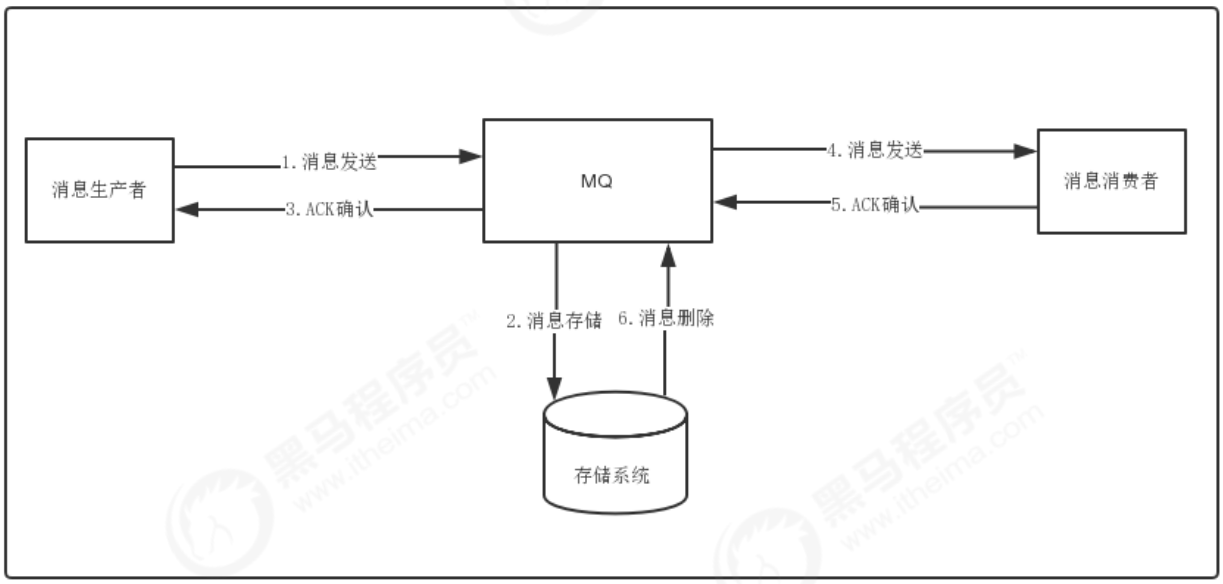
1. 高级功能

1.1 消息存储

分布式队列因为有高可靠性的要求，所以数据要进行持久化存储。



1. 消息生成者发送消息
2. MQ收到消息，将消息进行持久化，在存储中新增一条记录
3. 返回ACK给生产者
4. MQ push 消息给对应的消费者，然后等待消费者返回ACK
5. 如果消息消费者在指定时间内成功返回ack，那么MQ认为消息消费成功，在存储中删除消息，即执行第6步；如果MQ在指定时间内没有收到ACK，则认为消息消费失败，会尝试重新push消息,重复执行4、5、6步骤
6. MQ删除消息

1.1.1 存储介质

- 关系型数据库DB

Apache下开源的另外一款MQ—ActiveMQ（默认采用的KahaDB做消息存储）可选用JDBC的方式来做消息持久化，通过简单的xml配置信息即可实现JDBC消息存储。由于，普通关系型数据库（如Mysql）在单表数据量达到千万级别的情况下，其IO读写性能往往会出现瓶颈。在可靠性方面，该种方案非常依赖DB，如果一旦DB出现故障，则MQ的消息就无法落盘存储会导致线上故障

- 文件系统

  目前业界较为常用的几款产品（RocketMQ/Kafka/RabbitMQ）均采用的是消息刷盘至所部署虚拟机/物理机的文件系统来做持久化（刷盘一般可以分为异步刷盘和同步刷盘两种模式）。消息刷盘为消息存储提供了一种高效率、高可靠性和高性能的数据持久化方式。除非部署MQ机器本身或是本地磁盘挂了，否则一般是不会出现无法持久化的故障问题。

1.1.2 性能对比

文件系统>关系型数据库DB

1.1.3 消息的存储和发送

1）消息存储

磁盘如果使用得当，磁盘的速度完全可以匹配上网络 的数据传输速度。目前的高性能磁盘，顺序写速度可以达到600MB/s， 超过了一般网卡的传输速度。但是磁盘随机写的速度只有大概100KB/s，和顺序写的性能相差6000倍！因为有如此巨大的速度差别，好的消息队列系统会比普通的消息队列系统速度快多个数量级。RocketMQ的消息用顺序写，保证了消息存储的速度。

2）消息发送

Linux操作系统分为【用户态】和【内核态】，文件操作、网络操作需要涉及这两种形态的切换，免不了进行数据复制。

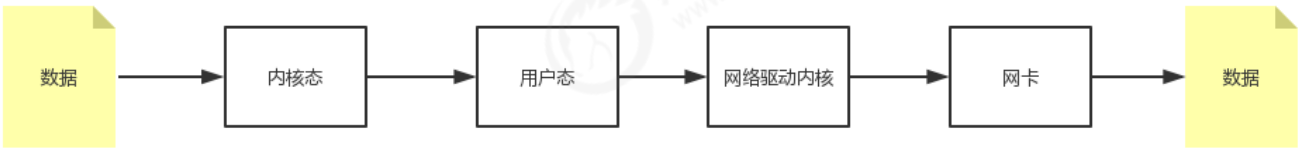
一台服务器 把本机磁盘文件的内容发送到客户端，一般分为两个步骤：

1）read；读取本地文件内容；

2）write；将读取的内容通过网络发送出去。

这两个看似简单的操作，实际进行了4 次数据复制，分别是：

1. 从磁盘复制数据到内核态内存；
2. 从内核态内存复 制到用户态内存；
3. 然后从用户态 内存复制到网络驱动的内核态内存；
4. 最后是从网络驱动的内核态内存复 制到网卡中进行传输。



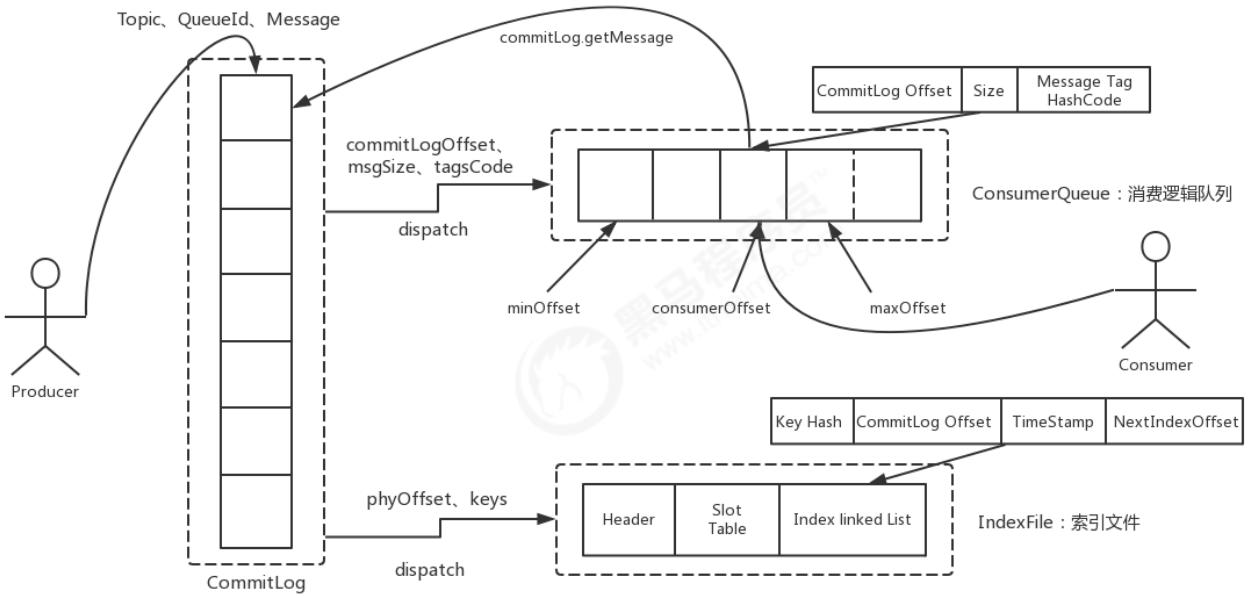
通过使用mmap的方式，可以省去向用户态的内存复制，提高速度。这种机制在Java中是通过MappedByteBuffer实现的

RocketMQ充分利用了上述特性，也就是所谓的“零拷贝”技术，提高消息存盘和网络发送的速度。

> 这里需要注意的是，采用MappedByteBuffer这种内存映射的方式有几个限制，其中之一是一次只能映射1.5~2G 的文件至用户态的虚拟内存，这也是为何RocketMQ默认设置单个CommitLog日志数据文件为1G的原因了

1.1.4 消息存储结构

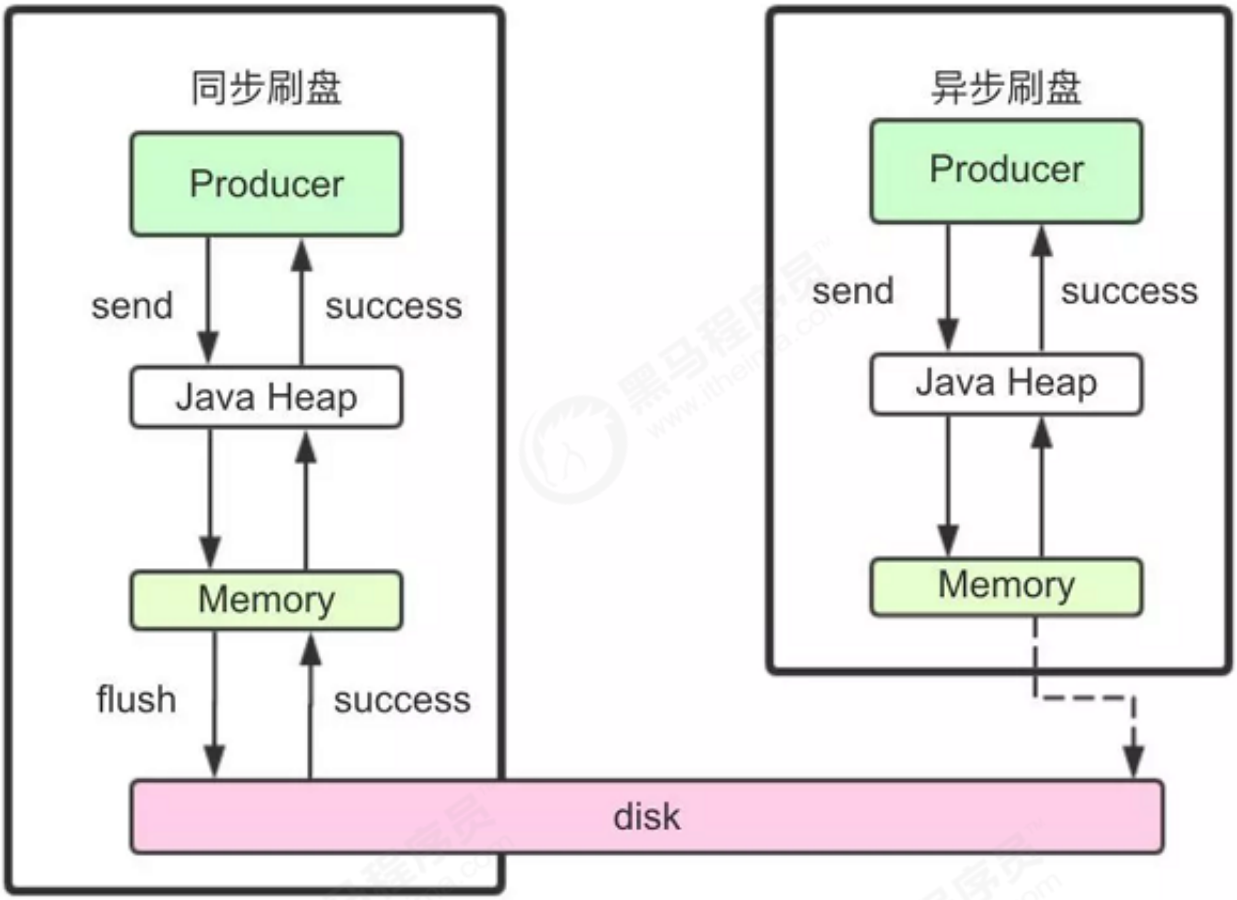
RocketMQ消息的存储是由ConsumeQueue和CommitLog配合完成 的，消息真正的物理存储文件是CommitLog，ConsumeQueue是消息的逻辑队列，类似数据库的索引文件，存储的是指向物理存储的地址。每 个Topic下的每个Message Queue都有一个对应的ConsumeQueue文件。



- CommitLog：存储消息的元数据
- ConsumerQueue：存储消息在CommitLog的索引
- IndexFile：为了消息查询提供了一种通过key或时间区间来查询消息的方法，这种通过IndexFile来查找消息的方法不影响发送与消费消息的主流程

1.1.5 刷盘机制

RocketMQ的消息是存储到磁盘上的，这样既能保证断电后恢复， 又可以让存储的消息量超出内存的限制。RocketMQ为了提高性能，会尽可能地保证磁盘的顺序写。消息在通过Producer写入RocketMQ的时 候，有两种写磁盘方式，分布式同步刷盘和异步刷盘。



1）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。

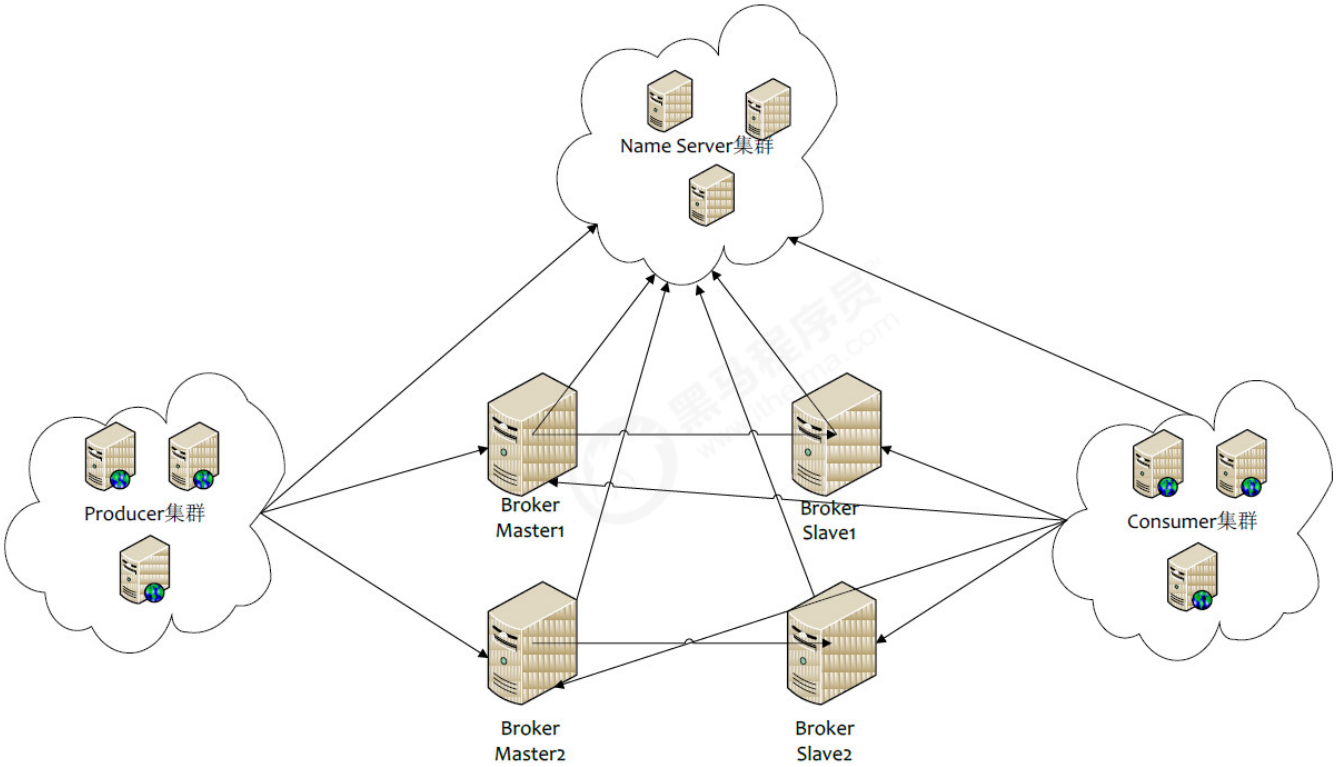
2）异步刷盘

在返回写成功状态时，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时，统一触发写磁盘动作，快速写入。

3）配置

同步刷盘还是异步刷盘，都是通过**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。

1.2 高可用性机制



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参数也会说明这个Broker是Master还是Slave。

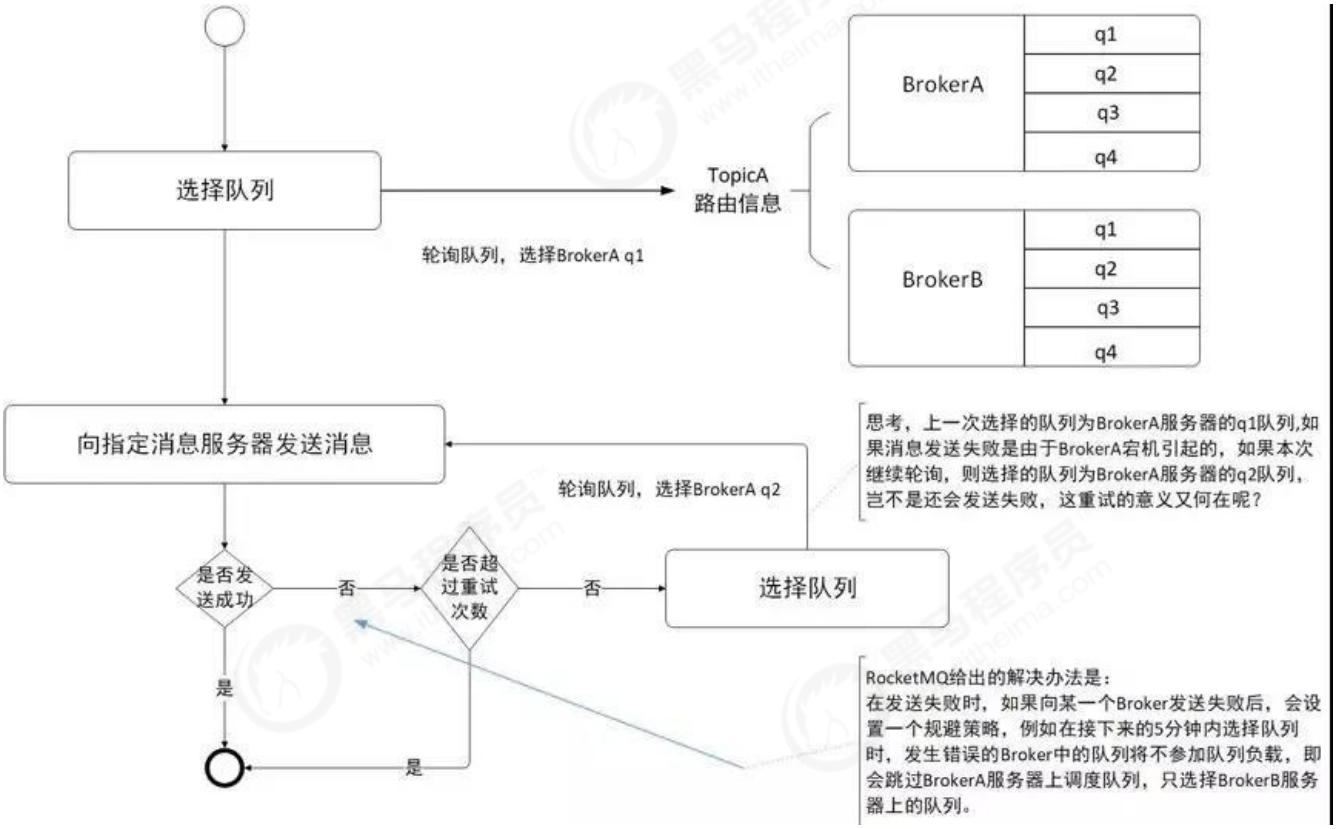
Master角色的Broker支持读和写，Slave角色的Broker仅支持读，也就是 Producer只能和Master角色的Broker连接写入消息；Consumer可以连接 Master角色的Broker，也可以连接Slave角色的Broker来读取消息。

1.2.1 消息消费高可用

在Consumer的配置文件中，并不需要设置是从Master读还是从Slave 读，当Master不可用或者繁忙的时候，Consumer会被自动切换到从Slave 读。有了自动切换Consumer这种机制，当一个Master角色的机器出现故障后，Consumer仍然可以从Slave读取消息，不影响Consumer程序。这就达到了消费端的高可用性。

1.2.2 消息发送高可用

在创建Topic的时候，把Topic的多个Message Queue创建在多个Broker组上（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 用后，其他组的Master仍然可用，Producer仍然可以发送消息。RocketMQ目前还不支持把Slave自动转成Master，如果机器资源不足， 需要把Slave转成Master，则要手动停止Slave角色的Broker，更改配置文 件，用新的配置文件启动Broker。



1.2.3 消息主从复制

如果一个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。

1）同步复制

同步复制方式是等Master和Slave均写 成功后才反馈给客户端写成功状态；

在同步复制方式下，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。

2）异步复制

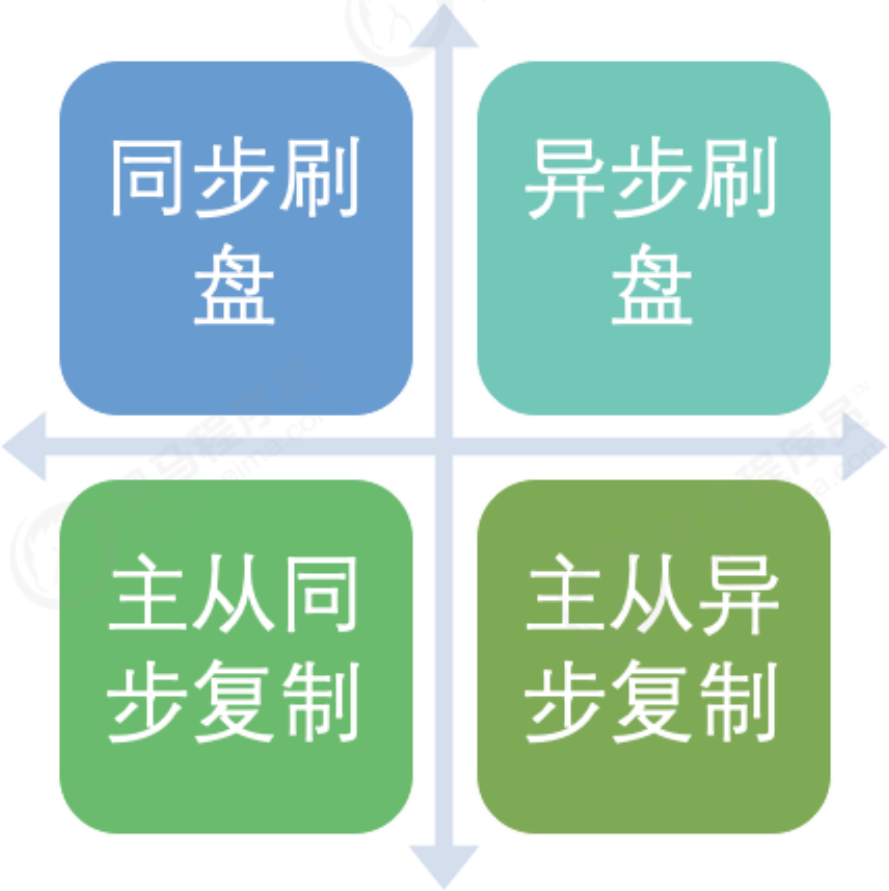
异步复制方式是只要Master写成功 即可反馈给客户端写成功状态。

在异步复制方式下，系统拥有较低的延迟和较高的吞吐量，但是如果Master出了故障，有些数据因为没有被写 入Slave，有可能会丢失；

3）配置

同步复制和异步复制是通过Broker配置文件里的brokerRole参数进行设置的，这个参数可以被设置成ASYNC_MASTER、 SYNC_MASTER、SLAVE三个值中的一个。

4）总结

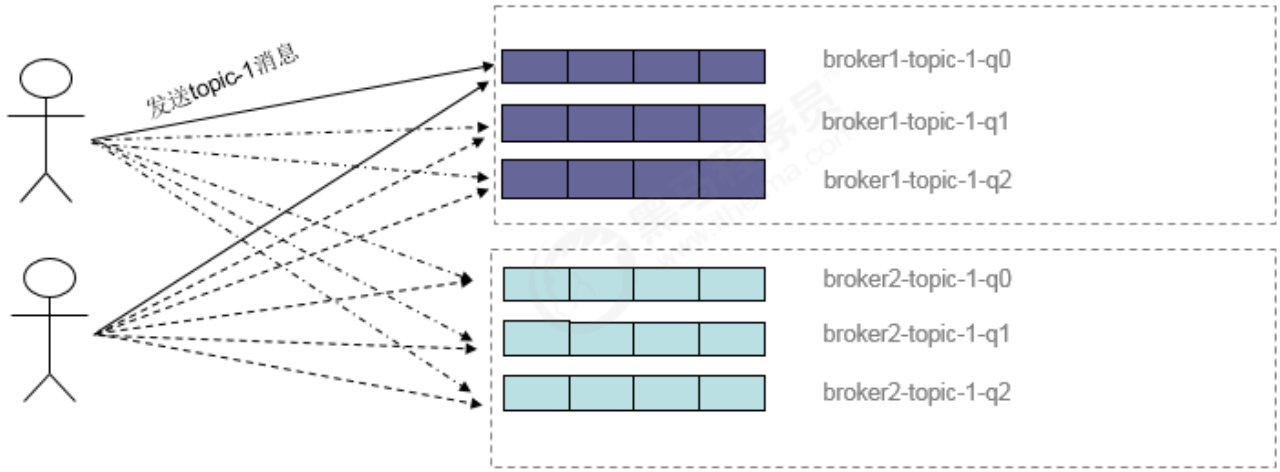


实际应用中要结合业务场景，合理设置刷盘方式和主从复制方式， 尤其是SYNC_FLUSH方式，由于频繁地触发磁盘写动作，会明显降低 性能。通常情况下，应该把Master和Save配置成ASYNC_FLUSH的刷盘 方式，主从之间配置成SYNC_MASTER的复制方式，这样即使有一台 机器出故障，仍然能保证数据不丢，是个不错的选择。

1.3 负载均衡

1.3.1 Producer负载均衡

Producer端，每个实例在发消息的时候，默认会轮询所有的message queue发送，以达到让消息平均落在不同的queue上。而由于queue可以散落在不同的broker，所以消息就发送到不同的broker下，如下图：



每个producer默认采用Roundbin方式轮训发送每个Queue

图中箭头线条上的标号代表顺序，发布方会把第一条消息发送至 Queue 0，然后第二条消息发送至 Queue 1，以此类推。

1.3.2 Consumer负载均衡

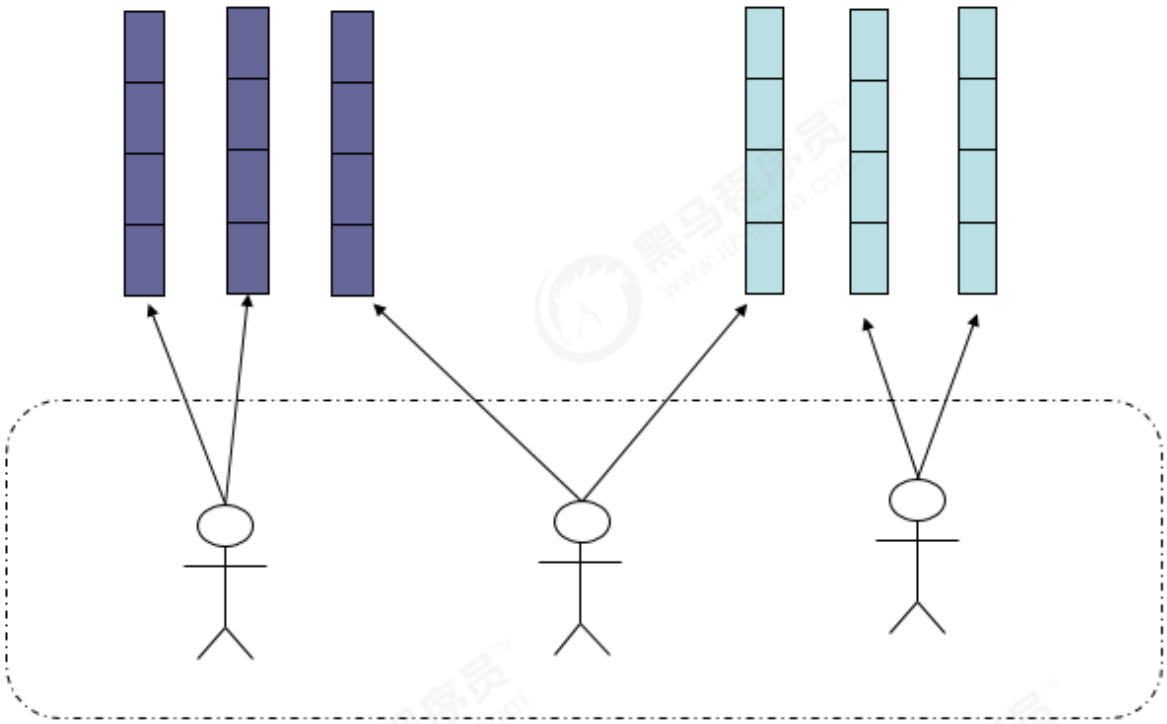
1）集群模式

在集群消费模式下，每条消息只需要投递到订阅这个topic的Consumer Group下的一个实例即可。RocketMQ采用主动拉取的方式拉取并消费消息，在拉取的时候需要明确指定拉取哪一条message queue。

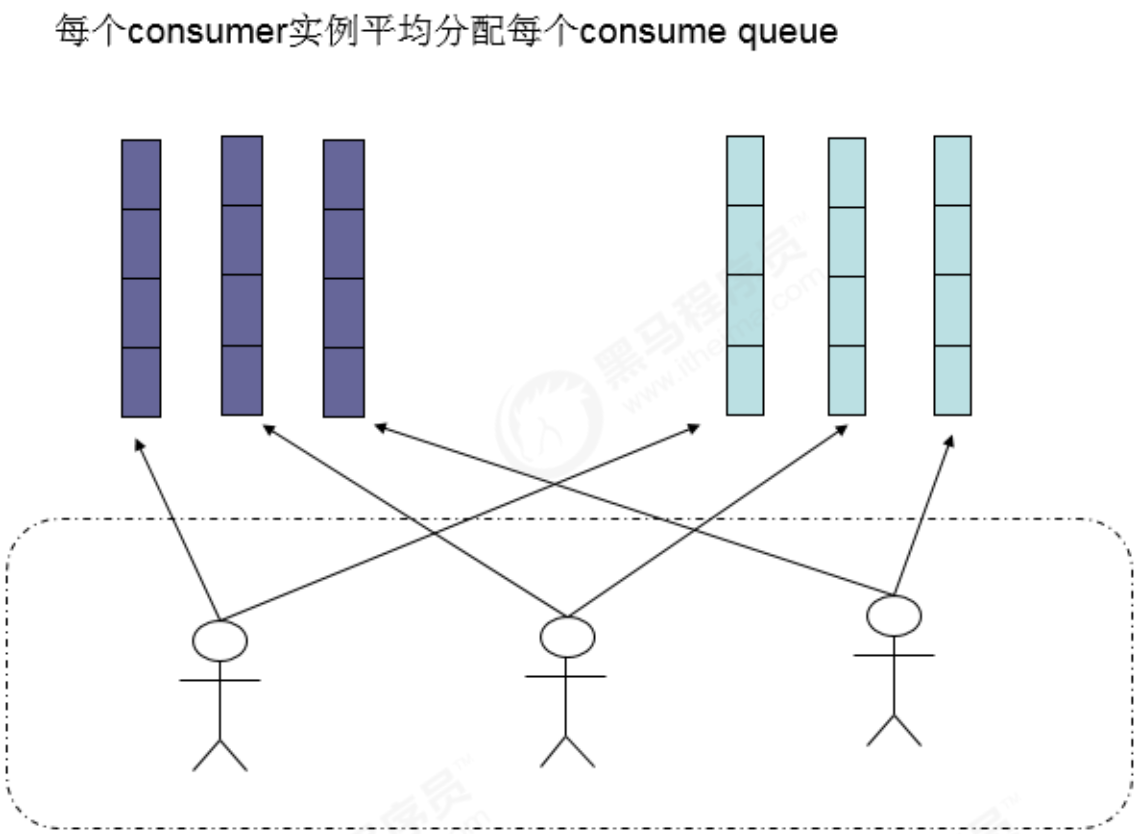
而每当实例的数量有变更，都会触发一次所有实例的负载均衡，这时候会按照queue的数量和实例的数量平均分配queue给每个实例。

默认的分配算法是AllocateMessageQueueAveragely，如下图：

每个consumer实例平均分配每个consume queue



还有另外一种平均的算法是AllocateMessageQueueAveragelyByCircle，也是平均分摊每一条queue，只是以环状轮流分queue的形式，如下图：



需要注意的是，集群模式下，queue都是只允许分配只一个实例，这是由于如果多个实例同时消费一个queue的消息，由于拉取哪些消息是consumer主动控制的，那样会导致同一个消息在不同的实例下被消费多次，所以算法上都是一个queue只分给一个consumer实例，一个consumer实例可以允许同时分到不同的queue。

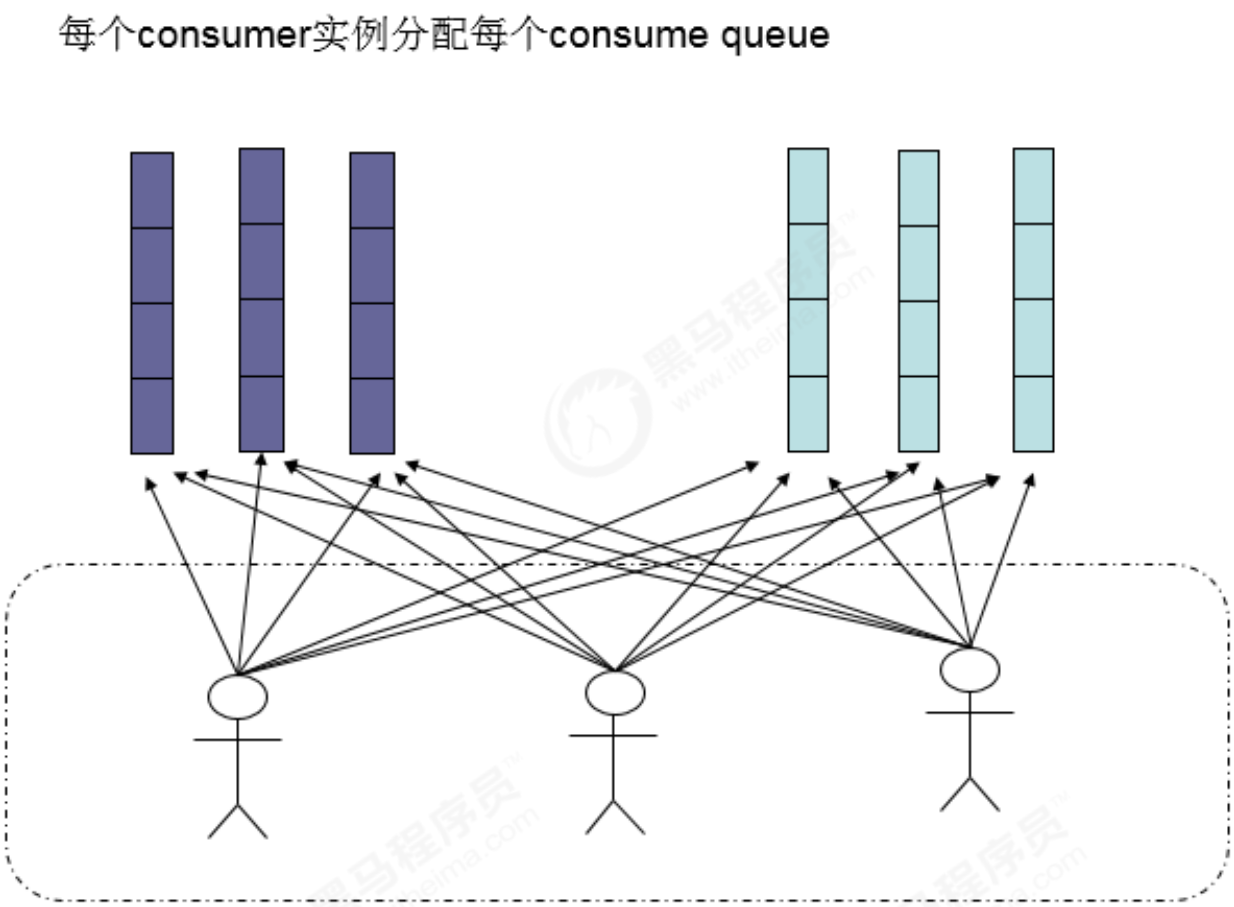
通过增加consumer实例去分摊queue的消费，可以起到水平扩展的消费能力的作用。而有实例下线的时候，会重新触发负载均衡，这时候原来分配到的queue将分配到其他实例上继续消费。

但是如果consumer实例的数量比message queue的总数量还多的话，多出来的consumer实例将无法分到queue，也就无法消费到消息，也就无法起到分摊负载的作用了。所以需要控制让queue的总数量大于等于consumer的数量。

2）广播模式

由于广播模式下要求一条消息需要投递到一个消费组下面所有的消费者实例，所以也就没有消息被分摊消费的说法。

在实现上，其中一个不同就是在consumer分配queue的时候，所有consumer都分到所有的queue。



1.4 消息重试

1.4.1 顺序消息的重试

对于顺序消息，当消费者消费消息失败后，消息队列 RocketMQ 会自动不断进行消息重试（每次间隔时间为 1 秒），这时，应用会出现消息消费被阻塞的情况。因此，在使用顺序消息时，务必保证应用能够及时监控并处理消费失败的情况，避免阻塞现象的发生。

1.4.2 无序消息的重试

对于无序消息（普通、定时、延时、事务消息），当消费者消费消息失败时，您可以通过设置返回状态达到消息重试的结果。

无序消息的重试只针对集群消费方式生效；广播方式不提供失败重试特性，即消费失败后，失败消息不再重试，继续消费新的消息。

1）重试次数

消息队列 RocketMQ 默认允许每条消息最多重试 16 次，每次重试的间隔时间如下：

第几次重试	与上次重试的间隔时间	第几次重试	与上次重试的间隔时间
1	10 秒	9	7 分钟
2	30 秒	10	8 分钟
3	1 分钟	11	9 分钟
4	2 分钟	12	10 分钟
5	3 分钟	13	20 分钟
6	4 分钟	14	30 分钟
7	5 分钟	15	1 小时
8	6 分钟	16	2 小时

如果消息重试 16 次后仍然失败，消息将不再投递。如果严格按照上述重试时间间隔计算，某条消息在一直消费失败的前提下，将会在接下来的 4 小时 46 分钟之内进行 16 次重试，超过这个时间范围消息将不再重试投递。

注意： 一条消息无论重试多少次，这些重试消息的 Message ID 不会改变。

2）配置方式

消费失败后，重试配置方式

集群消费方式下，消息消费失败后期望消息重试，需要在消息监听器接口的实现中明确进行配置（三种方式任选一种）：

- 返回 Action.ReconsumeLater （推荐）
- 返回 Null
- 抛出异常

```
public class MessageListenerImpl implements MessageListener {
    @Override
    public Action consume(Message message, ConsumeContext context) {
        //处理消息
        doConsumeMessage(message);
        //方式1：返回 Action.ReconsumeLater，消息将重试
        return Action.ReconsumeLater;
        //方式2：返回 null，消息将重试
        return null;
        //方式3：直接抛出异常， 消息将重试
        throw new RuntimeException("Consumer Message exceotion");
    }
}
```

消费失败后，不重试配置方式

集群消费方式下，消息失败后期望消息不重试，需要捕获消费逻辑中可能抛出的异常，最终返回 Action.CommitMessage，此后这条消息将不会再重试。

```
public class MessageListenerImpl implements MessageListener {
    @Override
    public Action consume(Message message, ConsumeContext context) {
        try {
            doConsumeMessage(message);
        } catch (Throwable e) {
            //捕获消费逻辑中的所有异常，并返回 Action.CommitMessage;
            return Action.CommitMessage;
        }
        //消息处理正常，直接返回 Action.CommitMessage;
        return Action.CommitMessage;
    }
}
```

自定义消息最大重试次数

消息队列 RocketMQ 允许 Consumer 启动的时候设置最大重试次数，重试时间间隔将按照如下策略：

- 最大重试次数小于等于 16 次，则重试时间间隔同上表描述。
- 最大重试次数大于 16 次，超过 16 次的重试时间间隔均为每次 2 小时。

```
Properties properties = new Properties();
//配置对应 Group ID 的最大消息重试次数为 20 次
properties.put(PropertyKeyConst.MaxReconsumeTimes,"20");
Consumer consumer =ONSFactory.createConsumer(properties);
```

> 注意：

- 消息最大重试次数的设置对相同 Group ID 下的所有 Consumer 实例有效。
- 如果只对相同 Group ID 下两个 Consumer 实例中的其中一个设置了 MaxReconsumeTimes，那么该配置对两个 Consumer 实例均生效。
- 配置采用覆盖的方式生效，即最后启动的 Consumer 实例会覆盖之前的启动实例的配置

获取消息重试次数

消费者收到消息后，可按照如下方式获取消息的重试次数：

```
public class MessageListenerImpl implements MessageListener {
    @Override
    public Action consume(Message message, ConsumeContext context) {
        //获取消息的重试次数
        System.out.println(message.getReconsumeTimes());
        return Action.CommitMessage;
    }
}
```

1.5 死信队列

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

在消息队列 RocketMQ 中，这种正常情况下无法被消费的消息称为死信消息（Dead-Letter Message），存储死信消息的特殊队列称为死信队列（Dead-Letter Queue）。

1.5.1 死信特性

死信消息具有以下特性

- 不会再被消费者正常消费。

- 有效期与正常消息相同，均为 3 天，3 天后会被自动删除。因此，请在死信消息产生后的 3 天内及时处理。

死信队列具有以下特性：

- 一个死信队列对应一个 Group ID， 而不是对应单个消费者实例。
- 如果一个 Group ID 未产生死信消息，消息队列 RocketMQ 不会为其创建相应的死信队列。
- 一个死信队列包含了对应 Group ID 产生的所有死信消息，不论该消息属于哪个 Topic。

1.5.2 查看死信信息

1. 在控制台查询出现死信队列的主题信息

1. 在消息界面根据主题查询死信消息

1. 选择重新发送消息

一条消息进入死信队列，意味着某些因素导致消费者无法正常消费该消息，因此，通常需要您对其进行特殊处理。排查可疑因素并解决问题后，可以在消息队列 RocketMQ 控制台重新发送该消息，让消费者重新消费一次。

1.6 消费幂等

消息队列 RocketMQ 消费者在接收到消息以后，有必要根据业务上的唯一 Key 对消息做幂等处理的必要性。

1.6.1 消费幂等的必要性

在互联网应用中，尤其在网络不稳定的情况下，消息队列 RocketMQ 的消息有可能会出现重复，这个重复简单可以概括为以下情况：

- 发送时消息重复

  当一条消息已被成功发送到服务端并完成持久化，此时出现了网络闪断或者客户端宕机，导致服务端对客户端应答失败。 如果此时生产者意识到消息发送失败并尝试再次发送消息，消费者后续会收到两条内容相同并且 Message ID 也相同的消息。

- 投递时消息重复

  消息消费的场景下，消息已投递到消费者并完成业务处理，当客户端给服务端反馈应答的时候网络闪断。 为了保证消息至少被消费一次，消息队列 RocketMQ 的服务端将在网络恢复后再次尝试投递之前已被处理过的消息，消费者后续会收到两条内容相同并且 Message ID 也相同的消息。

- 负载均衡时消息重复（包括但不限于网络抖动、Broker 重启以及订阅方应用重启）

  当消息队列 RocketMQ 的 Broker 或客户端重启、扩容或缩容时，会触发 Rebalance，此时消费者可能会收到重复消息。

1.6.2 处理方式

因为 Message ID 有可能出现冲突（重复）的情况，所以真正安全的幂等处理，不建议以 Message ID 作为处理依据。 最好的方式是以业务唯一标识作为幂等处理的关键依据，而业务的唯一标识可以通过消息 Key 进行设置：

```
Message message = new Message();
message.setKey("ORDERID_100");
SendResult sendResult = producer.send(message);
```

订阅方收到消息时可以根据消息的 Key 进行幂等处理：

```
consumer.subscribe("ons_test", "*", new MessageListener() {
    public Action consume(Message message, ConsumeContext context) {
        String key = message.getKey()
        // 根据业务唯一标识的 key 做幂等处理
    }
});
```

2. 源码分析

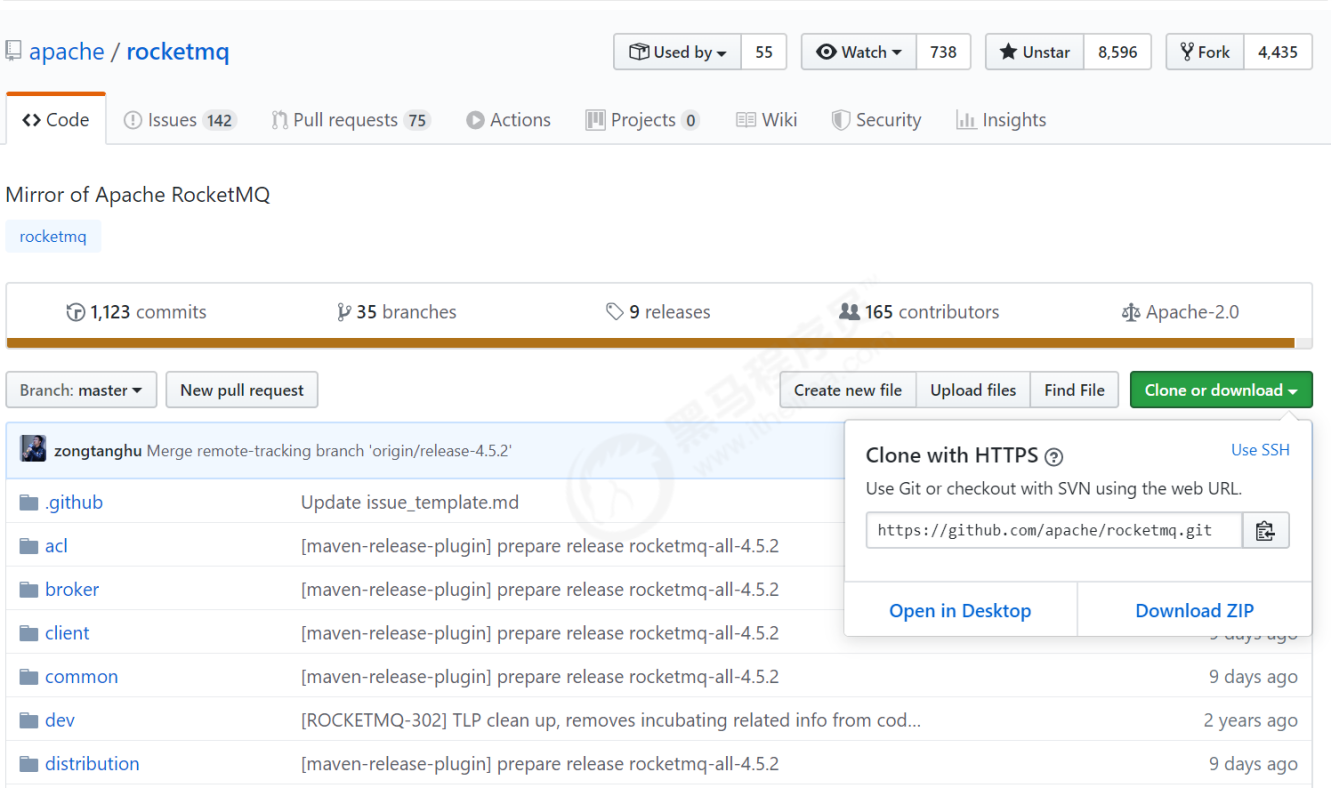
2.1 环境搭建

依赖工具

- JDK ： 1.8+
- Maven
- IntelliJ IDEA

2.1.1 源码拉取

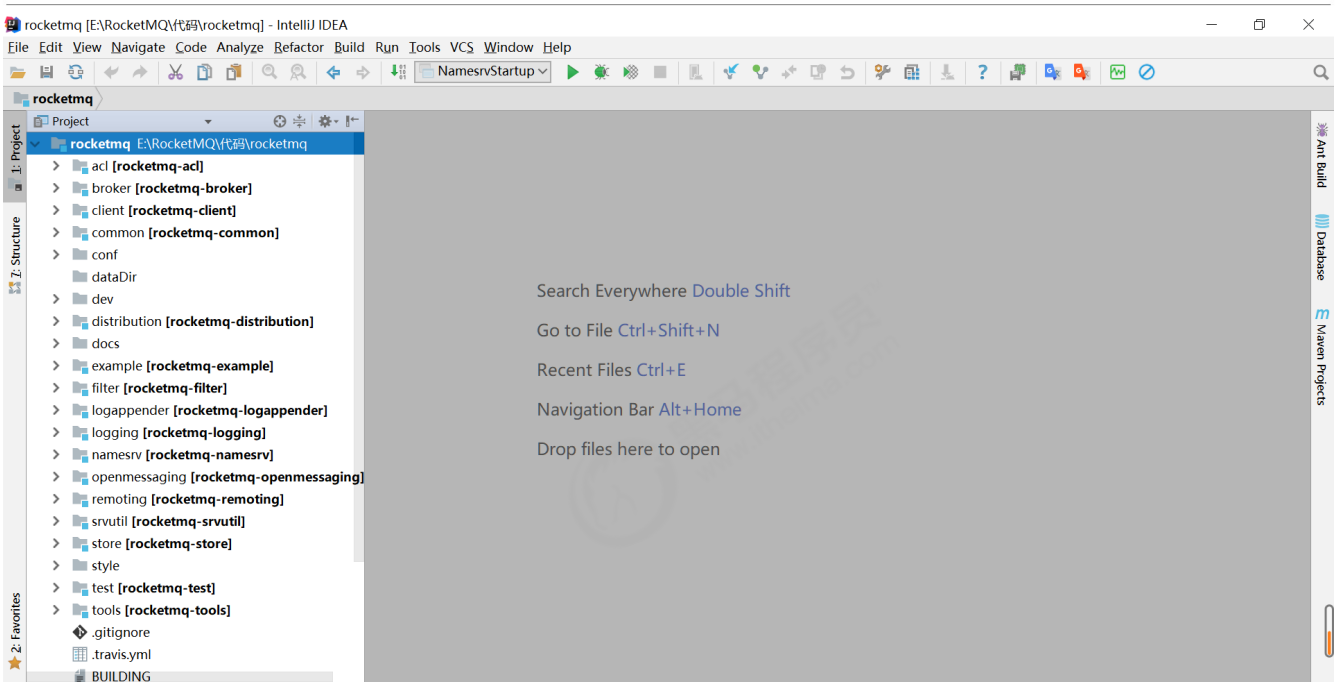
从官方仓库 https://github.com/apache/rocketmq `clone` 或者 `download` 源码。



源码目录结构：

- broker: broker 模块（broke 启动进程）
- client ：消息客户端，包含消息生产者、消息消费者相关类
- common ：公共包
- dev ：开发者信息（非源代码）
- distribution ：部署实例文件夹（非源代码）
- example: RocketMQ 例代码
- filter ：消息过滤相关基础类
- filtersrv：消息过滤服务器实现相关类（Filter启动进程）
- logappender：日志实现相关类
- namesrv：NameServer实现相关类（NameServer启动进程）
- openmessageing：消息开放标准
- remoting：远程通信模块，给予Netty
- srcutil：服务工具类
- store：消息存储实现相关类
- style：checkstyle相关实现
- test：测试相关类
- tools：工具类，监控命令相关实现类

2.1.2 导入IDEA

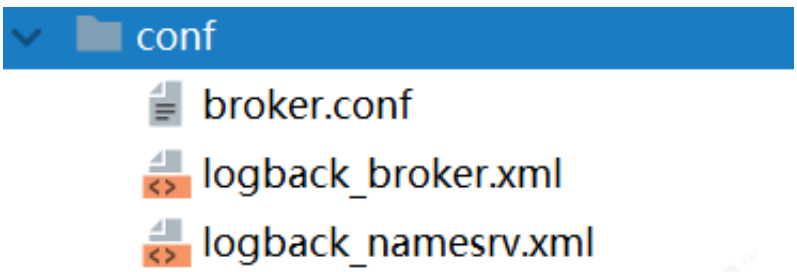


执行安装

```
clean install -Dmaven.test.skip=true
```

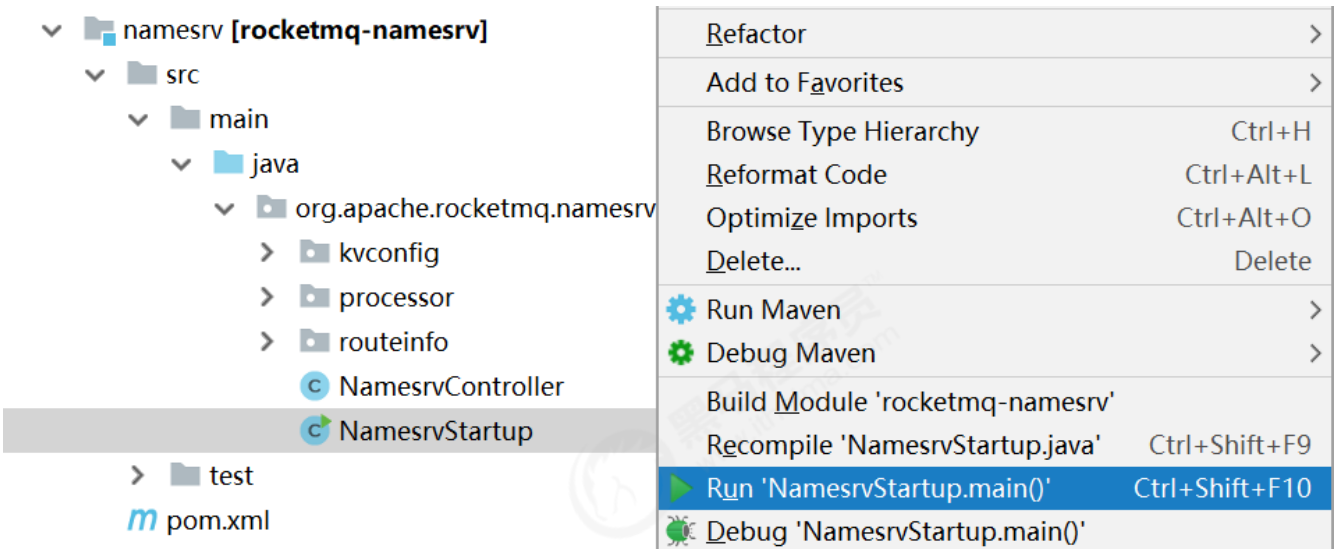
2.1.3 调试

创建`conf`配置文件夹,从`distribution`拷贝`broker.conf`和`logback_broker.xml`和`logback_namesrv.xml`

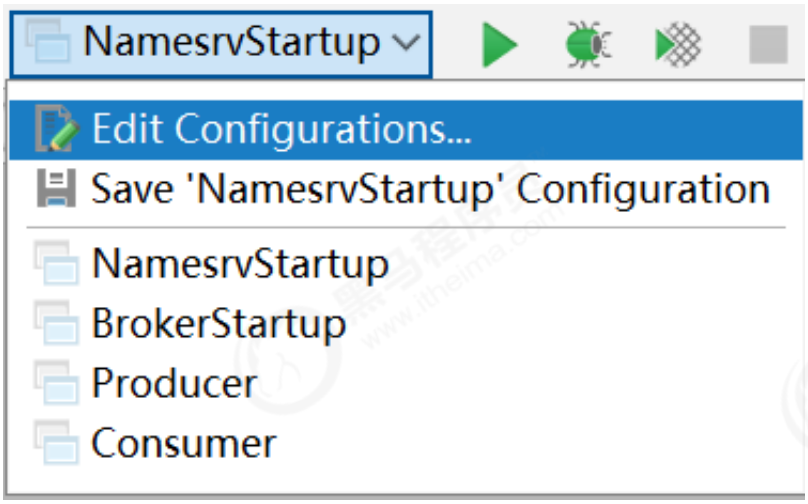


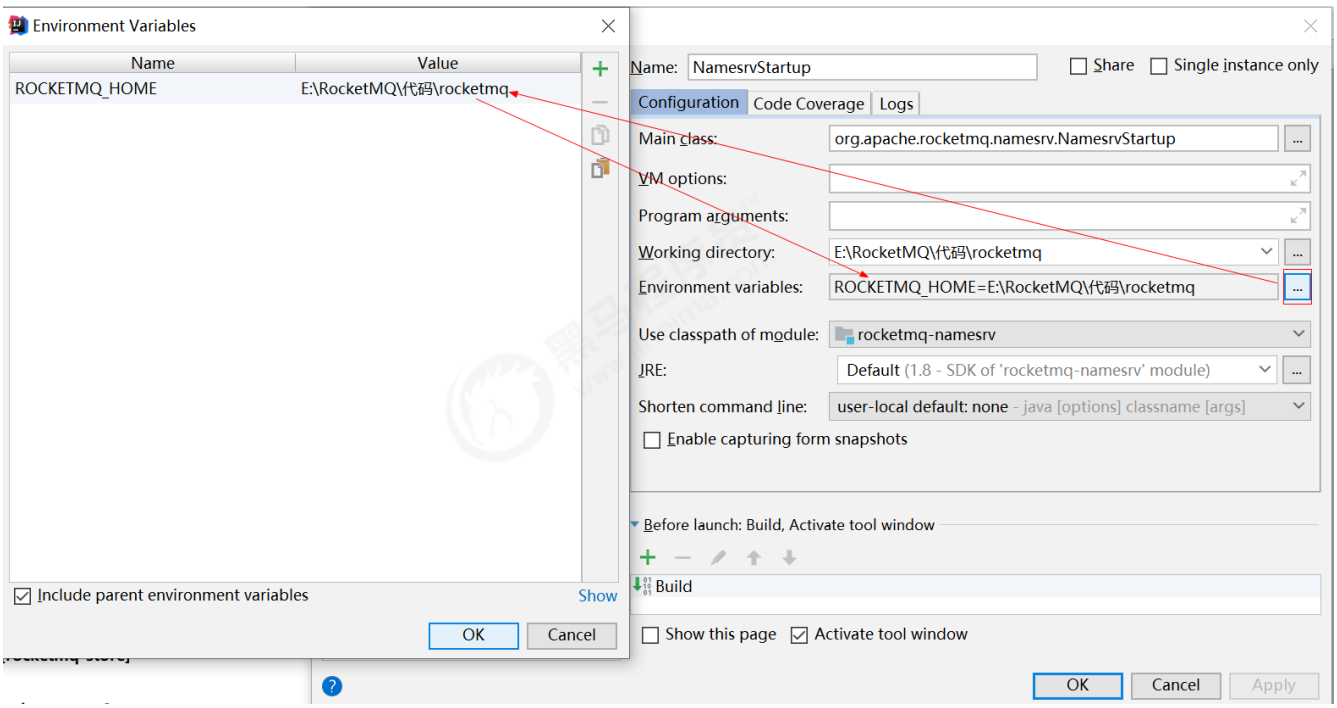
1）启动NameServer

- 展开namesrv模块，右键NamesrvStartup.java



- 配置**ROCKETMQ_HOME**





- 重新启动

  控制台打印结果

```
The Name Server boot success. serializeType=JSON
```

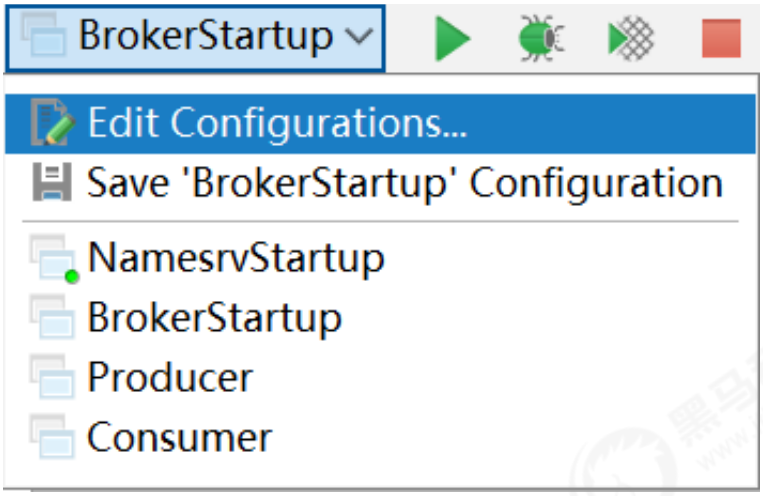
2）启动Broker

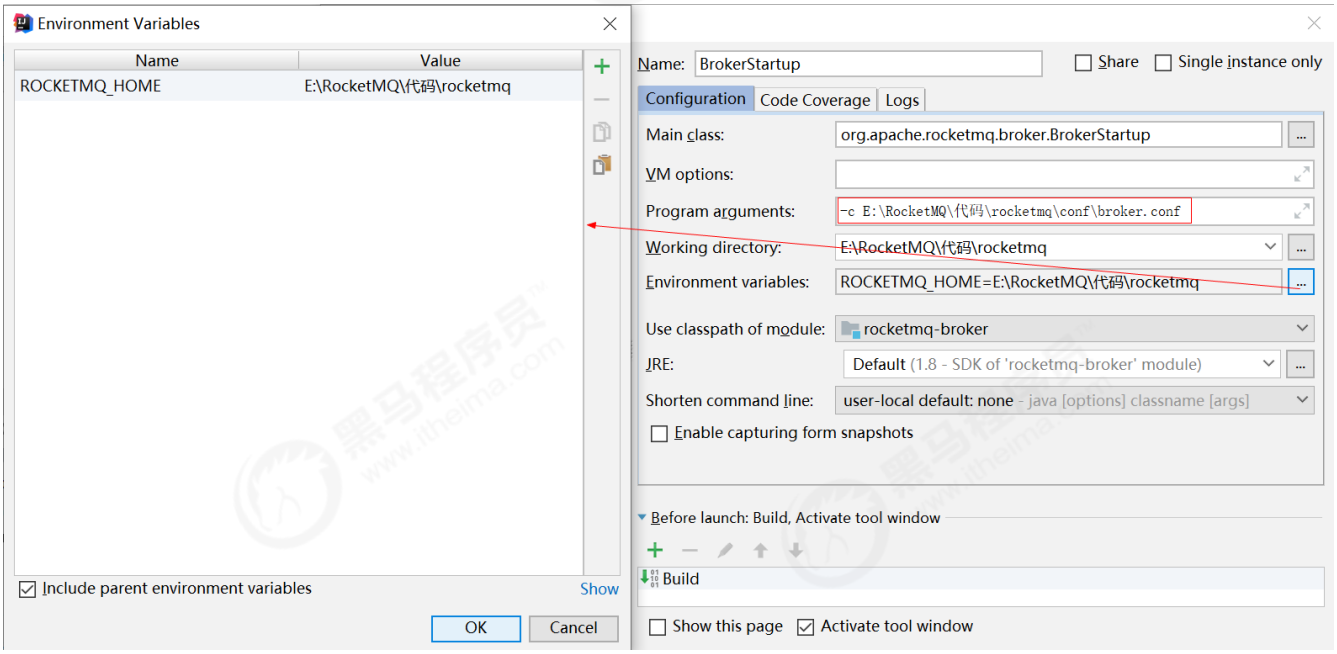
- `broker.conf`配置文件内容

```
brokerClusterName = DefaultCluster
brokerName = broker-a
brokerId = 0
# namesrvAddr地址
namesrvAddr=127.0.0.1:9876
deleteWhen = 04
fileReservedTime = 48
brokerRole = ASYNC_MASTER
flushDiskType = ASYNC_FLUSH
autoCreateTopicEnable=true

# 存储路径
storePathRootDir=E:\\RocketMQ\\data\\rocketmq\\dataDir
# commitLog路径
storePathCommitLog=E:\\RocketMQ\\data\\rocketmq\\dataDir\\commitlog
# 消息队列存储路径
storePathConsumeQueue=E:\\RocketMQ\\data\\rocketmq\\dataDir\\consumequeue
# 消息索引存储路径
storePathIndex=E:\\RocketMQ\\data\\rocketmq\\dataDir\\index
# checkpoint文件路径
storeCheckpoint=E:\\RocketMQ\\data\\rocketmq\\dataDir\\checkpoint
# abort文件存储路径
abortFile=E:\\RocketMQ\\data\\rocketmq\\dataDir\\abort
```

- 创建数据文件夹`dataDir`
- 启动`BrokerStartup`,配置`broker.conf`和`ROCKETMQ_HOME`





3）发送消息

- 进入example模块的`org.apache.rocketmq.example.quickstart`
- 指定Namesrv地址

```
DefaultMQProducer producer = new
DefaultMQProducer("please_rename_unique_group_name");
producer.setNamesrvAddr("127.0.0.1:9876");
```

- 运行`main`方法，发送消息

4）消费消息

- 进入example模块的`org.apache.rocketmq.example.quickstart`

- 指定Namesrv地址

```
DefaultMQPushConsumer consumer = new
DefaultMQPushConsumer("please_rename_unique_group_name_4");
consumer.setNamesrvAddr("127.0.0.1:9876");
```

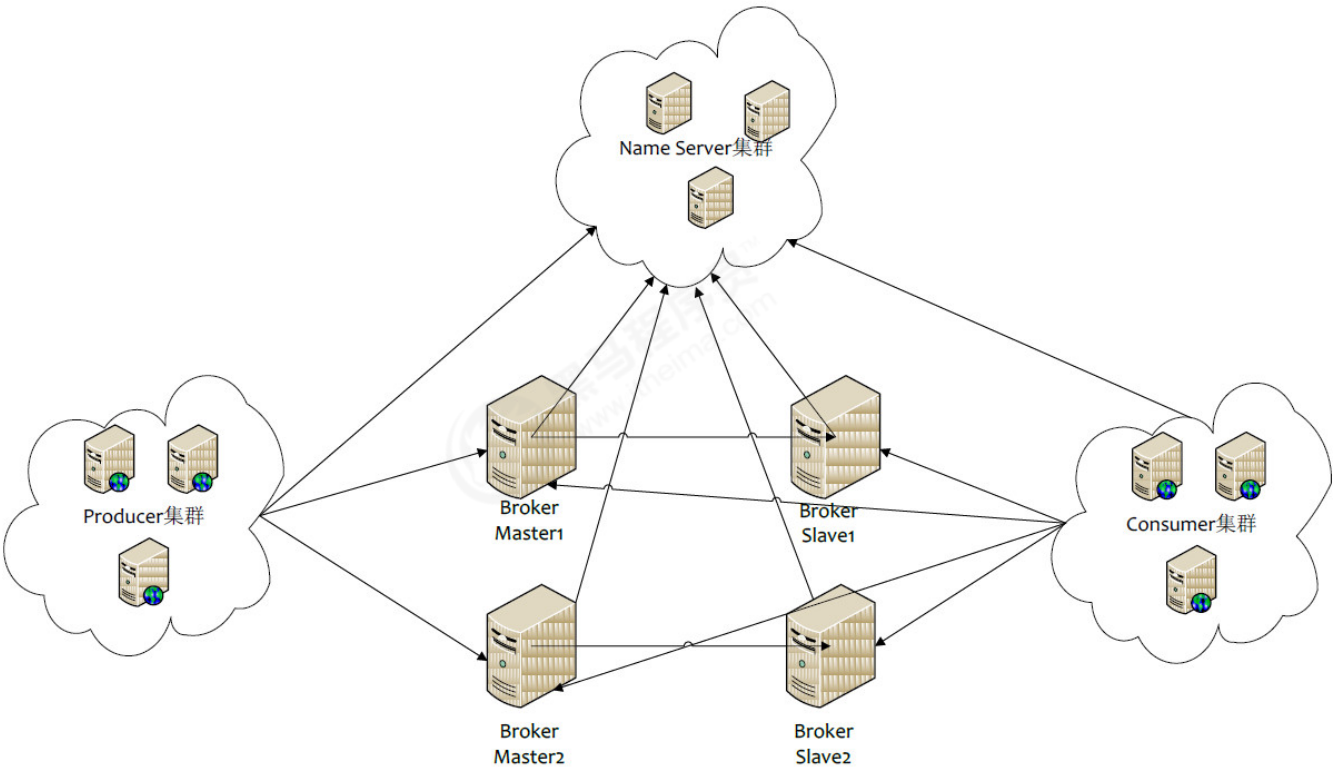
- 运行`main`方法，消费消息

2.2 NameServer

2.2.1 架构设计

消息中间件的设计思路一般是基于主题订阅发布的机制，消息生产者（Producer）发送某一个主题到消息服务器，消息服务器负责将消息持久化存储，消息消费者（Consumer）订阅该兴趣的主题，消息服务器根据订阅信息（路由信息）将消息推送到消费者（Push模式）或者消费者主动向消息服务器拉去（Pull模式），从而实现消息生产者与消息消费者解耦。为了避免消息服务器的单点故障导致的整个系统瘫痪，通常会部署多台消息服务器共同承担消息的存储。那消息生产者如何知道消息要发送到哪台消息服务器呢？如果某一台消息服务器宕机了，那么消息生产者如何在不重启服务情况下感知呢？

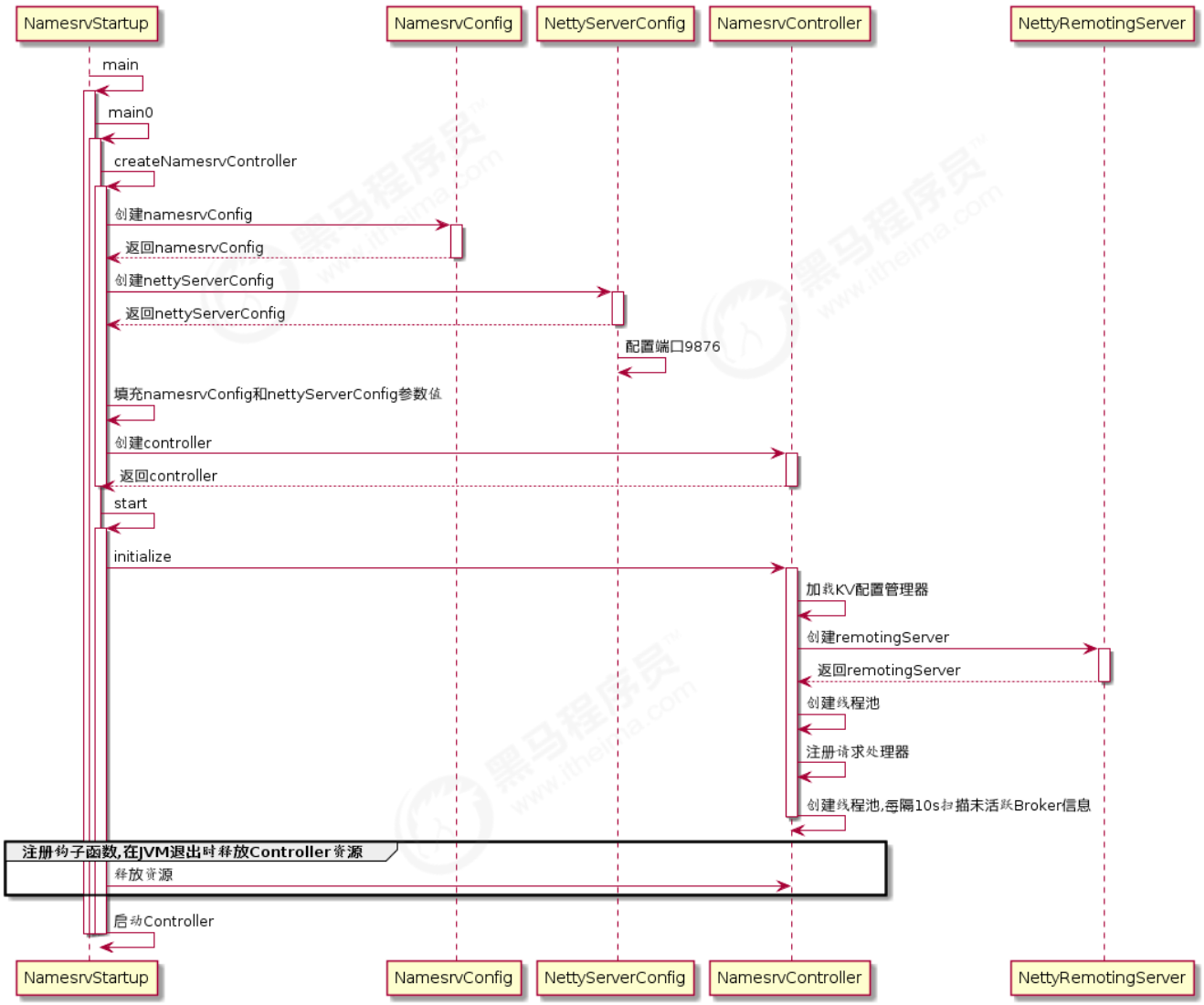
NameServer就是为了解决以上问题设计的。



Broker消息服务器在启动的时向所有NameServer注册，消息生产者（Producer）在发送消息时之前先从NameServer获取Broker服务器地址列表，然后根据负载均衡算法从列表中选择一台服务器进行发送。NameServer与每台Broker保持长连接，并间隔30S检测Broker是否存活，如果检测到Broker宕机，则从路由注册表中删除。但是路由变化不会马上通知消息生产者。这样设计的目的是为了降低NameServer实现的复杂度，在消息发送端提供容错机制保证消息发送的可用性。

NameServer本身的高可用是通过部署多台NameServer来实现，但彼此之间不通讯，也就是NameServer服务器之间在某一个时刻的数据并不完全相同，但这对消息发送并不会造成任何影响，这也是NameServer设计的一个亮点，总之，RocketMQ设计追求简单高效。

2.2.2 启动流程



启动类：`org.apache.rocketmq.namesrv.NamesrvStartup`

步骤一

解析配置文件，填充NameServerConfig、NettyServerConfig属性值，并创建NamesrvController

代码：***NamesrvController#createNamesrvController***

```
//创建NamesrvConfig
final NamesrvConfig namesrvConfig = new NamesrvConfig();
//创建NettyServerConfig
final NettyServerConfig nettyServerConfig = new NettyServerConfig();
//设置启动端口号
nettyServerConfig.setListenPort(9876);
//解析启动-c参数
if (commandLine.hasOption('c')) {
    String file = commandLine.getOptionValue('c');
    if (file != null) {
        InputStream in = new BufferedInputStream(new
FileInputStream(file));
        properties = new Properties();
        properties.load(in);
        MixAll.properties2Object(properties, namesrvConfig);
        MixAll.properties2Object(properties, nettyServerConfig);

        namesrvConfig.setConfigStorePath(file);

        System.out.printf("load config properties file OK, %s%n", file);
        in.close();
    }
}
//解析启动-p参数
if (commandLine.hasOption('p')) {
    InternalLogger console =
InternalLoggerFactory.getLogger(LoggerName.NAMESRV_CONSOLE_NAME);
    MixAll.printObjectProperties(console, namesrvConfig);
    MixAll.printObjectProperties(console, nettyServerConfig);
    System.exit(0);
}
//将启动参数填充到namesrvConfig,nettyServerConfig
MixAll.properties2Object(ServerUtil.commandLine2Properties(commandLine),
namesrvConfig);

//创建NameServerController
final NamesrvController controller = new NamesrvController(namesrvConfig,
nettyServerConfig);
```

NamesrvConfig属性

```
private String rocketmqHome =
System.getProperty(MixAll.ROCKETMQ_HOME_PROPERTY,
System.getenv(MixAll.ROCKETMQ_HOME_ENV));
private String kvConfigPath = System.getProperty("user.home") +
File.separator + "namesrv" + File.separator + "kvConfig.json";
private String configStorePath = System.getProperty("user.home") +
File.separator + "namesrv" + File.separator + "namesrv.properties";
private String productEnvName = "center";
private boolean clusterTest = false;
private boolean orderMessageEnable = false;
```

rocketmqHome：rocketmq主目录

kvConfig：NameServer存储KV配置属性的持久化路径

configStorePath：nameServer默认配置文件路径

orderMessageEnable：是否支持顺序消息

NettyServerConfig属性

```
private int listenPort = 8888;
private int serverWorkerThreads = 8;
private int serverCallbackExecutorThreads = 0;
private int serverSelectorThreads = 3;
private int serverOnewaySemaphoreValue = 256;
private int serverAsyncSemaphoreValue = 64;
private int serverChannelMaxIdleTimeSeconds = 120;
private int serverSocketSndBufSize = NettySystemConfig.socketSndbufSize;
private int serverSocketRcvBufSize = NettySystemConfig.socketRcvbufSize;
private boolean serverPooledByteBufAllocatorEnable = true;
private boolean useEpollNativeSelector = false;
```

listenPort：NameServer监听端口，该值默认会被初始化为9876
serverWorkerThreads：Netty业务线程池线程个数
serverCallbackExecutorThreads：Netty public任务线程池线程个数，Netty网络设计，根据业务类型会创建不同的线程池，比如处理消息发送、消息消费、心跳检测等。如果该业务类型未注册线程池，则由public线程池执行。
serverSelectorThreads：IO线程池个数，主要是NameServer、Broker端解析请求、返回相应的线程个数，这类线程主要是处理网路请求的，解析请求包，然后转发到各个

业务线程池完成具体的操作，然后将结果返回给调用方;
serverOnewaySemaphoreValue：send oneway消息请求并发读（Broker端参数）;
serverAsyncSemaphoreValue：异步消息发送最大并发度;
serverChannelMaxIdleTimeSeconds ：网络连接最大的空闲时间，默认120s。
serverSocketSndBufSize：网络socket发送缓冲区大小。
serverSocketRcvBufSize： 网络接收端缓存区大小。
serverPooledByteBufAllocatorEnable：ByteBuffer是否开启缓存;
useEpollNativeSelector：是否启用Epoll IO模型。

步骤二

根据启动属性创建NamesrvController实例，并初始化该实例。NameServerController实例为NameServer核心控制器

代码：***NamesrvController#initialize***

```
public boolean initialize() {
    //加载KV配置
    this.kvConfigManager.load();
    //创建NettyServer网络处理对象
    this.remotingServer = new NettyRemotingServer(this.nettyServerConfig,
this.brokerHousekeepingService);
    //开启定时任务:每隔10s扫描一次Broker,移除不活跃的Broker
    this.remotingExecutor =

Executors.newFixedThreadPool(nettyServerConfig.getServerWorkerThreads(),
new ThreadFactoryImpl("RemotingExecutorThread_"));
    this.registerProcessor();
    this.scheduledExecutorService.scheduleAtFixedRate(new Runnable() {
        @Override
        public void run() {

NamesrvController.this.routeInfoManager.scanNotActiveBroker();
        }
    }, 5, 10, TimeUnit.SECONDS);
    //开启定时任务:每隔10min打印一次KV配置
    this.scheduledExecutorService.scheduleAtFixedRate(new Runnable() {

        @Override
        public void run() {

NamesrvController.this.kvConfigManager.printAllPeriodically();
        }
    }, 1, 10, TimeUnit.MINUTES);
    return true;
}
```

步骤三

在JVM进程关闭之前，先将线程池关闭，及时释放资源

代码：***NamesrvStartup#start***

```
//注册JVM钩子函数代码
Runtime.getRuntime().addShutdownHook(new ShutdownHookThread(log, new
Callable<Void>() {
    @Override
    public Void call() throws Exception {
        //释放资源
        controller.shutdown();
        return null;
    }
}));
```

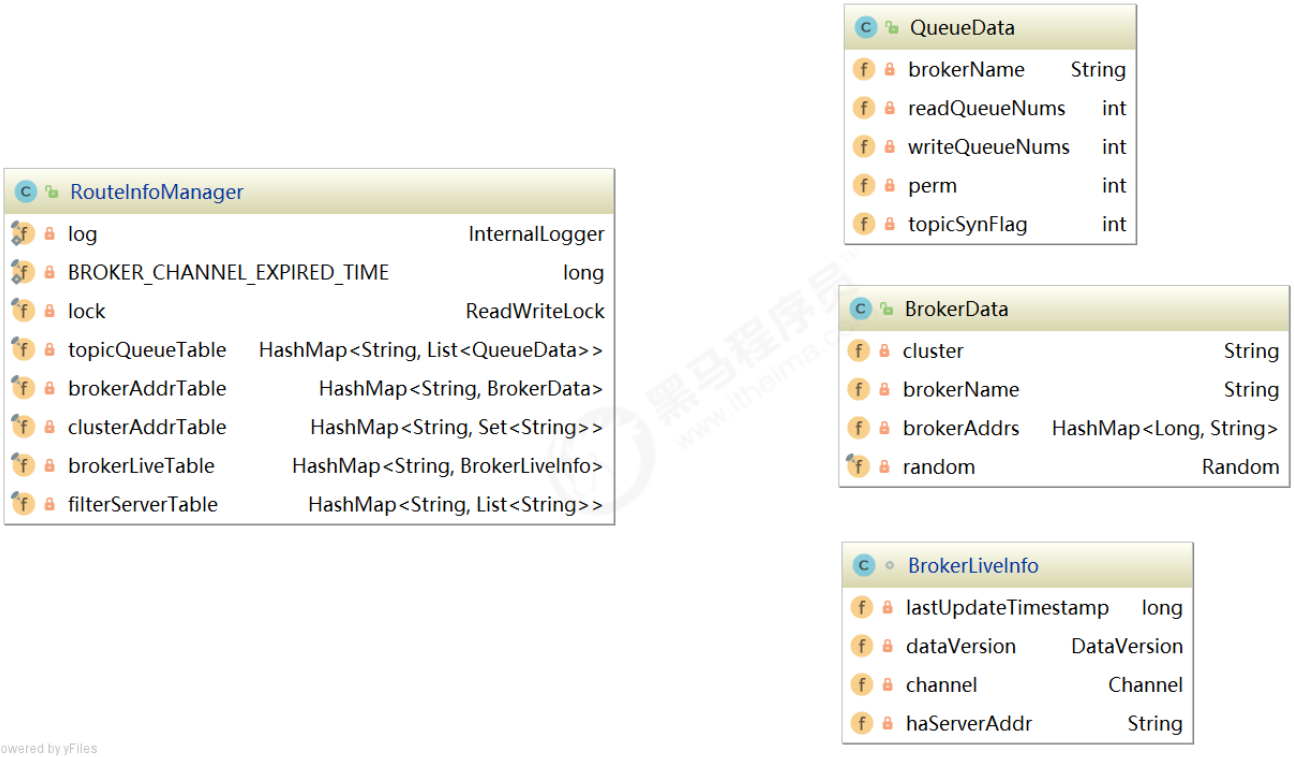
2.2.3 路由管理

NameServer的主要作用是为消息的生产者和消息消费者提供关于主题Topic的路由信息，那么NameServer需要存储路由的基础信息，还要管理Broker节点，包括路由注册、路由删除等。

2.2.3.1 路由元信息

代码：***RouteInfoManager***

```
private final HashMap<String/* topic */, List<QueueData>>
topicQueueTable;
private final HashMap<String/* brokerName */, BrokerData>
brokerAddrTable;
private final HashMap<String/* clusterName */, Set<String/* brokerName
*/>> clusterAddrTable;
private final HashMap<String/* brokerAddr */, BrokerLiveInfo>
brokerLiveTable;
private final HashMap<String/* brokerAddr */, List<String>/* Filter
Server */> filterServerTable;
```



topicQueueTable：Topic消息队列路由信息，消息发送时根据路由表进行负载均衡

brokerAddrTable：Broker基础信息，包括brokerName、所属集群名称、主备Broker地址

clusterAddrTable：Broker集群信息，存储集群中所有Broker名称

brokerLiveTable：Broker状态信息，NameServer每次收到心跳包是会替换该信息

filterServerTable：Broker上的FilterServer列表，用于类模式消息过滤。

> RocketMQ基于定于发布机制，一个Topic拥有多个消息队列，一个Broker为每一个主题创建4个读队列和4个写队列。多个Broker组成一个集群，集群由相同的多台Broker组成Master-Slave架构，brokerId为0代表Master，大于0为Slave。BrokerLiveInfo中的lastUpdateTimestamp存储上次收到Broker心跳包的时间。

```
topicQueueTable:{
    "topic1":[
        {
            "brokerName":"broker-a",
            "readQueueNums":4,
            "readQueueNums":4,
            "perm":6,   // 读写权限，具体含义请参考PermName
            "topicSynFlag":0 // topic同步标记，具体含义请参考TopicSysFlag
        },
        {
            "brokerName":"broker-b",
            "readQueueNums":4,
            "readQueueNums":4,
            "perm":6,   // 读写权限，具体含义请参考PermName
            "topicSynFlag":0 // topic同步标记，具体含义请参考TopicSysFlag
        }
    ],
    "topic other":[]
}
```

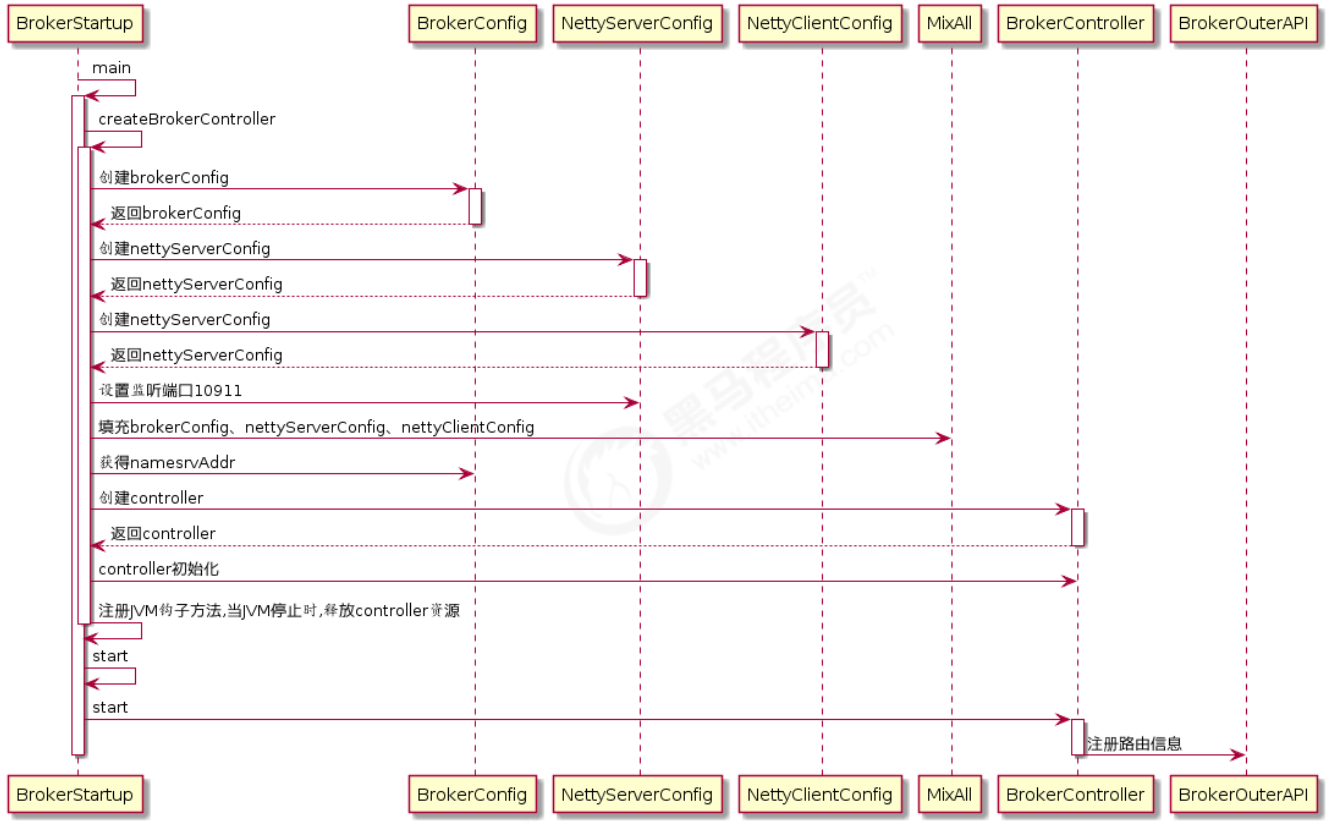
```
brokerAddrTable:{
   "broker-a": {
      "cluster":"c1",
      "brokerName":"broker-a",
      "brokerAddrs":{
          0:"192.168.56.1:10000",
          1:"192.168.56.2:10000",
      }
   },
   "broker-b": {
      "cluster":"c1",
      "brokerAddrs":{
          0:"192.168.56.3:10000",
          1:"192.168.56.4:10000"
      }
   }
}
```

```
brokerLiveTable : {
    "192.168.56.1:10000" : {
        "lastUpdateTimestamp":1518270318980,
        "dataVersion":versionObl,
        "channel":channelObj,
        "haServerAddr":"192.168.56.2:10000"
    },
    "192.168.56.2:10000" : {
        "lastUpdateTimestamp":1518270318980,
        "dataVersion":versionObl,
        "channel":channelObj,
        "haServerAddr":""
    },
    "192.168.56.3:10000" : {
        "lastUpdateTimestamp":1518270318980,
        "dataVersion":versionObl,
        "channel":channelObj,
        "haServerAddr":"192.168.56.4:10000"
    },
    "192.168.56.4:10000" : {
        "lastUpdateTimestamp":1518270318980,
        "dataVersion":versionObl,
        "channel":channelObj,
        "haServerAddr":""
    },
}
```

```
clusterAddrTable: {
    "c1" : [{"broker-a", "broker-b"}]
}
```

2.2.3.2 路由注册

1）发送心跳包



RocketMQ路由注册是通过Broker与NameServer的心跳功能实现的。Broker启动时向集群中所有的NameServer发送心跳信息，每隔30s向集群中所有NameServer发送心跳包，NameServer收到心跳包时会更新brokerLiveTable缓存中BrokerLiveInfo的lastUpdataTimeStamp信息，然后NameServer每隔10s扫描brokerLiveTable，如果连续120S没有收到心跳包，NameServer将移除Broker的路由信息同时关闭Socket连接。

代码：***BrokerController#start***

```
//注册Broker信息
this.registerBrokerAll(true, false, true);
//每隔30s上报Broker信息到NameServer
this.scheduledExecutorService.scheduleAtFixedRate(new Runnable() {

    @Override
    public void run() {
        try {
            BrokerController.this.registerBrokerAll(true, false,
brokerConfig.isForceRegister());
        } catch (Throwable e) {
            log.error("registerBrokerAll Exception", e);
        }
    }
}, 1000 * 10, Math.max(10000,
Math.min(brokerConfig.getRegisterNameServerPeriod(), 60000)),
                                                    TimeUnit.MILLISECONDS);
```

代码：***BrokerOuterAPI#registerBrokerAll***

```
//获得nameServer地址信息
List<String> nameServerAddressList =
this.remotingClient.getNameServerAddressList();
//遍历所有nameserver列表
if (nameServerAddressList != null && nameServerAddressList.size() > 0) {

    //封装请求头
    final RegisterBrokerRequestHeader requestHeader = new
RegisterBrokerRequestHeader();
    requestHeader.setBrokerAddr(brokerAddr);
    requestHeader.setBrokerId(brokerId);
    requestHeader.setBrokerName(brokerName);
    requestHeader.setClusterName(clusterName);
    requestHeader.setHaServerAddr(haServerAddr);
    requestHeader.setCompressed(compressed);
    //封装请求体
    RegisterBrokerBody requestBody = new RegisterBrokerBody();
    requestBody.setTopicConfigSerializeWrapper(topicConfigWrapper);
    requestBody.setFilterServerList(filterServerList);
    final byte[] body = requestBody.encode(compressed);
    final int bodyCrc32 = UtilAll.crc32(body);
    requestHeader.setBodyCrc32(bodyCrc32);
    final CountDownLatch countDownLatch = new
CountDownLatch(nameServerAddressList.size());
    for (final String namesrvAddr : nameServerAddressList) {
        brokerOuterExecutor.execute(new Runnable() {
            @Override
            public void run() {
                try {
                    //分别向NameServer注册
                    RegisterBrokerResult result =
registerBroker(namesrvAddr,oneway, timeoutMills,requestHeader,body);
                    if (result != null) {
                        registerBrokerResultList.add(result);
                    }

                    log.info("register broker[{}]to name server {} OK",
brokerId, namesrvAddr);
                } catch (Exception e) {

                    log.warn("registerBroker Exception, {}", namesrvAddr,
```

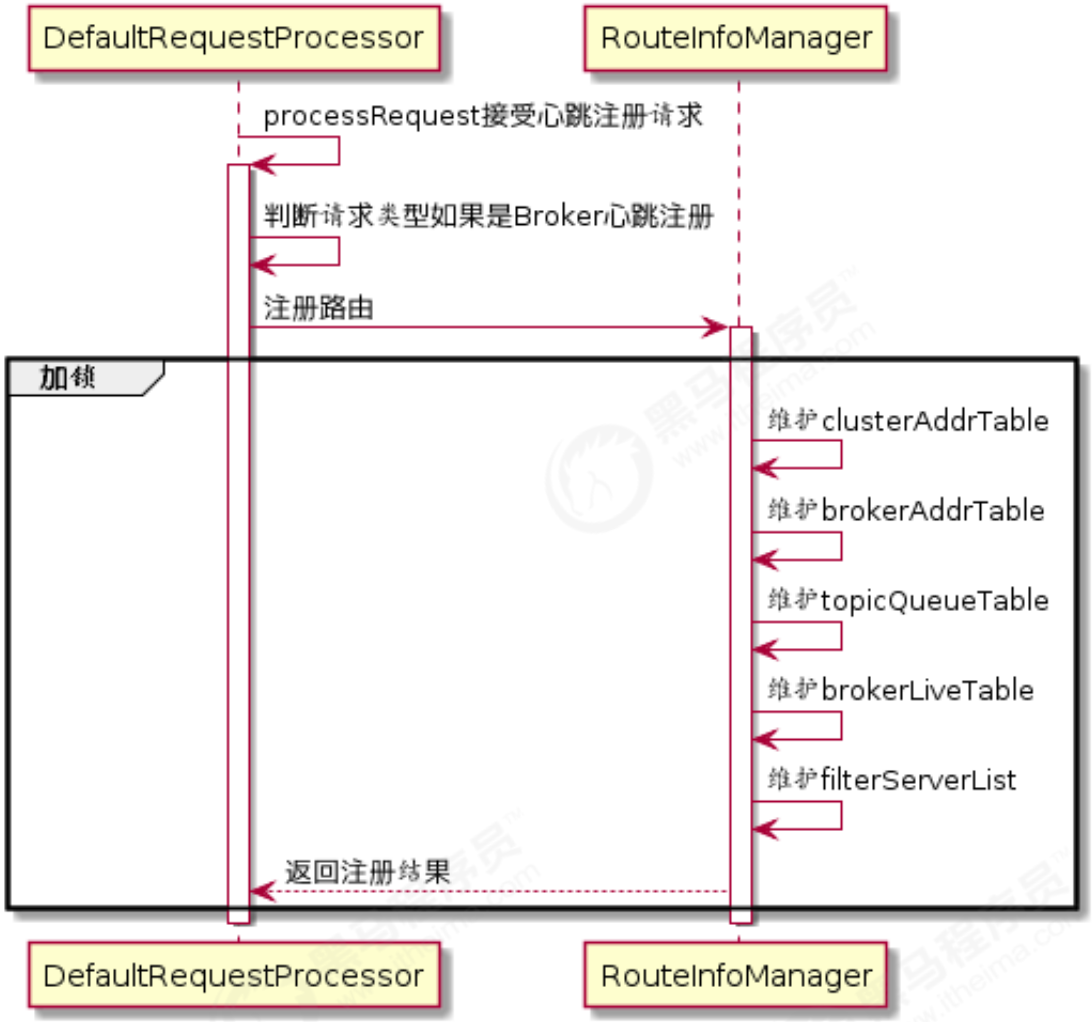
```
e);
                } finally {
                    countDownLatch.countDown();
                }
            }
        });
    }

    try {
        countDownLatch.await(timeoutMills, TimeUnit.MILLISECONDS);
    } catch (InterruptedException e) {
    }
}
```

代码：***BrokerOutAPI#registerBroker***

```
if (oneway) {
    try {
        this.remotingClient.invokeOneway(namesrvAddr, request,
timeoutMills);
    } catch (RemotingTooMuchRequestException e) {
        // Ignore
    }
    return null;
}
RemotingCommand response = this.remotingClient.invokeSync(namesrvAddr,
request, timeoutMills);
```

2）处理心跳包



`org.apache.rocketmq.namesrv.processor.DefaultRequestProcessor`网路处理类解析请求类型，如果请求类型是为***REGISTER_BROKER***，则将请求转发到`RouteInfoManager#regiesterBroker`

代码：***DefaultRequestProcessor#processRequest***

```
//判断是注册Broker信息
case RequestCode.REGISTER_BROKER:
    Version brokerVersion =
MQVersion.value2Version(request.getVersion());
    if (brokerVersion.ordinal() >= MQVersion.Version.V3_0_11.ordinal()) {
        return this.registerBrokerWithFilterServer(ctx, request);
    } else {
        //注册Broker信息
        return this.registerBroker(ctx, request);
    }
```

代码：***DefaultRequestProcessor#registerBroker***

```
RegisterBrokerResult result =
this.namesrvController.getRouteInfoManager().registerBroker(
    requestHeader.getClusterName(),
    requestHeader.getBrokerAddr(),
    requestHeader.getBrokerName(),
    requestHeader.getBrokerId(),
    requestHeader.getHaServerAddr(),
    topicConfigWrapper,
    null,
    ctx.channel()
);
```

代码：***RouteInfoManager#registerBroker***

维护路由信息

```
//加锁
this.lock.writeLock().lockInterruptibly();
//维护clusterAddrTable
Set<String> brokerNames = this.clusterAddrTable.get(clusterName);
if (null == brokerNames) {
    brokerNames = new HashSet<String>();
    this.clusterAddrTable.put(clusterName, brokerNames);
}
brokerNames.add(brokerName);
```

```
//维护brokerAddrTable
BrokerData brokerData = this.brokerAddrTable.get(brokerName);
//第一次注册,则创建brokerData
if (null == brokerData) {
    registerFirst = true;
    brokerData = new BrokerData(clusterName, brokerName, new
HashMap<Long, String>());
    this.brokerAddrTable.put(brokerName, brokerData);
}
//非第一次注册,更新Broker
Map<Long, String> brokerAddrsMap = brokerData.getBrokerAddrs();
Iterator<Entry<Long, String>> it = brokerAddrsMap.entrySet().iterator();
while (it.hasNext()) {
    Entry<Long, String> item = it.next();
    if (null != brokerAddr && brokerAddr.equals(item.getValue()) &&
brokerId != item.getKey()) {
        it.remove();
    }
}
String oldAddr = brokerData.getBrokerAddrs().put(brokerId, brokerAddr);
registerFirst = registerFirst || (null == oldAddr);
```

```
//维护topicQueueTable
if (null != topicConfigWrapper && MixAll.MASTER_ID == brokerId) {
    if (this.isBrokerTopicConfigChanged(brokerAddr,
topicConfigWrapper.getDataVersion()) ||
        registerFirst) {
        ConcurrentMap<String, TopicConfig> tcTable =
topicConfigWrapper.getTopicConfigTable();
        if (tcTable != null) {
            for (Map.Entry<String, TopicConfig> entry :
tcTable.entrySet()) {
                this.createAndUpdateQueueData(brokerName,
entry.getValue());
            }
        }
    }
}
```

代码：***RouteInfoManager#createAndUpdateQueueData***

```
private void createAndUpdateQueueData(final String brokerName, final
TopicConfig topicConfig) {
    //创建QueueData
    QueueData queueData = new QueueData();
    queueData.setBrokerName(brokerName);
    queueData.setWriteQueueNums(topicConfig.getWriteQueueNums());
    queueData.setReadQueueNums(topicConfig.getReadQueueNums());
    queueData.setPerm(topicConfig.getPerm());
    queueData.setTopicSynFlag(topicConfig.getTopicSysFlag());
    //获得topicQueueTable中队列集合
    List<QueueData> queueDataList =
this.topicQueueTable.get(topicConfig.getTopicName());
    //topicQueueTable为空,则直接添加queueData到队列集合
    if (null == queueDataList) {
        queueDataList = new LinkedList<QueueData>();
        queueDataList.add(queueData);
        this.topicQueueTable.put(topicConfig.getTopicName(),
queueDataList);
        log.info("new topic registered, {} {}",
topicConfig.getTopicName(), queueData);
    } else {
        //判断是否是新的队列
        boolean addNewOne = true;
        Iterator<QueueData> it = queueDataList.iterator();
        while (it.hasNext()) {
            QueueData qd = it.next();
            //如果brokerName相同,代表不是新的队列
            if (qd.getBrokerName().equals(brokerName)) {
                if (qd.equals(queueData)) {
                    addNewOne = false;
            } else {
                        log.info("topic changed, {} OLD: {} NEW: {}",
topicConfig.getTopicName(), qd,
                            queueData);
                        it.remove();
                    }
                }
            }
        //如果是新的队列,则添加队列到queueDataList

        if (addNewOne) {
```

```
                queueDataList.add(queueData);
            }
        }
    }
```

```
//维护brokerLiveTable
BrokerLiveInfo prevBrokerLiveInfo =
this.brokerLiveTable.put(brokerAddr,new BrokerLiveInfo(
    System.currentTimeMillis(),
    topicConfigWrapper.getDataVersion(),
    channel,
    haServerAddr));
```

```
//维护filterServerList
if (filterServerList != null) {
    if (filterServerList.isEmpty()) {
        this.filterServerTable.remove(brokerAddr);
    } else {
        this.filterServerTable.put(brokerAddr, filterServerList);
    }
}

if (MixAll.MASTER_ID != brokerId) {
    String masterAddr =
brokerData.getBrokerAddrs().get(MixAll.MASTER_ID);
    if (masterAddr != null) {
        BrokerLiveInfo brokerLiveInfo =
this.brokerLiveTable.get(masterAddr);
        if (brokerLiveInfo != null) {
            result.setHaServerAddr(brokerLiveInfo.getHaServerAddr());
            result.setMasterAddr(masterAddr);
        }
    }
}
```

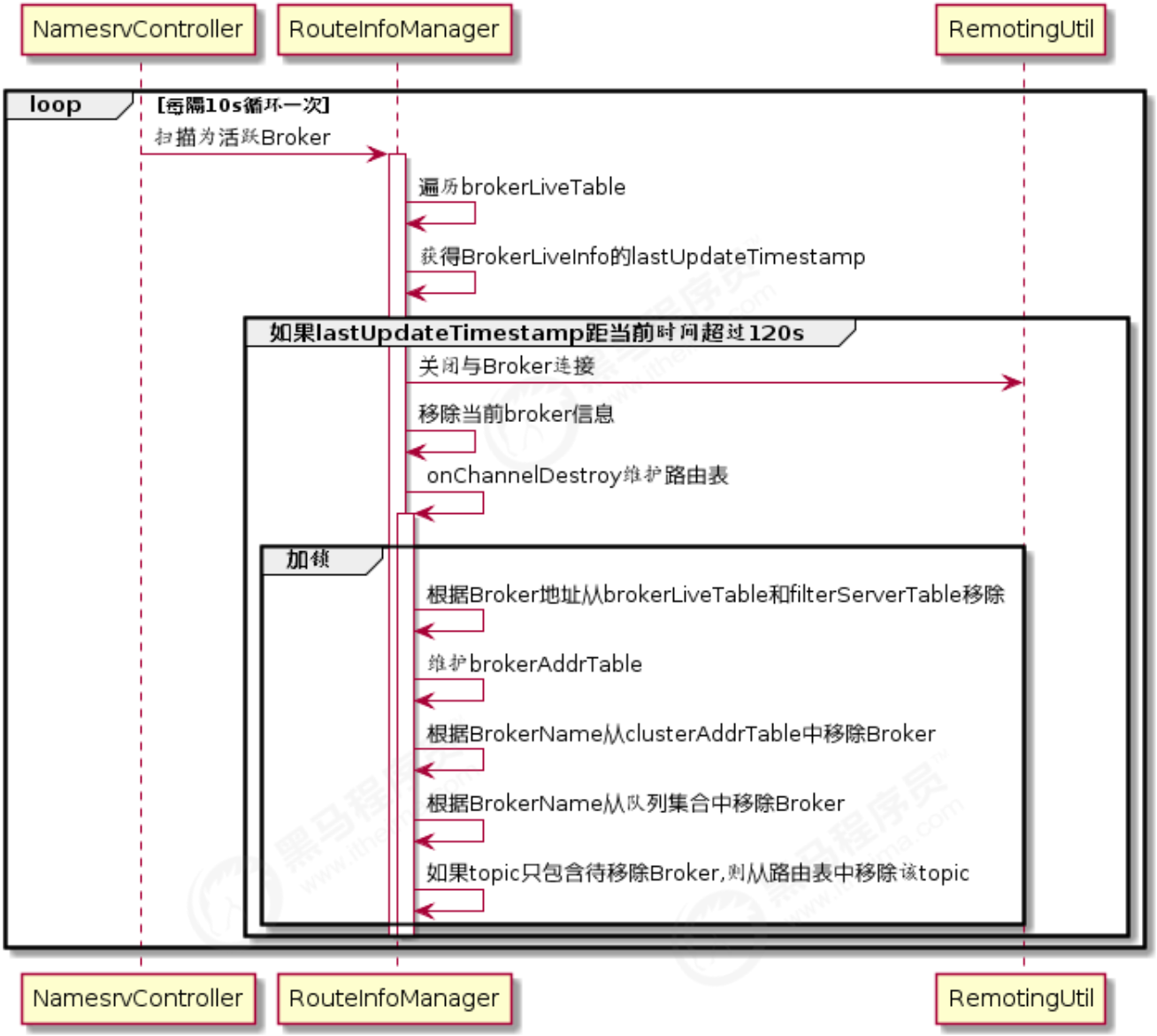
2.2.3.3 路由删除

`Broker`每隔30s向`NameServer`发送一个心跳包，心跳包包含`BrokerId`，`Broker`地址，`Broker`名称，`Broker`所属集群名称、`Broker`关联的`FilterServer`列表。但是如果`Broker`宕机，`NameServer`无法收到心跳包，此时`NameServer`如何来剔除这些失效的`Broker`呢？`NameServer`会每隔10s扫描`brokerLiveTable`状态表，如果`BrokerLive`的**lastUpdateTimestamp**的时间戳距当前时间超过120s，则认为`Broker`失效，移除该`Broker`，关闭与`Broker`连接，同时更新`topicQueueTable`、`brokerAddrTable`、`brokerLiveTable`、`filterServerTable`。

RocketMQ有两个触发点来删除路由信息：

- NameServer定期扫描brokerLiveTable检测上次心跳包与当前系统的时间差，如果时间超过120s，则需要移除broker。
- Broker在正常关闭的情况下，会执行unregisterBroker指令

这两种方式路由删除的方法都是一样的，就是从相关路由表中删除与该broker相关的信息。



代码：***NamesrvController#initialize***

```
//每隔10s扫描一次为活跃Broker
this.scheduledExecutorService.scheduleAtFixedRate(new Runnable() {

    @Override
    public void run() {
        NamesrvController.this.routeInfoManager.scanNotActiveBroker();
    }
}, 5, 10, TimeUnit.SECONDS);
```

代码：***RouteInfoManager#scanNotActiveBroker***

```
public void scanNotActiveBroker() {
    //获得brokerLiveTable
    Iterator<Entry<String, BrokerLiveInfo>> it =
this.brokerLiveTable.entrySet().iterator();
    //遍历brokerLiveTable
    while (it.hasNext()) {
        Entry<String, BrokerLiveInfo> next = it.next();
        long last = next.getValue().getLastUpdateTimestamp();
        //如果收到心跳包的时间距当时时间是否超过120s
        if ((last + BROKER_CHANNEL_EXPIRED_TIME) <
System.currentTimeMillis()) {
            //关闭连接
            RemotingUtil.closeChannel(next.getValue().getChannel());
            //移除broker
            it.remove();
            //维护路由表
            this.onChannelDestroy(next.getKey(),
next.getValue().getChannel());
        }
    }
}
```

代码：***RouteInfoManager#onChannelDestroy***

```
//申请写锁,根据brokerAddress从brokerLiveTable和filterServerTable移除
this.lock.writeLock().lockInterruptibly();
this.brokerLiveTable.remove(brokerAddrFound);
this.filterServerTable.remove(brokerAddrFound);
```

```
//维护brokerAddrTable
String brokerNameFound = null;
boolean removeBrokerName = false;
Iterator<Entry<String, BrokerData>> itBrokerAddrTable
=this.brokerAddrTable.entrySet().iterator();
//遍历brokerAddrTable
while (itBrokerAddrTable.hasNext() && (null == brokerNameFound)) {
    BrokerData brokerData = itBrokerAddrTable.next().getValue();
    //遍历broker地址
    Iterator<Entry<Long, String>> it =
brokerData.getBrokerAddrs().entrySet().iterator();
    while (it.hasNext()) {
        Entry<Long, String> entry = it.next();
        Long brokerId = entry.getKey();
        String brokerAddr = entry.getValue();
        //根据broker地址移除brokerAddr
        if (brokerAddr.equals(brokerAddrFound)) {
            brokerNameFound = brokerData.getBrokerName();
            it.remove();
            log.info("remove brokerAddr[{}, {}] from brokerAddrTable,
because channel destroyed",
                brokerId, brokerAddr);
            break;
        }
    }
    //如果当前主题只包含待移除的broker,则移除该topic
    if (brokerData.getBrokerAddrs().isEmpty()) {
        removeBrokerName = true;
        itBrokerAddrTable.remove();
        log.info("remove brokerName[{}] from brokerAddrTable, because
channel destroyed",
            brokerData.getBrokerName());
    }
}
```

```
//维护clusterAddrTable
if (brokerNameFound != null && removeBrokerName) {
    Iterator<Entry<String, Set<String>>> it =
this.clusterAddrTable.entrySet().iterator();
    //遍历clusterAddrTable
    while (it.hasNext()) {
        Entry<String, Set<String>> entry = it.next();
        //获得集群名称
        String clusterName = entry.getKey();
        //获得集群中brokerName集合
        Set<String> brokerNames = entry.getValue();
        //从brokerNames中移除brokerNameFound
        boolean removed = brokerNames.remove(brokerNameFound);
        if (removed) {
            log.info("remove brokerName[{}], clusterName[{}] from
clusterAddrTable, because channel destroyed",
                brokerNameFound, clusterName);

            if (brokerNames.isEmpty()) {
                log.info("remove the clusterName[{}] from
clusterAddrTable, because channel destroyed and no broker in this
cluster",
                    clusterName);
                //如果集群中不包含任何broker,则移除该集群
                it.remove();
            }

            break;
        }
    }
}
```

```
//维护topicQueueTable队列
if (removeBrokerName) {
    //遍历topicQueueTable
    Iterator<Entry<String, List<QueueData>>> itTopicQueueTable =
        this.topicQueueTable.entrySet().iterator();
    while (itTopicQueueTable.hasNext()) {
        Entry<String, List<QueueData>> entry = itTopicQueueTable.next();
        //主题名称
        String topic = entry.getKey();
        //队列集合
        List<QueueData> queueDataList = entry.getValue();
        //遍历该主题队列
        Iterator<QueueData> itQueueData = queueDataList.iterator();
        while (itQueueData.hasNext()) {
            //从队列中移除为活跃broker信息
            QueueData queueData = itQueueData.next();
            if (queueData.getBrokerName().equals(brokerNameFound)) {
                itQueueData.remove();
                log.info("remove topic[{} {}], from topicQueueTable,
because channel destroyed",
                    topic, queueData);
            }
        }
        //如果该topic的队列为空,则移除该topic
        if (queueDataList.isEmpty()) {
            itTopicQueueTable.remove();
            log.info("remove topic[{}] all queue, from topicQueueTable,
because channel destroyed",
                topic);
        }
    }
}
```

```
//释放写锁
finally {
    this.lock.writeLock().unlock();
}
```

2.2.3.4 路由发现

RocketMQ路由发现是非实时的，当Topic路由出现变化后，NameServer不会主动推送给客户端，而是由客户端定时拉取主题最新的路由。

代码：***DefaultRequestProcessor#getRouteInfoByTopic***

```
public RemotingCommand getRouteInfoByTopic(ChannelHandlerContext
```