

**NameServer注册中心职能**

NameServer是一个非常简单的路由注册中心，其角色类似Dubbo中的zookeeper，支持Broker的动态注册与发现。

* Broker管理，NameServer接受Broker集群的注册信息并且保存下来作为路由信息的基本数据。然后提供心跳检测机制，检查Broker是否还存活；
* 路由信息管理，每个NameServer将保存关于Broker集群的整个路由信息和用于客户端查询的队列信息。然后Producer和Conumser通过NameServer就可以知道整个Broker集群的路由信息，从而进行消息的投递和消费

**Broker集群:**

Broker用于接收生产者发送消息，或者消费者消费消息的请求。一个Broker集群由多组Master/Slave组成，Master可写可读，Slave只可以读，Master将写入的数据同步给Slave。

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。

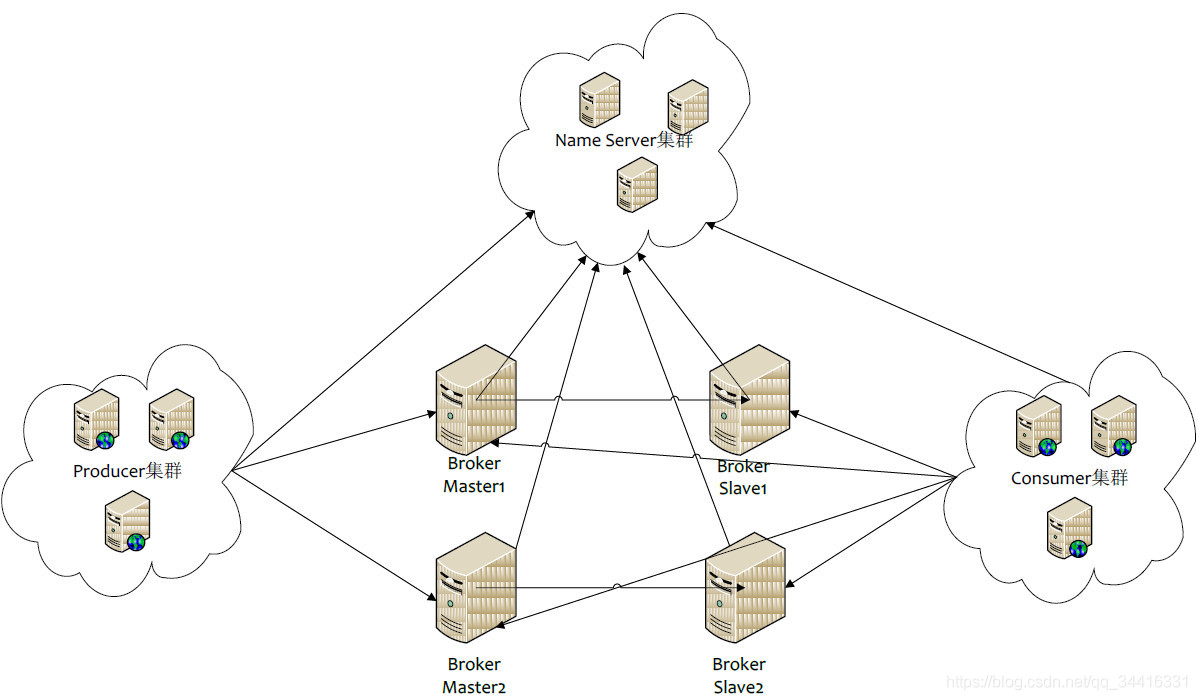
**Producer集群:**

消息的生产者，通过NameServer集群获得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。Producer只会将消息发送到Master节点上，因此只需要与Master节点建立连接。

**Consumer集群:**

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。

## RocketMQ高可用



### NameServer 高可用

由于 NameServer 节点是无状态的，且各个节点直接的数据是一致的，故存在多个 NameServer 节点的情况下，部分 NameServer 不可用也可以保证 MQ 服务正常运行

### BrokerServer 高可用

RocketMQ是通过 Master 和 Slave 的配合达到 BrokerServer 模块的高可用性的

一个 Master 可以配置多个 Slave，同时也支持配置多个 Master-Slave 组。

**当其中一个 Master 出现问题时：**

* 由于Slave只负责读，当 Master 不可用，它对应的 Slave 仍能保证消息被正常消费
* 由于配置多组 Master-Slave 组，其他的 Master-Slave 组也会保证消息的正常发送和消费

老版本的RocketMQ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。

新版本的RocketMQ，支持Slave自动转成Master。

### consumer高可用

Consumer 的高可用是依赖于 Master-Slave 配置的，由于 Master 能够支持读写消息，Slave 支持读消息，当 Master 不可用或繁忙时， Consumer 会被自动切换到从 Slave 读取(自动切换，无需配置)。

故当 Master 的机器故障后，消息仍可从 Slave 中被消费

### producer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。

## 可用性与可靠性

**3 可用性**

由于消息分布在各个broker上，一旦某个broker宕机，则该broker上的消息读写都会受到影响。所以rocketmq提供了master/slave的结构，salve定时从master同步数据，如果master宕机，则slave提供消费服务，但是不能写入消息，此过程对应用透明，由rocketmq内部解决。

这里有两个关键点：

* 一旦某个broker master宕机，生产者和消费者多久才能发现？受限于rocketmq的网络连接机制，默认情况下，最多需要30秒，但这个时间可由应用设定参数来缩短时间。这个时间段内，发往该broker的消息都是失败的，而且该broker的消息无法消费，因为此时消费者不知道该broker已经挂掉。
* 消费者得到master宕机通知后，转向slave消费，但是slave不能保证master的消息100%都同步过来了，因此会有少量的消息丢失。但是消息最终不会丢的，一旦master恢复，未同步过去的消息会被消费掉。

**4 可靠性**

* 所有发往broker的消息，有同步刷盘和异步刷盘机制，总的来说，可靠性非常高
* 同步刷盘时，消息写入物理文件才会返回成功，因此非常可靠
* 异步刷盘时，只有机器宕机，才会产生消息丢失，broker挂掉可能会发生，但是机器宕机崩溃是很少发生的，除非突然断电

## Broker消息的零丢失方案

### 同步刷盘、异步刷盘

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：

* 异步刷盘方式：

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，统一触发写磁盘操作，快速写入  
优点：性能高  
缺点：Master宕机，磁盘损坏的情况下，会丢失少量的消息, 导致MQ的消息状态和生产者/消费者的消息状态不一致

* 同步刷盘方式：

在返回应用写成功状态前，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，给应用返回消息写成功的状态。

优点：可以保持MQ的消息状态和生产者/消费者的消息状态一致  
缺点：性能比异步的低  
同步刷盘还是异步刷盘，是通过Broker配置文件里的flushDiskType参数设置的，这个参数被设置成SYNC\_FLUSH, ASYNC\_FLUSH中的一个。

### 同步复制、异步复制

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。

* 同步复制方式：

等Master和Slave均写成功后才反馈给客户端写成功状态  
优点：如果Master出故障，Slave上有全部的备份数据，容易恢复，消费者仍可以从Slave消费, 消息不丢失  
缺点：增大数据写入延迟，降低系统吞吐量，性能比异步复制模式略低，大约低10%左右，发送单个Master的响应时间会略高

* 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  
优点：系统拥有较低的延迟和较高的吞吐量. Master宕机之后，消费者仍可以从Slave消费，此过程对应用透明，不需要人工干预，性能同多个Master模式几乎一样  
缺点：如果Master出了故障，有些数据因为没有被写入Slave，而丢失少量消息。

若一个 Broker 组有一个 Master 和 Slave，消息需要从 Master 复制到 Slave 上，有同步复制和异步复制两种方式

|  | **同步复制** | **异步复制** |
| --- | --- | --- |
| **概念** | 即等 Master 和 Slave 均写成功后才反馈给客户端写成功状态 | 只要 Master 写成功，就反馈客户端写成功状态 |
| **可靠性** | 可靠性高，若 Master 出现故障，Slave 上有全部的备份数据，容易恢复 | 若 Master 出现故障，可能存在一些数据还没来得及写入 Slave，可能会丢失 |
| **效率** | 由于是同步复制，会增加数据写入延迟，降低系统吞吐量 | 由于只要写入 Master 即可，故数据写入延迟较低，吞吐量较高 |

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。

三个值的说明：

* sync\_master是同步方式，Master角色Broker中的消息要立刻同步过去。
* async\_master是异步方式，Master角色Broker中的消息通过异步处理的方式同步到Slave角色的机器上。
* SLAVE 表明当前是从节点，无需配置 brokerRole

### 消息零丢失方案

消息零丢失是一把双刃剑，要想用好，还是要视具体的业务场景，在性能和消息零丢失上做平衡。

实际应用中的推荐把Master和Slave设置成ASYNC\_FLUSH的异步刷盘方式，主从之间配置成SYNC\_MASTER的同步复制方式，这样即使有一台机器出故障，仍然可以保证数据不丢。

* 刷盘方式

Master和Slave都设置成ASYNC\_FLUSH的异步刷盘

* 复制方式

Master配置成SYNC\_MASTER 同步复制

异步刷盘能够避免频繁触发磁盘写操作，除非服务器宕机，否则不会造成消息丢失。

主从同步复制能够保证消息不丢失，即使 Master 节点异常，也能保证 Slave 节点存储所有消息并被正常消费掉。

## 消费幂等

**消费端实现消息幂等性**

RocketMQ 只能够保证消息丢失，但不能保证消息不重复投递，且由于高可用和高性能的考虑，应该在消费端实现消息幂等性。

那么 RocketMQ 是怎样解决消息重复的问题呢？还是“恰好”不解决。

造成消息重复的根本原因是：网络不可达。只要通过网络交换数据，就无法避免这个问题。所以解决这个问题的办法就是绕过这个问题。那么问题就变成了：如果消费端收到两条一样的消息，应该怎样处理？

* 消费端处理消息的业务逻辑保持幂等性
* 保证每条消息都有唯一编号且保证消息处理成功与去重表的日志同时出现
* 第1条很好理解，只要保持幂等性，不管来多少条重复消息，最后处理的结果都一样。
* 第2条原理就是利用一张日志表来记录已经处理成功的消息的ID，如果新到的消息ID已经在日志表中，那么就不再处理这条消息。

第1条解决方案，很明显应该在消费端实现，不属于消息系统要实现的功能。第2条可以消息系统实现，也可以业务端实现。正常情况下出现重复消息的概率其实很小，如果由消息系统来实现的话，肯定会对消息系统的吞吐量和高可用有影响，所以最好还是由业务端自己处理消息重复的问题，这也是 RocketMQ 不解决消息重复的问题的原因。

**RocketMQ 不保证消息不重复，如果你的业务需要保证严格的不重复消息，需要你自己在业务端去重。**

在消费端通过业务逻辑实现幂等性操作，最常用的方式就是唯一ID的形式，若已经消费过的消息就不进行处理。例如在秒杀系统中使用订单ID作为关键ID，分布式系统中常用雪花算法生成ID。