23 | RocketMQ客户端如何在集群中找到正确的节点？

2019-09-17 李玥

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

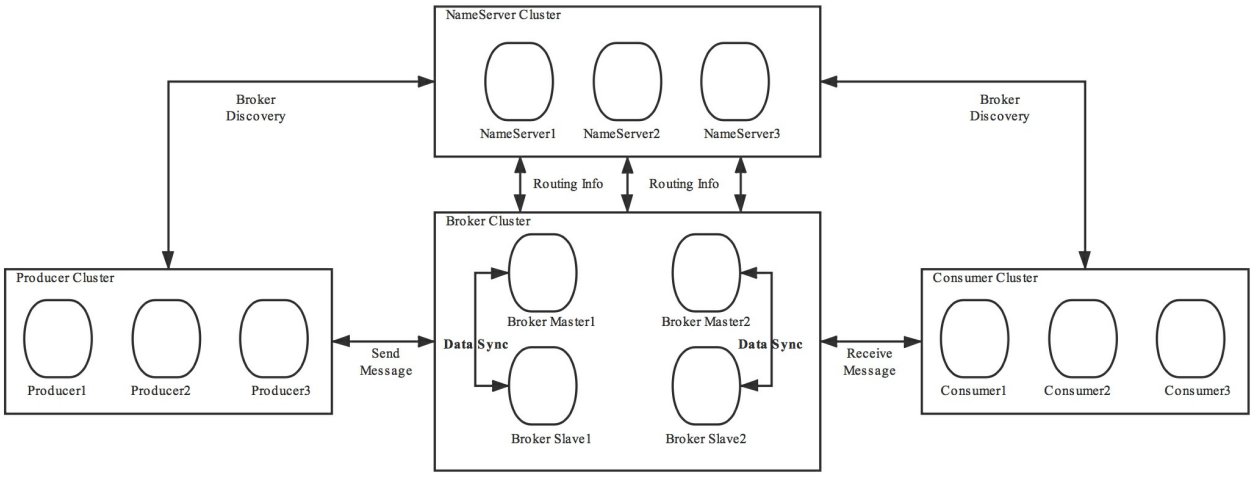
的Broker和客户端的。

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：

```
public class BrokerData implements Comparable<BrokerData> {
  // ...
  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;
  // ...
}
```

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：

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

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：

```
public class DefaultRequestProcessor implements NettyRequestProcessor {
    // ...
    @Override
    public RemotingCommand processRequest(ChannelHandlerContext ctx,
        RemotingCommand request) throws RemotingCommandException {
        // ...
        switch (request.getCode()) {
            // ...
            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 {
                    return this.registerBroker(ctx, request);
                }
            // ...
            default:
                break;
        }
        return null;
    }
    // ...
}
```

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：

```
public RegisterBrokerResult registerBroker(
    final String clusterName,
    final String brokerAddr,
    final String brokerName,
    final long brokerId,
    final String haServerAddr,
    final TopicConfigSerializeWrapper topicConfigWrapper
```

```
final TopicConfigSerializeWrapper topicConfigWrapper,
final List<String> filterServerList,
final Channel channel) {
RegisterBrokerResult result = new RegisterBrokerResult();
try {
    try {
        // 加写锁，防止并发修改数据
        this.lock.writeLock().lockInterruptibly();

        // 更新clusterAddrTable
        Set<String> brokerNames = this.clusterAddrTable.get(clusterName);
        if (null == brokerNames) {
            brokerNames = new HashSet<String>();
            this.clusterAddrTable.put(clusterName, brokerNames);
        }
        brokerNames.add(brokerName);

        // 更新brokerAddrTable
        boolean registerFirst = false;

        BrokerData brokerData = this.brokerAddrTable.get(brokerName);
        if (null == brokerData) {
            registerFirst = true; // 标识需要先注册
            brokerData = new BrokerData(clusterName, brokerName, new HashMap<Long, String>());
            this.brokerAddrTable.put(brokerName, brokerData);
        }
        Map<Long, String> brokerAddrsMap = brokerData.getBrokerAddrs();
        // 更新brokerAddrTable中的brokerData
        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();
            }
        }

        // 如果是新注册的Master Broker，或者Broker中的路由信息变了，需要更新topicQueueTable
        String oldAddr = brokerData.getBrokerAddrs().put(brokerId, brokerAddr);
```

```
registerFirst = registerFirst || (null == oldAddr);

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());
            }
        }
    }
}

// 更新brokerLiveTable
BrokerLiveInfo prevBrokerLiveInfo = this.brokerLiveTable.put(brokerAddr,
    new BrokerLiveInfo(
        System.currentTimeMillis(),
        topicConfigWrapper.getDataVersion(),
        channel,
        haServerAddr));
if (null == prevBrokerLiveInfo) {
    log.info("new broker registered, {} HAServer: {}", brokerAddr, haServerAddr);
}

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

// 如果是Slave Broker，需要在返回的信息中带上master的相关信息
```

```
            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);
                    }
                }
            }
        } finally {
            // 释放写锁
            this.lock.writeLock().unlock();
        }
    } catch (Exception e) {
        log.error("registerBroker Exception", e);
    }

    return result;
}
```

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：

```
public class TopicRouteData extends RemotingSerializable {
    // ...
    private List<QueueData> queueDatas;
    private List<BrokerData> brokerDatas;
    // ...
}
```

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：

```
public TopicRouteData pickupTopicRouteData(final String topic) {

    // 初始化返回数据topicRouteData
    TopicRouteData topicRouteData = new TopicRouteData();
    boolean foundQueueData = false;
    boolean foundBrokerData = false;
    Set<String> brokerNameSet = new HashSet<String>();
    List<BrokerData> brokerDataList = new LinkedList<BrokerData>();
    topicRouteData.setBrokerDatas(brokerDataList);

    HashMap<String, List<String>> filterServerMap = new HashMap<String, List<String>>();
    topicRouteData.setFilterServerTable(filterServerMap);

    try {
        try {

            // 加读锁
            this.lock.readLock().lockInterruptibly();
```

```
        //先获取主题对应的队列信息
        List<QueueData> queueDataList = this.topicQueueTable.get(topic);
        if (queueDataList != null) {

            // 把队列信息返回值中
            topicRouteData.setQueueDatas(queueDataList);
            foundQueueData = true;

            // 遍历队列，找出相关的所有BrokerName
            Iterator<QueueData> it = queueDataList.iterator();
            while (it.hasNext()) {
                QueueData qd = it.next();
                brokerNameSet.add(qd.getBrokerName());
            }

            // 遍历这些BrokerName，找到对应的BrokerData，并写入返回结果中
            for (String brokerName : brokerNameSet) {
                BrokerData brokerData = this.brokerAddrTable.get(brokerName);
                if (null != brokerData) {
                    BrokerData brokerDataClone = new BrokerData(brokerData.getCluster(), brokerData.getBrokerNa
                        .getBrokerAddrs().clone());
                    brokerDataList.add(brokerDataClone);
                    foundBrokerData = true;
                    for (final String brokerAddr : brokerDataClone.getBrokerAddrs().values()) {
                        List<String> filterServerList = this.filterServerTable.get(brokerAddr);
                        filterServerMap.put(brokerAddr, filterServerList);
                    }
                }
            }
        }
    } finally {
        // 释放读锁
        this.lock.readLock().unlock();
    }
} catch (Exception e) {
    log.error("pickupTopicRouteData Exception", e);
}
```

```
        log.debug("pickupTopicRouteData {} {}", topic, topicRouteData);

        if (foundBrokerData && foundQueueData) {
            return topicRouteData;
        }

        return null;
    }
```

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本身也是一个小集群，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。

精选留言

DFighting

0

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找不到对应的正常Broker的情况，因为NameServer不能保证保存了完整的Broker集群拓扑。路由发现算法虽然会导致一些时间的服务不可用，但在Broker集群体量很大的时候，肯定比独立NameServer好点。当然目前也可以考虑部署RocketMQ集群，让每个独立的NameServer服务部分区域的Broker的设计思路吧

2019-09-17

有铭 0

这整个就是一个微服务架构

2019-09-17

糖醋 0

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。

2019-09-17

业余草 0

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！

2019-09-17

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。

2019-09-17

Stalary 0

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

2019-09-17

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。

2019-09-17