RocketMQ

Consumer(消费者)、Producer(生产者)

- RocketMQ原生支持分布式，ActiveMQ原生存在单点性。
- 可以保证严格的消息顺序。
- 提供亿级消息堆积能力，并且依然保持写入低延迟。
- 丰富的消息拉取模式。
- 消息失败重试机制、高效的订阅者水平扩展能力，强大的API、事务机制。
- 分布式协调采用的Zookeeper,RockerMQ自己也实现了一个NameServer，更加轻量级、性能更好。

入门

消息队列最原始的就是 **数据结构的队列** “**先进先出**”的数据结构发展过来的。

> **特点**
>
> **应用解耦**：在微服务的时代，避免一个子系统出现问题，导致系统的其他功能无法使用的问题，使用消息队列进行应用解耦。
>
> **流量消峰**：通过消息队列，把大量的数据请求缓存起来，分散到相对长的一段时间内处理，能够大大提高系统的稳定性和用户体验。
>
> **消息分发**：数据的生产者只关心把各自的数据写入一个消息队列即可，数据的使用方根据各自的需求订阅感兴趣的数据，不同的团队所订阅的数据可以重复也可以不重复，互不干扰，也不必和数据产生关联。
>
> **消息一致性**、**动态扩容** 等
>
> **RocketMQ基于长轮询拉取方式 暂时还不懂其中的优点**

简单环境搭建

下载地址：http://rocketmq.apache.org/dowloading/releases

安装指南：https://blog.csdn.net/wangmx1993328/article/details/81536168

> 使用nohup sh bin/mqnamesrv & 报 **ignoring input and appending output to 'nohup.out'** 不要管他，正常性错误。
>
> 在使用这个**nohup sh bin/mqnamesrv &** 之前一定要在 bin/runservice.sh中修改他的环境配置：

```
#===================================================================================
# Java Environment Setting
#===================================================================================
export JAVA_HOME=/root/jdk/jdk1.8.0_212
export CLASSPATH=$JAVA_HOME/lib:$JAVA_HOME/jre/lib
error_exit ()
{
    echo "ERROR: $1 !!"
    exit 1
}

#[ ! -e "$JAVA_HOME/bin/java" ] && JAVA_HOME=$HOME/jdk/java
#[ ! -e "$JAVA_HOME/bin/java" ] && JAVA_HOME=/usr/java
#[ ! -e "$JAVA_HOME/bin/java" ] && error_exit "Please set the JAVA_HOME variable in your environment, We need java(x64)
!"

#export JAVA_HOME
export JAVA="$JAVA_HOME/bin/java"
export BASE_DIR=$(dirname $0)/..
export CLASSPATH=.:${BASE_DIR}/conf:${CLASSPATH}
echo "$CLASSPATH"
#===================================================================================
# JVM Configuration
#===================================================================================
JAVA_OPT="${JAVA_OPT} -server -Xms4g -Xmx4g -Xmn2g -XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=320m"
JAVA_OPT="${JAVA_OPT} -XX:+UseConcMarkSweepGC -XX:+UseCMSCompactAtFullCollection -XX:CMSInitiatingOccupancyFraction=70
-XX:+CMSParallelRemarkEnabled -XX:SoftRefLRUPolicyMSPerMB=0 -XX:+CMSClassUnloadingEnabled -XX:SurvivorRatio=8  -XX:-Use
ParNewGC"
```

新加，不然会出现找不到java的情况

去掉

如果出现地址被占用：①：重启虚拟机 ②：找到nameserver端口默认是9876，并且 netstat -tln | grep 9876。然后干掉他。

第二个坑

如果tail -f ~/logs/rocketLogs/broker.log 出现找不到该文件，表示需要修改JVM的配置：

runserver.sh 将原来的4G改小，至于改到多少合适，暂时不知，

```
export JAVA="$JAVA_HOME/bin/java"
export BASE_DIR=$(dirname $0)/..
export CLASSPATH=.:${BASE_DIR}/conf:${CLASSPATH}
echo "$CLASSPATH"
#===================================================================================
# JVM Configuration
#===================================================================================
JAVA_OPT="${JAVA_OPT} -server -Xms256m -Xmx256m -Xmn125m -XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=320m"
JAVA_OPT="${JAVA_OPT} -XX:+UseConcMarkSweepGC -XX:+UseCMSCompactAtFullCollection -XX:CMSInitiatingOccupancyFraction=70
-XX:+CMSParallelRemarkEnabled -XX:SoftRefLRUPolicyMSPerMB=0 -XX:+CMSClassUnloadingEnabled -XX:SurvivorRatio=8  -XX:-Use
ParNewGC"
JAVA_OPT="${JAVA_OPT} -verbose:gc -Xloggc:/dev/shm/rmq_srv_gc.log -XX:+PrintGCDetails"
JAVA_OPT="${JAVA_OPT} -XX:-OmitStackTraceInFastThrow"
"runserver.sh" 51L, 2561C                                                    41,56         70%
```

runbroker.sh将原来的8G改小：

```
export JAVA="$JAVA_HOME/bin/java"
export BASE_DIR=$(dirname $0)/..
export CLASSPATH=.:${BASE_DIR}/conf:${CLASSPATH}

#===================================================================================
# JVM Configuration
#===================================================================================
JAVA_OPT="${JAVA_OPT} -server -Xms256m -Xmx256m -Xmn125m"
JAVA_OPT="${JAVA_OPT} -XX:+UseG1GC -XX:G1HeapRegionSize=16m -XX:G1ReservePercent=25 -XX:InitiatingHeapOc
30 -XX:SoftRefLRUPolicyMSPerMB=0 -XX:SurvivorRatio=8"
JAVA_OPT="${JAVA_OPT} -verbose:gc -Xloggc:/dev/shm/mq_gc_%p.log -XX:+PrintGCDetails -XX:+PrintGCDateStam
ApplicationStoppedTime -XX:+PrintAdaptiveSizePolicy"
JAVA_OPT="${JAVA_OPT} -XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=5 -XX:GCLogFileSize=30m"
"runbroker.sh" 62L, 2875C                                                    39,
```

启用RocketMq

```
nohup sh bin/mqnamesrv &

tail -f ~/logs/rocketmqLogs/namesrv.log

nohup sh bin/mqbroker -n localhost:9876 &

tail -f ~/logs/rocketmqLogs/broker.log
```

使用命令行发送和接收消息

```
export NAMESRV_ADDR=localhost:9876
sh bin/tools.sh org.apache.rocketmq.example.quickstart.Producer
sh bin/tools.sh org.apache.rocketmq.example.quickstart.Consumer
```

关闭消息队列

打开的时候，先NameServer再是Broker。关闭就是先Broker在NameServer。

```
sh bin/mqshutdown broker
sh bin/mqshutdown namesrv
```

生产环境下的配置和使用

RocketMQ四部分组成：**生产者Producer、消费者Consumer、暂存处Broker、协调者NameServer(好像也可以使用Zookeeper)。**

启动RocketMq的顺序是先启动NameServer再启动Broker。启动完成后，这个消息队列可以提供服务了。对于消息生产和消费主要是 Producer和Consumer负责。

> 为了消除单点故障，增加可靠性或者增大吞吐量，可以在多台机器上部署多个NameServer和Broker。为每个Broker部署一个或者多个Slave。

Topic: 不同类型的消息使用不同的Topic来区分。

Message Queue：在有Topic将消息分类后，还存在性能问题。如果一个Topic所涉及的数据量非常大，需要能支持并行处理的机器来提高处理速度，这个时候一个Topic可以根据需求设置一个或者多个Message Queue。Message Queue 类似分区或者Partition。Topic有了多个Message Queue后，消息可以并行的向Message Queue 发送消息。同理消费者也可以并行的处理消息。

在两台机器上都装上rocketMq，比如我的地址 A和B。

```
首先启动NameServer
nohup sh bin/mqnamesrv &
```

在地址A机器上，Master和Slave(在conf/2m-2s-sync下)配置如下：

> 2m-2s-sync : 同步双写
>
> 2,m-2s-async：异步双写
>
> 2m-noslave：多master模式

Master

```
# NameServer的地址 可以地址使用;分割
namesrvAddr=A:9876;B:9876
#所属集群的名字
brokerClusterName=DefaultCluster
#Broker的名称，Master和Slave通过使用相同的Broker名称来表明互相的关系。说明某个Slave是那个Master的Slave
brokerName=broker-a
# 一个Master Broker可以有多少个Slave、0表示Master，大于0表示不同的Slave的ID
brokerId=0
#和fileReservedTime想配合的参数，表明在几点做消息删除的动作。04，表示凌晨4点操作。
deleteWhen=04
# 在磁盘保存消息的时长，超时自动删除。单位是小时
fileReservedTime=48
# Broker规则有：SYNC_MASTER：表示Master和Slave之间同步消息机制、ASYNC_MASTER：表示异步复制、SLAVE。
brokerRole=SYNC_MASTER
# 表示刷盘策略，分为SYNC_FLUSH，同步刷盘，消息真正写入磁盘在返回成功状态、和ASYNC_FLUSH：异步刷盘，消息写入page_cache后返回成功状态两种
flushDiskType=ASYNC_FLUSH
# Broker监听的端口号，如果一台机器上，启动多个Broker，则要设置不同的端口号避免冲突
listenPort=10911
# 存储消息和配置的目录
storePathRootDir=/home/rocketmq/store-a

```

Slave

```
namesrvAddr=A:9876;B:9876
brokerClusterName=DefaultCluster
brokerName=broker-b
brokerId=1
deleteWhen=04
fileReservedTime=48
brokerRole=SLAVE
flushDiskType=ASYNC_FLUSH
listenPort=11011
storePathRootDir=/home/rocketmq/store-b

```

在地址B的机器上

Master

```
namesrvAddr=192.168.9.110:9876;192.168.9.111:9876
brokerClusterName=DefaultCluster
brokerName=broker-b
brokerId=0
deleteWhen=04
fileReservedTime=48
brokerRole=SYNC_MASTER
flushDiskType=ASYNC_FLUSH
listenPort=10911
storePathRootDir=/home/rocketmq/store-b

```

Slave

```
namesrvAddr=192.168.9.110:9876;192.168.9.111:9876
brokerClusterName=DefaultCluster
brokerName=broker-a
brokerId=1
deleteWhen=04
fileReservedTime=48
brokerRole=SLAVE
flushDiskType=ASYNC_FLUSH
listenPort=11011
storePathRootDir=/home/rocketmq/store-a

```

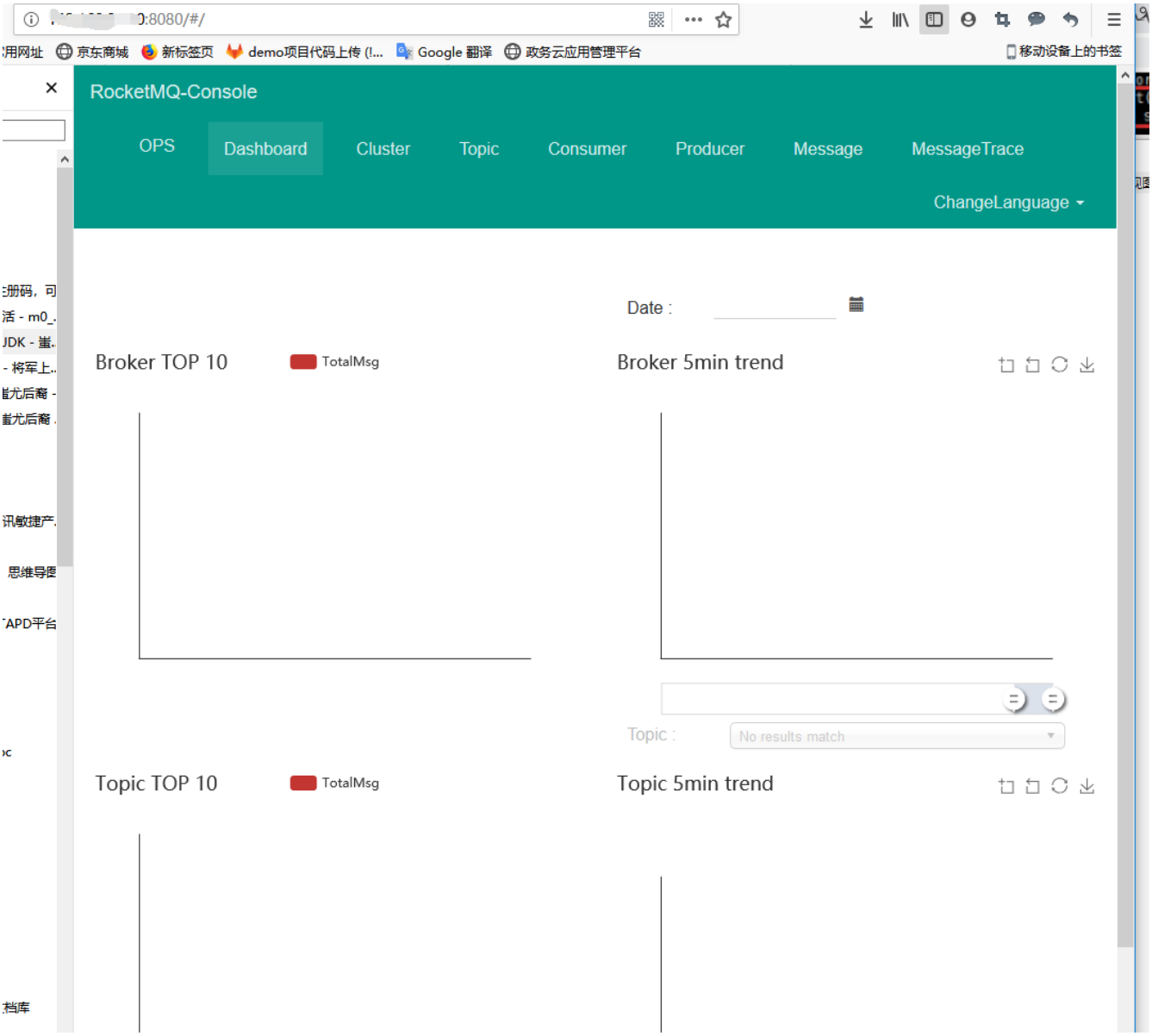
```
启动broker

nohup sh bin/mqbroker -c 配置文件 &
```

安装rocketMq_Console

https://blog.csdn.net/lwf006164/article/details/91628252

> mvn clean package -Dmaven.test.skip=true #maven打包
>
> 使用 java -jar target/xxx.jar & 启动
>
> 然后访问A:8080(首先是自己在A中安装的)

运行



测试一下集群

新建一个maven项目并且引入rocket的client

```
1  public class SyncConsumer {
2
3      public static void main(String[] args) throws MQClientException {
4          DefaultMQPushConsumer defaultMQPushConsumer = new
   DefaultMQPushConsumer("pleaserenameuniquegroupname");
5          defaultMQPushConsumer.setNamesrvAddr("192.168.9.110:9876");
6
    defaultMQPushConsumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_OFFSET)
   ;
7          defaultMQPushConsumer.subscribe("TopicTest", "*");
8          defaultMQPushConsumer.registerMessageListener(new
   MessageListenerConcurrently() {
9              @Override
10             public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> list,
```

```
                                                        ConsumeConcurrentlyContext
consumeConcurrentlyContext) {
                System.out.printf(Thread.currentThread().getName() + "Receive New
Messages:"
                        + list + "%n");
                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
            }
        });
        defaultMQPushConsumer.start();
    }

}
```

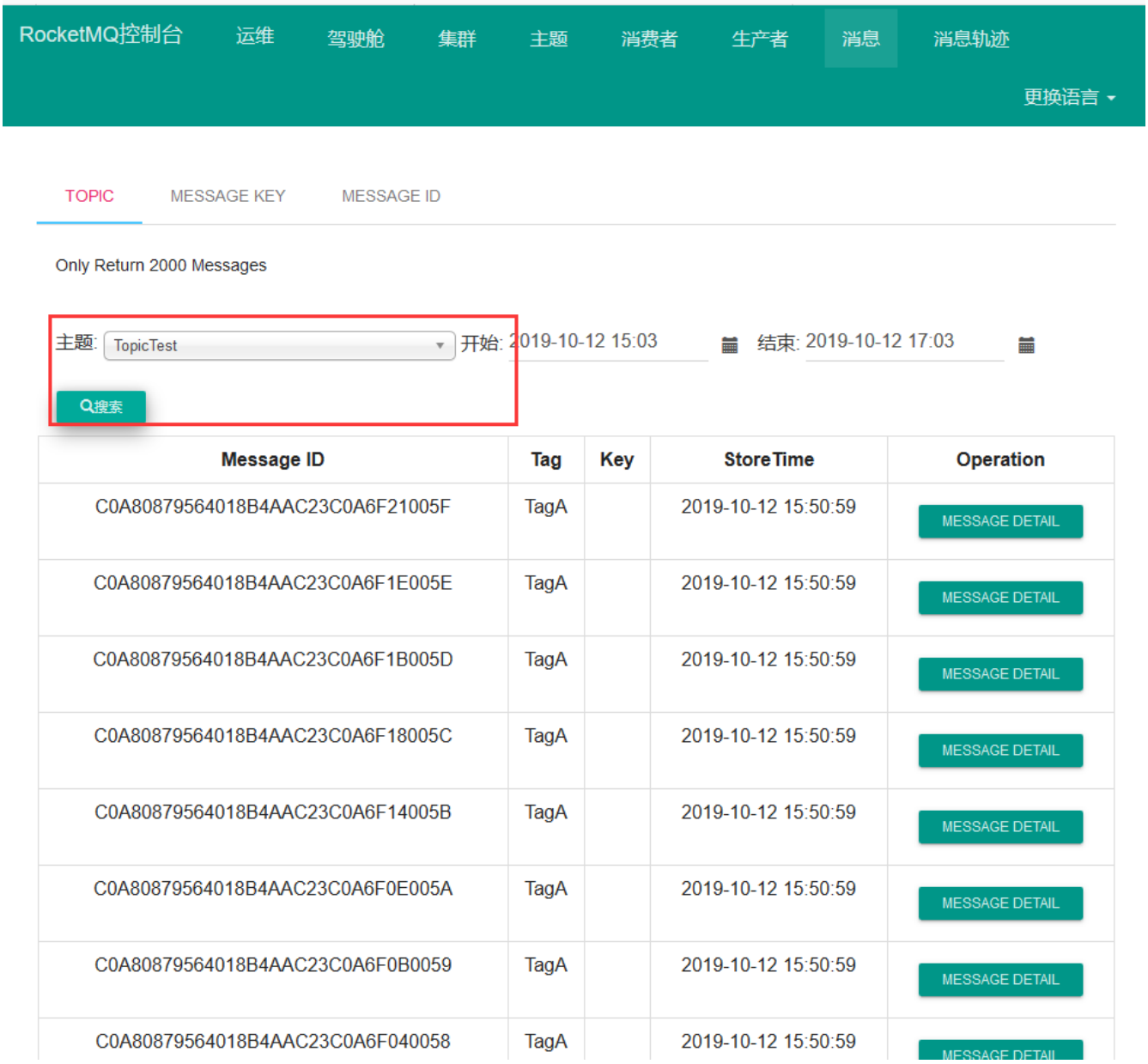
```
public class SyncProducer {

    public static void main(String[] args) throws MQClientException,
UnsupportedEncodingException, RemotingException, InterruptedException,
MQBrokerException {
        DefaultMQProducer producer = new
DefaultMQProducer("pleaserenameuniquegroupname");
        // 192.168.9.110:9876;192.168.9.111:9876
        producer.setNamesrvAddr("192.168.9.110:9876");
        producer.start();
        for (int i = 0; i < 100; i++) {
            Message msg = new Message("TopicTest",
                    "TagA",
                    ("hello rocketMq" + i).getBytes(RemotingHelper.DEFAULT_CHARSET));
            SendResult sendResult = producer.send(msg);
            System.out.println("cc" + sendResult);
        }
        producer.shutdown();
    }


}
```

出现的问题：

> sendDefaultImpl call timeout:因为broker部署在虚拟机，并且虚拟双网卡，client无法正常连接服务端。
>
> 在配置文件中指定主机Ip
>
> brokerIP1=192.168.0.110

```
# See the License for the specific language governing permissions and
# limitations under the License.
namesrvAddr=192.168.9.110:9876;192.168.9.111:9876
brokerClusterName=DefaultCluster
brokerName=broker-b
brokerId=0
deleteWhen=04
fileReservedTime=48
brokerRole=SYNC_MASTER
flushDiskType=ASYNC_FLUSH
listenPort=10911
storePathRootDir=/home/rocketmq/store-b
brokerIP1=192.168.9.111
~
~
~
~
```

运行后：



常用命令

MQAdmin自带的命令行管理工具，在bin目录下

消费者

消费者可分为两种类型，一种是DefaultMQPushConsumer：由系统控制读取操作，收到消息后自动调用传入处理方法来处理，一种是DefaultMQPullConsumer：读取的大部分的功能能由使用者自主控制。

DefaultMQPushConsumer：

主要是设置好各种传入处理消息函数，自动保存Offset，而且加入新的DefaultMQPushConsumer后会做负载均衡。

```
consumer.setMessageModel(MessageModel.CLUSTERING);
```

设置消息模式：Clustering模式：同一个ConsumerGroup(GroupName相同)里的每个Consumer只消费所订阅消息的一部分。同一个ConsumerGoup里所有的Consumer消费的内容合起来才是所订阅Topic内容的整体，从而达到负债均衡的目的。 Broadcasting：广播模式，同一个ConsumerGroup里的每个Consumer都能消费所订阅Topic的全部消息。一个消息被多次分发，被多个Consumer消费

```
consumer.subscribe("TopicTest", "*");//*换成 "tag1"||"tag2"||"tag3"

consumer.setConsumeMessageBatchMaxSize(size) // 设置消费者每次拉取的大小
```

消费指定Topic 下的Tag

DefaultMQPushConsumer处理流程：

> PullRequest:在PushConsumer 中使用PullRequest，通过长轮询方式达到push效果的方法，长轮询方式既有pull的优点，又兼有push的实时性。

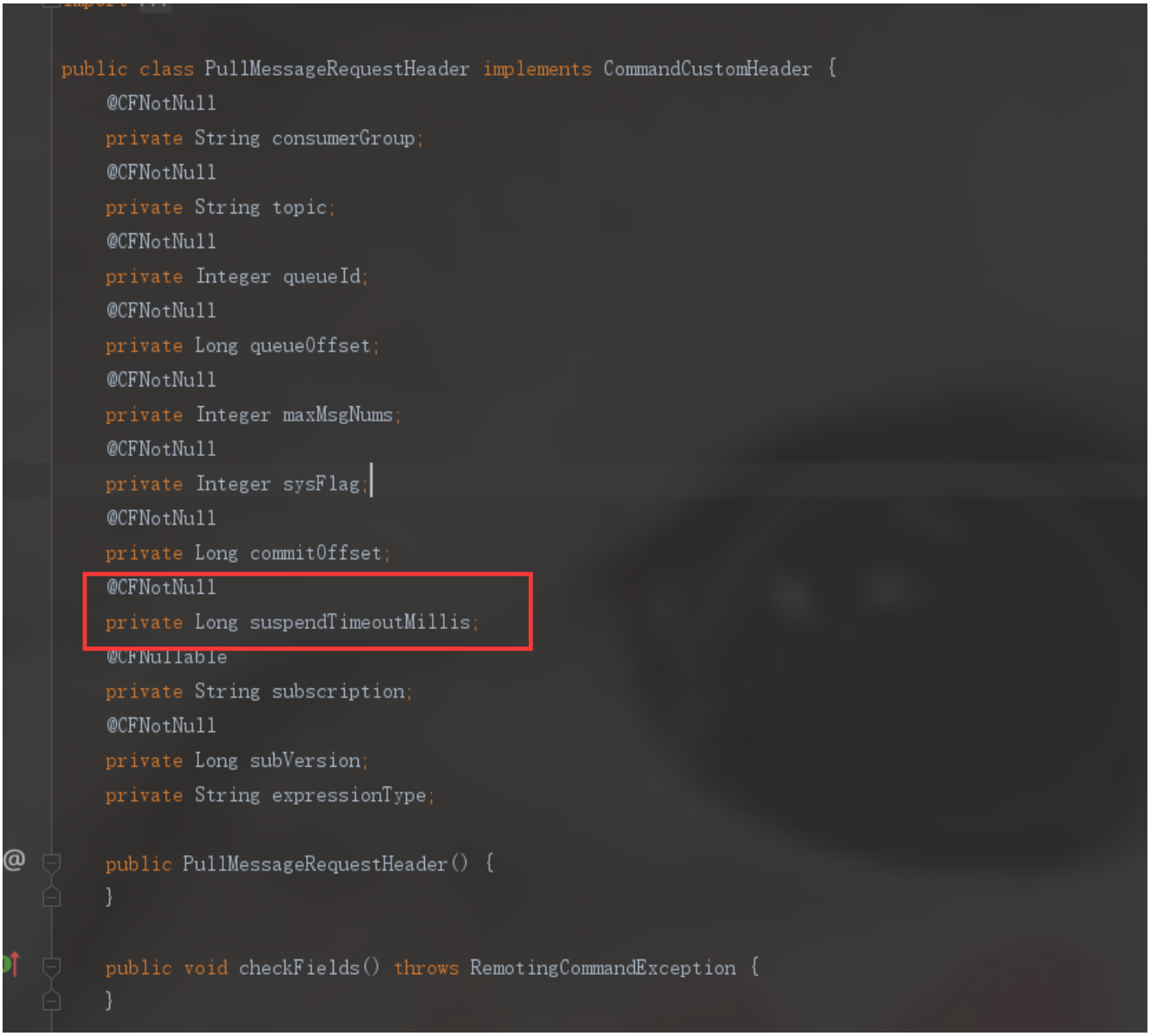
push 方式：是Server端收到消息后，主动把消息推送给Client，实时性高。

弊端：①：加大Server端的工作量，影响Server的性能。②：client不同的处理能力，导致client的状态不受Server控制。

pull方式：是Client循环的从Server拉取消息，client想拉就拉。

弊端：循环的时间间隔不好设定，间隔太短就处于一个“忙等”的状态，浪费资源；每个pull的时间间隔过长，Server有消息的时候，有可能没有及时处理。

> "长轮询"的方式通过client端和server端的配合，达到拥有pull的优点，又能达到保证实时性的目的。



```java
public class PullMessageRequestHeader implements CommandCustomHeader {
    @CFNotNull
    private String consumerGroup;
    @CFNotNull
    private String topic;
    @CFNotNull
    private Integer queueId;
    @CFNotNull
    private Long queueOffset;
    @CFNotNull
    private Integer maxMsgNums;
    @CFNotNull
    private Integer sysFlag;
    @CFNotNull
    private Long commitOffset;
    @CFNotNull
    private Long suspendTimeoutMillis;
    @CFNullable
    private String subscription;
    @CFNotNull
    private Long subVersion;
    private String expressionType;

    public PullMessageRequestHeader() {
    }

    public void checkFields() throws RemotingCommandException {
    }
```

requestHeader.serSuspendTimeoutMillis(brokerSuspendMaxTimeMillis) 作用是设置Broker最长的阻塞时间(默认是15s)，在没有新消息的时候才阻塞，有消息会立刻返回

DefaultMQPushConsumer的流量控制

PushConsumer有个线程池，消息处理的逻辑在各个线程里同时执行。

但是Pull获得的消息，如果直接提交到线程池里去执行，很难监控和控制。RocketMQ定义了一个快照类ProcessQueue来解决这些问题，在PushConsumer运行的时候，每个Message Queue都会有个对应的ProcessQueue对象，保存了这个MessageQueue消息处理的状态快照。

PushConsumer会判断获取但还未处理的消息个数、消息总大小，Offset的跨度，任何一个值超过设计的大小就隔一段时间再拉取消息，从而达到流量控制的目的。ProcessQueue还可以辅助实现顺序消费的逻辑。

DefaultMQPullConsumer

使用DefaultMQPullConsumer像使用DefaultMQPushConsumer一样需要设置各种参数，写出来消息的函数，同时还需要做额外的事情。PullConsumer主要体现在Pull上

```java
// 创建一个消费者
private final DefaultMQPullConsumer pullConsumer = new
DefaultMQPullConsumer("TestRocketMQPushConsumer2");
// 创建一个快照 来保存消息的Offset
private final Map<MessageQueue, Long> OFFSE_TABLE = new HashMap<>();

pullConsumer.start();
// 获得MessageQueue
Set<MessageQueue> mqs =pullConsumer.fetchSubscribeMessageQueues("TopicTest");
// 遍历一遍得到的消息集合
for (MessageQueue mq : mqs) {
    // 获得Offset
    long offset = pullConsumer.fetchConsumeOffset(mq, true);
    log.info("####Consumer from the queue : " + mq);
    SINGLE_MQ:
    while (true) {
        // 获得pull的结果集
         PullResult pullResult = pullConsumer.pullBlockIfNotFound(mq, null,
                           getMessageQueueOffset(mq), 32);
         log.info("####pullResult:" + pullResult);
        // 将得到的mq存入map
         putMessageQueueOffset(mq, pullResult.getNextBeginOffset());
         switch (pullResult.getPullStatus()) {
              case FOUND:
                   break;
                // 没有新消息，那么再次遍历
              case NO_NEW_MSG:
                   break SINGLE_MQ;
              case NO_MATCHED_MSG:
                   break;
              case OFFSET_ILLEGAL:
                   break;
              default:
                  break;
            }
      }
}
```

Consumer的启动和关闭

消息队列一般是提供一个不间断的持续性服务，在Consumer使用过程中，如何才能优雅的启动和关闭，并且不漏掉或者重复消费消息。

PullConsumer来说，主动权比较高，保证数据准确，那么在退出的时候要把Offset写入磁盘保存，下次加载的时候读取。

DefaultMQPushConsumer退出，要调用shutdown函数，以便释放资源、保存Offset等。这个调用要放到应用的退出逻辑中。

> PushConsumer在启动的时候，会做各种配置检查，然后连接NameServer获取Topic信息。但是如果遇到异常，比如无法连接NameServer，程序仍然不报错，但是会有警告（日志有WARN信息）。因为分布式系统中，某个机器出现问题，整体服务依然可用。所以PushConsumer被设计成当发现某个连接异常不立刻退出，而且不断尝试重连。
>
> 如果需要在DefaultMQPushConsumer启动的时候，及时暴露配置问题。那么在Consumer.start()语句后调用：consumer.fetchSubscribeMessageQueues("Topic"),如果配置信息不准确的话，这个语句会报MQClientException异常

不同类型的消费者

DefaultMQProducer

生产者向消息队列写入消息，不同的业务场景需要生产者采用不同的策略，比如：同步发送、延迟发送、发送事务消息等。

> - 设置InstanceName , producer.setInstanceName 当一个Jvm需要启动多个Producer的时候，通过设置不同的InstanceName区分，不设置的话系统使用默认名称“DEFAULT”。
> - producer.setRetryTimesWhenSendAsyncFailed 表示发送失败重试次数，当网络异常的时候，这个次数影响消息的重复投递次数。

消息的发送有同步和异步的形式，消息发送的返回状态有4种：FLUSH_DISK_TIMEOUT(没有在规定的时间完成刷盘。需要将Broker的策略设置成SYNC_FLUSH才会报这个错误)、FLUSH_SLAVE_TIMEOUT(在主备方式下，并且Broker被设置成SYNC_MASTER方式，没有在指定时间完成主从同步)、SLAVE_NOT_AVAILABLE（在主备方式下，并且Broker被设置成SYNC_MASTER，但是没有找到被配置成Slave的Broker）、SEND_OK(以上都没有出现问题)。

发送延迟消息

在创建Message对象的时候，setDelayTimeLevel(int)。不支持设置任意值。(1s/ 5s/1Os/30s/Im/ 2m/3m/4m/5m/6m/7m/8m/9m/ Om/20m/ 30m/1 h/2h).

自定义消息发送规则

一个Topic会有多个Message Queue，如果使用Producer的默认配置，这个Producer会流向各个Message Queue发送消息。Consumer在消费的时候，会根据负载均衡策略，消费被分配到的Message Queue，如果不经过特定的设置，某条消息被发往哪个Message Queue，被哪个

Consumer消费是未知的。

为了解决这个问题，可以使用MessageQueueSelector

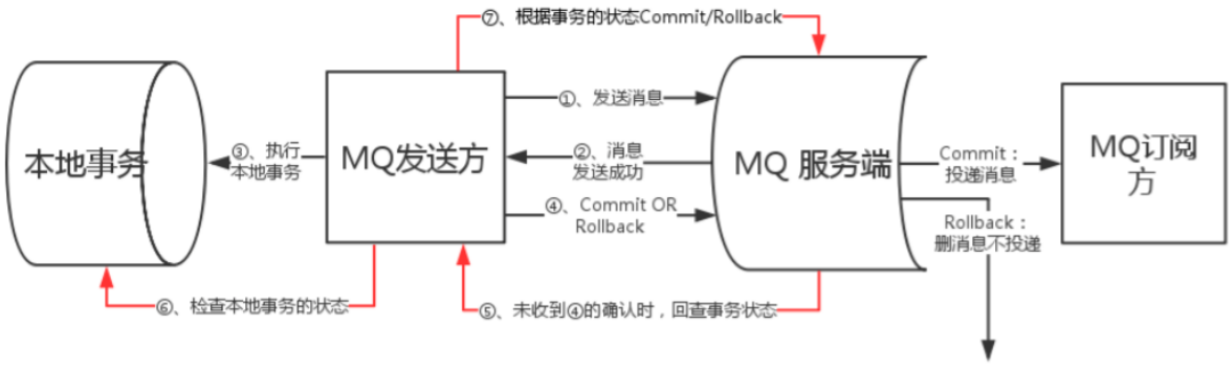
```
public class OrderMessageQueueSelector implements MessageQueueSelector {
    @Override
    public MessageQueue select(List<MessageQueue> list, Message message, Object o) {
        int id = Integer.parseInt(o.toString());
        int idMainIndex = id / 100;
        int size = list.size();
        int index = idMainIndex % size;
        return list.get(index);
    }
}
```

发消息的时候，把MessageQueueSelect的对象作为参数，使用public SendResult send(Message msg,MessageQueueSelector,arg)。

事务

同时成功或者同时失败，TransactionMQProducer继承与DefaultMQProducer。

A->B转账100元，例子如下：



A向B转账，那么首先A向B发送B要增加100的消息，然后自己减100，自己减100成功后，服务器，向B暴露消息，让B能够收到消息，最后处理B增加的逻辑。

步骤如上图：

①：A发送消息：“B add 100 ”到服务器。

②：服务器反馈是否收到消息。

③：A执行本地事务: "A minus 100"。

④：如果事务成功->commit 消息，失败->rollback消息

⑤-⑦：是第④步骤没有收到消息的一个回查情况。

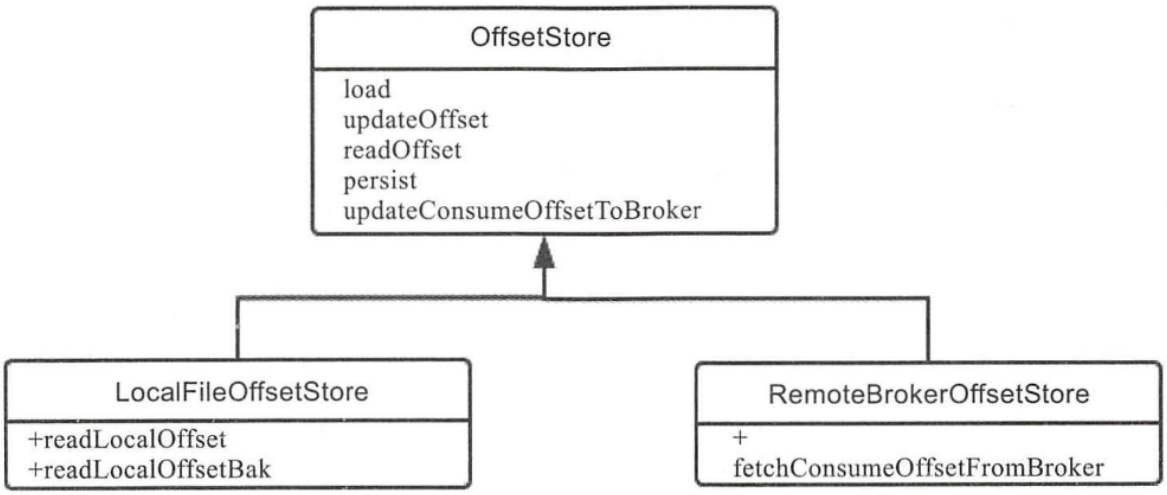
当服务器收到消息，Rollback类型的消息不进行投递，Commit才进行投递。

然后B消费消息，进行相应的事务。整套流程，两边的数据最终会一致性，而不是强一致性。

如何存储队列位置信息

Offset：一个Topic会有多个Message Queue，Offset就是在某个队列的位置：

在CLUSTERING模式下：每个Consumer只消费消息的一部分，之间互不干扰，这种情况，Broker端存储和控制Offset的值，使用RemoteBrokerOffserStore



在DefaultMQPushConsumer里的BROADCASTING(广播模式)下，每个Consumer都会接收到这个Topic的全部消息，各个Consumer间相互没有干扰，RocketMQ使用LocalFileOffsetStore,把Offset存到本地。

> Offsetstore的内容
>
> "OffsetTable":{{"brokerNarne":"localhost","Queue Id":1,"Topic":"brokerl") : 1 , {"brokerNarne""local host ","Queue Id":2,"Topic":"brokerl"):2 , { "brokerNarne":"localhost","Queue Id": 0, "Topic " : "brokerl") : 3))

```
@Slf4j
public class LocalOffsetStoreExt {
    private final String groupName;

    private final String storePath;

    private ConcurrentMap<MessageQueue, AtomicLong> offsetTable = new
ConcurrentHashMap<>();

    public LocalOffsetStoreExt(String groupName, String storePath) {
        this.groupName = groupName;
        this.storePath = storePath;
    }

    /**
     *  根据 storePath 加载本地的所有MessageQueue的offset
     * */
    public void load() {
        OffsetSerializeWrapper offsetSerializeWrapper = this.readLocalOffset();
        if (offsetSerializeWrapper != null && offsetSerializeWrapper.getOffsetTable()
!= null) {
            offsetTable.putAll(offsetSerializeWrapper.getOffsetTable());
            for (MessageQueue mq : offsetSerializeWrapper.getOffsetTable().keySet()) {
                AtomicLong offset = offsetSerializeWrapper.getOffsetTable().get(mq);
                log.info("load consumer's offset ,{}{}{}", this.groupName, mq,
offset.get());
            }
        }
```

```
26      }
27
28      /**
29       *  更新offset
30       * */
31      public void updateOffset(MessageQueue mq, long offset) {
32          if (mq != null) {
33              AtomicLong offsetOld = this.offsetTable.get(mq);
34              if (null == offsetOld) {
35                  this.offsetTable.putIfAbsent(mq, new AtomicLong(offset));
36              } else {
37                  offsetOld.set(offset);
38              }
39          }
40      }
41
42      /**
43       *  根据MessageQueue 获取Offset
44       * */
45      public long readOffset(final MessageQueue mq) {
46          if (mq != null) {
47              AtomicLong offset = this.offsetTable.get(mq);
48              if (offset != null) {
49                  return offset.get();
50              }
51          }
52          return 0;
53      }
54
55      /**
56       *  将新的偏移量持久化到本地
57       * */
58      public void persistAll(Set<MessageQueue> mqs) {
59          if (null == mqs || mqs.isEmpty()) {
60              return;
61          }
62          OffsetSerializeWrapper offsetSerializeWrapper = new OffsetSerializeWrapper();
63
64          for (Map.Entry<MessageQueue, AtomicLong> entry : this.offsetTable.entrySet())
    {
65              if (mqs.contains(entry.getKey())) {
66                  AtomicLong offset = entry.getValue();
67                  offsetSerializeWrapper.getOffsetTable().put(entry.getKey(), offset);
68              }
69          }
70          String jsonString = offsetSerializeWrapper.toJson(true);
71          if (jsonString != null) {
72              try {
73                  MixAll.string2File(jsonString,this.storePath);
74              } catch (IOException e) {
75                  e.printStackTrace();
76              }
77          }
```

```
78
79      }
80
81
82      /**
83       * 获取本地的offset
84       */
85      private OffsetSerializeWrapper readLocalOffset() {
86          String content = null;
87          try {
88              content = MixAll.file2String(this.storePath);
89          } catch (IOException e) {
90              e.printStackTrace();
91          }
92          if (null == content || content.length() == 0) {
93              return null;
94          } else {
95              return OffsetSerializeWrapper.fromJson(content,
    OffsetSerializeWrapper.class);
96          }
97      }
98  }
99
```

设置Consumer读取消息的位置

```
1  setConsumerFromWhere()// 设置从什么地方开始读
   ConsumerFromWhere.CONSUMER_FROM_FIRST_OFFSET。CONSUMER_FROM_LAST_OFFSET、
   CONSUMER_FROM_TIMESTAMP(时间戳精确到秒)
```

> 设置读取位置不是每次有效，优先级低于Offset Store

分布式消息队列的协作者

NameServer

是mq的状态服务器，集群的各个组件通过它来了解全局的信息。同时，每个角色机器都要定期向NameServer上报自己的状态，超时不上报的话，NameServer就会认为该不可用。

NameServer可以部署多个，相互之间独立，其他角色同时向多个NameServer上报状态信息，从而达到热备份的目的。

消息队列的核心机制

Broker是RocketMQ的核心，大部分"重量级"工作都是由Broker完成的，包括接收Producer发过来的消息、处理Consumer的消息请求、消息的持久化存储、消息的HA机制以及服务端过滤功能等。

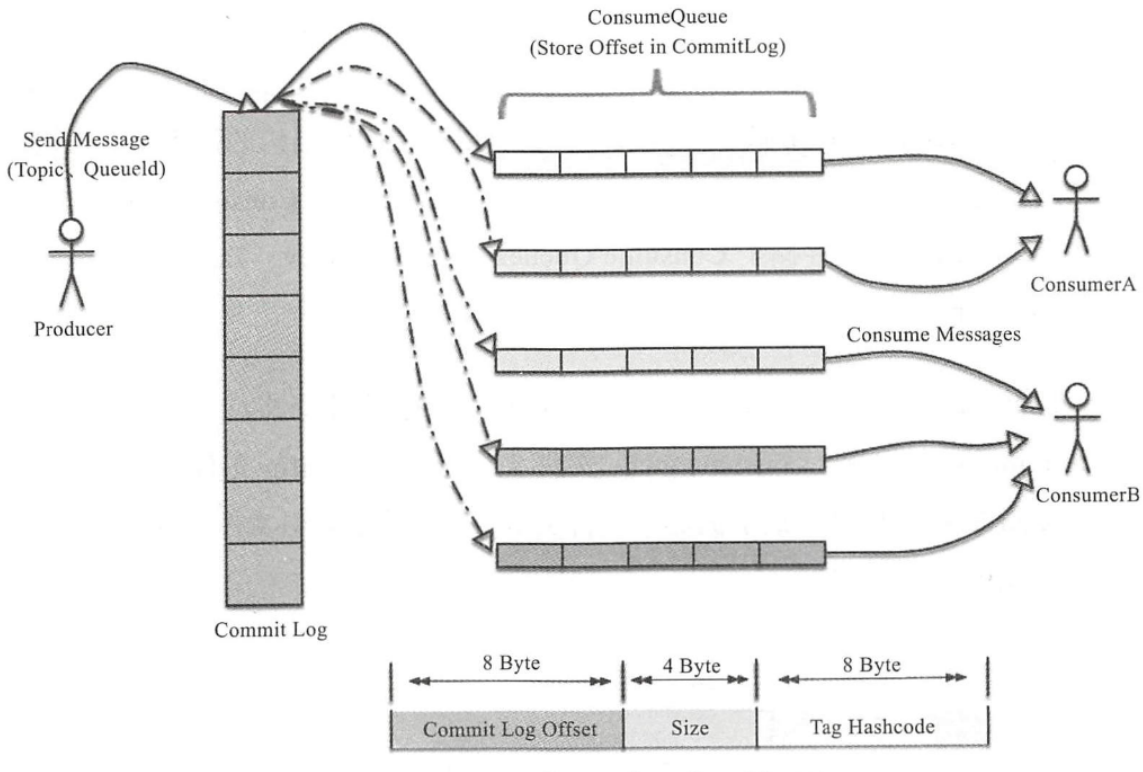
> 磁盘的顺序读写可达： 600MB/S

> 磁盘的随机读写只有：100KB/s

Linux系统分为"用户态"和"内核态"，文件操作和网络操作都需要涉及这两种形态的切换，免不了进行数据复制，一台服务器把本机磁盘文件的内容发送到客户端，一般两个步骤：read（file,temp_buf,len）、write(socket,temp_buf,len)

> 这两个步骤实际进行了4次数据复制。
>
> ①：磁盘 copy -> 内核态内存
>
> ②：内核态内存复制 copy->用户态内存(完成了read)
>
> ③：用户态内存 copy-> 网络驱动的内核态内存
>
> ④：网络驱动的内核态内存 copy-> 网卡(完成write)

RocketMQ消息的存储是由ConsumerQueue和CommitLog配合完成的。



> 消息真正的物理存储文件是CommitLog
>
> ConsumerQueue是消息的逻辑队列，类似数据库索引文件，存储的是指向物理存储的地址。每个Topic下的每个MessageQueue都有一个对应的ConsumerQueue文件。
>
> ConsumerQueue地址：$ {$storeRoot} \consumequeue\$ {topicName}\ $ { queueId}\ $ {fileName}

CommitLog是以物理文件的方式存放，每台Broker上的CommitLog被本机器所有的ConsumerQueue共享，文件地址：${user.home}\store\$ { commitlog}\${fileName}.在CommitLog中，一个消息的存储长度是不固定的，RocketMQ采取一些机制，尽量向CommitLog中顺序写，但是随机读。ConsumeQueue的内容也会被写到磁盘里作持久存储。

> 优点：
>
> ①：CommitLog顺序写，效率高

> ②：虽然随机读，但是利用操作系统的pagecache机制，可以批量的从磁盘读取，作为cache存到内存中，加速后续的读取速度。
>
> ③：为了保证完全的顺序写，需要ConsumerQueue这个中间结构，因为ConsumerQueue只存偏移量信息，尺寸有限，在实际情况中，大部分的ConsumeQueue能够被完全读入内存，故这个中间结构的操作速度很快，可以认为是内存读取的速度。此外为了保证CommitLog和ConsumeQueue的一致性，CommitLog里存储Consume Queues、Message Key、Tag等所有信息，即使ConsumeQueue丢失，也可以通过commitLog完全恢复出来。

高可用机制

RocketMQ分布式集群是通过Master和Slave的配合达到高可用性的。

Master和Slave的区别：在Broker的配置文件中，参数brokerId的值为0，表明这个Broker是Master，大于0表明这个Broker是Slave，同时brokerRole参数也会说明该Broker是Master还是Slave。

> Master : 支持读和写。
>
> Slave：仅支持读。
>
> 也就是Producer只能和Master角色的Broker连接写入消息，Comsumer则是可以连接Master和Slave来读。
>
> 消费者的高可用：Consumer不需要设置，是从Master读还是从Slave读，当Master不可用或者繁忙时候，Consumer会自动切换到Slave读。
>
> 生产者的高可用：创建Topic的时候，把Topic的多个Message Queue创建在多个Broker上(相同Broker名称，不同brokerId的机器组成一个Broker组)，这样当一个Broker组的Master不可用，还有其他的Master任何用。
> ps： 或者，如果机器资源不足，将Slave转成Master，目前RocketMq不支持自动转，需要手动更改Slave的配置，再重新启动

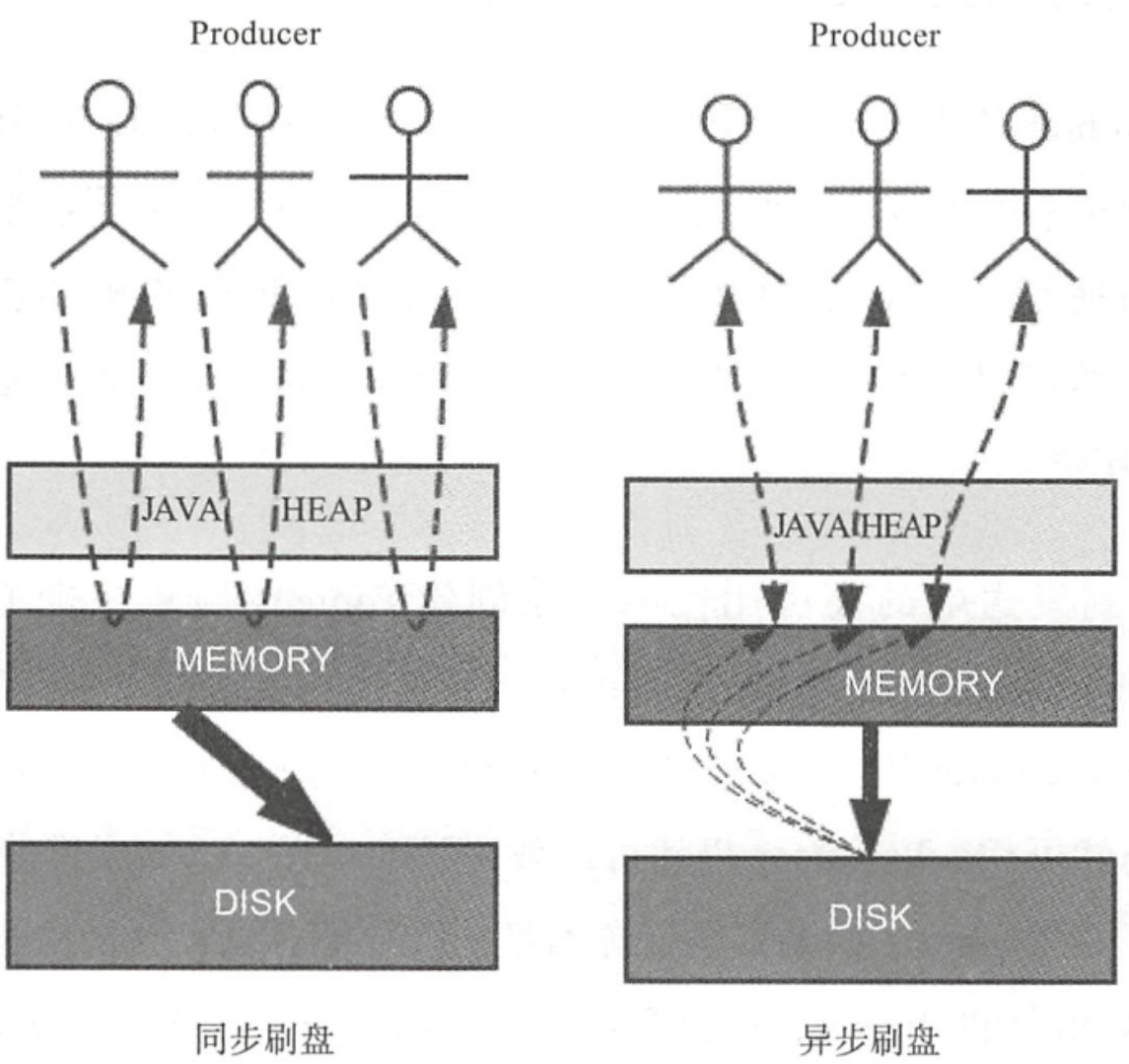
同步刷盘和异步刷盘

目的

消息存储到磁盘上，要保证断电后能够恢复、又可以让存储的消息超过内存的限制。

异步刷盘：在返回写成功状态的时候，消息可能只是被写入了内存的PAGECACHE，写操作的返回快，吞吐量大；当内存里的消息量积累到一定程度的时候，统一触发写磁盘动作，快速写入。

同步刷盘：在返回写成功状态时，消息已经被写入磁盘。具体流程是，消息写入内存的PAGECACHE后，立刻通知刷盘线程刷盘，然后等待刷盘完成，刷盘线程执行完成后唤醒等待的线程，返回消息写成功的状态。



> 在Broker配置文件里的flushDiskType参数设置，值是 SYNC_FLUSH、ASYNC_FLUSH。

同步复制和异步复制

Broker组有Master和Slave，消息需要从Master复制到Slave上，有同步和异步两种复制方式。

同步复制方式：Master和Slave均写成功后才反馈给客户端写成功状态。

异步复制方式：只要Master写成功即可反馈给客户端写成功状态。

> 优劣：
>
> 异步复制，系统拥有较低的延迟和较高的吞吐量，但是如果Master出现故障，有些数据因为没有被写入Slave，有可能会丢失。在同步复制的方式下，如果Master出故障，Slave上有全部的备份数据，容易恢复，但是同步复制会增大数据写入延迟，降低系统吞吐量。

> 在Broker配置文件里的brokerRole进行配置：ASYNC_MASTER、SYNC_MASTER、SLAVE三个值中的一个。

通常情况：刷盘：异步刷盘（ASYNC_FLUSH），复制：同步复制（SYNC_MASTER）,这样尽管有一台机器出故障，仍能保证数据不丢失。

可靠性优先的使用场景

顺序消息

就是发送按照顺序来，消费也按照顺序去。

> 一个Topic默认有8个写队列，消费者也会有多个Consumer，每个Consumer可能会启动多个线程并行处理，所以要想有序。那么写队列只能设置为1，Consumer的并发也只能是1。

实现消息有序，就是Producer发送消息的时候，只往一个MessageQueue中发送，Consumer一个一个的处理即可。

实现生产者顺序写主要是实现： MessageQueueSelector这个接口。

```
public class OrderMessageQueueSelector implements MessageQueueSelector {
    @Override
    public MessageQueue select(List<MessageQueue> list, Message message, Object o) {
        int id = Integer.parseInt(o.toString());
        int idMainIndex = id / 100;
        int size = list.size();
        int index = idMainIndex % size;
        return list.get(index);
    }
}

```

实现消费者有序消费，实现的是MessageListenerOrderly 而不是：MessageListenerConcurrently或者MessageListener了。并且仍然可以使用：setConsumerThreadMin、setConsumerThreadMax(设置线程数)、setPullBatchSize(一次从broker拉取的消息数量，默认32)、setConsumerMessageBatchMaxSize(Consumer的Executor(也就是调用MessageListener处理的地方)一次传入的消息数List msgs 这个链表的最大长度)。

> MessageListenerOrderly 并不是简单的禁止并发处理。在MessageListenerOrderly的实现中，为每个Consumer Queue加个锁，消费每个消息前，需要先获得这个消息对应的Consumer Queue所对应的锁，这样保证了同一时间，同一个Consumer Queue的消息不被并发消费，但不同Consumer Queue的消息可以并发处理。

消息重复问题

解决消息重复的方法：

①：保证消费逻辑的幂等性(多次调用和第一次调用效果相同)

②：维护一个已消费消息的记录，消费前查询这个消息是否被消费过。

> 消息重复出现的原因：
>
> 消息重复一般情况不会发生，但是如果消息量巨大，网络有波动，消息重复是个大概率事件。比如Producer的函数setRetryTimesWhenSendFailed，设置在同步方式下自动重试的次数，默认是2，这样当第一次发送消息时，Broker端接收到了消息但是没有正确返回发送成功的状态，就造成了消息重复。

动态增减机器

一个消息队列集群由多台机器组成，持续稳定的提供服务，因为业务需求或者硬件故障，经常需要增加或减少各个角色的机器。

动态增减NameServer

NameServer功能： ①：各个Broker定期上报自己状态信息到NameServer；②：包括Producer、Consumer 以及命令行工具，通过NameServer获取最新的状态信息。在启动Broker之前，必须告诉他们NameServer的地址，为提高可靠性，建议启动多个NameServer(占用资源不多)。

设置NameServer地址：

①：代码设置，比如：`consumer.setNamesrvAddr(namesrvAddr)` `producer.setNamesrvAddr(namesrvAddr)`。或者在mqadmin命令行工具中 通过 `-n name-server-ip1:port;name-server2-ip:port` .

②：使用Java启动参数设置，对应的option是rocketmq.namesrv.addr。

③：通过linuc环境变量设置，在启动前设置：NAMESRV_ADDR。

④：通过Http服务来设置，当上述方法都没有使用，程序会向一个Http地址发送获取NameServer地址，默认URL是`http://jmenv.tbsite. net: 8080/rocketmq/nsaddr`，通过rocketm.namesrv.domain参数来覆盖jemenv.tbsite.net;通过rocketmq.namesrv.domain.subgroup来覆盖nsaddr。

> 方法4，看起来繁琐，但他是唯一动态增加NameServer，无需要重启其他组件。使用这种方式后其他组件每隔2分钟请求一次该URL，获取最新的NameServer地址。

动态增加Broker

增加Broker：①：把新建的Topic指定到新的Broker机器上，均衡利用资源；②：通过updateTopic命令来更改现有的Topic配置，在新加的Broker上创建一个新的读写队列。

> Topic：TestTopic，因为数据量增加需要扩容，新的broker是192.168.0.1：9876，执行下面命令
>
> `sh ./bin/mqadmin updateTopic -b 192.168.0.1:9876 -t TestTopic -n 192.168.0.100:9876`,结果是在新增的Broker机器上，为TestTopic新创建8个读写队列。

减少Broker：

情况一：一个Topic只有一个Master Broker，关掉这个Broker，消息的发送必定受到影响，需要在停止这个Broker前，停止发送消息。

情况二：一个Topic有多个Master Broker，停了其中一个，对于是否会丢失消息，和Producer生产者的发送方式有关(同步发送，会有个重试逻辑，一个Broker停了，会向另一个Broker发消息，不会丢失消息。异步发送就会丢失)。对于异步发送来说，Producer.setRetryTimesWhenSendFailed不会生效。

> 如果Producer程序能够暂停，在有一个Master和一个Slave情况下也可以顺利切换。消费者不受到影响

linux的kill pid 可以正确的关闭Broker，但是不能使用kill-9.

linux 的kill 命令和，mqshutdown broker原理一样的

各种奇怪的问题

①：Broker正常关闭

可控情况，内存的数据不会丢失。如果重启过程中有持续运行的Consumer，Master机器出现故障后，Consumer会自动连接到对应的Slave，不会有消息丢失和偏差。当Master角色机器重启后，Consumer会重新连接到Master（在Master启动的时候，如果Consumer正在Slave消费消息，不要停止Consumer。）

其他硬件损坏，或者Broker异常Crash，属于不可控情况，为了避免这种问题，为了确保消息的高可靠。

应该设置为：

a：多Master、每个Master带有Slave

b：主从之间设置成SYNC_MASTER

c：Producer用同步方式写

d：刷盘策略设置成SYNC_FLUSH

消息优先级

严格来说，RocketMq没有优先级一说，如果想要优先处理某一类型消息，建议新开Topic或者Tag。

吞吐量优先

但是当服务器出现异常时会增大丢消息的概率。

提高吞吐量的方式：`①：过滤掉无关数据。②：提高Consumer的处理性能。③：提高Producer的生产效率。④：对服务器的CpU、网络进行调优。⑤：设置异步刷盘或者异步复制。⑥：负载均衡`

过滤

在Broker端进行消息过滤，可以减少无效消息发送到Consumer，少占用网络带宽提高吞吐量。消息过滤的三种方式：

a: 消息的Tag和Key。Tag每个消息只能有一个Tag，Key是唯一标识。两者的使用场景不同，Tag用在Consumer的代码中，用来进行服务器端的过滤，Key主要是用于通过命令行查询消息。

b: 用SQL表达式进行过滤。 在producer发送消息的时候，putUserProperty函数来增加多个自定义属性。

```
Producer 消息发送端

Message mqMsg = new Message(topic, tags, keys, JSON.toJSONBytes(data));
        mqMsg.putUserProperty("a",tags);


Consumer 消费者消费端
consumer.subscribe("TopicTest", MessageSelector.bySql(" a between 0 and 3"));
```

SQL支持如下语法：

```
数字对比 ： >、>=、<、<=、BETWEEN、=;
字符串对比： =、<>、IN
IS NULL or IS NOT NULL
逻辑符号 AND、OR、NOT
```

c.Filter Server方式过滤。实现MessageFilter接口，但是由于代码姿势不正确导致服务器挂掉或者大量占用Broker机器资源。

提高Consumer的处理能力

Consumer处理能力比较差时，导致消息积压。提高Consumer的几种方式：

a:提高消费并行度：加机器或者在机器中增加多个Consumer实例(Consumer的数量不要超过Topic下的Read Queue数量，超过的Consumer实例收不到消息。)或者在一个Consumer内增加并行度来提高吞吐量(setConsumerThreadMin,setConsumerThreadMax).

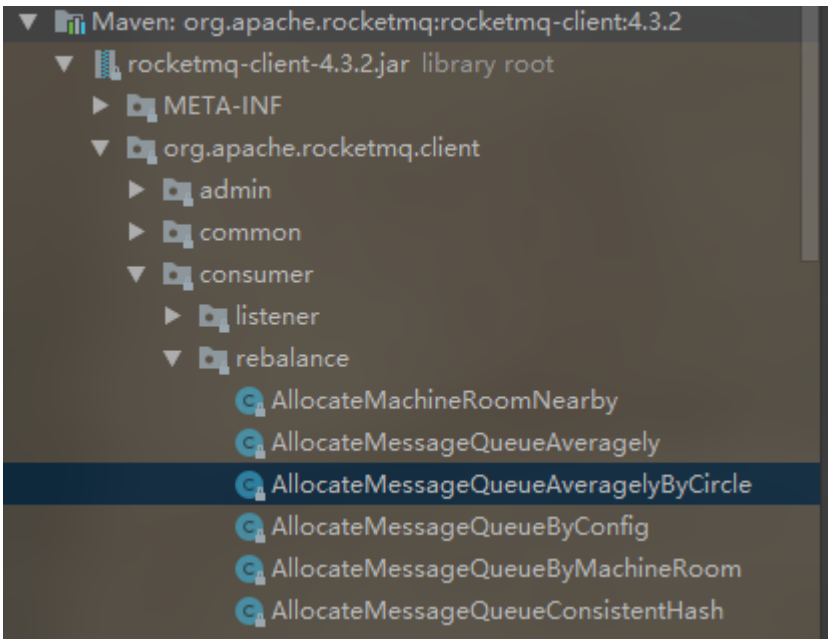
b：批量方式进行消费。设置 setConsumerMessageBatchMaxSize,一次性拉取处理的消息。

c：消息积压严重，跳过不重要的消息。

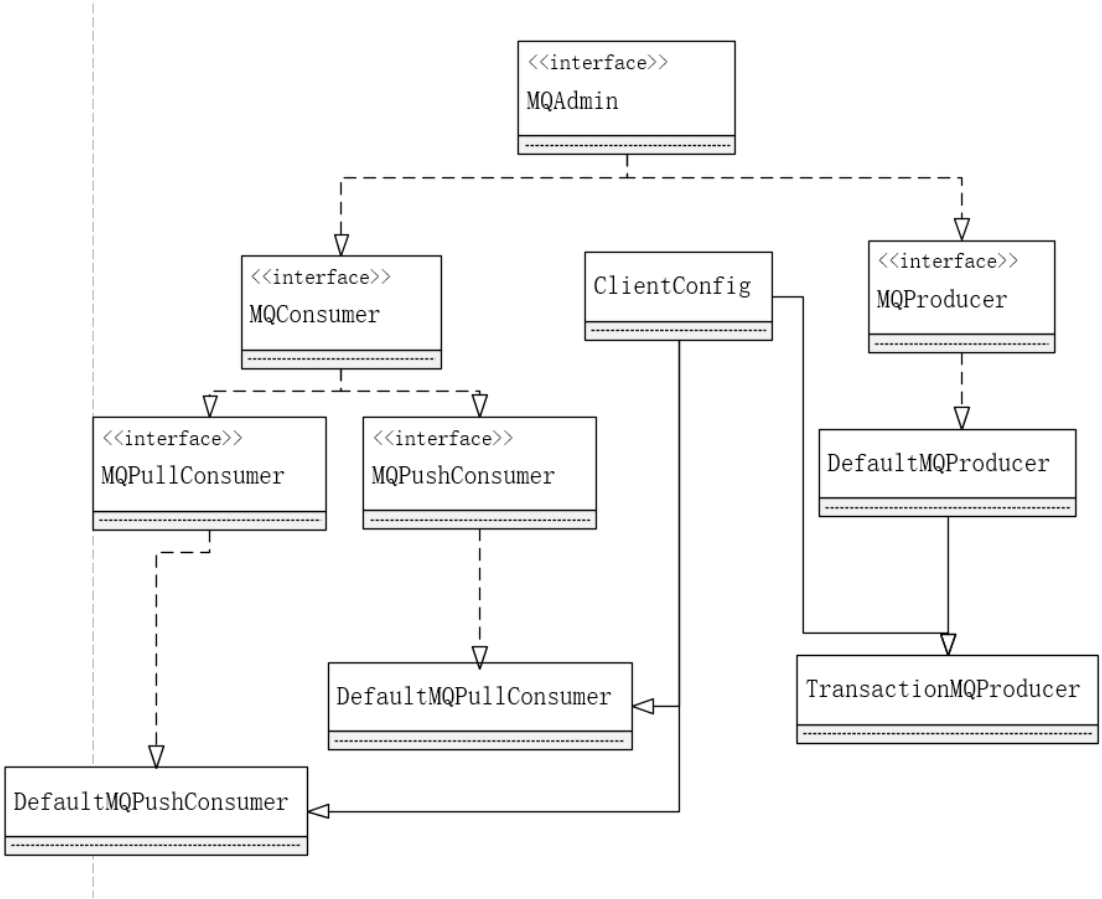
Consumer的负载均衡

负载均衡的前提条件：知道必要的全局信息，Consumer的个数等。

DefaultMQPushConsumer的负载均衡过程不需要使用者操心，客户端程序会自动处理，每个DefaultMQPushConsumer启动后，会马上触发一个doRebalance动作；而且在同一个ConsumerGroup里加入新的DefaultMQPushConsumer时，各个Consumer都会被触发doRebalance动作。负载均衡的算法有5种：



DefaultMQPullConsumer的负载均衡：Pull Consumer可以看到所有的Message Queue，而且从哪个MessageQueue读取消息，读取消息时的Offset都由使用者控制，使用者可以实现任何特殊方式的负载均衡。


<<interface>>
MQAdmin
<<interface>>
MQConsumer
ClientConfig
<<interface>>
MQProducer
<<interface>>
MQPullConsumer
<<interface>>
MQPushConsumer
DefaultMQProducer
DefaultMQPullConsumer
TransactionMQProducer
DefaultMQPushConsumer