**RocketMQ**

**第一节、安装以及角色**

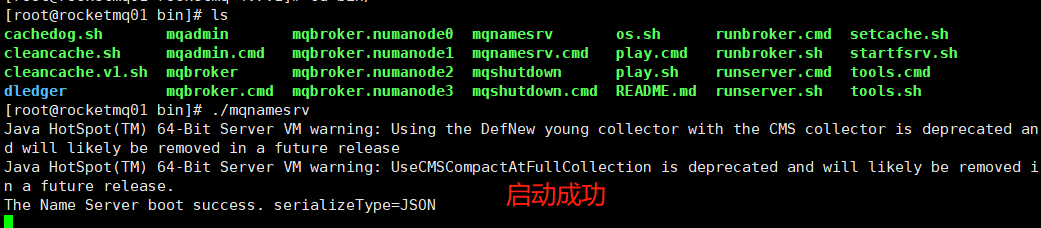
**1.1安装broker**

maven配置阿里云仓库

配置环境变量

export M2\_HOME=/root/soft/maven/apache-maven-3.6.3
export PATH=$PATH:$M2\_HOME/bin

./mqnamesrv 启动成功



**1.2安装web客户端**

rocketmq-externals-master下载扩展包 进入rocketmq-console目录
mvn clean package -Dmaven.test.skip=true 执行编译
执行target目录下rocketmq-console-ng-1.0.1.jar
java -jar rocketmq-console-ng-1.0.1.jar --rocketmq.config.namesrvAddr=127.0.0.1:9876
//临时关闭防火墙
service iptables stop

**2.rocketmq的角色**

**2.1 nameserver**

注册中心。

nameserver是服务发现者，集群中的各个角色（producer、broker、consumer等）都需要定时想nameserver上报自己的状态，以便互相发现彼此，超时不上报的话，nameserver会把它从列表中剔除。

nameserver是无状态的。可以部署多个，当多个nameserver存在的时候，其他角色同时向他们上报信息，以保证高可用。

nameServer因为是无状态的，所以集群间互不通信，没有主备的概念。

nameserver内存式存储，nameserver中的broker、topic等信息默认不会持久化。

**2.2broker**

Broker面向producer和consumer接受和发送消息。

1.向nameserver提交自己的信息。

2.是消息中间件的消息存储、转发服务器。

3.每个Broker节点，在启动时，都会遍历NameServer列表，与每个NameServer建立长连接，注册自己的信息，之后定时上报。

**2.3集群broker**

Broker高可用，可以配成Master/Slave结构，Master可写可读，Slave只可以读，Master将写入的数据同步给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多机负载，可以部署多个broker，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。

**2.4producer**

消息的生产者，通过nameserver集群中的其中一个节点（随机选择）建立长连接，获得Topic的路由信息，包括Topic下面有哪些Queue，这些Queue分布在哪些Broker上等，接下来向提供Topic服务的Master建立长连接，且定时向Master发送心跳。

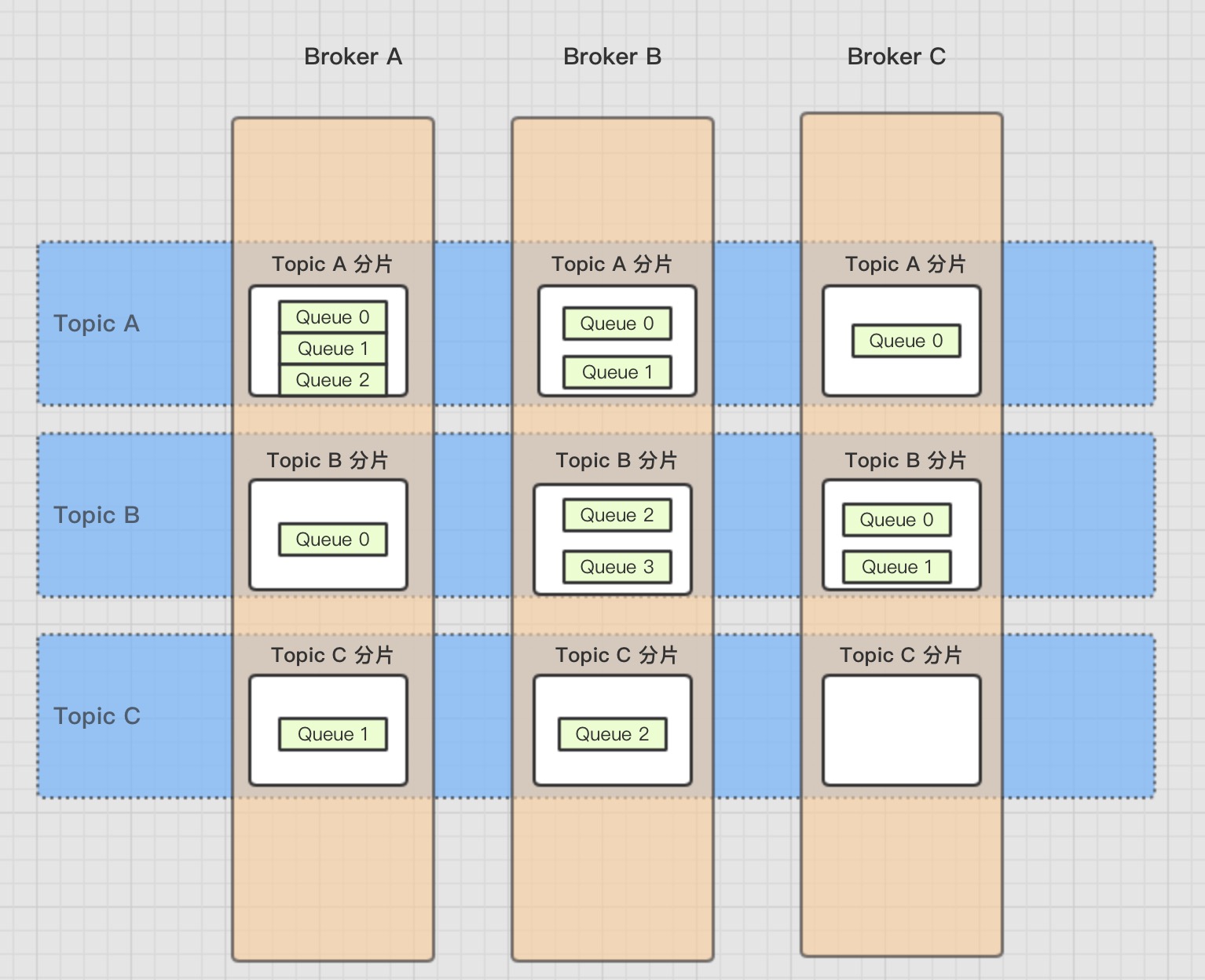
每次需要连接broker的时候。都要先连接nameserver，通过nameserver获取分配的broker地址。

**2.5consumer**

消息的消费者，通过NameServer集群获得Topic的路由信息，连接到对应的Broker上消费消息。注意，由于Master和Slave都可以读取消息，因此Consumer会与Master和Slave都建立连接。

**3.topic**

对比JSM中的Topic和Queue，Topic是一个逻辑上的概念，实际上Message是在每个Broker上以Queue的形式记录。topic下会有多个queue。



**第二节、API**

**1.代码demo**

public class Producer {
public static void main(String[] args) throws Exception {
// 设置producer组
DefaultMQProducer defaultMQProducer = new DefaultMQProducer("producerGroup01");
// 设置nameserver的服务地址
defaultMQProducer.setNamesrvAddr("192.168.73.200:9876");
// 启动
defaultMQProducer.start();
String message = "rocketmq的第一条消息，发送时间：" + System.currentTimeMillis();
// topic 消息将要发送到的地址
// body 消息中的具体数据
Message msg = new Message("topic01", message.getBytes());
// 同步消息发送，等到broker的ack之后才会结束
defaultMQProducer.send(msg);
// 可以批量发送
// 批量消息要求必要具有同一topic、相同消息配置
// 不支持延时消息
// 建议一个批量消息最好不要超过1MB大小
// 如果不确定是否超过限制，可以手动计算大小分批发送
List<Message> list = new ArrayList<>();
list.add(msg);
defaultMQProducer.send(list);
// 关闭连接
defaultMQProducer.shutdown();
}
}

public class Consumer {
public static void main(String[] args) throws Exception {
// 设置消费组
DefaultMQPushConsumer defaultMQPushConsumer = new DefaultMQPushConsumer("consumerGroup01");
defaultMQPushConsumer.setNamesrvAddr("192.168.73.200:9876");
// 设置消息过滤器，\*为不过啦
String messageSelector = "\*";
// 订阅topic的消息
defaultMQPushConsumer.subscribe("topic01", messageSelector);
defaultMQPushConsumer.registerMessageListener(new MessageListenerConcurrently() {
@Override
public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs, ConsumeConcurrentlyContext context) {
for (int i = 0; i < msgs.size(); i++) {
MessageExt messageExt = msgs.get(i);
String message = new String(messageExt.getBody());
System.out.println(message);
}
// 默认情况下 这条消息只会被 一个consumer 消费到 点对点
// message 状态修改
// ack
// 返回消费成功
return ConsumeConcurrentlyStatus.CONSUME\_SUCCESS;
}
});
defaultMQPushConsumer.start();
System.out.println("Consumer start...");
}
}

**2.批量发送**

批量消息要求必要具有同一topic、相同消息配置

不支持延时消息

建议一个批量消息最好不要超过1MB大小

如果不确定是否超过限制，可以手动计算大小分批发送

**3.异步发送**

public class AsyncProducer {
public static void main(String[] args) throws Exception {
// 设置producer组
DefaultMQProducer defaultMQProducer = new DefaultMQProducer("producerGroup01");
// 设置nameserver的服务地址
defaultMQProducer.setNamesrvAddr("192.168.73.200:9876");
// 启动
defaultMQProducer.start();
String message = "rocketmq的第一条消息，发送时间：" + System.currentTimeMillis();
// topic 消息将要发送到的地址
// body 消息中的具体数据
Message msg = new Message("topic01", message.getBytes());
// 只发送消息，不等待服务器响应，只发送请求不等待应答。此方式发送消息的过程耗时非常短，一般在微秒级别
// defaultMQProducer.sendOneway(msg);
// 失败重投次数
defaultMQProducer.setRetryTimesWhenSendAsyncFailed(2);
// 异步消息发送，不能在send方法外shutdown。因为ack是异步返回的。会有时间差
defaultMQProducer.send(msg, new SendCallback() {
@Override
public void onSuccess(SendResult sendResult) {
System.out.println("发送成功");
}
@Override
public void onException(Throwable e) {
System.out.println("发送失败");
}
});
}
}

**4.顺序消费**

队列先天支持FIFO模型，单一生产和消费者下只要保证使用`MessageListenerOrderly`监听器即可

顺序消费表示消息消费的顺序同生产者为每个消息队列发送的顺序一致，所以如果正在处理全局顺序是强制性的场景，需要确保使用的主题只有一个消息队列。

并行消费不再保证消息顺序，消费的最大并行数量受每个消费者客户端指定的线程池限制。

那么只要顺序的发送，再保证一个线程只去消费一个队列上的消息，那么他就是有序的。

跟普通消息相比，顺序消息的使用需要在producer的send()方法中添加MessageQueueSelector接口的实现类，并重写select选择使用的队列，因为顺序消息局部顺序，需要将所有消息指定发送到同一队列中。

**保证有序参与因素**

- FIFO

- 队列内保证有序

- 消费线程

for (int i = 0; i < 20; i++) {
Message msg = new Message("myTopic", null, "1234", ("测试" + i).getBytes());
defaultMQProducer.send(msg, new MessageQueueSelector() {
@Override
public MessageQueue select(List<MessageQueue> mqs, Message msg, Object arg) {
return mqs.get(0);
}
}, "");
}

defaultMQPushConsumer.registerMessageListener(new MessageListenerOrderly() {
@Override
public ConsumeOrderlyStatus consumeMessage(List<MessageExt> msgs, ConsumeOrderlyContext context) {
System.out.println(msgs.size());
for (MessageExt string : msgs) {
String str = new String(string.getBody());
System.out.println(str);
}
return ConsumeOrderlyStatus.SUCCESS;
}
});

**5.消费模式**

集群消费模式 广播消费模式

默认为集群消费模式

|  |
| --- |
| consumer.setMessageModel(MessageModel.BROADCASTING); |
| consumer.setMessageModel(MessageModel.CLUSTERING); |

集群消息是指集群化部署消费者

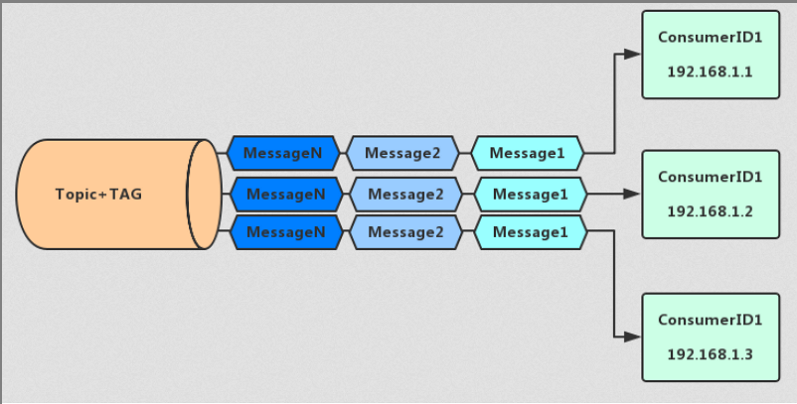
当使用集群消费模式时，MQ 认为任意一条消息只需要被集群内的任意一个消费者处理即可。

特点：

- 每条消息只需要被处理一次，broker只会把消息发送给消费集群中的一个消费者

- 在消息重投时，不能保证路由到同一台机器上

- 消费状态由broker维护



当使用广播消费模式时，MQ 会将每条消息推送给集群内所有注册过的客户端，保证消息至少被每台机器消费一次。

特点：

- 消费进度由consumer维护

- 保证每个消费者消费一次消息

- 消费失败的消息不会重投

**6.消息过滤**

同一组，不能关注不同的topic以及tag等的过滤，否则消息会乱

TAG

Message msg = new Message("TopicTest","TagA" ,("Hello RocketMQ " ).getBytes(RemotingHelper.DEFAULT\_CHARSET));

consumer中订阅TAG

consumer.subscribe("TopicTest", "TagA");// \* 代表订阅Topic下的所有消息

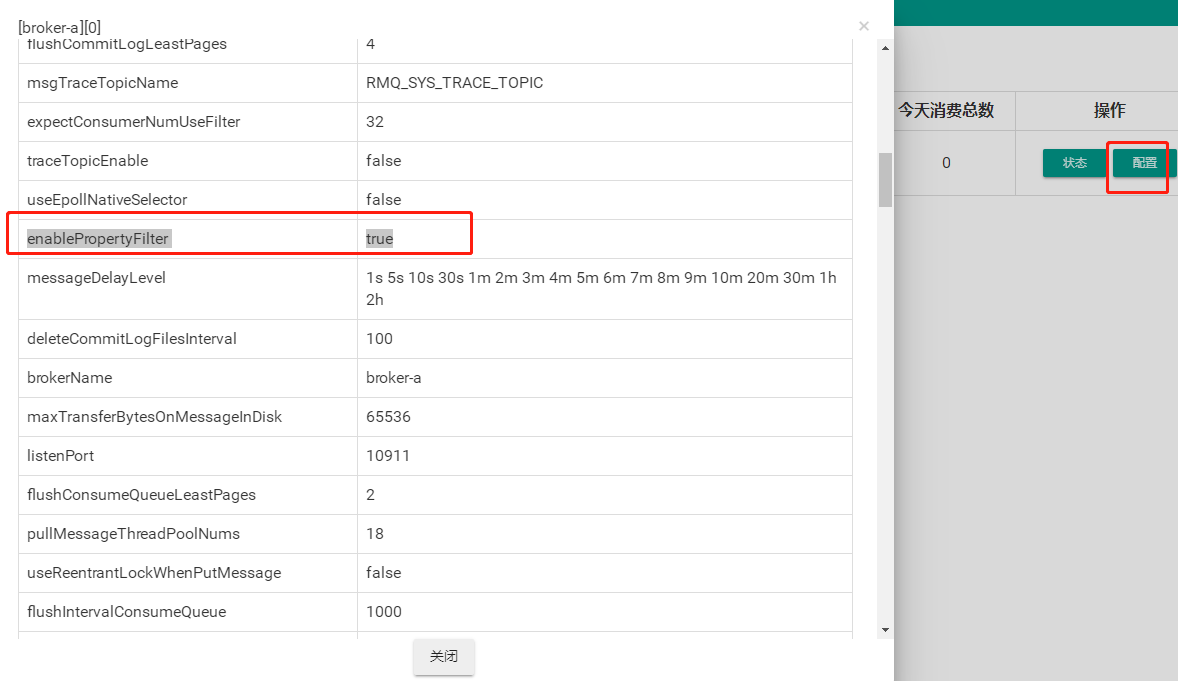
KEYS

开启sql过滤

在`broker.conf `中添加配置

enablePropertyFilter=true
启动broker 加载指定配置文件
../bin/mqbroker -n 192.168.150.113:9876 -c broker.conf

启动成功后



实例

发送
Message message = new Message("myTopic003", "TAG-B","KEY-xx",("xxooxx:" + i ).getBytes());
message.putUserProperty("age", String.valueOf(i));
接收
MessageSelector selector = MessageSelector.bySql("order > 5");
consumer.subscribe("xxoo3", selector);

|  |
| --- |
| 语法 |
| RocketMQ只定义了一些基本的语法来支持这个功能。 你也可以很容易地扩展它. |
| 1. 数字比较, 像 `>`, `>=`, `<`, `<=`, `BETWEEN`, `=`; |
| 2. 字符比较, 像 `=`, `<>`, `IN`; |
| 3. `IS NULL` 或者 `IS NOT NULL`; |
| 4. 逻辑运算`AND`, `OR`, `NOT`; |
| 常量类型是: |
| 1. 数字, 像123, 3.1415; |
| 2. 字符串, 像‘abc’,必须使用单引号; |
| 3. `NULL`, 特殊常数; |
| 4. 布尔常量, `TRUE` 或`FALSE`; |

**7.重试机制**

只有在消息模式为MessageModel.CLUSTERING集群模式时，Broker才会自动进行重试，广播消息不重试

重投使用`messageDelayLevel`

默认值

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

/\*\*
\* Timeout for sending messages.默认超时时间
\*/
private int sendMsgTimeout = 3000;
// 异步发送时 重试次数，默认 2
producer.setRetryTimesWhenSendAsyncFailed(1);
// 同步发送时 重试次数，默认 2
producer.setRetryTimesWhenSendFailed(1);
// 是否向其他broker发送请求 默认false
producer.setRetryAnotherBrokerWhenNotStoreOK(true);

**第三节、API以及事务**

**1.分布式事务**

rocketmq使用两阶段提交

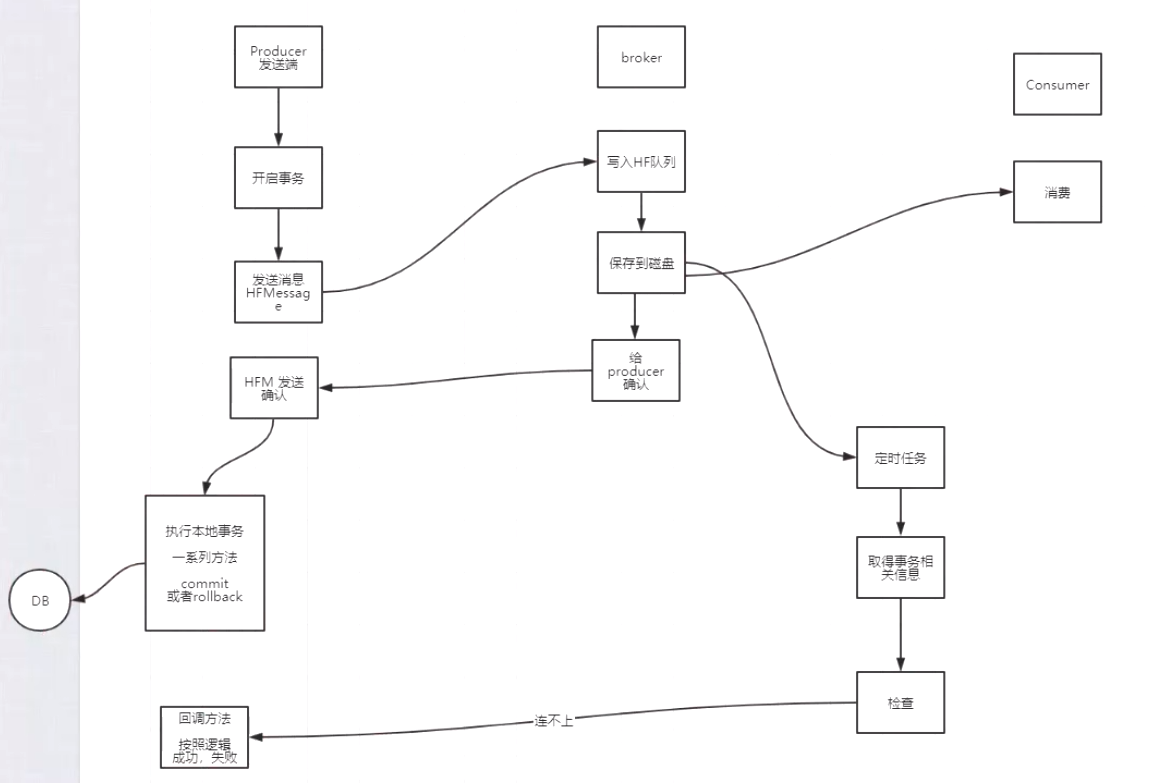
producer产生消息之后，产生事务，此时将消息发送给broker。写入broker的HF队列。消息的状态是不可用的。写入到HF队列之后，会刷到磁盘中，此时broker给producer发送确认消息。

producer收到确认消息之后，代表HFM消息发送成功，此时可以开启本地真正的事务。一系列操作成功之后，再给broker发送成功确认的消息，HFM消息变成可用状态，消费端可以进行消费。

1.如果本地事务执行时间比较久，broker会开启一个定时任务，定时向producer发送消息，确认程序是执行失败了，还是就是执行时间比较久。此时producer会有一个回调方法，可以手动编码返回给定时任务程序执行的状况。如果定时任务一直拿不到结果，这条半消息就会被废弃了。

2.如果本地事务执行失败，会在回调方法内向broker发送消息状态。

HF队列中的消息会写到磁盘中进行保存，定时任务会定时进行清除。



**事务消息**

分布式系统中的事务可以使用TCC（Try、Confirm、Cancel）、2pc来解决分布式系统中的消息原子性

RocketMQ 4.3+提供分布事务功能，通过 RocketMQ 事务消息能达到分布式事务的最终一致

**RocketMQ实现方式**

**Half Message：**预处理消息，当broker收到此类消息后，会存储到RMQ\_SYS\_TRANS\_HALF\_TOPIC的消息消费队列中

**检查事务状态：**Broker会开启一个定时任务，消费RMQ\_SYS\_TRANS\_HALF\_TOPIC队列中的消息，每次执行任务会向消息发送者确认事务执行状态（提交、回滚、未知），如果是未知，等待下一次回调。

**超时：**如果超过回查次数，默认回滚消息

**TransactionListener的两个方法**

**executeLocalTransaction**

半消息发送成功触发此方法来执行本地事务

**checkLocalTransaction**

broker将发送检查消息来检查事务状态，并将调用此方法来获取本地事务状态

**本地事务执行状态**

**LocalTransactionState.COMMIT\_MESSAGE**

执行事务成功，确认提交

**LocalTransactionState.ROLLBACK\_MESSAGE**

回滚消息，broker端会删除半消息

**LocalTransactionState.UNKNOW**

暂时为未知状态，等待broker回查

**2.Group**

消费单位以group为主。消息会发给关注同一topic的多个group。并且group中只要有一个消费成功，就代表group消费成功。

**3.使用限制**

事务消息不支持延时消息和批量消息。

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。

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。

事务性消息可能不止一次被检查或消费。

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。

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。

**4.代码demo**

public class Producer {
public static void main(String[] args) {
TransactionMQProducer transactionMQProducer = new TransactionMQProducer("transactionGroup1");
transactionMQProducer.setNamesrvAddr("192.168.73.40:9876");
// 设置事务监听器
transactionMQProducer.setTransactionListener(new TransactionListener() {
// 执行本地事务
@Override
public LocalTransactionState executeLocalTransaction(Message msg, Object arg) {
System.out.println(arg);
if (Integer.valueOf(arg.toString()) == 1) {
return LocalTransactionState.COMMIT\_MESSAGE;
} else if (Integer.valueOf(arg.toString()) == 2) {
return LocalTransactionState.ROLLBACK\_MESSAGE;
}
return LocalTransactionState.UNKNOW;
}
// 消息状态回查
@Override
public LocalTransactionState checkLocalTransaction(MessageExt msg) {
String string = new String(msg.getBody());
System.out.println(string);
return LocalTransactionState.COMMIT\_MESSAGE;
}
});
try {
transactionMQProducer.start();
} catch (MQClientException e) {
e.printStackTrace();
}
for (int i = 0; i < 3; i++) {
Message msg = new Message("myTopic", ("测试" + i).getBytes());
try {
transactionMQProducer.sendMessageInTransaction(msg, i);
} catch (MQClientException e) {
e.printStackTrace();
}
}
}
}

**第四、五节、源码**

**1.offset**

每个broker中的queue在收到消息时会记录offset，初始值为0,每记录一条消息offset会递增+1

minOffset 最小值

maxOffset 最大值

consumerOffset 消费者消费的进度（位置）

diffTotal 消费积压，未被消费的消息数量

**2.DefaultMQPushConsumer和DefaultMQPullConsumer**

在消费端，我们可以视情况来控制消费过程

DefaultMQPushConsumer 由系统自动控制过程，

DefaultMQPullConsumer 大部分功能需要手动控制

**3.集群消息的消费负载均衡**

在集群消费模式下（clustering）相同的group中的每个消费者只消费topic中的一部分内容，group中的所有消费者都参与消费过程，每个消费者 消费的内容不重复，从而达到负载均衡的效果。

使用DefaultMQPushConsumer，新启动的消费者自动参与负载均衡。

**4.长轮询**

consumer向broker发起请求，如果broker有消息，就直接给。如果没有，这个链接就挂起。等到有消息之后，再将消息发送给consumer。

consumer消费完毕一条消息之后，再发起请求，请求下一条消息。主动权掌握在consumer端。

**长连接**是连接一旦建立，永远不断开，push方式推送。

Consumer -> Broker RocketMQ采用的长轮询建立连接

consumer的处理能力Broker不知道

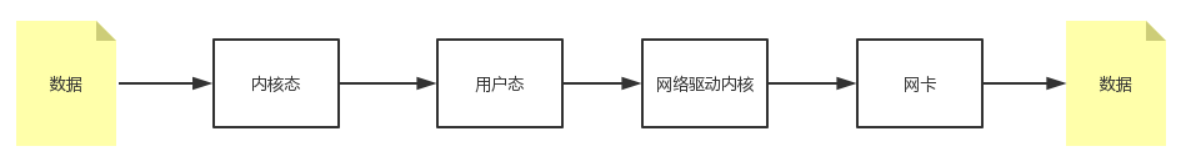
直接推送消息 broker端压力较大

采用长连接有可能consumer不能及时处理推送过来的数据

pull主动权在consumer手里

**第六节、消息存储机制**

**1.数据零拷贝技术**



从磁盘复制数据到内核态内存；

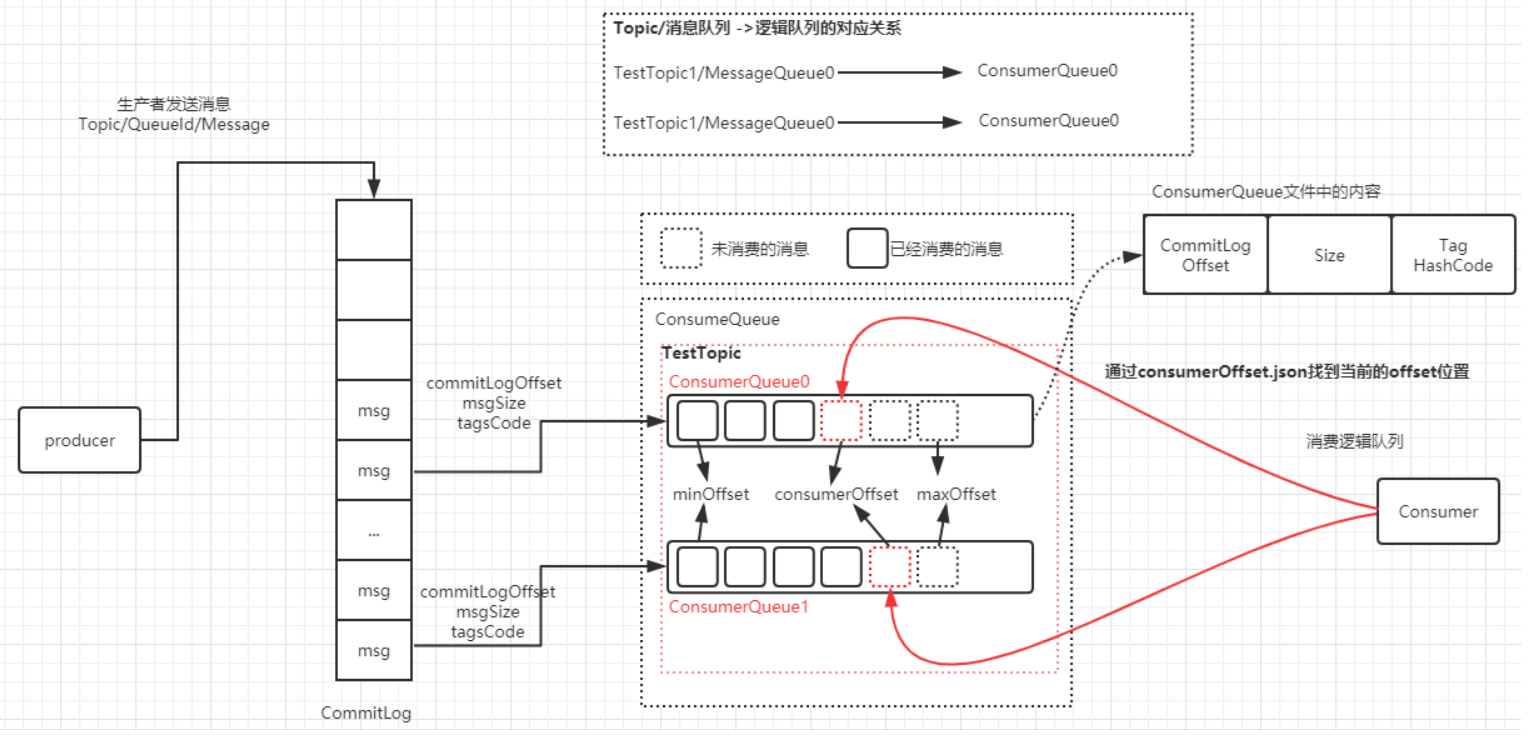
从内核态内存复 制到用户态内存；

然后从用户态 内存复制到网络驱动的内核态内存；

最后是从网络驱动的内核态内存复 制到网卡中进行传输。

MappedByteBuffer底层使用了操作系统的mmap机制，可以省略掉将数据复制到用户态这一步骤，可以直接从内核kernel将文件发送到网卡。但是MappedByteBuffer内存映射只能映射1.5G-2G大小，使用的是虚拟内存，所以RocketMq设定commitLog大小为1G.

**2.存储结构**

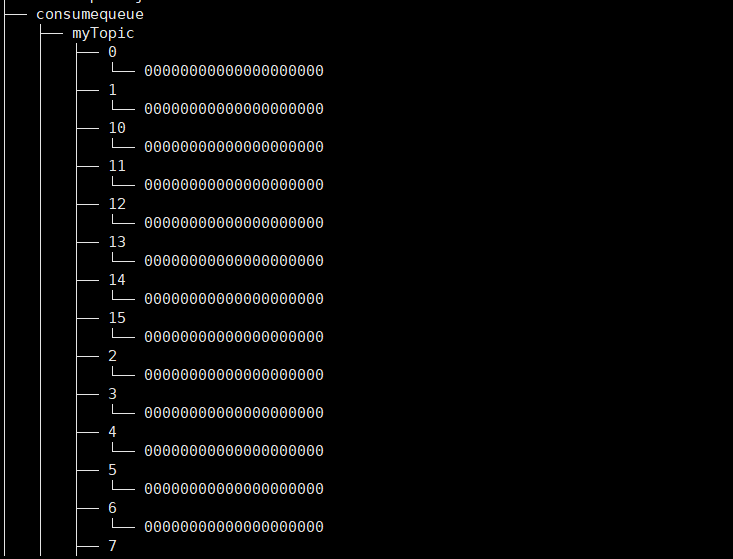


**2.1commitLog**

数据存储的真正的物理单元。一个commitLog大小为1g，会把所有收到的消息，按照顺序存储进commitLog，并且每条消息还会存储offset偏移量，方便通过consumeQueue快速定位到具体的某条消息。

**2.2consumequeue**

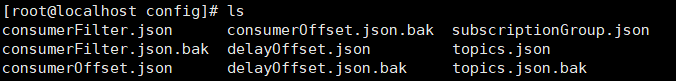
consumequeue的物理结构是按照每个topic的名称进行分类，树状结构的文件夹，每个topic下还会按照每个queue的序号进行分类，每个queue下是具体的文件。文件内容包括：offset偏移量，size大小，tag标签。



**2.3index**

indexFile提供了通过key和时间戳来查找消息的功能。

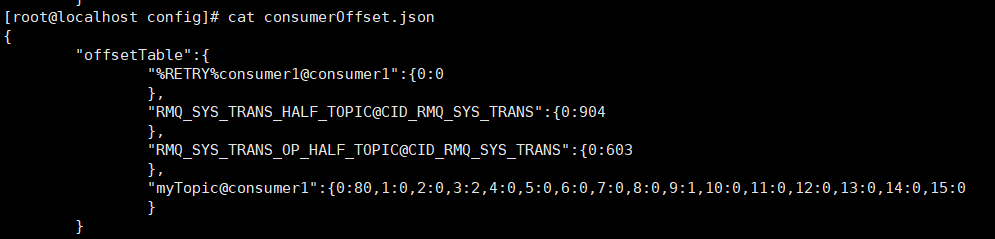
**2.4config**



以json格式存储消费信息

**consumerFilter.json**---消息过滤器

**consumerOffset.json**---客户端的消费进度



**delayOffset.json**---延迟消息进度

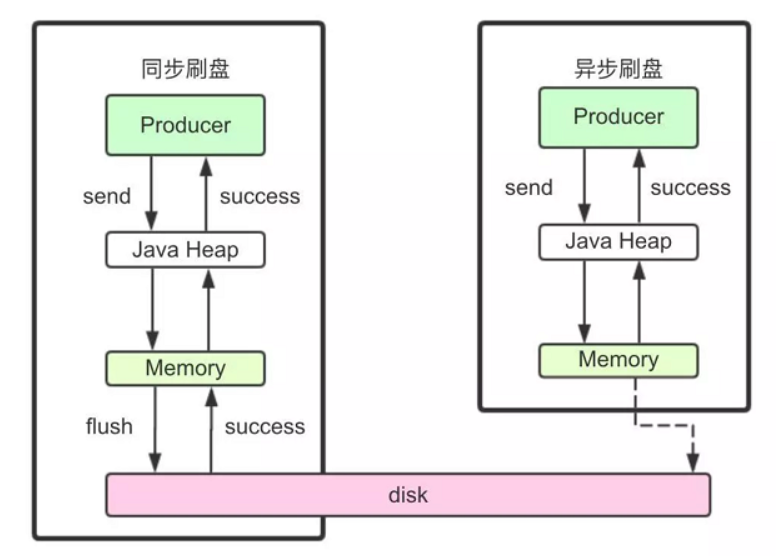
**subscriptionGroup.json**---group的订阅数据



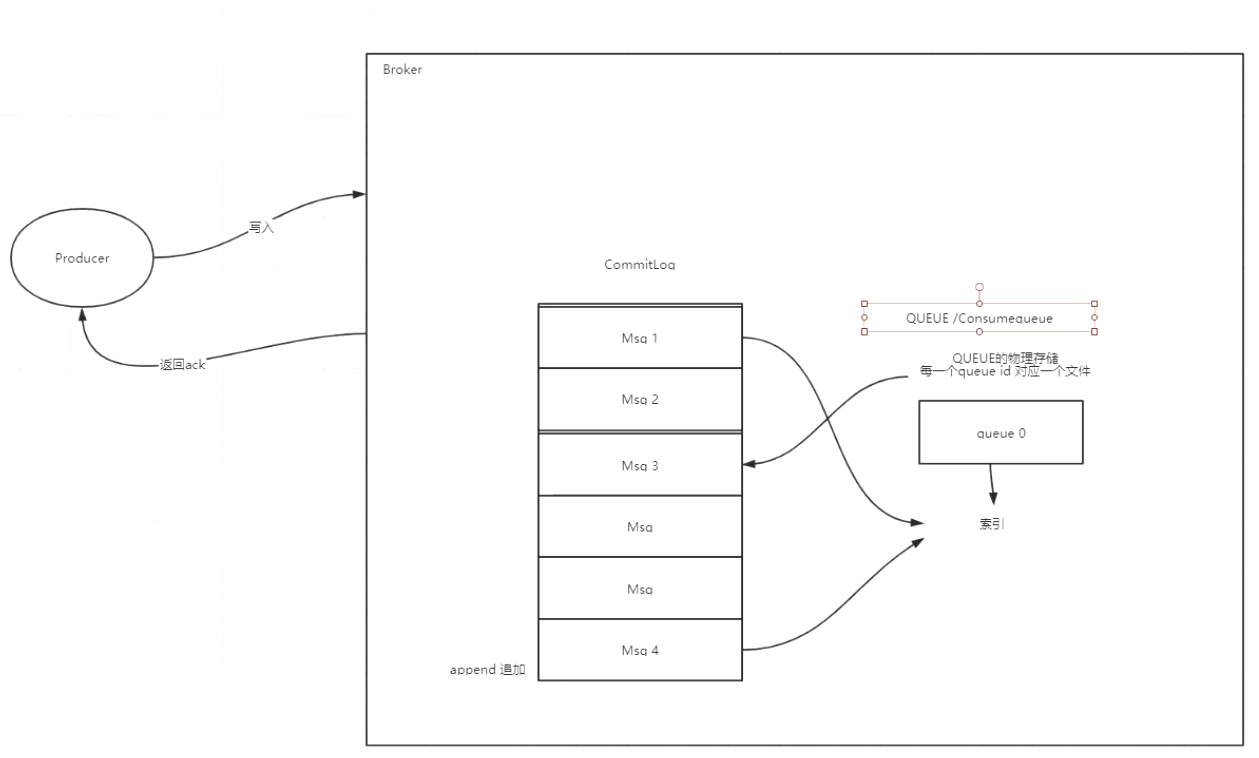
**topics.jsonTopic**---的配置信息



**3.刷盘机制**



消息写入broker之后，进行持久化



**3.1同步刷盘**

消息发送到broker之后，持久化到磁盘上，再返回给producer响应。

**3.2异步刷盘**

消息发送到broker之后，直接返回给producer响应，当内存中的消息堆积到一定程度再执行刷盘持久化操作。

通过参数可以控制。Broker配置文件里的flushDiskType参数设置的，这个参数被配置成SYNC\_FLUSH、ASYNC\_FLUSH中的一个。

**第七节、集群搭建**

**1.注意**

至少3个节点才能组件集群。

**2.配置**

提供了默认的配置

/root/temp/rocketmq/bat/rocketmq-4.9.0/conf/2m-2s-sync

启动的时候加载配置

./mqbroker -c ../conf/2m-2s-sync/broker-a.properties

dleger提供自动选举，高可用

#集群整体的name
brokerClusterName=DefaultCluster
#小集群的name
brokerName=broker-a
#默认0为master，但是在dleger模式下，是失效的
brokerId=0
#凌晨4点删除commitlog
deleteWhen=04
#commitlog多久未使用时，执行删除
fileReservedTime=48
#主节点角色 数据异步复制ASYNC\_MASTER、 同步复制SYNC\_MASTER、SLAVE
brokerRole=SYNC\_MASTER
#刷盘机制，异步
flushDiskType=ASYNC\_FLUSH
namesrvAddr=192.168.73.40:9876;192.168.73.41:9876;192.168.73.42:9876;
192.168.73.50:9876;192.168.73.51:9876;192.168.73.52:9876
# dleger
#启用
enableDLegerCommitLog = true
dLegerGroup = broker-a
# 兄弟节点的地址
dLegerPeers = n0-192.168.73.40:40911;n1-192.168.73.41:40911;n2-192.168.73.42:40911
#自己的id
dLegerSelfId = n0
#发送消息的线程数量
sendMessageThreadPoolNums = 4



**3.MQAdmin**

**第八节、面试题**

**1.为什么要使用MQ？**

答：因为项目比较大，做了分布式系统，所有远程服务调用请求都是**同步执行**经常出问题，所以引入了mq

1.解耦合。没有强依赖关系。

比如A系统和B系统，需要同时上线，才能提供服务，挂掉一个，另一个也不可用了。使用消息中间件，可以对消息做一下缓存。即使挂掉一个，另一个也是可用的。

2.异步执行，不需要同步，可以降低系统响应时间。

3.削峰：请求达到峰值之后，避免压垮后端service，引入mq，可以使service可以保持固定速率进行消费。

**2.RocketMQ Broker中的消息被消费后会立即删除吗？**

不会，每条消息都会持久化到commitLog中，每次consumer进行消费的时候，会记录下offest偏移量。只有当整个commitLog默认48小时没有消费的时候，才会定时默认在凌晨4点进行整个文件的删除。

**3.那么消息会堆积吗？什么时候清理过期消息？**

4.6版本默认48小时后会删除不再使用的CommitLog文件

检查这个文件最后访问时间

判断是否大于过期时间

指定时间删除，默认凌晨4点

**4.RocketMQ消费模式有几种？**

由消费者进行控制，分为集群模式、广播模式。

集群模式：一组同一group下的consumer对同一topic，只有一个consumer会收到消息。不同group都会收到消息。group内只要有一个consumer消费了，就代表消费成功。会有重试机制。

广播模式：group下所有的consumer都会收到消息，但是没有重试机制。

**5.消费消息时使用的是push还是pull？**

在刚开始的时候就要决定使用哪种方式消费，两种：DefaultLitePullConsumerImpl--拉，DefaultMQPushConsumerImpl--推

两个实现 `DefaultLitePullConsumerImpl` `DefaultMQPushConsumerImpl`都实现了MQConsumerInner接口接口

名称上看起来是一个推，一个拉，但实际底层实现都是采用的**长轮询机制**，即拉取方式。当broker端有消息的时候，会通知consumer。

broker端属性 longPollingEnable 标记是否开启长轮询。默认开启

**6.说一说几种常见的消息同步机制？**

**push：**如果broker主动推送消息的话有可能push速度快，消费速度慢的情况，那么就会造成消息在consumer端堆积过多，同时又不能被其他consumer消费的情况

**pull：**轮训时间间隔，固定值的话会造成资源浪费

**长轮询：**长连接 短连接（每秒） 长轮询

**7.broker如何处理拉取请求的？**

consumer首次请求broker

broker中是否有符合条件的消息

有 ->

响应consumer

等待下次consumer的请求

没有

挂起consumer的请求，即不断开连接，也不返回数据

挂起时间长短，写死在代码里的吗？长轮询写死，短轮询可以配

使用consumer的offset，

DefaultMessageStore#ReputMessageService#run方法

每隔1ms检查commitLog中是否有新消息，有的话写入到pullRequestTable

当有新消息的时候返回请求

PullRequestHoldService 来Hold连接，每隔5s执行一次检查pullRequestTable有没有消息，有的话立即推送

**8.RocketMQ如何做负载均衡？**

基础点：每个topic会在broker端创建queue。可以手动在broker端创建topic对应的queue，均匀分布在不同的broker上，达到负载均衡的效果。

**8.1producer负载均衡**

producer send的时候，可以主动选择要发送的queue。通过MessageQueueSelector接口，可以自定义queue负载逻辑，默认的有byhash、byRondom。也可以自己实现。

默认使用的是随机选择。

**8.2consumer负载均衡**

consumer可以设置AllocateMessageQueueStragegy策略，选择目标queue。consumer的数量最好和Message Queue的数量对等或者是倍数，不然可能会有消费倾斜。

平均分配策略(默认)(AllocateMessageQueueAveragely)

环形分配策略(AllocateMessageQueueAveragelyByCircle)

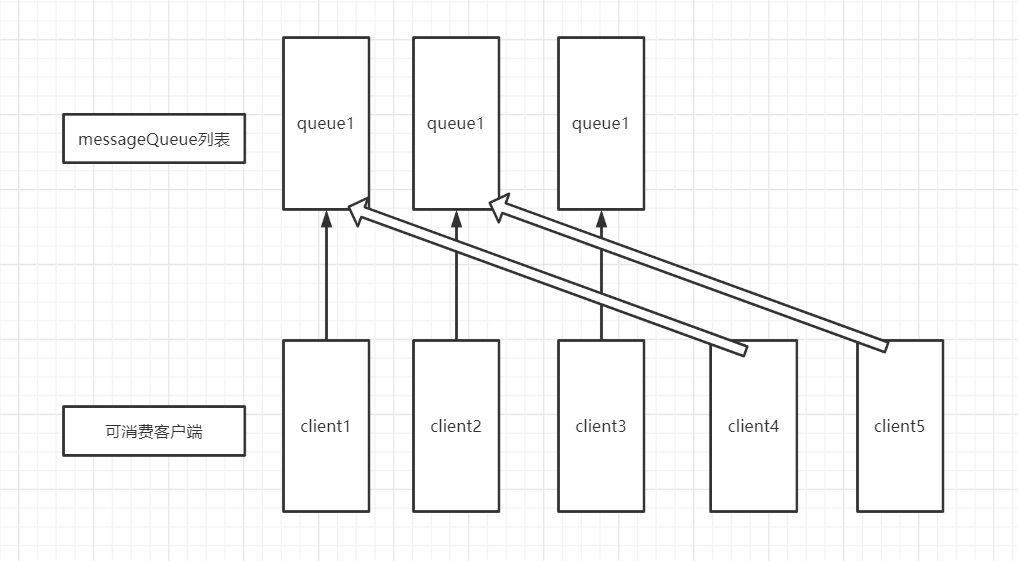
手动配置分配策略(AllocateMessageQueueByConfig)

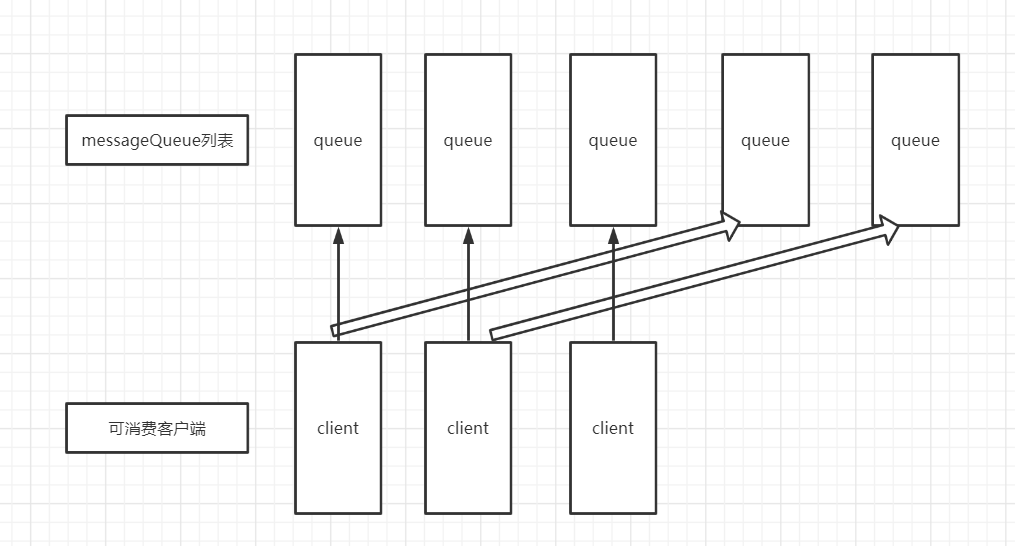
机房分配策略(AllocateMessageQueueByMachineRoom)

一致性哈希分配策略(AllocateMessageQueueConsistentHash)

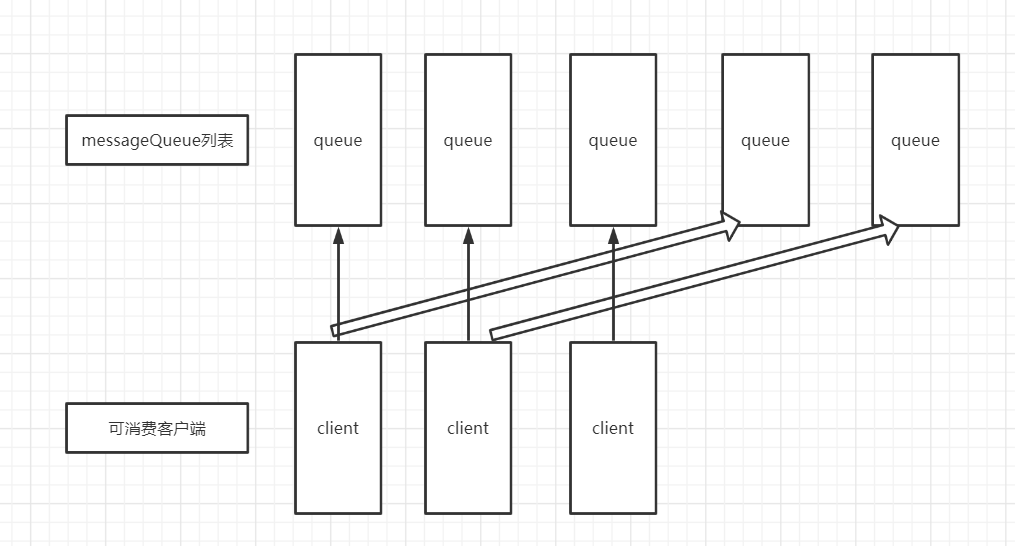
靠近机房策略(AllocateMachineRoomNearby)

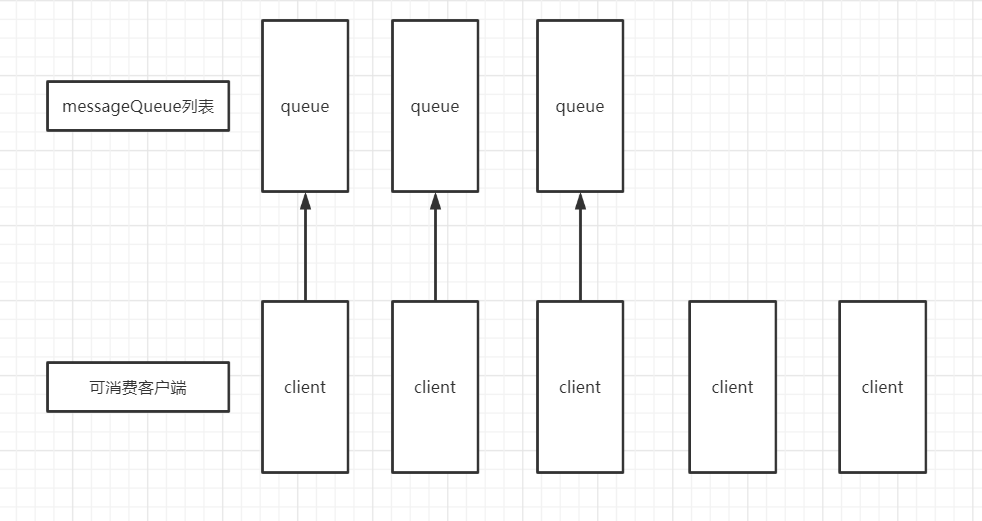
**平均分配**





环形分配





**9.消息丢失**

producer在发送同步/异步可靠消息后，会接收到SendResult，表示消息发送成功，SendResult其中属性sendStatus表示了broker是否真正完成了消息存储，

当sendStatus!="ok"的时候，应该重新发送消息，避免丢失。

**10.消息重复消费**

影响消息正常发送和消费的重要原因是网络的不确定性。

可能是因为consumer首次启动引起重复消费，所以需要设置consumer.setConsumeFromWhere

这个设置只对一个新的consumeGroup第一次启动时有效,设置从头消费还是从最新处开始消费，已有的group消费记录，在broker中有记录。

并且发送的每一条消息，都要有唯一的key。

**11.引起重复消费的原因**

producer发送消息到broker，但是broker没有返回给producer响应，导致producer重复发送。

consumer消费消息的时候，返回给broker的ack，因为网络等原因，没有发送成功，broker就会重投

解决：

数据库唯一主键做校验

单机情况下，缓存map

redis校验

**12.如何让RocketMQ保证消息的顺序消费**

同一topic

同一个QUEUE

发消息的时候一个线程去发送消息

消费的时候 一个线程 消费一个queue里的消息或者使用MessageListenerOrderly

多个queue 只能保证单个queue里的顺序

场景：数据同步的时候，新增、修改必须按照顺序来

https://blog.csdn.net/qq\_41893274/article/details/112617531

**13.RocketMQ如何保证消息不丢失**

生产端在发送消息的同时，要做好本地记录，以防止broker因为宕机、网络等原因消息丢失

broker端做好刷盘机制，同步刷盘、同步到slave上

consumer做好数据存库。定时任务检查消费失败的消息，进行重复消费。并且consumer在消费的时候，如果是事务消息，没有返回ack消息还会重投。

**14.rocketMq的消息堆积如何处理**

1.准备一个临时的topic

2.一台新的consumer，将堆积的消息转发到临时的topic下。

3.上线几台新的consumer，处理临时topic下的数据。

因为做消息转发，速度是非常快的，所以可以很速度的将堆积的消息转发到临时topic，并且原有的consumer也可以继续消费，不影响业务。

**堆积时间过长消息超时了？**

RocketMQ中的消息只会在commitLog被删除的时候才会消失，不会超时
**堆积的消息会不会进死信队列？**

不会，消息在消费失败后会进入重试队列（%RETRY%+consumergroup），多次（默认16）才会进入死信队列（%DLQ%+consumergroup）

**15.你们用的是RocketMQ?RocketMQ很大的一个特点是对分布式事务的支持，你说说他在分布式事务支持这块机制的底层原理?**

rocketmq采用两阶段提交来控制事务，producer发送事务消息到broker，会进入RMQ\_SYS\_TRANS\_HALF\_TOPIC半消息队列，此时消息是不可用的，只有

producer发送确认ack，消息才会可用。

发送事务消息之后，会执行回调，进入executeLocalTransaction方法，执行本地事务，此方法会返回success、rollback、unknown。

当回复unknown或者broker一直得不到反馈时，会回调checkLocalTransaction，检查是否可以提交事务。当检查16次（默认）一直失败，

则会把消息转入死信队列。

16.高吞吐量下如何优化生产者和消费者的性能?

**第九节、全部参数解释**

#4.7.1版本
#所属集群名字
brokerClusterName=rocketmq-cluster
#broker名字，名字可重复,为了管理,每个master起一个名字,他的slave同他,eg:Amaster叫broker-a,他的slave也叫broker-a
brokerName=broker-a
#0 表示 Master，>0 表示 Slave
brokerId=0
brokerIp=127.0.0.1
#nameServer地址，分号分割
namesrvAddr=127.0.0.1:9876
#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=8
#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=48
#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
destroyMapedFileIntervalForcibly=120000
redeleteHangedFileInterval=120000
#检测物理文件磁盘空间
diskMaxUsedSpaceRatio=88
#存储路径
storePathRootDir=D:\\rocketMqTestData\\store
#commitLog 存储路径
storePathCommitLog=D:\\rocketMqTestData\\store\\commitlog
#消费队列存储路径存储路径
storePathConsumeQueue=D:\\rocketMqTestData\\store\\consumequeue
#消息索引存储路径
storePathIndex=D:\\rocketMqTestData\\store\\index
#checkpoint 文件存储路径
storeCheckpoint=D:\\rocketMqTestData\\store\\checkpoint
#abort 文件存储路径
abortFile=D:\\rocketMqTestData\\store\\abort
#限制的消息大小
maxMessageSize=65536
flushCommitLogLeastPages=4
flushConsumeQueueLeastPages=2
flushCommitLogThoroughInterval=10000
flushConsumeQueueThoroughInterval=60000
#Broker 的角色
brokerRole=ASYNC\_MASTER
#刷盘方式
flushDiskType=ASYNC\_FLUSH
checkTransactionMessageEnable=false
#发消息线程池数量
sendMessageTreadPoolNums=128
#拉消息线程池数量
pullMessageTreadPoolNums=128