

23 | RocketMQ客户端如何在集群中找到正确的节点？

2019-09-17 李玥

消息队列高手课

[进入课程 >](#)



讲述：李玥

时长 13:53 大小 19.09M



你好，我是李玥。

我们在《[21 | RocketMQ Producer 源码分析：消息生产的实现过程](#)》这节课中，讲解 RocketMQ 的生产者启动流程时提到过，生产者只要配置一个接入地址，就可以访问整个集群，并不需要客户端配置每个 Broker 的地址。RocketMQ 会自动根据要访问的主题名称和队列序号，找到对应的 Broker 地址。如果 Broker 发生宕机，客户端还会自动切换到新的 Broker 节点上，这些对于用户代码来说都是透明的。

这些功能都是由 NameServer 协调 Broker 和客户端共同实现的，其中 NameServer 的作用是最关键的。

展开来讲，不仅仅是 RocketMQ，任何一个弹性分布式集群，都需要一个类似于 NameServer 服务，来帮助访问集群的客户端寻找集群中的节点，这个服务一般称为 NamingService。比如，像 Dubbo 这种 RPC 框架，它的注册中心就承担了 NamingService 的职责。在 Flink 中，则是 JobManager 承担了 NamingService 的职责。

也就是说，这种使用 NamingService 服务来协调集群的设计，在分布式集群的架构设计中，是一种非常通用的方法。你在学习这节课之后，不仅要掌握 RocketMQ 的 NameServer 是如何实现的，还要能总结出通用的 NamingService 的设计思想，并能应用于其他分布式系统的设计中。

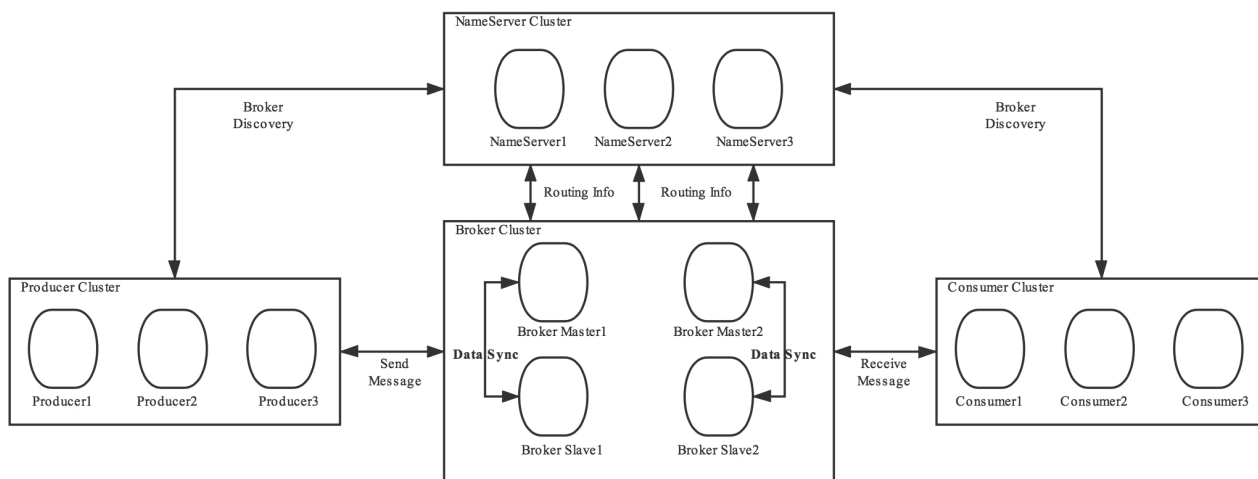
这节课，我们一起来分析一下 NameServer 的源代码，看一下 NameServer 是如何协调集群中众多的 Broker 和客户端的。

NameServer 是如何提供服务的？

在 RocketMQ 中，NameServer 是一个独立的进程，为 Broker、生产者和消费者提供服务。NameServer 最主要的功能就是，为客户端提供寻址服务，协助客户端找到主题对应的 Broker 地址。此外，NameServer 还负责监控每个 Broker 的存活状态。

NameServer 支持只部署一个节点，也支持部署多个节点组成一个集群，这样可以避免单点故障。在集群模式下，NameServer 各节点之间是不需要任何通信的，也不会通过任何方式互相感知，每个节点都可以独立提供全部服务。

我们一起通过这个图来看一下，在 RocketMQ 集群中，NameServer 是如何配合 Broker、生产者和消费者一起工作的。这个图来自[RocketMQ 的官方文档](#)。



每个 Broker 都需要和所有的 NameServer 节点进行通信。当 Broker 保存的 Topic 信息发生变化的时候，它会主动通知所有的 NameServer 更新路由信息，为了保证数据一致性，Broker 还会定时给所有的 NameServer 节点上报路由信息。这个上报路由信息的 RPC 请求，也同时起到 Broker 与 NameServer 之间的心跳作用，NameServer 依靠这个心跳来确定 Broker 的健康状态。

因为每个 NameServer 节点都可以独立提供完整的服务，所以，对于客户端来说，包括生产者和消费者，只需要选择任意一个 NameServer 节点来查询路由信息就可以了。客户端在生产或消费某个主题的消息之前，会先从 NameServer 上查询这个主题的路由信息，然后根据路由信息获取到当前主题和队列对应的 Broker 物理地址，再连接到 Broker 节点上进行生产或消费。

如果 NameServer 检测到与 Broker 的连接中断了，NameServer 会认为这个 Broker 不再能提供服务。NameServer 会立即把这个 Broker 从路由信息中移除掉，避免客户端连接到一个不可用的 Broker 上去。而客户端在与 Broker 通信失败之后，会重新去 NameServer 上拉取路由信息，然后连接到其他 Broker 上继续生产或消费消息，这样就实现了自动切换失效 Broker 的功能。

此外，NameServer 还提供一个类似 Redis 的 KV 读写服务，这个不是主要的流程，我们不展开讲。

接下来我带你一起分析 NameServer 的源代码，看一下这些服务都是如何实现的。

NameServer 的总体结构

由于 NameServer 的结构非常简单，排除 KV 读写相关的类之后，一共只有 6 个类，这里面直接给出这 6 个类的说明：

NamesrvStartup：程序入口。

NamesrvController：NameServer 的总控制器，负责所有服务的生命周期管理。


RouteInfoManager：NameServer 最核心的实现类，负责保存和管理集群路由信息。

BrokerHousekeepingService：监控 Broker 连接状态的代理类。

DefaultRequestProcessor：负责处理客户端和 Broker 发送过来的 RPC 请求的处理器。

ClusterTestRequestProcessor：用于测试的请求处理器。

RouteInfoManager 这个类中保存了所有的路由信息，这些路由信息都是保存在内存中，并且没有持久化的。在代码中，这些路由信息保存在 RouteInfoManager 的几个成员变量中：

 复制代码

```
1 public class BrokerData implements Comparable<BrokerData> {
2     // ...
3     private final HashMap<String/* topic */, List<QueueData>> topicQueueTable;
4     private final HashMap<String/* brokerName */, BrokerData> brokerAddrTable;
5     private final HashMap<String/* clusterName */, Set<String/* brokerName */>> clusterAddrTable;
6     private final HashMap<String/* brokerAddr */, BrokerLiveInfo> brokerLiveTable;
7     private final HashMap<String/* brokerAddr */, List<String>/* Filter Server */> filterTable;
8     // ...
9 }
```

以上代码中的这 5 个 Map 对象，保存了集群所有的 Broker 和主题的路由信息。

topicQueueTable 保存的是主题和队列信息，其中每个队列信息对应的类 QueueData 中，还保存了 brokerName。需要注意的是，这个 brokerName 并不真正是某个 Broker 的物理地址，它对应的一组 Broker 节点，包括一个主节点和若干个从节点。

brokerAddrTable 中保存了集群中每个 brokerName 对应 Broker 信息，每个 Broker 信息用一个 BrokerData 对象表示：

```
1 public class BrokerData implements Comparable<BrokerData> {
2     private String cluster;
3     private String brokerName;
4     private HashMap<Long/* brokerId */, String/* broker address */> brokerAddrs;
5     // ...
6 }
```

BrokerData 中保存了集群名称 cluster, brokerName 和一个保存 Broker 物理地址的 Map: brokerAddrs, 它的 Key 是 BrokerID, Value 就是这个 BrokerID 对应的 Broker 的物理地址。

下面这三个 map 相对没那么重要, 简单说明如下:

brokerLiveTable 中, 保存了每个 Broker 当前的动态信息, 包括心跳更新时间, 路由数据版本等等。

clusterAddrTable 中, 保存的是集群名称与 BrokerName 的对应关系。

filterServerTable 中, 保存了每个 Broker 对应的消息过滤服务的地址, 用于服务端消息过滤。

可以看到, 在 NameServer 的 RouteInfoManager 中, 主要的路由信息就是由 topicQueueTable 和 brokerAddrTable 这两个 Map 来保存的。

在了解了总体结构和数据结构之后, 我们再来看一下实现的流程。

NameServer 如何处理 Broker 注册的路由信息?

首先来看一下, NameServer 是如何处理 Broker 注册的路由信息的。

NameServer 处理 Broker 和客户端所有 RPC 请求的入口方法

是: “DefaultRequestProcessor#processRequest”, 其中处理 Broker 注册请求的代码如下:


```
1 public class DefaultRequestProcessor implements NettyRequestProcessor {
2     // ...
```

```

3      @Override
4      public RemotingCommand processRequest(ChannelHandlerContext ctx,
5          RemotingCommand request) throws RemotingCommandException {
6          // ...
7          switch (request.getCode()) {
8              // ...
9              case RequestCode.REGISTER_BROKER:
10                 Version brokerVersion = MQVersion.value2Version(request.getVersion());
11                 if (brokerVersion.ordinal() >= MQVersion.Version.V3_0_11.ordinal()) {
12                     return this.registerBrokerWithFilterServer(ctx, request);
13                 } else {
14                     return this.registerBroker(ctx, request);
15                 }
16                 // ...
17                 default:
18                     break;
19             }
20             return null;
21         }
22         // ...
23     }

```

这是一个非常典型的处理 Request 的路由分发器，根据 request.getCode() 来分发请求到对应的处理器中。Broker 发给 NameServer 注册请求的 Code 为 REGISTER_BROKER，在代码中根据 Broker 的版本号不同，分别有两个不同的处理实现方法：“registerBrokerWithFilterServer” 和 “registerBroker”。这两个方法实现的流程是差不多的，实际上都是调用了 “RouteInfoManager#registerBroker” 方法，我们直接看这个方法的代码：

 复制代码

```

1 public RegisterBrokerResult registerBroker(
2     final String clusterName,
3     final String brokerAddr,
4     final String brokerName,
5     final long brokerId,
6     final String haServerAddr,
7     final TopicConfigSerializeWrapper topicConfigWrapper,
8     final List<String> filterServerList,
9     final Channel channel) {
10     RegisterBrokerResult result = new RegisterBrokerResult();
11     try {
12         try {
13             // 加写锁，防止并发修改数据
14             this.lock.writeLock().lockInterruptibly();
15

```



```

16 // 更新 clusterAddrTable
17 Set<String> brokerNames = this.clusterAddrTable.get(clusterName);
18 if (null == brokerNames) {
19     brokerNames = new HashSet<String>();
20     this.clusterAddrTable.put(clusterName, brokerNames);
21 }
22 brokerNames.add(brokerName);
23
24 // 更新 brokerAddrTable
25 boolean registerFirst = false;
26
27 BrokerData brokerData = this.brokerAddrTable.get(brokerName);
28 if (null == brokerData) {
29     registerFirst = true; // 标识需要先注册
30     brokerData = new BrokerData(clusterName, brokerName, new HashMap<Long,
31     this.brokerAddrTable.put(brokerName, brokerData);
32 }
33 Map<Long, String> brokerAddrsMap = brokerData.getBrokerAddrs();
34 // 更新 brokerAddrTable 中的 brokerData
35 Iterator<Entry<Long, String>> it = brokerAddrsMap.entrySet().iterator();
36 while (it.hasNext()) {
37     Entry<Long, String> item = it.next();
38     if (null != brokerAddr && brokerAddr.equals(item.getValue()) && brokerI
39         it.remove();
40     }
41 }
42
43 // 如果是新注册的 Master Broker, 或者 Broker 中的路由信息变了, 需要更新 topicQu
44 String oldAddr = brokerData.getBrokerAddrs().put(brokerId, brokerAddr);
45 registerFirst = registerFirst || (null == oldAddr);
46
47 if (null != topicConfigWrapper
48     && MixAll.MASTER_ID == brokerId) {
49     if (this.isBrokerTopicConfigChanged(brokerAddr, topicConfigWrapper.getT
50         || registerFirst) {
51         ConcurrentMap<String, TopicConfig> tcTable =
52             topicConfigWrapper.getTopicConfigTable();
53         if (tcTable != null) {
54             for (Map.Entry<String, TopicConfig> entry : tcTable.entrySet())
55                 this.createAndUpdateQueueData(brokerName, entry.getValue())
56             }
57         }
58     }
59 }
60
61 // 更新 brokerLiveTable
62 BrokerLiveInfo prevBrokerLiveInfo = this.brokerLiveTable.put(brokerAddr,
63     new BrokerLiveInfo(
64         System.currentTimeMillis(),
65         topicConfigWrapper.getDataVersion(),
66         channel,
67         haServerAddr));

```

```

68         if (null == prevBrokerLiveInfo) {
69             log.info("new broker registered, {} HAServer: {}", brokerAddr, haServer
70         }
71
72         // 更新 filterServerTable
73         if (filterServerList != null) {
74             if (filterServerList.isEmpty()) {
75                 this.filterServerTable.remove(brokerAddr);
76             } else {
77                 this.filterServerTable.put(brokerAddr, filterServerList);
78             }
79         }
80
81         // 如果是 Slave Broker, 需要在返回的信息中带上 master 的相关信息
82         if (MixAll.MASTER_ID != brokerId) {
83             String masterAddr = brokerData.getBrokerAddrs().get(MixAll.MASTER_ID);
84             if (masterAddr != null) {
85                 BrokerLiveInfo brokerLiveInfo = this.brokerLiveTable.get(masterAddr
86                 if (brokerLiveInfo != null) {
87                     result.setHaServerAddr(brokerLiveInfo.getHaServerAddr());
88                     result.setMasterAddr(masterAddr);
89                 }
90             }
91         }
92     } finally {
93         // 释放写锁
94         this.lock.writeLock().unlock();
95     }
96 } catch (Exception e) {
97     log.error("registerBroker Exception", e);
98 }
99
100 return result;
101 }


```

上面这段代码比较长，但总体结构很简单，就是根据 Broker 请求过来的路由信息，依次对比并更新 clusterAddrTable、brokerAddrTable、topicQueueTable、brokerLiveTable 和 filterServerTable 这 5 个保存集群信息和路由信息的 Map 对象中的数据。

另外，在 RouteInfoManager 中，这 5 个 Map 作为一个整体资源，使用了一个读写锁来做并发控制，避免并发更新和更新过程中读到不一致的数据问题。这个读写锁的使用方法，和我们在之前的课程《[17 | 如何正确使用锁保护共享数据，协调异步线程？](#)》中讲到的方法是一样的。

客户端如何寻找 Broker?

下面我们来看一下，NameServer 如何帮助客户端来找到对应的 Broker。对于客户端来说，无论是生产者还是消费者，通过主题来寻找 Broker 的流程是一样的，使用的也是同一份实现。客户端在启动后，会启动一个定时器，定期从 NameServer 上拉取相关主题的路由信息，然后缓存在本地内存中，在需要的时候使用。每个主题的路由信息用一个 TopicRouteData 对象来表示：


 复制代码

```
1 public class TopicRouteData extends RemotingSerializable {
2     // ...
3     private List<QueueData> queueDatas;
4     private List<BrokerData> brokerDatas;
5     // ...
6 }
```

其中，queueDatas 保存了主题中的所有队列信息，brokerDatas 中保存了主题相关的所有 Broker 信息。客户端选定了队列后，可以在对应的 QueueData 中找到对应的 BrokerName，然后用这个 BrokerName 找到对应的 BrokerData 对象，最终找到对应的 Master Broker 的物理地址。这部分代码在 org.apache.rocketmq.client.impl.factory.MQClientInstance 这个类中，你可以自行查看。

下面我们看一下在 NameServer 中，是如何实现根据主题来查询 TopicRouteData 的。

NameServer 处理客户端请求和处理 Broker 请求的流程是一样的，都是通过路由分发器将请求分发的对应的处理方法中，我们直接看具体的实现方法 RouteInfoManager#pickupTopicRouteData：

 复制代码

```
1 public TopicRouteData pickupTopicRouteData(final String topic) {
2
3     // 初始化返回数据 topicRouteData
4     TopicRouteData topicRouteData = new TopicRouteData();
5     boolean foundQueueData = false;
6     boolean foundBrokerData = false;
7     Set<String> brokerNameSet = new HashSet<String>();
8     List<BrokerData> brokerDataList = new LinkedList<BrokerData>();
```

```

9      topicRouteData.setBrokerDatas(brokerDataList);
10
11      HashMap<String, List<String>> filterServerMap = new HashMap<String, List<String>>()
12      topicRouteData.setFilterServerTable(filterServerMap);
13
14      try {
15          try {
16
17              // 加读锁
18              this.lock.readLock().lockInterruptibly();
19
20              // 先获取主题对应的队列信息
21              List<QueueData> queueDataList = this.topicQueueTable.get(topic);
22              if (queueDataList != null) {
23
24                  // 把队列信息返回值中
25                  topicRouteData.setQueueDatas(queueDataList);
26                  foundQueueData = true;
27
28                  // 遍历队列，找出相关的所有 BrokerName
29                  Iterator<QueueData> it = queueDataList.iterator();
30                  while (it.hasNext()) {
31                      QueueData qd = it.next();
32                      brokerNameSet.add(qd.getBrokerName());
33                  }
34
35                  // 遍历这些 BrokerName，找到对应的 BrokerData，并写入返回结果中
36                  for (String brokerName : brokerNameSet) {
37                      BrokerData brokerData = this.brokerAddrTable.get(brokerName);
38                      if (null != brokerData) {
39                          BrokerData brokerDataClone = new BrokerData(brokerData.getClusterName(),
40                              brokerData.getBrokerAddrs().clone());
41                          brokerDataList.add(brokerDataClone);
42                          foundBrokerData = true;
43                          for (final String brokerAddr : brokerDataClone.getBrokerAddrs()) {
44                              List<String> filterServerList = this.filterServerTable.get(brokerAddr);
45                              filterServerMap.put(brokerAddr, filterServerList);
46                          }
47                      }
48                  }
49              } finally {
50                  // 释放读锁
51                  this.lock.readLock().unlock();
52              }
53          } catch (Exception e) {
54              log.error("pickupTopicRouteData Exception", e);
55          }
56      }
57
58      log.debug("pickupTopicRouteData {} {}", topic, topicRouteData);
59
60      if (foundBrokerData && foundQueueData) {

```

```
61         return topicRouteData;
62     }
63
64     return null;
65 }
```

这个方法的实现流程是这样的：

1. 初始化返回的 topicRouteData 后，☐获取读锁。
2. 在 topicQueueTable 中获取主题对应的队列信息，并写入返回结果中。
3. 遍历队列，找出相关的所有 BrokerName。
4. 遍历这些 BrokerName，从 brokerAddrTable 中找到对应的 BrokerData，并写入返回结果中。
5. 释放读锁并返回结果。

小结

这节课我们一起分析了 RocketMQ NameServer 的源代码，NameServer 在集群中起到的一个核心作用就是，为客户端提供路由信息，帮助客户端找到对应的 Broker。

每个 NameServer 节点上都保存了集群所有 Broker 的路由信息，可以独立提供服务。Broker 会与所有 NameServer 节点建立长连接，定期上报 Broker 的路由信息。客户端会选择连接某一个 NameServer 节点，定期获取订阅主题的路由信息，用于 Broker 寻址。

NameServer 的所有核心功能都是在 RouteInfoManager 这个类中实现的，这类中使用了几个 Map 来在内存中保存集群中所有 Broker 的路由信息。

我们还一起分析了 RouteInfoManager 中的两个比较关键的方法：注册 Broker 路由信息的方法 registerBroker，以及查询 Broker 路由信息的方法 pickupTopicRouteData。

建议你仔细读一下这两个方法的代码，结合保存路由信息的几个 Map 的数据结构，体会一下 RocketMQ NameServer 这种简洁的设计。

把以上的这些 NameServer 的设计和实现方法抽象一下，我们就可以总结出通用的 NamingService 的设计思想。

NamingService 负责保存集群内所有节点的路由信息，NamingService 本身也是一个集群，由多个 NamingService 节点组成。这里我们所说的“路由信息”也是一种通用的抽象，含义是：“客户端需要访问的某个特定服务在哪个节点上”。

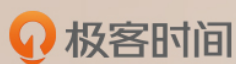
集群中的节点主动连接 NamingService 服务，注册自身的路由信息。给客户端提供路由寻址服务的方式可以有两种，一种是客户端直接连接 NamingService 服务查询路由信息，另一种是，客户端连接集群内任意节点查询路由信息，节点再从自身的缓存或者从 NamingService 上进行查询。

掌握了以上这些 NamingService 的设计方法，将会非常有助于你理解其他分布式系统的架构，当然，你也可以把这些方法应用到分布式系统的设计中去。

思考题

今天的思考题是这样的，在 RocketMQ 的 NameServer 集群中，各节点之间不需要互相通信，每个节点都可以独立的提供服务。课后请你想一想，这种独特的集群架构有什么优势，又有什么不足？欢迎在评论区留言写下你的想法。

感谢阅读，如果你觉得这篇文章对你有一些启发，也欢迎把它分享给你的朋友。



消息队列高手课

从源码角度全面解析 MQ 的设计与实现

李玥

京东零售技术架构部资深架构师



新版升级：点击「👤请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

上一篇 22 | Kafka和RocketMQ的消息复制实现的差异点在哪？

下一篇 24 | Kafka的协调服务ZooKeeper：实现分布式系统的“瑞士军刀”

精选留言 (9)

写留言



Better me

2019-09-19

NamingService集群有点像去中心化的结构设计，每个节点保存所有数据，很好的保证了节点的可用性，但每个节点之间不互相通信，很难确保节点间的数据一致性。想问下老师主题(和其中队列)在broker节点的分布情况是怎样的

展开

作者回复: 主题和队列是分散到所有Broker节点上的，每个Broker只保存自己负责的那部分主题和队列信息。并且实际上在集群中，元数据最终是以Broker上保存的信息为准的。



1



益军

2019-09-20

优点: nameserver本身设计为无状态，实现简单，
缺点: broker客户端通信成本复杂，适合在客户端环境完全可控的情况下设计。namesrv一致性无法保证，需要定时幂等性心跳保持最终一致性。



康师傅

2019-09-19

对于rocketmq和kafka而言，都有自己的“注册中心”，但对于rabbitmq而言，它的集群允许你连接到集群中的任何一台进行生产消费，即便队列master所在节点并不是你连接的这台，rabbitmq内部会帮你进行中转，但这会有一个很大的弊端，就是节点间会有较大的流量并且不可控，并且整体的性能会受影响

...

展开

作者回复: 理论上是可以的，但权衡工作量来说，最好还是换一个MQ吧。





leslie

2019-09-18

老师最近的课都是在啃代码且非常整体性：学习上是越来越辛苦了，总要翻阅和梳理知识才能明白题目可能的问题。

节点之间不互相通信其实减少了网络开销以及相互的等待确认的过程从而节约了时间,不会互相影响互相继承：换个角度来思考这个问题其实就像是我们用虚拟化一样，RocketMQ的这种NameServer就像是docker/Kubernetes，各自出了问题不会影响其它的docke...
展开 ▾



DFighting

2019-09-17

多个NameServer独立对外提供服务是一种用冗余的路由注册来换维持集群式的NameServer间数据一致性和高可用性的方式，集群部署不可避免需要在数据一致性和高可用性间平衡，这会给程序设计、编码和后溪维护带来很大的代价，因为路由信息本就不会太多，所以选择了前者。

不过我更偏向于后者，单节点独立提供服务肯定会出现某个节点请求当前的NameServer...
展开 ▾



有铭

2019-09-17

这整个就是一个微服务架构

展开 ▾



糖醋 ☹

2019-09-17

nnameserver各个节点独立不通信，是ap的思路。

各个节点总是可用，但是节点之间不通信，有可能由于网络原因，某个节点的路由信息可能会不一致。

客户端拉去所有节点的路由信息，可以弥补某个节点路由信息不一致的情况。

展开 ▾

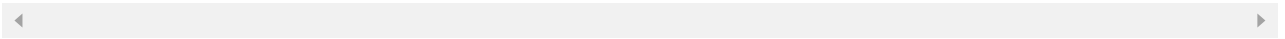


业余草

2019-09-17

线上环境突发消息延迟2个小时，该如何尽快解决？以及后期如何避免这类问题？说说你的思路和经验！

作者回复: 这个我在之前的课程中讲到过, 首先需要先看一下是消费慢还是生产慢, 如果是生产慢, 一般需要扩容Producer的节点数量, 如果是消费慢, 需要扩容队列数和Consumer数量。



Stalary

2019-09-17

老师, 我想问一下, 如果起了很多个NameServer, 都保持长连接的话是不是开销会较大呢, 为什么没有采用订阅发布的模式去更新broker呢, 是因为即时性吗

展开 ∨

作者回复: 这又是一个设计选择而已, NameServer只是负责存储一下元数据, 数据量不大, 处理请求的TPS也不高, 所以没必要启动很多个NameServer, 所以并不会存在你说的很多个NameServer的情况。

