RocketMq 源码剖析(很混乱，未整理)

RocketMq 工程主要有 Broker、client、store、namesrv、remoting、common、research、tools 等子工程。

Broker:服务端，上层接受 consumer 与 producer 的 message request，并进行处理。HA 的基本单元，支持同步双写和异步复制。下层主要调用 store 层服务进行数据的持久化(commitlog)。

Client：consumer 与 producer 的 API 层。提供了与 broker 交互的接口。

Store：核心存储，主要使用 mapFile（内存映射）进行数据管理，顺序写随机读。支持 offset 存储。

Namesrv:还未看。。。

Remoting:

Broker 层笔记：

BrokerController 是 Broker 的心脏：

三个配置管理：BrokerConfig（Broker 本身的配置） 、NettyServerConfig 与 NettyClientConfig（Broker 通信配置，包括发送请求和接收请求）、MessageStoreConfig(存储配置，主要是持久化时的路径等相关信息)

管理器：ConsumerOffsetManager（消费进度存储管理）、ConsumerManager（消费者管理）、ProducerManager(生产者管理)、SubscriptionGroupManager(订阅关系管理，比如 Tag 设置，是否需要过滤)、RebalanceLockManager(队列锁分配管理)、DigestLogManager(统计管理)、TopicConfigManager（Topic 配置管理）

两个主要的属性，一个是<topicName,TopicConfig>Hash 结构，一个是 DataVersion(包含一个时间戳，和递增的 Counter 计数器)

每一个 broker 在初始化时，都有一个 TopicConfigManager 实例，而且会默认加载几个系统级的 Topic。

SELF_TEST_TOPIC、TBW102、BenchmarkTest、以集群名字作为 topic Name 的 topic (TBW102 和集群名字作为 topicName 的 topic 是无法发送 message 的,这个默认的 topic 很重要，因为很多新产生的 topic 在权限和过滤类型上都要继承这个默认的 topic 的配置)

```
 */
public enum TopicFilterType {
    /**
     * 每个消息只能有一个Tag
     */
    SINGLE_TAG,
    /**
     * 每个消息可以有多个Tag（暂时不支持，后续视情况支持）<br>
     * 为什么暂时不支持？<br>
     * 此功能可能会对用户造成困扰，且方案并不完美，所以暂不支持
     */
    MULTI_TAG
}
```

TopicConfigManager(broker 本地的 topic 配置管理)：

有几个重要的函数：

createTopicInSendMessageMethod(如果 topic 不存在，尝试创建,创建之后会持久化为 json 格

式的文件)
updateTopicConfig（更新 hash 结构）等
在更新 topic 结构时，这里有一个很重要的动作：

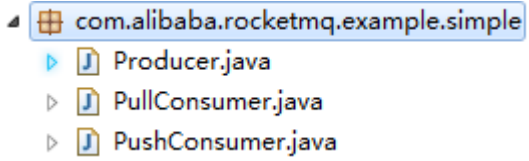
```
this.brokerController.registerBrokerAll();
```

服务：
ClientHousekeepingService（检查客户端连接服务）、…
一个一个来：
首先看 MassageStoreConfig，这是 Broker 进行持久化时需要依赖的配置类，里面是各种存储目录，很多路径是写死的，所以在开发测试时，可以通过这些目录里的文件，查看当前 Message 的存储情况。
CommitLog 存储在哪里：Broker 服务器里的{userhome}/store/目录下，有所有持久化的数据文件：

```
[zhuangwei.zw@dev136090.sqa.cm6 store]$ ll
total 20
-rw-r--r-- 1 zhuangwei.zw users    0 Dec 21 10:31 abort
-rw-r--r-- 1 zhuangwei.zw users 4096 Dec 28 14:48 checkpoint
drwxr-xr-x 2 zhuangwei.zw users 4096 Dec 21 10:35 commitlog
drwxr-xr-x 2 zhuangwei.zw users 4096 Dec 28 14:48 config
drwxr-xr-x 3 zhuangwei.zw users 4096 Dec 21 10:35 consumequeue
drwxr-xr-x 2 zhuangwei.zw users 4096 Dec 21 10:35 index
[zhuangwei.zw@dev136090.sqa.cm6 store]$ pwd
/home/zhuangwei.zw/store
```

Commitlog 是消息真正存储的地方
Consumequeue 是消息队列数据存储的位置
Index 是索引文件存放的位置
Checkpoint 是检查点信息的数据
abortFile 异常退出产生的文件
重要参数：commitlog 文件每个大小为 1G
ConsumeQueue

Broker 的几个关键 processor:
PullMessageProcessor:拉消息处理器
SendMessageProcessor：处理客户端发送消息的请求处理器
这里通过 example 中的例子，解释一下推消息与拉消息的差别



1. 拉消息：(建立与 broker 之间的长轮询，由 consumer 直接从队列里拉取需要的数据)
   发出拉消息的请求类 DefaultMQPullConsumer。这个类是个包装类，其实真正处理拉的动作的是 DefaultMQPullConsumerImpl 类。
   这里先介绍 DefaultMQPullConsumerImpl 类里面的主要功能：
   重要属性：

MQClientFactory ：

1. 用来管理 producer 与 consumer 的类，这个类功能很强大，而且每个客户端都无论产生多少个 consumer 或者 producer，都只会共享一个 MQClientFactory，这个实例包含网络连接和线程资源等。包含了三个关键的 map 结构 :producerTable （group:producer）、consumerTable （group:consumer）、adminExtTable（group:MQAdminExtInner），这些实例都是在一个客户端下的所有创建的 producer、consumer、admin。topicRouteTable(topic:TopicRouteData) 是从 name Server 拿到的 Topic 路由信息。以 topic 为维度，汇总了当前所有的 borker （name 、 ip 等）、 queue(brokerName 、 readQueueNums 、 writeQueueNums、perm 等信息)。MQClientFactory 都会定期从 nameserver 更新这个路由信息。
2. 这个类里有一个 pullMessageService 服务。这个服务建立了一个长轮询拉消息服务，起了一个单线程用来异步拉取数据。

2. 真正拉消息的实现是在 DefaultMQPullConsumerImpl 实例中的 pullmessage 函数，这里介绍整个过程：
    1. 每一个 consumer 客户端都会维护一个客户端 offsetstore 实例，作为获取客户端消费消息的进度，有两种方式可以获取 offset 信息(本地 local 文件和 broker offset 文件)
       LocalFileOffsetStore:
       user.home/.rocketmq_offsets/clientid/groupname/offsets.json 文件

Store 层笔记：

零拷贝方式：1.mmap+write 方式，这种方式适用于小块文件传输，不能很好的利用 DMA，所以也就消耗了 cpu 资源，更多的需要程序控制内存，编程比较复杂，需要避免 jvm crash 问题。

2. sendFile 方式，使用的是 DMA，消耗 cpu 比较少，适用大块文件传输，不需要考虑 jvm crash 问题。但是效率很低，而且无法使用异步 IO。

存储层中有几个很重要的类：CommitLog、ConsumeQueue、MapedFile、MapedFileQueue、DefaultMessageStore 等，想要弄清楚 rocketMq 的持久化机制，这几个类的关系必须很清楚。

MapedFile:最基本的存储单元。内存映射到文件中。

重要参数：

全局：TotalMapedVitualMemory JVM 中虚拟内存总大小

全局: TotalMapedFiles mapedFile 文件个数

fileName 映射的文件名(物理文件名)

fileFromOffset 映射的起始偏移

fileSize 文件大小 (文件大小)

file 真正映射的物理文件

mappedBytebuffer 映射到内存中的一部分数据

wrotePostion 当前写到什么位置 (后面着重介绍)

commitedPosition 当前刷磁盘到什么位置(后面着重介绍)

fileChannel MappedByteBuffer 与 file 之间的管道。

storeTimestamp 最后一条消息存储时间

新建 MapedFile 时几个注意点：

1. mappedByteBuffer 会从 0~fileSize 位置之间映射文件数据到内存
2. TotalMapedVitualMemory= TotalMapedVitualMemory+fileSize
3. TotalMapedFiles= TotalMapedFiles+1
4. fileFromOffset 设置为 Long.parseLong(this.file.getName())，文件名其实就是该文件在逻辑上的 offset 值

AppendMessage （commitlog）如何追加数据：

先获取 wrotePosition 判断是否小于 fileSize，如果小于，则往内存里写 message。调用 appendMessage 函数的接口，必须保证 wrotePosition 不能大于 fileSize。此时 append 的只是写入到内存中，还未真正刷盘。Append 真正调用的是 commitLog 的 doAppend 函数。

doAppend 函数的具体流程：

1. 写入位点=文件名对应的 fileFromOffset+mapedFile 对应的 wroteOffset。
2. 生成 msgId （这个很重要,作为返回值的一部分属性，指定了真实 commitlog 存储的 ip 地址和 offset 等信息）这个 id 是定长，为 16 字节。msgId 格式为：8 字节的 IP 和端口+8 字节的 offset（上面的写入位点）
3. 生成 key，这个 key 主要是用来定位 ConsumeQueue 信息的，commitLog 维护一个 topickey、offset 的 map 结构。Key=Topic+"-"+queueid。
4. 如果是事务性消息需要特殊处理
5. 序列化消息

```
final int msgLen = 4 // 1 TOTALSIZE
        + 4 // 2 MAGICCODE
        + 4 // 3 BODYCRC
        + 4 // 4 QUEUEID
        + 4 // 5 FLAG
        + 8 // 6 QUEUEOFFSET
        + 8 // 7 PHYSICALOFFSET
        + 4 // 8 SYSFLAG
        + 8 // 9 BORNTIMESTAMP
        + 8 // 10 BORNHOST
        + 8 // 11 STORETIMESTAMP
        + 8 // 12 STOREHOSTADDRESS
        + 4 // 13 RECONSUMETIMES
        + 8 // 14 Prepared Transaction Offset
        + 4 + bodyLength // 14 BODY
        + 1 + topicLength // 15 TOPIC
        + 2 + propertiesLength // 16 propertiesLength
        + 0;
```

一条消息序列化时的协议

如果消息的大小超过 mapedFile 的剩余空间，则

```
this.resetMsgStoreItemMemory(maxBlank);
// 1 TOTALSIZE
this.msgStoreItemMemory.putInt(maxBlank);
// 2 MAGICCODE
this.msgStoreItemMemory.putInt(CommitLog.BlankMagicCode);
// 3 剩余空间可能是任何值
//
```

会写进去剩余空间大小和 END 魔数，然后返回。

如果大小未超过剩余空间，消息会写入缓冲区(还未刷盘，意味着这个时候宕机那么新写的数据就丢失了)。

消息刷盘：

查看是否可以满足刷盘条件：

1. 文件写满，则立即刷盘
2. 未刷盘的数据大于一定的 page 数才可以(每个 pagecache 4M)
3. 资源是否允许

selectMapedBuffer 读分区函数比较简单，指定 pos 和 size，然后从 bytebuffer 获取。

MapedFileQueue：存储队列，定时删除数据，无限增长

重要属性：

DeleteFileBatchMax 定时删除时，最多可以删除多少个文件

storePath 上文提到的 consumequeue 路径下 topic/queueid/的目录

mapedFileSize 文件大小

mapedFiles MapedFile 数组

readWriteLock 数组的读写锁

allocateMapedFileService 预分配 MapedFile 对象的服务

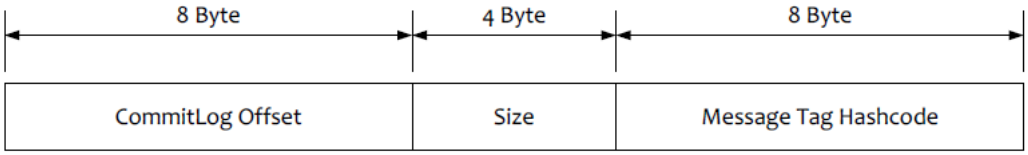
committedWhere 刷盘刷到哪里

storeTimestamp 最后一条消息存储时间

load 函数：这个函数的作用是将 store/topic/queueid/下的所有 file 映射为 mapedFile 数组。

getLastMappedFile 获取最后一个 MapedFile 对象，如果没有则创建。这里有一个需要注意的地方就是 mapedFile 的命名规范。就是全局的 offset 值左补 0 对齐至 20 位。

CQStoreUnitSize：MapedFileQueue 中每一次写入需要占用的内存大小，为 20 字节。



也就是说，MapedFileQueue 中的每一个 mapedFile 都以 20 字节划分(Message Tag HashCode 是 Tag 的 hash 值)，记录的是消息存储的元数据和 tag 标签数据，最后的数据查询是在 commitLog 中(通过逻辑队列层实现了所谓的顺序写，随机读，但是每一次都得先读 MapedFileQueue 再读 Commit Log，增加了开销，而且要保证 commitlog 与 MapedFileQueue 的一致性，增加了编程的复杂度)

MapedFileQueue 的这种结构写函数是在 consumeQueue 类的 putMessagePostionInfo 函数里实现的。整个函数，其实就是把这个结构写入到 MapedFile 里

ConsumeQueue 是 MapedFileQueue 队列的封装，是消费队列的实现。

ConsumeQueue:

重要属性：DefaultMessageStore 存储层对象

消息存储的默认实现，包括本机的 commitLog 实例，(同一个 topic 能不能跨 broker?)ConsumeQueue 集合、刷盘服务、清理文件服务(commitlog 的文件)、清理逻辑文件服务（consumeQueue 队列的描述文件）,HA 服务、等

Commitlog 中的 putMessage 函数中有同步双写（slave）

topic 的信息发布到 namesrv，客户端获取 topic 的信息，分别到不同的（master-slave 组）进行消费
广播消费是保存在 consumer 本地的
集群消费才保存在 broker 上

注意点：
Push consumer:consumer 对象注册一个 listener 接口，一旦有消息，就回调这个接口
Pull consumer:consumer 主动从 broker 拉消息。
消费分为广播消费和集群消费:广播消费是，每一个 consumer 都可以接收到消息，集群消费是均摊消息。
消息顺序性：分为普通顺序和严格顺序，顺序指的是按照单线程 producer 产生的顺序进行消费。普通顺序不能保证在宕机或者重启的情况下顺序的一致性，因为重启后队列的定位会出现短暂的错乱。严格顺序，牺牲了分布式的 failover 特性，也就是如果有一台宕机了，相当于集群都不可用，可以使用同步双写，但是在切换服务器的时候会造成短暂时间的服务不可用。

HA 实现：
由 HAService 服务实现，负责同步双写，异步复制功能。（这里同步的是）
HAConnection：有两个 socketService，一个是 WriteSocketService,是用来向 slave 写数据，ReadSocketService 是用来读取 slave 请求的。

SlaveSynchronize: Slave 从 Master 同步信息（非消息）
Broker（slave）在初始化时，会去同步 master 的信息。

```
public void syncAll() {
    this.syncTopicConfig();
    this.syncConsumerOffset();
    this.syncDelayOffset();
    this.syncSubscriptionGroupConfig();
}
```

syncTopicConfig:从 master 获取 TopicConfig 信息，这里有一个 DataVersion，主要是用来验证同步成功，通过
Namesrv 实现：
Namesrv 其实只需要关注 NamesrvController、KVConfigManager、RouteInfoManager 三个类即可。第一个是整个 namesrv 控制器，管理底层所有关于 topic、broker 路由信息、定期更新、处理请求。KVCOnfigManager 维护了一个二维 Map，键为 Namespace，值为键值 Hash 结构。这个数据结构可以通过 json 格式 load 到内存，也可以持久化，我们可以调用其中的 api 进行相应处理。RouteInfoManager 维护了四个很重要的数据结构，包括每个 topic 下的队列信息、每个 brokerName 下的 Broker 信息、每个 clusterName 下的 brokerName 信息、每个 brokerAddr 下的 Broker 具体连接很版本信息。这四个结构，是相关的，比如你知道了一个

topic，就可以查出相应的 brokername，知道了 brokerName 就可以查出 BrokerData(每个 brokername 下可以有多个 Broker 实例，包括 master 和 slaver),我们可以查出 brokerName 是在哪个集群下，而且可以获取的具体的管道连接实例。这些路由信息值存在内存中，nameSrv 宕机后就会消失，broker 会定期推送新数据。

Offset 消费进度管理：

MQPullConsumer 与 MQPushConsumer 都包含一个 OffsetStore，这个就是消费进度存储的功能类，可以通过客户端配置创建相应的实例，目前有两个类型：LocalFileOffsetStore 和 RemoteBrokerOffsetStore。前者直接存储在本地，后者则存储在远端的 Broker，比较可靠。

先看看 LocalFileOffsetStore:

1. 持久化到{user.home}/.rocketmq_offsets/{clientid}/{groupname}/offsets.json
2. load 函数是从默认路径下加载 offsetTable 到内存，updateOffset 是用来更新 offsetTable 结构内容的。readOffset 会根据是否先从内存读还是从存储器读，来读取相应 MessageQueue 的消费进度。PersistAll 是持久化队列消费进度到存储器。removeOffset 暂时还没实现。
3. RemoteBrokerOffsetStore 会根据 MessageQueue 结构从 broker 获取 offset 信息
   详细介绍这个获取过程，主要是调用了 fetchConsumeOffsetFromBroker

```
HashMap<Long/* brokerId */, String/* address */> map = this.brokerAddrTable.get(brokerNa
if (map != null && !map.isEmpty()) {
    // TODO 如果有多个Slave，可能会每次都选中相同的Slave，这里需要优化
    FOR_SEG: for (Map.Entry<Long, String> entry : map.entrySet()) {
        Long id = entry.getKey();
        brokerAddr = entry.getValue();
        if (brokerAddr != null) {
            found = true;
            if (MixAll.MASTER_ID == id) {
                slave = false;
                break FOR_SEG;
            }
            else {
                slave = true;
            }
            break;

        }
    } // end of for
}
```

   查询机器的顺序和逻辑（疑问）
4. 如果上一步获取的 Broker 为空，则客户端会更新 NameServer 的路由信息。然后再查找。此处对 Name Server 压力过大。
5. 与找到的 BrokerServer 建立连接，向 BrokerServer 发出获取 offset 的请求。此处是异步的。

Producer 产生消息流程：

在 rocketmq-client 有一个 producer 默认实现：DefaultMQProducerImp 类，这个是客户端 producer 的默认实现，这里说明一些关键点：

逻辑队列刷盘服务 FlushConsumeQueueService：定期将 consumerQueue 队列上的数据刷到磁盘上，可以指定刷多少 page。核心是调用 consumeQueue 的 commit 函数实现的，MapedFileQueue 将剩下的未刷盘的 mapedFile 刷新到磁盘中。

索引构建：IndexFile、IndexHeader、IndexService、QueryOffsetResult
这四个类之间的关系是，IndexFile 包括 IndexHeader（索引头），IndexService 包含一个索引队列(IndexFile 数组)，是索引服务的核心类，QueryOffsetResult 是查询索引返回的结果类
先看索引头结构：INDEX_HEADER_SIZE=40,说明索引头是 40 字节。里面包括：
8 字节的 beginTimestamp、8 字节的 endTimestamp、8 字节的启始偏移量(beginPhyOffset)、8 字节的结束偏移量（endPhyOffset），4 字节的槽数量(hashSlotCount)、4 字节的索引个数(indexCount)
着重看一下 IndexFile:

```
int fileTotalSize =
        IndexHeader.INDEX_HEADER_SIZE + (hashSlotNum * HASH_SLOT_SIZE) + (indexNum * INDEX_SIZE);
```

这一句代码说明，IndexFile 所映射的文件大小为(索引头 40 字节+槽数量*槽大小 4 字节（槽记录的是在写入最后一个索引后的 indexCount 信息） + 索引个数*索引结构大小 20 字节)
Putkey 函数：(这个 key 是啥意思，这个索引最后用来干嘛)
参数：key（topic+”#”+key）,phyoffset(commitlog 的偏移量)，storeTimestamp 存储时间戳

1. 首先将 key 取 hash 值，hash 后的值与槽数量取模，得到槽位置，读取 slotvalue 值，判断是否大于当前的索引头的 indexCount（其实必须小于 1）或者小于 0，如果是，则赋予 0。
2. 找到真正索引的位置：索引头大小+槽数量*槽大小+索引个数+索引大小
3. 写入 hash 值、偏移量、存储时间和初始时间戳(索引头的初始时间)的差值、槽的值(当前索引头的 IndexCount 值)
4. 索引头 IndexCount+1、索引头槽数量+1、索引头更新最后一次写入的 commitlog 的偏移量、索引头更新最后一次存储的时间戳

在 IndexService 中有一个函数 queryOffset 函数，用来获取 topic+key 组合下组成的键所对应的 begin~end 时戳下的 commitlog 偏移量。调用这个函数的是 MessageStore 的 queryMessage，这个函数通过建立索引来获取 topic 的 commitlog，这样可以快速定位相应的 topic 下的数据。
我们来看看这个函数的实现：
首先是 queryMessage:
通过调用 indexService 的 queryOffset 函数，获取所有 topic+key 下的偏移量，读取 commitlog，返回对应的 message 信息。
具体如何查询：
遍历所有的 index 索引文件(从最后一个遍历)，每一个 index 文件都是一个 indexFile 实例，然后调用的是 selectPhyOffset 函数，首先要定位槽的位置，通过取模可以获取槽的位置，

```
int slotPos = Math.abs(keyHash) % this.hashSlotNum;
int absSlotPos = IndexHeader.INDEX_HEADER_SIZE + slotPos * HASH_SLOT_SIZE;
```

读取当前槽的值，这个值就是 index 信息的 indexCount,通过一个 for 循环不断向前遍历，读取相应的索引信息

```
int absIndexPos =
        IndexHeader.INDEX_HEADER_SIZE + this.hashSlotNum * HASH_SLOT_SIZE
                + nextIndexToRead * INDEX_SIZE;

int keyHashRead = this.mappedByteBuffer.getInt(absIndexPos);
long phyOffsetRead = this.mappedByteBuffer.getLong(absIndexPos + 4);
int timeDiff = this.mappedByteBuffer.getInt(absIndexPos + 4 + 8);
int prevIndexRead = this.mappedByteBuffer.getInt(absIndexPos + 4 + 8 + 4);
```

整个过程是需要锁文件的，所以效率比较慢。在读的时候是无法写的。

Broker 如何感知其他的 borker 的存在：首先必须先找到 NameServer:

优先级由高到低，高优先级会覆盖低优先级。

一、代码中指定 Name Server 地址

```
producer.setNamesrvAddr("192.168.0.1:9876;192.168.0.2:9876");
或
consumer.setNamesrvAddr("192.168.0.1:9876;192.168.0.2:9876");
```

二、Java 启动参数中指定 Name Server 地址

```
-Drocketmq.namesrv.addr=192.168.0.1:9876;192.168.0.2:9876
```

三、环境变量指定 Name Server 地址

```
export NAMESRV_ADDR=192.168.0.1:9876;192.168.0.2:9876
```

四、HTTP 静态服务器寻址（默认）

客户端启动后，会定时访问一个静态 HTTP 服务器，地址如下：

http://jmenv.tbsite.net:8080/rocketmq/nsaddr

这个 URL 的返回内容如下

```
192.168.0.1:9876;192.168.0.2:9876
```

客户端默认每隔 2 分钟访问一次这个 HTTP 服务器，并更新本地的 Name Server 地址。

URL 已经在代码中写死，可通过修改/etc/hosts 文件来改变要访问的服务器，例如在/etc/hosts 增加如下配置

```
10.232.22.67    jmenv.taobao.net
```

推荐使用 HTTP 静态服务器寻址方式，好处是客户端部署简单，且 Name Server 集群可以热升级。

```
public static final String WS_DOMAIN_NAME = System.getProperty("rocketmq.namesrv.domain",
    "jmenv.tbsite.net");
public static final String MESSAGE_COMPRESS_LEVEL = "rocketmq.message.compressLevel";
// http://jmenv.tbsite.net:8080/rocketmq/nsaddr
public static final String WS_ADDR = "http://" + WS_DOMAIN_NAME + ":8080/rocketmq/nsaddr";
```

写死了，真恶心啊

可以指定多个 NameServer，以；分割

每一个 Broker 都可以调用 registerBroker 向多有的 namesrvList 注册自己。

```
try {
    RegisterBrokerResult result =
            this.registerBroker(namesrvAddr, clusterName, brokerAddr, brokerName, brokerId,
                haServerAddr, topicConfigWrapper);
    if (result != null) {
        registerBrokerResult = result;
    }
```

看一下 nameServer 是如何注册每个 Broker，在 NameSvere 工程中：
DefaultRequestProcessor 处理类。
registerBroker 函数：首先，它会获取传过来的 TopicConfigSerializeWrapper 实例，然后调用 RouteInfoManager 的 registerBroker 函数,这个函数会先将 clusterName 加入到 map 对象中，然后更新主备信息(通过 brokerName 获取 BrokerDate,更新这个结构后，从新 add 到 brokerAddrTable HashMap 中)，更新 topic 信息，这个只针对主 broker（slave 会从 master 主动读取 Topic 信息）。调用 createAndUpdateQueueData 函数更新队列信息

重建同名的 topic，还会收到被删除的 topic 的原来消息
？
那岂不是原来本来想删掉的数据，在 topic 重建之后还是会再收到？
就是说：我删掉了 topic，其实消息数据还是存在的，当我重建的时候，想要不再接收原来的历史数据，我就得选择从删除 topic 的那个时间点开始订阅
这样才能保证数据都是新的是吧
不能了，　广播不支持失败重试。
广播只 push 一次，