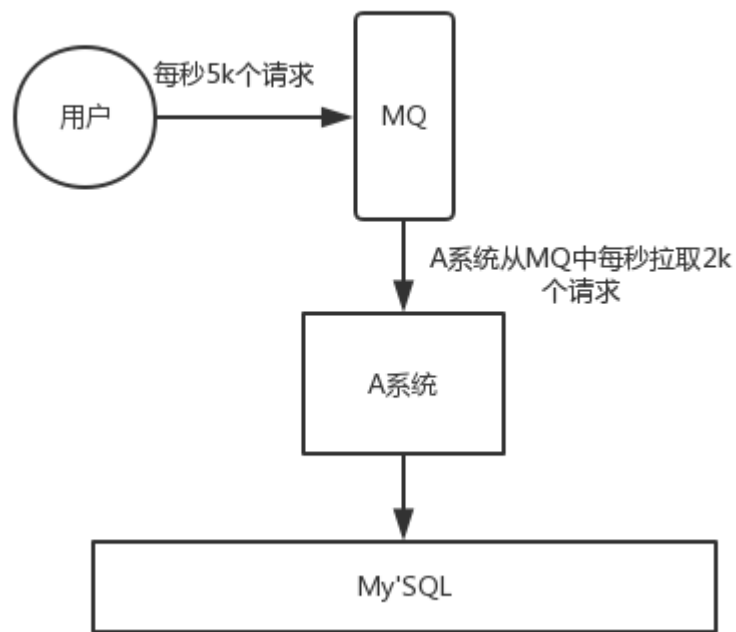
使用消息队列解耦合，系统的耦合性就会提高了。比如物流系统发生故障，需要几分钟才能来修复，在这段时间内，物流系统要处理的数据被缓存到消息队列中，用户的下单操作正常完成。当物流系统回复后，补充处理存在消息队列中的订单消息即可，终端系统感知不到物流系统发生过几分钟故障。



- 流量削峰



应用系统如果遇到系统请求流量的瞬间猛增，有可能会将系统压垮。有了消息队列可以将大量请求缓存起来，分散到很长一段时间处理，这样可以大大提到系统的稳定性和用户体验。

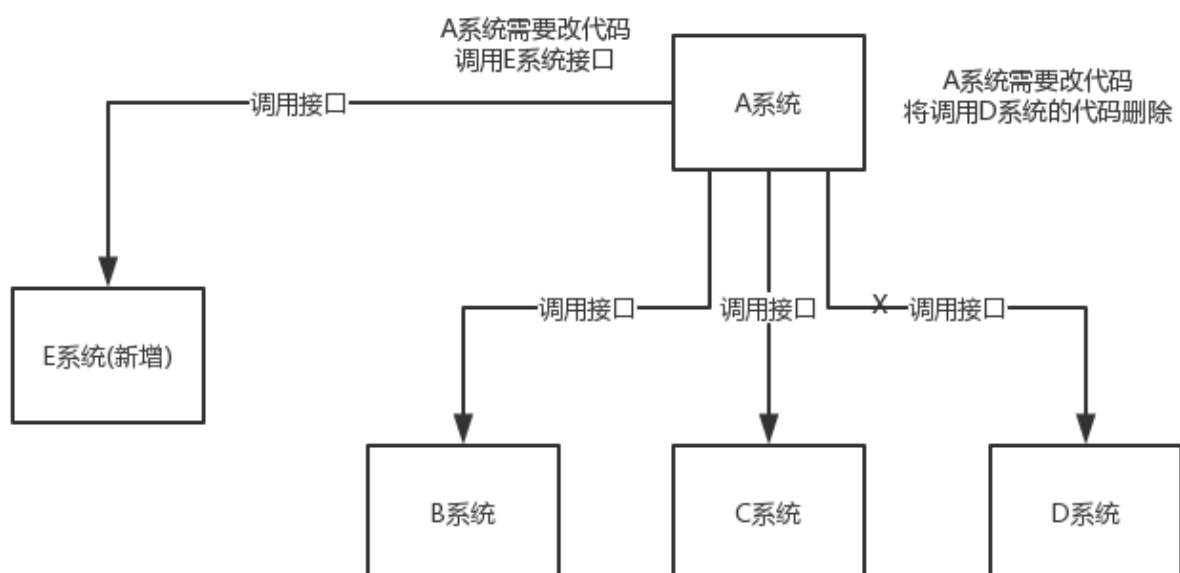


一般情况，为了保证系统的稳定性，如果系统负载超过阈值，就会阻止用户请求，这会影响用户体验，而如果使用消息队列将请求缓存起来，等待系统处理完毕后通知用户下单完毕，这样总下单体验要好。

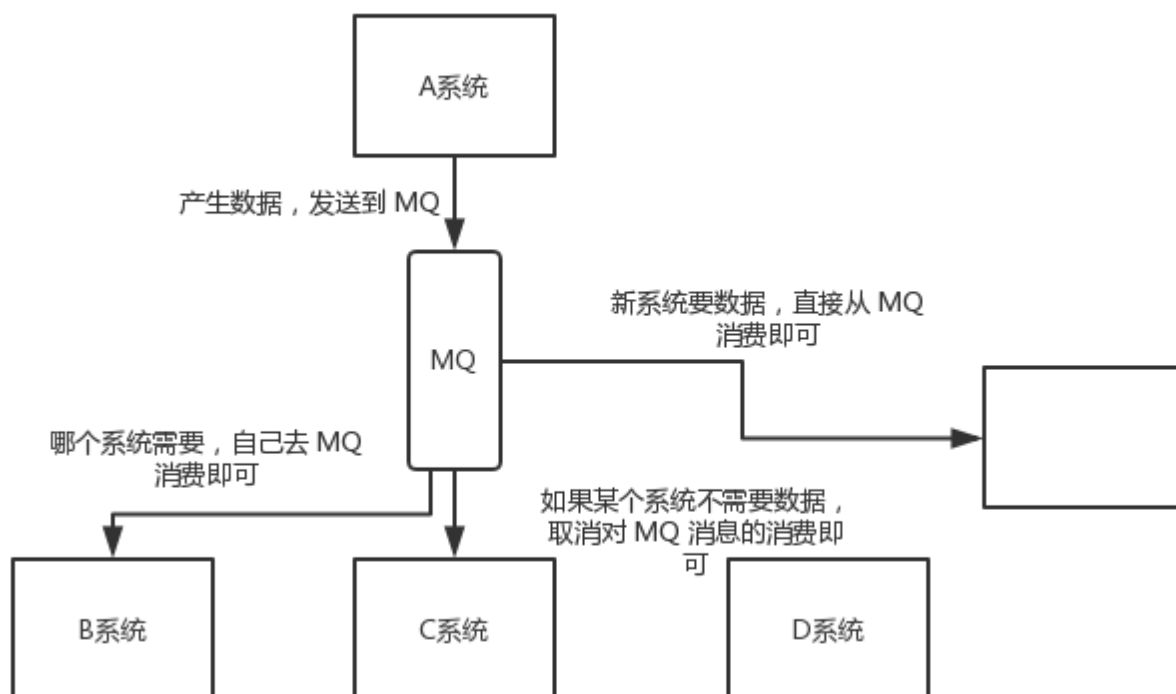
处于经济考量目的：

业务系统正常时段的QPS如果是1000，流量最高峰是10000，为了应对流量高峰配置高性能的服务器显然不划算，这时可以使用消息队列对峰值流量削峰

- 数据分发



通过消息队列可以让数据在多个系统更加之间进行流通。数据的产生方不需要关心谁来使用数据，只需要将数据发送到消息队列，数据使用方直接在消息队列中直接获取数据即可



1.2 MQ的优点和缺点

优点：解耦、削峰、数据分发

缺点包含以下几点：

- 系统可用性降低
系统引入的外部依赖越多，系统稳定性越差。一旦MQ宕机，就会对业务造成影响。
如何保证MQ的高可用？
- 系统复杂度提高
MQ的加入大大增加了系统的复杂度，以前系统间是同步的远程调用，现在是通过MQ进行异步调用。
如何保证消息没有被重复消费？怎么处理消息丢失情况？那么保证消息传递的顺序性？
- 一致性问题
A系统处理完业务，通过MQ给B、C、D三个系统发消息数据，如果B系统、C系统处理成功，D系统处理失败。
如何保证消息数据处理的一致性？

1.3 各种MQ产品的比较

常见的MQ产品包括Kafka、ActiveMQ、RabbitMQ、RocketMQ。

特性	ActiveMQ	RabbitMQ	RocketMQ	kafka
开发语言	java	erlang	java	scala
单机吞吐量	万级	万级	10万级	10万级
时效性	ms级	us级	ms级	ms级以内
可用性	高(主从架构)	高(主从架构)	非常高(分布式架构)	非常高(分布式架构)
功能特性	成熟的产品，在很多公司得到应用；有较多的文档；各种协议支持较好	基于erlang开发，所以并发能力很强，性能极其好，延时很低;管理界面较丰富	MQ功能比较完备，扩展性佳	只支持主要的MQ功能，像一些消息查询，消息回溯等功能没有提供，毕竟是为大数据准备的，在大数据领域应用广。

2. RocketMQ快速入门

RocketMQ是阿里巴巴2016年MQ中间件，使用Java语言开发，在阿里内部，RocketMQ承接了例如“双11”等高并发场景的消息流转，能够处理万亿级别的消息。

2.1 准备工作

2.1.1 下载RocketMQ

RocketMQ最新版本：4.5.1

[下载地址](#)

2.2.2 环境要求

- Linux64位系统
- JDK1.8(64位)
- 源码安装需要安装Maven 3.2.x

2.2 安装RocketMQ

2.2.1 安装步骤

本教程以二进制包方式安装

1. 解压安装包
2. 进入安装目录

2.2.2 目录介绍

- bin: 启动脚本, 包括shell脚本和CMD脚本
- conf: 实例配置文件, 包括broker配置文件、logback配置文件等
- lib: 依赖jar包, 包括Netty、commons-lang、FastJSON等

2.3 启动RocketMQ

1. 启动NameServer

```
# 1.启动NameServer
nohup sh bin/mqnamesrv &
# 2.查看启动日志
tail -f ~/logs/rocketmqlogs/namesrv.log
```

2. 启动Broker

```
# 1.启动Broker
nohup sh bin/mqbroker -n localhost:9876 &
# 2.查看启动日志
tail -f ~/logs/rocketmqlogs/broker.log
```

- 问题描述:

RocketMQ默认的虚拟机内存较大, 启动Broker如果因为内存不足失败, 需要编辑如下两个配置文件, 修改JVM内存大小

```
# 编辑runbroker.sh和runserver.sh修改默认JVM大小
vi runbroker.sh
vi runserver.sh
```

- 参考设置:

```
JAVA_OPT="{JAVA_OPT} -server -Xms256m -Xmx256m -Xmn128m -XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=320m"
```

2.4 测试RocketMQ

2.4.1 发送消息

```
# 1.设置环境变量
export NAMESRV_ADDR=localhost:9876
# 2.使用安装包的Demo发送消息
sh bin/tools.sh org.apache.rocketmq.example.quickstart.Producer
```

2.4.2 接收消息

```
# 1.设置环境变量
export NAMESRV_ADDR=localhost:9876
# 2.接收消息
sh bin/tools.sh org.apache.rocketmq.example.quickstart.Consumer
```

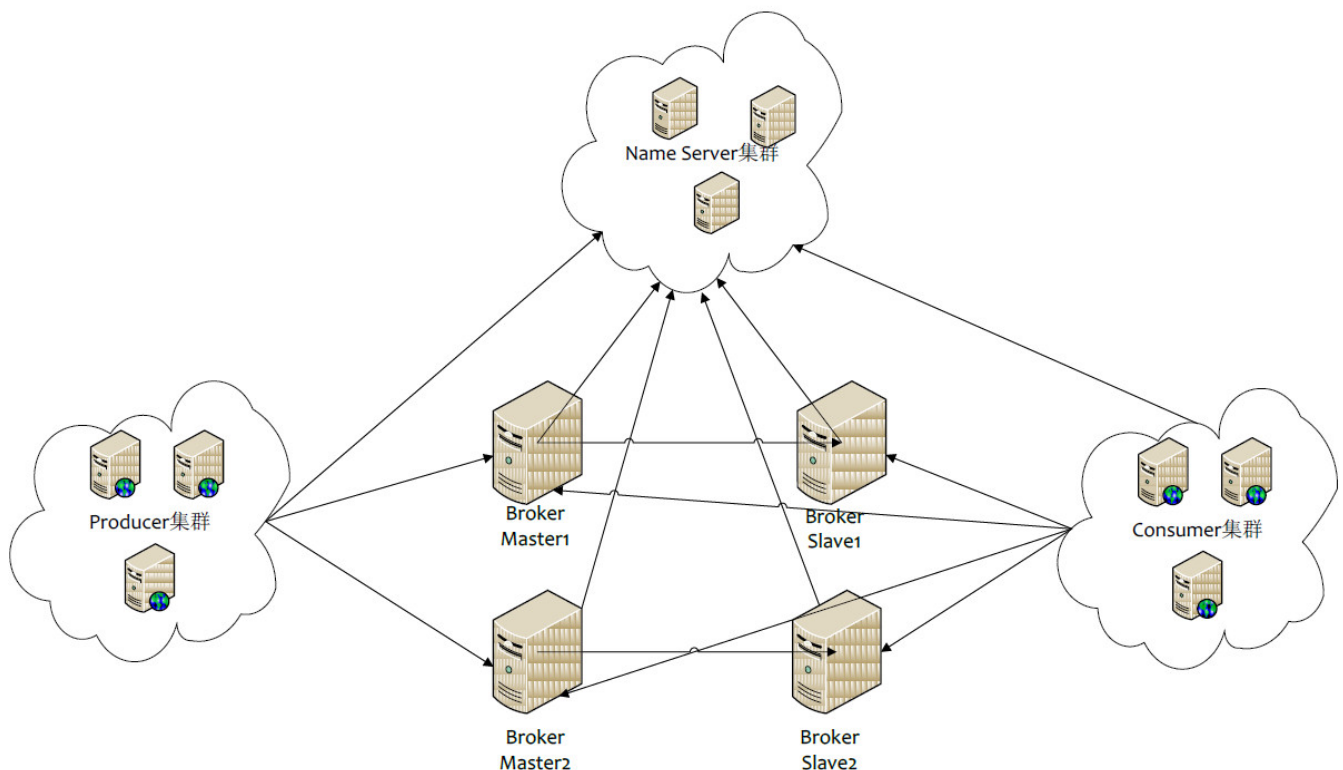
2.5 关闭RocketMQ

```
# 1.关闭NameServer
sh bin/mqshutdown namesrv
# 2.关闭Broker
sh bin/mqshutdown broker
```

3. RocketMQ集群搭建

3.1 各角色介绍

- Producer：消息的发送者；举例：发信者
- Consumer：消息接收者；举例：收信者
- Broker：暂存和传输消息；举例：邮局
- NameServer：管理Broker；举例：各个邮局的管理机构
- Topic：区分消息的种类；一个发送者可以发送消息给一个或者多个Topic；一个消息的接收者可以订阅一个或者多个Topic消息
- Message Queue：相当于是Topic的分区；用于并行发送和接收消息



3.2 集群搭建方式

3.2.1 集群特点

- 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。
- 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，订阅规则由Broker配置决定。

3.2.3 集群模式

1) 单Master模式

这种方式风险较大，一旦Broker重启或者宕机时，会导致整个服务不可用。不建议线上环境使用,可以用于本地测试。

2) 多Master模式

一个集群无Slave，全是Master，例如2个Master或者3个Master，这种模式的优缺点如下：

- 优点：配置简单，单个Master宕机或重启维护对应用无影响，在磁盘配置为RAID10时，即使机器宕机不可恢复情况下，由于RAID10磁盘非常可靠，消息也不会丢（异步刷盘丢失少量消息，同步刷盘一条不丢），性能最高；
- 缺点：单台机器宕机期间，这台机器上未被消费的消息在机器恢复之前不可订阅，消息实时性会受到影响。

3) 多Master多Slave模式（异步）

每个Master配置一个Slave，有多对Master-Slave，HA采用异步复制方式，主备有短暂消息延迟（毫秒级），这种模式的优缺点如下：

- 优点：即使磁盘损坏，消息丢失的非常少，且消息实时性不会受影响，同时Master宕机后，消费者仍然可以从Slave消费，而且此过程对应用透明，不需要人工干预，性能同多Master模式几乎一样；
- 缺点：Master宕机，磁盘损坏情况下会丢失少量消息。

4) 多Master多Slave模式（同步）

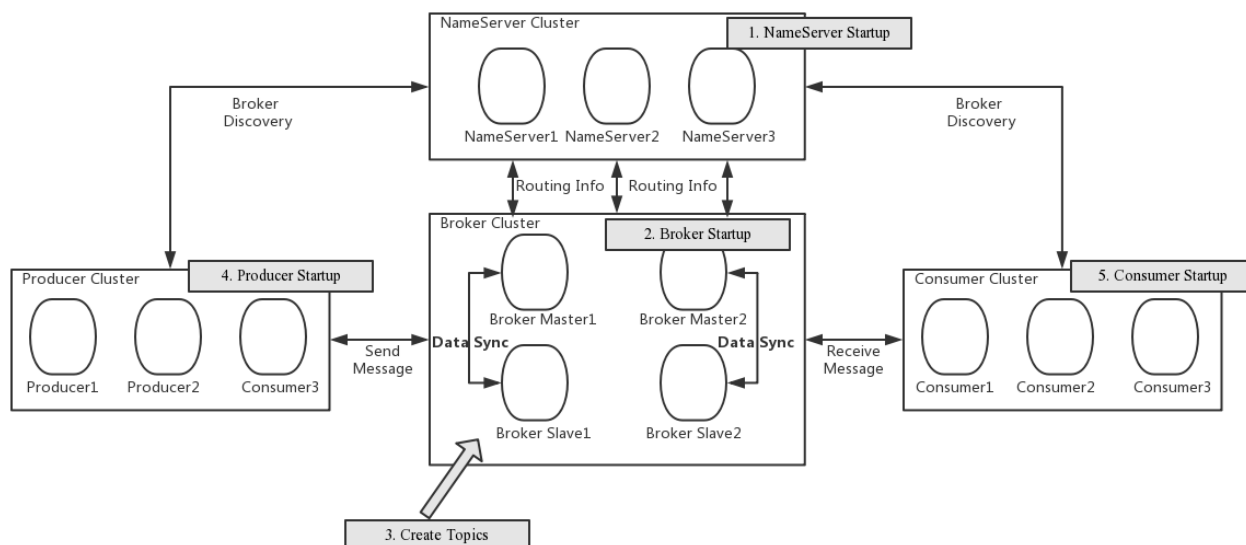
每个Master配置一个Slave，有多对Master-Slave，HA采用同步双写方式，即只有主备都写成功，才向应用返回成功，这种模式的优缺点如下：

- 优点：数据与服务都无单点故障，Master宕机情况下，消息无延迟，服务可用性与数据可用性都非常高；
- 缺点：性能比异步复制模式略低（大约低10%左右），发送单个消息的RT会略高，且目前版本在主节点宕机后，备机不能自动切换为主机。

3.3 双主双从集群搭建

3.3.1 总体架构

消息高可用采用2m-2s（同步双写）方式



3.3.2 集群工作流程

1. 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。
2. 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。
3. 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。
4. 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。
5. 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。

3.3.3 服务器环境

序号	IP	角色	架构模式
1	192.168.25.135	nameserver、brokerserver	Master1、Slave2
2	192.168.25.138	nameserver、brokerserver	Master2、Slave1

3.3.4 Host添加信息

```
vim /etc/hosts
```

配置如下:

```
# nameserver
192.168.25.135 rocketmq-nameserver1
192.168.25.138 rocketmq-nameserver2
# broker
192.168.25.135 rocketmq-master1
192.168.25.135 rocketmq-slave2
192.168.25.138 rocketmq-master2
192.168.25.138 rocketmq-slave1
```

配置完成后, 重启网卡

```
systemctl restart network
```

3.3.5 防火墙配置

宿主机需要远程访问虚拟机的rocketmq服务和web服务，需要开放相关的端口号，简单粗暴的方式是直接关闭防火墙

```
# 关闭防火墙
systemctl stop firewalld.service
# 查看防火墙的状态
firewall-cmd --state
# 禁止firewall开机启动
systemctl disable firewalld.service
```

或者为了安全，只开放特定的端口号，RocketMQ默认使用3个端口：9876、10911、11011。如果防火墙没有关闭的话，那么防火墙就必须开放这些端口：

- nameserver 默认使用 9876 端口
- master 默认使用 10911 端口
- slave 默认使用11011 端口

执行以下命令：

```
# 开放name server默认端口
firewall-cmd --remove-port=9876/tcp --permanent
# 开放master默认端口
firewall-cmd --remove-port=10911/tcp --permanent
# 开放slave默认端口（当前集群模式可不开启）
firewall-cmd --remove-port=11011/tcp --permanent
# 重启防火墙
firewall-cmd --reload
```

3.3.6 环境变量配置

```
vim /etc/profile
```

在profile文件的末尾加入如下命令

```
#set rocketmq
ROCKETMQ_HOME=/usr/local/rocketmq/rocketmq-all-4.4.0-bin-release
PATH=$PATH:$ROCKETMQ_HOME/bin
export ROCKETMQ_HOME PATH
```

输入:wq! 保存并退出，并使得配置立刻生效：

```
source /etc/profile
```

3.3.7 创建消息存储路径

```
mkdir /usr/local/rocketmq/store
mkdir /usr/local/rocketmq/store/commitlog
mkdir /usr/local/rocketmq/store/consumequeue
mkdir /usr/local/rocketmq/store/index
```

3.3.8 broker配置文件

1) master1

服务器：192.168.25.135

```
vi /usr/soft/rocketmq/conf/2m-2s-sync/broker-a.properties
```

修改配置如下：

```
#所属集群名字
brokerClusterName=rocketmq-cluster
#broker名字，注意此处不同的配置文件填写的不一样
brokerName=broker-a
#0 表示 Master, >0 表示 Slave
brokerId=0
#nameServer地址，分号分割
namesrvAddr=rocketmq-nameserver1:9876;rocketmq-nameserver2:9876
#在发送消息时，自动创建服务器不存在的topic，默认创建的队列数
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
#删除文件时间点，默认凌晨 4点
deleteWhen=04
#文件保留时间，默认 48 小时
fileReservedTime=120
#commitLog每个文件的大小默认1G
mappedFileSizeCommitLog=1073741824
#ConsumeQueue每个文件默认存30w条，根据业务情况调整
mappedFileSizeConsumeQueue=300000
#destroyMappedFileIntervalForcibly=120000
#redelateHangedFileInterval=120000
#检测物理文件磁盘空间
diskMaxUsedSpaceRatio=88
#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
#abort 文件存储路径
abortFile=/usr/local/rocketmq/store/abort
#限制的消息大小
maxMessageSize=65536
#flushCommitLogLeastPages=4
#flushConsumeQueueLeastPages=2
#flushCommitLogThoroughInterval=10000
#flushConsumeQueueThoroughInterval=60000
#Broker 的角色
#- ASYNC_MASTER 异步复制Master
#- SYNC_MASTER 同步双写Master
#- SLAVE
brokerRole=SYNC_MASTER
#刷盘方式
#- ASYNC_FLUSH 异步刷盘
#- SYNC_FLUSH 同步刷盘
flushDiskType=SYNC_FLUSH
#checkTransactionMessageEnable=false
```

```
#发消息线程池数量
#sendMessageThreadPoolNums=128
#拉消息线程池数量
#pullMessageThreadPoolNums=128
```

2) slave2

服务器: 192.168.25.135

```
vi /usr/soft/rocketmq/conf/2m-2s-sync/broker-b-s.properties
```

修改配置如下:

```
#所属集群名字
brokerClusterName=rocketmq-cluster
#broker名字, 注意此处不同的配置文件填写的不一样
brokerName=broker-b
#0 表示 Master, >0 表示 Slave
brokerId=1
#nameServer地址, 分号分割
namesrvAddr=rocketmq-nameserver1:9876;rocketmq-nameserver2:9876
#在发送消息时, 自动创建服务器不存在的topic, 默认创建的队列数
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=11011
#删除文件时间点, 默认凌晨 4点
deleteWhen=04
#文件保留时间, 默认 48 小时
fileReservedTime=120
#commitLog每个文件的大小默认1G
mappedFileSizeCommitLog=1073741824
#ConsumeQueue每个文件默认存30w条, 根据业务情况调整
mappedFileSizeConsumeQueue=300000
#destroyMappedFileIntervalForcibly=120000
#redeleteHangedFileInterval=120000
#检测物理文件磁盘空间
diskMaxUsedSpaceRatio=88
#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
#abort 文件存储路径
abortFile=/usr/local/rocketmq/store/abort
#限制的消息大小
maxMessageSize=65536
#flushCommitLogLeastPages=4
#flushConsumeQueueLeastPages=2
#flushCommitLogThoroughInterval=10000
#flushConsumeQueueThoroughInterval=60000
#Broker 的角色
```

```
#- ASYNC_MASTER 异步复制Master
#- SYNC_MASTER 同步双写Master
#- SLAVE
brokerRole=SLAVE
#刷盘方式
#- ASYNC_FLUSH 异步刷盘
#- SYNC_FLUSH 同步刷盘
flushDiskType=ASYNC_FLUSH
#checkTransactionMessageEnable=false
#发消息线程池数量
#sendMessageThreadPoolNums=128
#拉消息线程池数量
#pullThreadPoolNums=128
```

3) master2

服务器：192.168.25.138

```
vi /usr/soft/rocketmq/conf/2m-2s-sync/broker-b.properties
```

修改配置如下：

```
#所属集群名字
brokerClusterName=rocketmq-cluster
#broker名字，注意此处不同的配置文件填写的不一样
brokerName=broker-b
#0 表示 Master, >0 表示 Slave
brokerId=0
#nameServer地址，分号分割
namesrvAddr=rocketmq-nameserver1:9876;rocketmq-nameserver2:9876
#在发送消息时，自动创建服务器不存在的topic，默认创建的队列数
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
#删除文件时间点，默认凌晨 4点
deleteWhen=04
#文件保留时间，默认 48 小时
fileReservedTime=120
#commitLog每个文件的大小默认1G
mappedFileSizeCommitLog=1073741824
#ConsumeQueue每个文件默认存30W条，根据业务情况调整
mappedFileSizeConsumeQueue=300000
#destroyMappedFileIntervalForcibly=120000
#redeleteHangedFileInterval=120000
#检测物理文件磁盘空间
diskMaxUsedSpaceRatio=88
#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
```

```

#abort 文件存储路径
abortFile=/usr/local/rocketmq/store/abort
#限制的消息大小
maxMessageSize=65536
#flushCommitLogLeastPages=4
#flushConsumeQueueLeastPages=2
#flushCommitLogThoroughInterval=10000
#flushConsumeQueueThoroughInterval=60000
#Broker 的角色
#- ASYNC_MASTER 异步复制Master
#- SYNC_MASTER 同步双写Master
#- SLAVE
brokerRole=SYNC_MASTER
#刷盘方式
#- ASYNC_FLUSH 异步刷盘
#- SYNC_FLUSH 同步刷盘
flushDiskType=SYNC_FLUSH
#checkTransactionMessageEnable=false
#发消息线程池数量
#sendMessageThreadPoolNums=128
#拉消息线程池数量
#pullMessageThreadPoolNums=128

```

4) slave1

服务器：192.168.25.138

```
vi /usr/soft/rocketmq/conf/2m-2s-sync/broker-a-s.properties
```

修改配置如下：

```

#所属集群名字
brokerClusterName=rocketmq-cluster
#broker名字，注意此处不同的配置文件填写的不一样
brokerName=broker-a
#0 表示 Master, >0 表示 Slave
brokerId=1
#nameServer地址，分号分割
namesrvAddr=rocketmq-nameserver1:9876;rocketmq-nameserver2:9876
#在发送消息时，自动创建服务器不存在的topic，默认创建的队列数
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=11011
#删除文件时间点，默认凌晨 4点
deleteWhen=04
#文件保留时间，默认 48 小时
fileReservedTime=120
#commitLog每个文件的大小默认1G
mappedFileSizeCommitLog=1073741824
#ConsumeQueue每个文件默认存30W条，根据业务情况调整
mappedFileSizeConsumeQueue=300000
#destroyMappedFileIntervalForcibly=120000
#redelateHangedFileInterval=120000
#检测物理文件磁盘空间
diskMaxUsedSpaceRatio=88
#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
#abort 文件存储路径
abortFile=/usr/local/rocketmq/store/abort
#限制的消息大小
maxMessageSize=65536
#flushCommitLogLeastPages=4
#flushConsumeQueueLeastPages=2
#flushCommitLogThoroughInterval=10000
#flushConsumeQueueThoroughInterval=60000
#Broker 的角色
#- ASYNC_MASTER 异步复制Master
#- SYNC_MASTER 同步双写Master
#- SLAVE
brokerRole=SLAVE
#刷盘方式
#- ASYNC_FLUSH 异步刷盘
#- SYNC_FLUSH 同步刷盘
flushDiskType=ASYNC_FLUSH
#checkTransactionMessageEnable=false
#发消息线程池数量
#sendMessageThreadPoolNums=128
#拉消息线程池数量
#pullMessageThreadPoolNums=128

```

3.3.9 修改启动脚本文件

1) runbroker.sh

```
vi /usr/local/rocketmq/bin/runbroker.sh
```

需要根据内存大小进行适当的对JVM参数进行调整：

```

#=====
# 开发环境配置 JVM Configuration
JAVA_OPT="{JAVA_OPT} -server -Xms256m -Xmx256m -Xmn128m"

```

####2) runserver.sh

```
vim /usr/local/rocketmq/bin/runserver.sh
```

```

JAVA_OPT="{JAVA_OPT} -server -Xms256m -Xmx256m -Xmn128m -XX:MetaspaceSize=128m -
XX:MaxMetaspaceSize=320m"

```

3.3.10 服务启动

1) 启动NameServe集群

分别在192.168.25.135和192.168.25.138启动NameServer

```

cd /usr/local/rocketmq/bin
nohup sh mqnamesrv &

```

2) 启动Broker集群

- 在192.168.25.135上启动master1和slave2

master1:

```
cd /usr/local/rocketmq/bin
nohup sh mqbroker -c /usr/local/rocketmq/conf/2m-2s-syncbroker-a.properties &
```

slave2:

```
cd /usr/local/rocketmq/bin
nohup sh mqbroker -c /usr/local/rocketmq/conf/2m-2s-sync/broker-b-s.properties &
```

- 在192.168.25.138上启动master2和slave2

master2

```
cd /usr/local/rocketmq/bin
nohup sh mqbroker -c /usr/local/rocketmq/conf/2m-2s-sync/broker-b.properties &
```

slave1

```
cd /usr/local/rocketmq/bin
nohup sh mqbroker -c /usr/local/rocketmq/conf/2m-2s-sync/broker-a-s.properties &
```

3.3.11 查看进程状态

启动后通过JPS查看启动进程

```
[root@localhost bin]# jps
2374 NamesrvStartup
2839 BrokerStartup
2735 BrokerStartup
2879 Jps
```

3.3.12 查看日志

```
# 查看nameServer日志
tail -500f ~/logs/rocketmqlogs/namesrv.log
# 查看broker日志
tail -500f ~/logs/rocketmqlogs/broker.log
```

3.4 mqadmin管理工具

3.4.1 使用方式

进入RocketMQ安装位置，在bin目录下执行./mqadmin {command} {args}

###3.4.2 命令介绍

####1) Topic相关

名称	含义	命令选项	说明
----	----	------	----

updateTopic	创建更新Topic配置	-b	Broker 地址，表示 topic 所在 Broker，只支持单台Broker，地址为ip:port
		-c	cluster 名称，表示 topic 所在集群（集群可通过 clusterList 查询）
		-h-	打印帮助
		-n	NameServer服务地址，格式 ip:port
		-p	指定新topic的读写权限(W=2 R=4 WR=6)
		-r	可读队列数（默认为 8）
deleteTopic	删除Topic	-w	可写队列数（默认为 8）
		-t	topic 名称（名称只能使用字符 ^[a-zA-Z0-9_-]+\$ ）
		-c	cluster 名称，表示删除某集群下的某个 topic （集群 可通过 clusterList 查询）
		-h	打印帮助
topicList	查看 Topic 列表信息	-n	NameServer 服务地址，格式 ip:port
		-t	topic 名称（名称只能使用字符 ^[a-zA-Z0-9_-]+\$ ）
		-h	打印帮助

		-c	不配置-c只返回topic列表，增加-c返回clusterName, topic, consumerGroup信息，即topic的所属集群和订阅关系，没有参数
		-n	NameServer 服务地址，格式 ip:port
		-t	topic 名称
		-h	打印帮助
topicRoute	查看 Topic 路由信息	-n	NameServer 服务地址，格式 ip:port
		-t	topic 名称
		-h	打印帮助
topicStatus	查看 Topic 消息队列offset	-n	NameServer 服务地址，格式 ip:port
		-t	topic 名称
		-h	打印帮助
topicClusterList	查看 Topic 所在集群列表	-n	NameServer 服务地址，格式 ip:port
		-t	topic 名称
		-h	打印帮助
updateTopicPerm	更新 Topic 读写权限	-n	NameServer 服务地址，格式 ip:port
		-b	Broker 地址，表示 topic 所在 Broker，只支持单台Broker，地址为ip:port
		-p	指定新 topic 的读写权限(W=2 R=4 WR=6)

		-c	cluster 名称，表示 topic 所在集群（集群可通过 clusterList 查询），-b 优先，如果没有 -b，则对集群中所有 Broker 执行命令
		-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
updateOrderConf	从 NameServer 上创建、删除、获取特定命名空间的 kv 配置，目前还未启用	-t	topic，键
		-v	orderConf，值
		-m	method，可选 get、put、delete
		-t	topic 名称
		-h	打印帮助
allocateMQ	以平均负载算法计算消费者列表负载消息队列的负载结果	-n	NameServer 服务地址，格式 ip:port
		-i	ipList，用逗号分隔，计算这些 ip 去负载 Topic 的消息队列
		-h	打印帮助
statsAll	打印 Topic 订阅关系、TPS、积累量、24h 读写总量等信息	-n	NameServer 服务地址，格式 ip:port
		-a	是否只打印活跃 topic
		-t	指定 topic

####2) 集群相关

名称	含义	命令选项	说明
----	----	------	----

clusterList	查看集群信息，集群、BrokerName、BrokerId、TPS等信息	-m	打印更多信息 (增加打印出如下信息 #InTotalYest, #OutTotalYest, #InTotalToday, #OutTotalToday)
		-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
		-i	打印间隔，单位秒
		-a	amount，每次探测的总数，RT = 总时间 / amount
clusterRT	发送消息检测集群各Broker RT。消息发往 \${BrokerName} Topic。	-s	消息大小，单位B
		-c	探测哪个集群
		-p	是否打印格式化日志，以 分割，默认不打印
		-h	打印帮助
		-m	所属机房，打印使用
		-i	发送间隔，单位秒
		-n	NameServer 服务地址，格式 ip:port

####3) Broker相关

名称	含义	命令选项	说明
updateBrokerConfig	更新 Broker 配置文件，会修改Broker.conf	-b	Broker 地址，格式为ip:port
		-c	cluster 名称
		-k	key 值
		-v	value 值
		-h	打印帮助

		-n	NameServer 服务地址，格式ip:port
		-b	Broker 地址，地址为ip:port
brokerStatus	查看 Broker 统计信息、运行状态（你想要的信息几乎都在里面）	-h	打印帮助
		-n	NameServer 服务地址，格式ip:port
		-b	Broker 地址，地址为ip:port
		-t	请求超时时间
brokerConsumeStats	Broker中各个消费者的消费情况，按Message Queue维度返回Consume Offset, Broker Offset, Diff, Timestamp等信息	-l	diff阈值，超过阈值才打印
		-o	是否为顺序topic，一般为false
		-h	打印帮助
		-n	NameServer 服务地址，格式ip:port
		-b	Broker 地址，地址为ip:port
getBrokerConfig	获取Broker配置	-n	NameServer 服务地址，格式ip:port
		-b	Broker 地址，地址为ip:port
wipeWritePerm	从NameServer上清除Broker写权限	-n	NameServer 服务地址，格式ip:port
		-h	打印帮助
		-n	NameServer 服务地址，格式ip:port
cleanExpiredCQ	清理Broker上过期的Consume Queue，如果手动减少对列数可能产生过期队列	-h	打印帮助
		-b	Broker 地址，地址为ip:port
		-c	集群名称
cleanUnusedTopic	清理Broker上不使用的Topic，从内存中释放Topic的Consume Queue，如果手动删除Topic会产生不使用的Topic	-n	NameServer 服务地址，格式ip:port

		-h	打印帮助
		-b	Broker 地址，地址为ip:port
		-c	集群名称
		-n	NameServer 服务地址，格式ip:port
sendMsgStatus	向Broker发消息，返回发送状态和RT	-h	打印帮助
		-b	BrokerName，注意不同于Broker地址
		-s	消息大小，单位B
		-c	发送次数

####4) 消息相关

名称	含义	命令选项	说明
queryMsgById	根据offsetMsgId查询msg，如果使用开源控制台，应使用offsetMsgId，此命令还有其他参数，具体作用请阅读QueryMsgByIdSubCommand。	-h	msgId 打印帮助
		-n	NameServer 服务地址，格式ip:port
		-k	msgKey
		-t	Topic 名称
queryMsgByKey	根据消息 Key 查询消息	-h	打印帮助
		-n	NameServer 服务地址，格式ip:port
queryMsgByOffset	根据 Offset 查询消息		Broker 名称，（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)
		-b	
		-i	query 队列 id

		-o	offset 值
		-t	topic 名称
		-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
		-h	打印帮助
queryMsgByUniqueKey	根据msgId查询，msgId不同于offsetMsgId，区别详见常见运维问题。-g, -d配合使用，查到消息后尝试让特定的消费者消费消息并返回消费结果	-n	NameServer 服务地址，格式 ip:port
		-i	unique msg id
		-g	consumerGroup
		-d	clientId
		-t	topic名称
		-h	打印帮助
checkMsgSendRT	检测向topic发消息的RT，功能类似clusterRT	-n	NameServer 服务地址，格式 ip:port
		-t	topic名称
		-a	探测次数
		-s	消息大小
		-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
sendMessage	发送一条消息，可以根据配置发往特定Message Queue，或普通发送。	-t	topic名称
		-p	body，消息体
		-k	keys
		-c	tags
		-b	BrokerName
		-i	queueId
consumeMessage	消费消息。可以根据offset、开始&结束时间戳、消息队列消费消息，配置不同执行不同消费逻辑，详见ConsumeMessageCommand。	-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
		-t	topic名称
		-b	BrokerName
		-o	从offset开始消费
		-i	queueId
		-g	消费者分组
		-s	开始时间戳，格式详见-h
		-d	结束时间戳

		-c	消费多少条消息
		-h	打印帮助
		-n	NameServer 服务地址, 格式 ip:port
		-t	topic名称
printMsg	从Broker消费消息并打印, 可选时间段	-c	字符集, 例如UTF-8
		-s	subExpress, 过滤表达式
		-b	开始时间戳, 格式参见-h
		-e	结束时间戳
		-d	是否打印消息体
		-h	打印帮助
		-n	NameServer 服务地址, 格式 ip:port
		-t	topic名称
		-i	queueId
		-a	BrokerName
printMsgByQueue	类似printMsg, 但指定Message Queue	-c	字符集, 例如UTF-8
		-s	subExpress, 过滤表达式
		-b	开始时间戳, 格式参见-h
		-e	结束时间戳
		-p	是否打印消息
		-d	是否打印消息体
		-f	是否统计tag数量并打印
resetOffsetByTime	按时间戳重置offset, Broker和consumer都会重置	-h	打印帮助
		-n	NameServer 服务地址, 格式 ip:port
		-g	消费者分组
		-t	topic名称
		-s	重置为此时间戳对应的offset

是否强制重置，如果false，只支持回溯offset，如果true，不管时间戳对应offset与consumeOffset关系

-f

是否重置c++客户端offset

-c

5) 消费者、消费组相关

名称	含义	命令选项	说明
consumerProgress	查看订阅组消费状态，可以查看具体的client IP的消息积累量	-g	消费者所属组名
		-s	是否打印client IP
		-h	打印帮助
		-n	NameServer 服务地址，格式ip:port
consumerStatus	查看消费者状态，包括同一个分组中是否都是相同的订阅，分析Process Queue是否堆积，返回消费者jstack结果，内容较多，使用者参见ConsumerStatusSubCommand	-h	打印帮助
		-n	NameServer 服务地址，格式ip:port
		-g	consumer group
		-i	clientId
		-s	是否执行jstack
		-g	消费者所属组名
		-t	查询主题
getConsumerStatus	获取 Consumer 消费进度	-i	Consumer 客户端 ip
		-n	NameServer 服务地址，格式ip:port
updateSubGroup	更新或创建订阅关系	-h	打印帮助
		-n	NameServer 服务地址，格式ip:port

-h	打印帮助
-b	Broker地址
-c	集群名称
-g	消费者分组名称
-s	分组是否允许消费
-m	是否从最小offset开始消费
-d	是否是广播模式
-q	重试队列数量
-r	最大重试次数
-i	当slaveReadEnable开启时有效，且还未达到从slave消费时建议从哪个BrokerId消费，可以配置备机id，主动从备机消费
-w	如果Broker建议从slave消费，配置决定从哪个slave消费，配置BrokerId，例如1
-a	当消费者数量变化时是否通知其他消费者负载均衡
-n	NameServer 服务地址，格式ip:port

deleteSubGroup	从Broker删除订阅关系
cloneGroupOffset	在目标群组中使用源群组的offset

-h	打印帮助
-b	Broker地址
-c	集群名称
-g	消费者分组名称
-n	NameServer 服务地址，格式ip:port
-h	打印帮助
-s	源消费者组

-d	目标消费者组
-t	topic名称
-o	暂未使用

6) 连接相关

名称	含义	命令选项	说明
		-g	消费者所属组名
consumerConnec tion	查询 Consumer 的网络连接	-n	NameServer 服务地址，格式 ip:port
		-h	打印帮助
		-g	生产者所属组名
		-t	主题名称
producerConnec tion	查询 Producer 的网络连接	-n	NameServer 服务地址，格式 ip:port
		-h	打印帮助

7) NameServer相关

名称	含义	命令选项	说明
		-s	命名空间
		-k	key
		-v	value
updateKvConfig	更新NameServer的kv配置，目前还未使用	-n	NameServer 服务地址，格式 ip:port
		-h	打印帮助
		-s	命名空间
		-k	key
deleteKvConfig	删除NameServer的kv配置	-n	NameServer 服务地址，格式 ip:port
		-h	打印帮助
getNamesrvConfig	获取NameServer配置	-n	NameServer 服务地址，格式 ip:port
		-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
updateNamesrvConfig	修改NameServer配置	-h	打印帮助
		-k	key
		-v	value

8) 其他

名称	含义	命令选项	说明
startMonitoring	开启监控进程，监控消息误删、重试队列消息数等	-n -h	NameServer 服务地址，格式ip:port 打印帮助

3.4.3 注意事项

- 几乎所有命令都需要配置-n表示NameServer地址，格式为ip:port
- 几乎所有命令都可以通过-h获取帮助
- 如果既有Broker地址（-b）配置项又有clusterName（-c）配置项，则优先以Broker地址执行命令；如果不配置Broker地址，则对集群中所有主机执行命令

























3.5 集群监控平台搭建

3.5.1 概述

RocketMQ有一个对其扩展的开源项目[incubator-rocketmq-externals](https://github.com/apache/rocketmq-externals)，这个项目中有一个子模块叫rocketmq-console，这个便是管理控制台项目了，先将[incubator-rocketmq-externals](https://github.com/apache/rocketmq-externals)拉到本地，因为我们需要自己对rocketmq-console进行编译打包运行。

名称



-  dev
-  rocketmq-client-dotnet
-  rocketmq-client-php
-  rocketmq-cloudevents-binding
-  rocketmq-connect-activemq
-  rocketmq-connect-runtime
-  rocketmq-console
-  rocketmq-docker
-  rocketmq-flink
-  rocketmq-flume
-  rocketmq-hbase
-  rocketmq-iot-bridge
-  rocketmq-jms
-  rocketmq-mysql
-  rocketmq-php
-  rocketmq-prometheus-exporter
-  rocketmq-redis
-  rocketmq-replicator
-  rocketmq-sentinel
-  rocketmq-serializer
-  rocketmq-spark
-  .gitignore
-  .travis.yml
-  README.md

3.5.2 下载并编译打包

```
git clone https://github.com/apache/rocketmq-externals
cd rocketmq-console
mvn clean package -Dmaven.test.skip=true
```

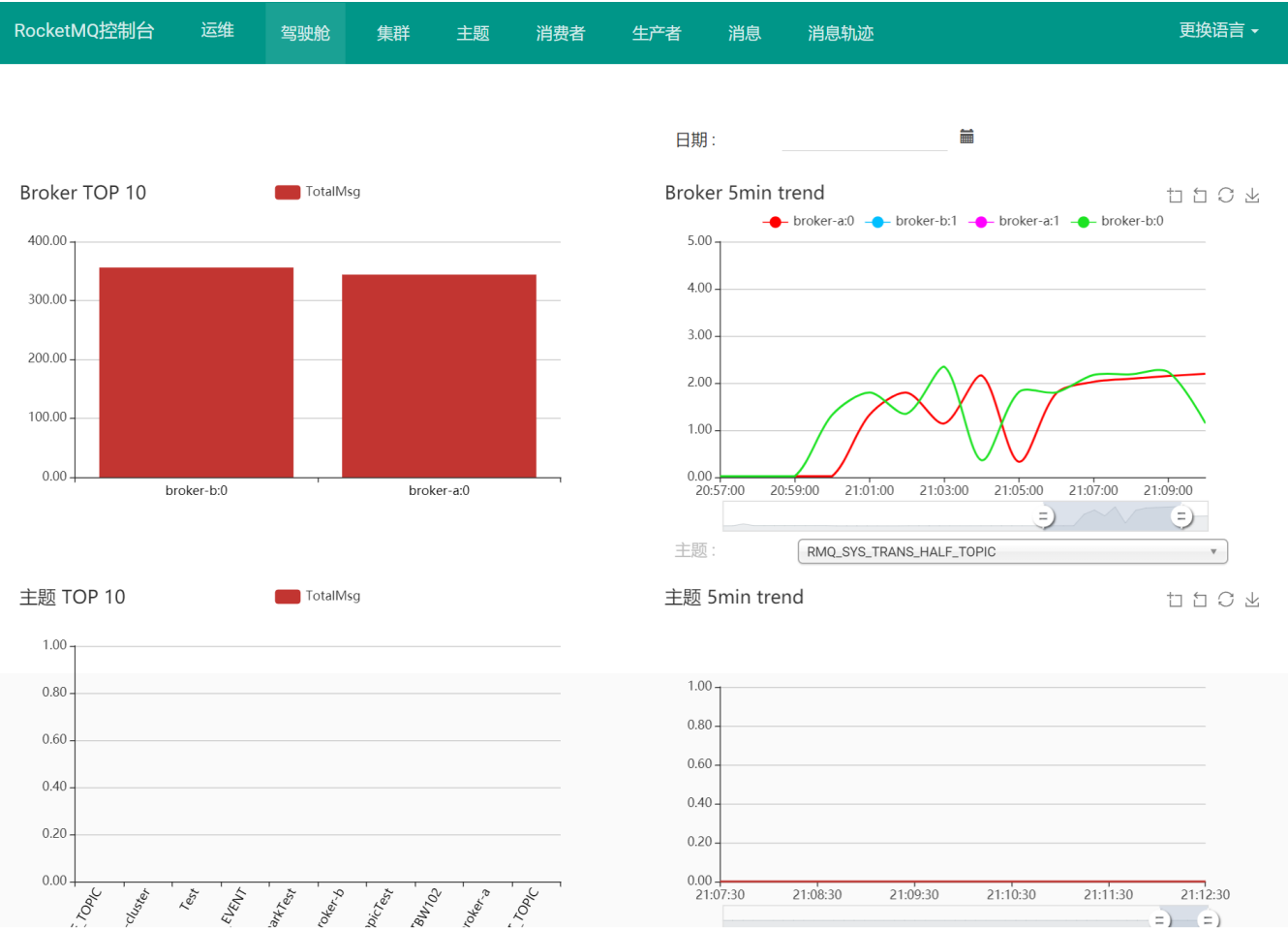
注意：打包前在rocketmq-console中配置namesrv集群地址：

rocketmq.config.namesrvAddr=192.168.25.135:9876;192.168.25.138:9876

启动rocketmq-console:

java -jar rocketmq-console-ng-1.0.0.jar

启动成功后，我们就可以通过浏览器访问http://localhost:8080进入控制台界面了，如下图：



集群状态:

RocketMQ控制台 运维 驾驶舱 集群 主题 消费者 生产者 消息 消息轨迹 更换语言										
集群: rocketmq-cluster										
分片	编号	地址	版本	生产消息TPS	消费消息TPS	昨日生产总数	昨日消费总数	今天生产总数	今天消费总数	操作
broker-b	0(master)	192.168.25.138:10911	V4_4_0	0.00	0.00	0	0	152	356	<button>状态</button> <button>配置</button>
broker-a	0(master)	192.168.25.135:10911	V4_4_0	0.00	0.00	0	0	148	344	<button>状态</button> <button>配置</button>

4. 消息发送样例

- 导入MQ客户端依赖

```
<dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-client</artifactId>
    <version>4.4.0</version>
</dependency>
```

- 消息发送者步骤分析

1. 创建消息生产者producer，并制定生产者组名
2. 指定Nameserver地址
3. 启动producer
4. 创建消息对象，指定主题Topic、Tag和消息体
5. 发送消息
6. 关闭生产者producer

- 消息消费者步骤分析

1. 创建消费者Consumer，制定消费者组名
2. 指定Nameserver地址
3. 订阅主题Topic和Tag
4. 设置回调函数，处理消息
5. 启动消费者consumer

4.1 基本样例

4.1.1 消息发送

1) 发送同步消息

这种可靠性同步地发送方式使用的比较广泛，比如：重要的消息通知，短信通知。

```
public class SyncProducer {
    public static void main(String[] args) throws Exception {
        // 实例化消息生产者Producer
        DefaultMQProducer producer = new DefaultMQProducer("please_rename_unique_group_name");
        // 设置Nameserver的地址
        producer.setNamesrvAddr("localhost:9876");
        // 启动Producer实例
        producer.start();
        for (int i = 0; i < 100; i++) {
            // 创建消息，并指定Topic, Tag和消息体
            Message msg = new Message("TopicTest" /* Topic */,
                "TagA" /* Tag */,
                ("Hello RocketMQ " + i).getBytes(RemotingHelper.DEFAULT_CHARSET) /* Message
body */
            );
            // 发送消息到一个Broker
            SendResult sendResult = producer.send(msg);
            // 通过sendResult返回消息是否成功送达
            System.out.printf("%s\n", sendResult);
        }
        // 如果不再发送消息，关闭Producer实例。
        producer.shutdown();
    }
}
```

2) 发送异步消息

异步消息通常用在响应时间敏感的业务场景，即发送端不能容忍长时间地等待Broker的响应。

```

public class AsyncProducer {
    public static void main(String[] args) throws Exception {
        // 实例化消息生产者Producer
        DefaultMQProducer producer = new DefaultMQProducer("please_rename_unique_group_name");
        // 设置NameServer的地址
        producer.setNamesrvAddr("localhost:9876");
        // 启动Producer实例
        producer.start();
        producer.setRetryTimesWhenSendAsyncFailed(0);
        for (int i = 0; i < 100; i++) {
            final int index = i;
            // 创建消息, 并指定Topic, Tag和消息体
            Message msg = new Message("TopicTest",
                "TagA",
                "OrderID188",
                "Hello world".getBytes(RemotingHelper.DEFAULT_CHARSET));
            // SendCallback接收异步返回结果的回调
            producer.send(msg, new SendCallback() {
                @Override
                public void onSuccess(SendResult sendResult) {
                    System.out.printf("%-10d OK %s %n", index,
                        sendResult.getMsgId());
                }
                @Override
                public void onException(Throwable e) {
                    System.out.printf("%-10d Exception %s %n", index, e);
                    e.printStackTrace();
                }
            });
        }
        // 如果不再发送消息, 关闭Producer实例。
        producer.shutdown();
    }
}

```

3) 单向发送消息

这种方式主要用在不特别关心发送结果的场景, 例如日志发送。

```

public class OnewayProducer {
    public static void main(String[] args) throws Exception{
        // 实例化消息生产者Producer
        DefaultMQProducer producer = new DefaultMQProducer("please_rename_unique_group_name");
        // 设置NameServer的地址
        producer.setNamesrvAddr("localhost:9876");
        // 启动Producer实例
        producer.start();
        for (int i = 0; i < 100; i++) {
            // 创建消息, 并指定Topic, Tag和消息体
            Message msg = new Message("TopicTest" /* Topic */,
                "TagA" /* Tag */,
                ("Hello RocketMQ " + i).getBytes(RemotingHelper.DEFAULT_CHARSET) /* Message
body */
            );
            // 发送单向消息, 没有任何返回结果
            producer.sendOneway(msg);
        }
        // 如果不再发送消息, 关闭Producer实例。
        producer.shutdown();
    }
}

```



```
}  
}
```

4.1.2 消费消息

1) 负载均衡模式

消费者采用负载均衡方式消费消息，多个消费者共同消费队列消息，每个消费者处理的消息不同

```
public static void main(String[] args) throws Exception {  
    // 实例化消息生产者,指定组名  
    DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("group1");  
    // 指定Namesrv地址信息.  
    consumer.setNamesrvAddr("localhost:9876");  
    // 订阅Topic  
    consumer.subscribe("Test", "*");  
    //负载均衡模式消费  
    consumer.setMessageModel(MessageModel.CLUSTERING);  
    // 注册回调函数, 处理消息  
    consumer.registerMessageListener(new MessageListenerConcurrently() {  
        @Override  
        public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs,  
                                                         ConsumeConcurrentlyContext context) {  
            System.out.printf("%s Receive New Messages: %s %n",  
                               Thread.currentThread().getName(), msgs);  
            return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;  
        }  
    });  
    //启动消息者  
    consumer.start();  
    System.out.printf("Consumer Started.%n");  
}
```

2) 广播模式

消费者采用广播的方式消费消息，每个消费者消费的消息都是相同的

```
public static void main(String[] args) throws Exception {  
    // 实例化消息生产者,指定组名  
    DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("group1");  
    // 指定Namesrv地址信息.  
    consumer.setNamesrvAddr("localhost:9876");  
    // 订阅Topic  
    consumer.subscribe("Test", "*");  
    //广播模式消费  
    consumer.setMessageModel(MessageModel.BROADCASTING);  
    // 注册回调函数, 处理消息  
    consumer.registerMessageListener(new MessageListenerConcurrently() {  
        @Override  
        public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs,  
                                                         ConsumeConcurrentlyContext context) {  
            System.out.printf("%s Receive New Messages: %s %n",  
                               Thread.currentThread().getName(), msgs);  
            return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;  
        }  
    });  
    //启动消息者  
    consumer.start();  
}
```

```

        System.out.printf("Consumer Started.%n");
    }
}

```

4.2 顺序消息

消息有序指的是可以按照消息的发送顺序来消费(FIFO)。RocketMQ可以严格的保证消息有序，可以分为分区有序或者全局有序。

顺序消费的原理解析，在默认的情况下消息发送会采取Round Robin轮询方式把消息发送到不同的queue(分区队列)；而消费消息的时候从多个queue上拉取消息，这种情况发送和消费是不能保证顺序。但是如果控制发送的顺序消息只依次发送到同一个queue中，消费的时候只从这个queue上依次拉取，则就保证了顺序。当发送和消费参与的queue只有一个，则是全局有序；如果多个queue参与，则为分区有序，即相对每个queue，消息都是有序的。

下面用订单进行分区有序的示例。一个订单的顺序流程是：创建、付款、推送、完成。订单号相同的消息会被先后发送到同一个队列中，消费时，同一个OrderId获取到的肯定是同一个队列。

4.2.1 顺序消息生产

```

/**
 * Producer, 发送顺序消息
 */
public class Producer {

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

        producer.setNamesrvAddr("127.0.0.1:9876");

        producer.start();

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

        // 订单列表
        List<OrderStep> orderList = new Producer().buildOrders();

        Date date = new Date();
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        String dateStr = sdf.format(date);
        for (int i = 0; i < 10; i++) {
            // 加个时间前缀
            String body = dateStr + " Hello RocketMQ " + orderList.get(i);
            Message msg = new Message("TopicTest", tags[i % tags.length], "KEY" + i,
            body.getBytes());

            SendResult sendResult = producer.send(msg, new MessageQueueSelector() {
                @Override
                public MessageQueue select(List<MessageQueue> mqs, Message msg, Object arg) {
                    Long id = (Long) arg; //根据订单id选择发送queue
                    long index = id % mqs.size();
                    return mqs.get((int) index);
                }
            }, orderList.get(i).getOrderId()); //订单id

            System.out.println(String.format("SendResult status:%s, queueId:%d, body:%s",
            sendResult.getSendStatus(),
            sendResult.getMessageQueue().getQueueId(),
            body));
        }
    }
}

```

```

        producer.shutdown();
    }

    /**
     * 订单的步骤
     */
    private static class OrderStep {
        private long orderId;
        private String desc;

        public long getOrderId() {
            return orderId;
        }

        public void setOrderId(long orderId) {
            this.orderId = orderId;
        }

        public String getDesc() {
            return desc;
        }

        public void setDesc(String desc) {
            this.desc = desc;
        }

        @Override
        public String toString() {
            return "OrderStep{" +
                "orderId=" + orderId +
                ", desc='" + desc + '\'' +
                '}';
        }
    }

    /**
     * 生成模拟订单数据
     */
    private List<OrderStep> buildOrders() {
        List<OrderStep> orderList = new ArrayList<OrderStep>();

        OrderStep orderDemo = new OrderStep();
        orderDemo.setOrderId(15103111039L);
        orderDemo.setDesc("创建");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(15103111065L);
        orderDemo.setDesc("创建");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(15103111039L);
        orderDemo.setDesc("付款");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(15103117235L);
        orderDemo.setDesc("创建");
    }

```

```

        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(15103111065L);
        orderDemo.setDesc("付款");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(15103117235L);
        orderDemo.setDesc("付款");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(15103111065L);
        orderDemo.setDesc("完成");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(15103111039L);
        orderDemo.setDesc("推送");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(15103117235L);
        orderDemo.setDesc("完成");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(15103111039L);
        orderDemo.setDesc("完成");
        orderList.add(orderDemo);

        return orderList;
    }
}

```

4.2.2 顺序消费消息

```

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

    public static void main(String[] args) throws Exception {
        DefaultMQPushConsumer consumer = new
            DefaultMQPushConsumer("please_rename_unique_group_name_3");
        consumer.setNamesrvAddr("127.0.0.1:9876");
        /**
         * 设置Consumer第一次启动是从队列头部开始消费还是队列尾部开始消费<br>
         * 如果非第一次启动，那么按照上次消费的位置继续消费
         */
        consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_OFFSET);

        consumer.subscribe("TopicTest", "TagA || TagC || TagD");

        consumer.registerMessageListener(new MessageListenerOrderly() {

            Random random = new Random();

            @Override

```

```

        public ConsumeOrderlyStatus consumeMessage(List<MessageExt> msgs,
ConsumeOrderlyContext context) {
            context.setAutoCommit(true);
            for (MessageExt msg : msgs) {
                // 可以看到每个queue有唯一的consume线程来消费，订单对每个queue(分区)有序
                System.out.println("consumeThread=" + Thread.currentThread().getName() +
"queueId=" + msg.getQueueId() + ", content:" + new String(msg.getBody()));
            }

            try {
                //模拟业务逻辑处理中...
                TimeUnit.SECONDS.sleep(random.nextInt(10));
            } catch (Exception e) {
                e.printStackTrace();
            }
            return ConsumeOrderlyStatus.SUCCESS;
        }
    }
});

consumer.start();

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

```

4.3 延时消息

比如电商里，提交了一个订单就可以发送一个延时消息，1h后去检查这个订单的状态，如果还是未付款就取消订单释放库存。

4.3.1 启动消息消费者

```

public class ScheduledMessageConsumer {
    public static void main(String[] args) throws Exception {
        // 实例化消费者
        DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("ExampleConsumer");
        // 订阅Topics
        consumer.subscribe("TestTopic", "*");
        // 注册消息监听者
        consumer.registerMessageListener(new MessageListenerConcurrently() {
            @Override
            public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> messages,
ConsumeConcurrentlyContext context) {
                for (MessageExt message : messages) {
                    // Print approximate delay time period
                    System.out.println("Receive message[msgId=" + message.getMsgId() + "] " +
(System.currentTimeMillis() - message.getStoreTimestamp()) + "ms later");
                }
                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
            }
        });
        // 启动消费者
        consumer.start();
    }
}

```

4.3.2 发送延时消息

```

public class ScheduledMessageProducer {
    public static void main(String[] args) throws Exception {
        // 实例化一个生产者来产生延时消息
        DefaultMQProducer producer = new DefaultMQProducer("ExampleProducerGroup");
        // 启动生产者
        producer.start();
        int totalMessagesToSend = 100;
        for (int i = 0; i < totalMessagesToSend; i++) {
            Message message = new Message("TestTopic", ("Hello scheduled message " +
i).getBytes());
            // 设置延时等级3,这个消息将在10s之后发送(现在只支持固定的几个时间,详看delayTimeLevel)
            message.setDelayTimeLevel(3);
            // 发送消息
            producer.send(message);
        }
        // 关闭生产者
        producer.shutdown();
    }
}

```

###4.3.3 验证

您将会看到消息的消费比存储时间晚10秒

4.3.4 使用限制

```

// org/apache/rocketmq/store/config/MessageStoreConfig.java
private String messageDelayLevel = "1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m 30m 1h 2h";

```

现在RocketMq并不支持任意时间的延时，需要设置几个固定的延时等级，从1s到2h分别对应着等级1到18

4.4 批量消息

批量发送消息能显著提高传递小消息的性能。限制是这些批量消息应该有相同的topic，相同的waitStoreMsgOK，而且不能是延时消息。此外，这一批消息的总大小不应超过4MB。

4.4.1 发送批量消息

如果您每次只发送不超过4MB的消息，则很容易使用批处理，样例如下：

```

String topic = "BatchTest";
List<Message> messages = new ArrayList<>();
messages.add(new Message(topic, "TagA", "OrderID001", "Hello world 0".getBytes()));
messages.add(new Message(topic, "TagA", "OrderID002", "Hello world 1".getBytes()));
messages.add(new Message(topic, "TagA", "OrderID003", "Hello world 2".getBytes()));
try {
    producer.send(messages);
} catch (Exception e) {
    e.printStackTrace();
    //处理error
}

```

如果消息的总长度可能大于4MB时，这时候最好把消息进行分割

```

public class ListSplitter implements Iterator<List<Message>> {
    private final int SIZE_LIMIT = 1024 * 1024 * 4;
    private final List<Message> messages;
    private int currIndex;

```

```

public ListSplitter(List<Message> messages) {
    this.messages = messages;
}
@Override
public boolean hasNext() {
    return currIndex < messages.size();
}
@Override
public List<Message> next() {
    int nextIndex = currIndex;
    int totalSize = 0;
    for (; nextIndex < messages.size(); nextIndex++) {
        Message message = messages.get(nextIndex);
        int tmpSize = message.getTopic().length() + message.getBody().length;
        Map<String, String> properties = message.getProperties();
        for (Map.Entry<String, String> entry : properties.entrySet()) {
            tmpSize += entry.getKey().length() + entry.getValue().length();
        }
        tmpSize = tmpSize + 20; // 增加日志的开销20字节
        if (tmpSize > SIZE_LIMIT) {
            //单个消息超过了最大的限制
            //忽略,否则会阻塞分裂的进程
            if (nextIndex - currIndex == 0) {
                //假如下一个子列表没有元素,则添加这个子列表然后退出循环,否则只是退出循环
                nextIndex++;
            }
            break;
        }
        if (tmpSize + totalSize > SIZE_LIMIT) {
            break;
        } else {
            totalSize += tmpSize;
        }
    }
    List<Message> subList = messages.subList(currIndex, nextIndex);
    currIndex = nextIndex;
    return subList;
}
}
//把大的消息分裂成若干个小的消息
ListSplitter splitter = new ListSplitter(messages);
while (splitter.hasNext()) {
    try {
        List<Message> listItem = splitter.next();
        producer.send(listItem);
    } catch (Exception e) {
        e.printStackTrace();
        //处理error
    }
}
}

```

4.5 过滤消息

在大多数情况下，TAG是一个简单而有用的设计，其可以来选择您想要的消息。例如：

```

DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("CID_EXAMPLE");
consumer.subscribe("TOPIC", "TAGA || TAGB || TAGC");

```

消费者将接收包含TAGA或TAGB或TAGC的消息。但是限制是一个消息只能有一个标签，这对于复杂的场景可能不起作用。在这种情况下，可以使用SQL表达式筛选消息。SQL特性可以通过发送消息时的属性来进行计算。在RocketMQ定义的语法下，可以实现一些简单的逻辑。下面是一个例子：

```
-----
| 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 |
-----
```

4.5.1 SQL基本语法

RocketMQ只定义了一些基本语法来支持这个特性。你也可以很容易地扩展它。

- 数值比较，比如：>, >=, <, <=, BETWEEN, =;
- 字符比较，比如：=, <>, IN;
- IS NULL 或者 IS NOT NULL;
- 逻辑符号 AND, OR, NOT;

常量支持类型为：

- 数值，比如：123, 3.1415;
- 字符，比如：'abc'，必须用单引号包裹起来；
- NULL，特殊的常量
- 布尔值，TRUE 或 FALSE

只有使用push模式的消费者才能用使用SQL92标准的sql语句，接口如下：

```
public void subscribe(finalString topic, final MessageSelector messageSelector)
```

4.5.2 消息生产者

发送消息时，你能通过putUserProperty来设置消息的属性

```
DefaultMQProducer producer = new DefaultMQProducer("please_rename_unique_group_name");
producer.start();
Message msg = new Message("TopicTest",
    tag,
    ("Hello RocketMQ " + i).getBytes(RemotingHelper.DEFAULT_CHARSET)
);
// 设置一些属性
msg.putUserProperty("a", String.valueOf(i));
SendResult sendResult = producer.send(msg);

producer.shutdown();
```

4.5.3 消息消费者

用MessageSelector.bySql来使用sql筛选消息

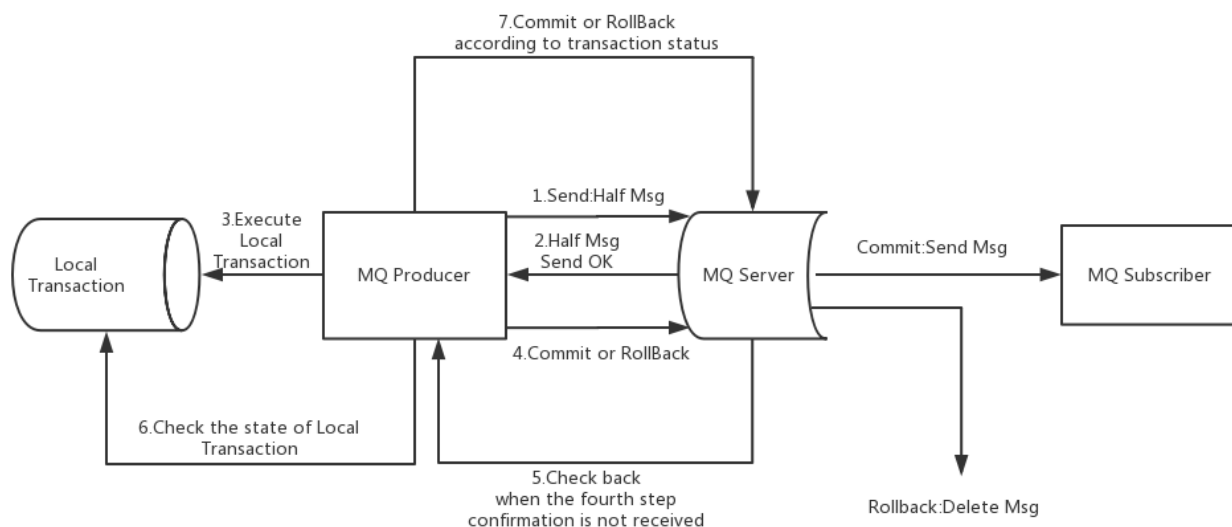

```

DefaultMQPushConsumer consumer = new
DefaultMQPushConsumer("please_rename_unique_group_name_4");
// 只有订阅的消息有这个属性a, a >=0 and a <= 3
consumer.subscribe("TopicTest", MessageSelector.bySql("a between 0 and 3"));
consumer.registerMessageListener(new MessageListenerConcurrently() {
    @Override
    public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs,
ConsumeConcurrentlyContext context) {
        return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
    }
});
consumer.start();

```

4.6 事务消息

###4.6.1 流程分析



上图说明了事务消息的大致方案，其中分为两个流程：正常事务消息的发送及提交、事务消息的补偿流程。

####1) 事务消息发送及提交

- (1) 发送消息（half消息）。
- (2) 服务端响应消息写入结果。
- (3) 根据发送结果执行本地事务（如果写入失败，此时half消息对业务不可见，本地逻辑不执行）。
- (4) 根据本地事务状态执行Commit或者Rollback（Commit操作生成消息索引，消息对消费者可见）

2) 事务补偿

- (1) 对没有Commit/Rollback的事务消息（pending状态的消息），从服务端发起一次“回查”
- (2) Producer收到回查消息，检查回查消息对应的本地事务的状态
- (3) 根据本地事务状态，重新Commit或者Rollback

其中，补偿阶段用于解决消息Commit或者Rollback发生超时或者失败的情况。

3) 事务消息状态

事务消息共有三种状态，提交状态、回滚状态、中间状态：

- TransactionStatus.CommitTransaction: 提交事务，它允许消费者消费此消息。
- TransactionStatus.RollbackTransaction: 回滚事务，它代表该消息将被删除，不允许被消费。
- TransactionStatus.Unknown: 中间状态，它代表需要检查消息队列来确定状态。

###4.6.1 发送事务消息

1) 创建事务性生产者

使用 TransactionMQProducer类创建生产者，并指定唯一的 ProducerGroup，就可以设置自定义线程池来处理这些检查请求。执行本地事务后、需要根据执行结果对消息队列进行回复。回传的事务状态在请参考前一节。

```
public class Producer {
    public static void main(String[] args) throws MQClientException, InterruptedException {
        //创建事务监听器
        TransactionListener transactionListener = new TransactionListenerImpl();
        //创建消息生产者
        TransactionMQProducer producer = new TransactionMQProducer("group6");
        producer.setNamesrvAddr("192.168.25.135:9876;192.168.25.138:9876");
        //生产者这是监听器
        producer.setTransactionListener(transactionListener);
        //启动消息生产者
        producer.start();
        String[] tags = new String[]{"TagA", "TagB", "TagC"};
        for (int i = 0; i < 3; i++) {
            try {
                Message msg = new Message("TransactionTopic", tags[i % tags.length], "KEY" +
i,
                    ("Hello RocketMQ " + i).getBytes(RemotingHelper.DEFAULT_CHARSET));
                SendResult sendResult = producer.sendMessageInTransaction(msg, null);
                System.out.printf("%s\n", sendResult);
                TimeUnit.SECONDS.sleep(1);
            } catch (MQClientException | UnsupportedEncodingException e) {
                e.printStackTrace();
            }
        }
        //producer.shutdown();
    }
}
```

2) 实现事务的监听接口

当发送半消息成功时，我们使用 executeLocalTransaction 方法来执行本地事务。它返回前一节中提到的三个事务状态之一。checkLocalTranscation 方法用于检查本地事务状态，并回应消息队列的检查请求。它也是返回前一节中提到的三个事务状态之一。

```
public class TransactionListenerImpl implements TransactionListener {

    @Override
    public LocalTransactionState executeLocalTransaction(Message msg, Object arg) {
        System.out.println("执行本地事务");
        if (StringUtils.equals("TagA", msg.getTags())) {
            return LocalTransactionState.COMMIT_MESSAGE;
        } else if (StringUtils.equals("TagB", msg.getTags())) {
```

```

        return LocalTransactionState.ROLLBACK_MESSAGE;
    } else {
        return LocalTransactionState.UNKNOW;
    }
}

@Override
public LocalTransactionState checkLocalTransaction(MessageExt msg) {
    System.out.println("MQ检查消息Tag【"+msg.getTags()+"】的本地事务执行结果");
    return LocalTransactionState.COMMIT_MESSAGE;
}
}

```

4.6.2 使用限制

1. 事务消息不支持延时消息和批量消息。
2. 为了避免单个消息被检查太多次而导致半队列消息累积，我们默认将单个消息的检查次数限制为 15 次，但是用户可以通过 Broker 配置文件的 transactionCheckMax 参数来修改此限制。如果已经检查某条消息超过 N 次的话（N = transactionCheckMax）则 Broker 将丢弃此消息，并在默认情况下同时打印错误日志。用户可以通过重写 AbstractTransactionCheckListener 类来修改这个行为。
3. 事务消息将在 Broker 配置文件中的参数 transactionMsgTimeout 这样的特定时间长度之后被检查。当发送事务消息时，用户还可以通过设置用户属性 CHECK_IMMUNITY_TIME_IN_SECONDS 来改变这个限制，该参数优先于 transactionMsgTimeout 参数。
4. 事务性消息可能不止一次被检查或消费。
5. 提交给用户的目标主题消息可能会失败，目前这依日志的记录而定。它的高可用性通过 RocketMQ 本身的高可用性机制来保证，如果希望确保事务消息不丢失、并且事务完整性得到保证，建议使用同步的双重写入机制。
6. 事务消息的生产者 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，事务消息允许反向查询、MQ 服务器能通过它们的生产者 ID 查询到消费者。