

图文 106 Broker是如何发送定时心跳的，以及如何进行故障感知？

158 人次阅读 2020-02-27 10:49:06

详情 评论

Broker是如何发送定时心跳的，以及如何进行故障感知？



狸猫技术

进店逛

相关频道



从 0 开
间件实站
已更新1



继《从零开始带你成为JVM实战高手》后，阿里资深技术专家携新作再度出山，重磅推荐：

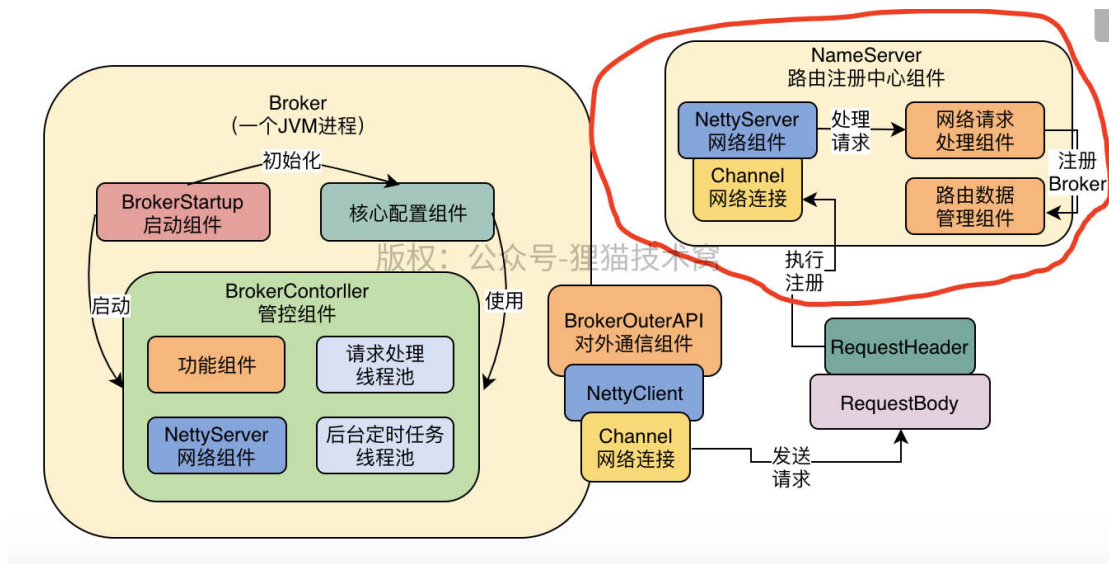
(点击下方蓝字试听)

[《从零开始带你成为MySQL实战优化高手》](#)

昨天我们已经讲解了NameServer处理Broker注册请求的源码流程，大家已经知道了，NameServer核心其实就是基于Netty服务器来接收Broker注册请求，然后交给DefaultRequestProcessor这个请求处理组件，来处理Broker注册请求。

而真正的Broker注册的逻辑是放在RouteInfoManager这个路由数据管理组件里来进行实现的，最终Broker路由数据都会存放在RouteInfoManager内部的一些Map数据结构组成的路由数据表中。

我们看下图，就是一个示意。



今天我们就来讲讲，Broker是如何定时发送心跳到NameServer，让NameServer感知到Broker一直都存活着，然后如果Broker一段时间没有发送心跳到NameServer，那么NameServer是如何感知到Broker已经挂掉了。

首先我们看一下Broker中的发送注册请求给NameServer的一个源码入口，其实就是在BrokerController.start()方法中，在BrokerController启动的时候，他其实并不是仅仅发送一次注册请求，而是启动了一个定时任务，会每隔一段时间就发送一次注册请求。

```

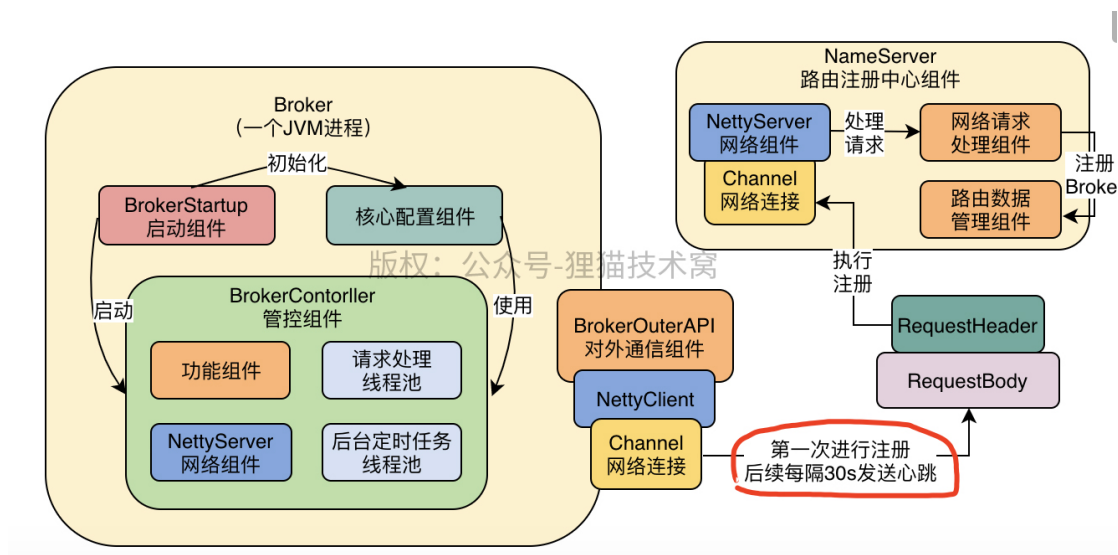
1 this.scheduledExecutorService.scheduleAtFixedRate(new Runnable() {
2
3     @Override
4     public void run() {
5         try {
6             BrokerController.this.registerBrokerAll(true, false, brokerConfig.isForceRegister());
7         } catch (Throwable e) {
8             log.error("registerBrokerAll Exception", e);
9         }
10    }
11 },
12 1000 * 10,
13 Math.max(10000, Math.min(brokerConfig.getRegisterNameServerPeriod(), 60000)), TimeUnit.MILLISECONDS);
14

```

上面这块代码，其实是启动了一个定时调度的任务，他默认是每隔30s就会执行一次Broker注册的过程，上面的registerNameServerPeriod是一个配置，他默认的值就是30s一次。

所以其实大家看到这里就会明白，默认情况下，第一次发送注册请求就是在进行注册，就是我们上一讲讲的内容，他会把Broker路由数据放入到NameServer的RouteInfoManager的路由数据表里去。

但是后续每隔30s他都会发送一次注册请求，这些后续定时发送的注册请求，其实本质上就是Broker发送心跳给NameServer了，我们看下图示意。



那么后续每隔30s, Broker就发送一次注册请求, 作为心跳来发送给NameServer的时候, NameServer对后续重复发送过来的注册请求 (也就是心跳), 是如何进行处理的呢?

说到这里，我今天来带大家看一下RouteInfoManager的注册方法逻辑。

上一次是给大家留了作业，想必很多人可能都已经看出一点感悟来了，今天就我们一起来分析一下，下面是RouteInfoManager的注册Broker的源码。

```

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             // 这里是加写锁，同一时间，只能是一个线程来执行
14             this.lock.writeLock().lockInterruptibly();
15
16             // 下面这里是根据clusterName获取了一个set集合
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             // 然后直接就把brokerName扔到了这个set集合里去
23             // 这就是在维护一个集群里有哪些broker存在的一个set数据结构
24             // 假如你后续每隔30s发送注册请求作为心跳，这里是没影响的
25             // 因为同样一个brokerName反复发送，这里set集合是自动去重的！
26             brokerNames.add(brokerName);
27
28             boolean registerFirst = false;
29

```

```

30 // 这里是根据brokerName获取到BrokerData
31 // 他用一个brokerAddrTable作为核心路由数据表
32 // 这里存放了所有的Broker的详细的路由数据
33 BrokerData brokerData = this.brokerAddrTable.get(brokerName);
34
35 // 如果第一次发送注册请求，这里就是null
36 // 那么就会封装一个BrokerData，放入到这个路由数据表里去
37 // 其实这个就是核心的Broker注册过程
38
39 // 如果后续每隔30s发送注册请求作为心跳，这里是没影响的
40 // 因为明显你重复发送注册请求的时候，这个BrokerData已经存在了
41 // 他不会重复进行处理的
42 if (null == brokerData) {
43     registerFirst = true;
44     brokerData = new BrokerData(clusterName, brokerName, new HashMap<Long, String>());
45     this.brokerAddrTable.put(brokerName, brokerData);
46 }
47
48 // 下面这里是对路由数据做一些处理，暂时不用管
49 Map<Long, String> brokerAddrsMap = brokerData.getBrokerAddrs();
50 //Switch slave to master: first remove <1, IP:PORT> in namesrv, then add <0, IP:PORT>
51 //The same IP:PORT must only have one record in brokerAddrTable
52 Iterator<Entry<Long, String>> it = brokerAddrsMap.entrySet().iterator();
53 while (it.hasNext()) {
54     Entry<Long, String> item = it.next();
55     if (null != brokerAddr && brokerAddr.equals(item.getValue()) && brokerId != item.getKey()) {
56         it.remove();
57     }
58 }
59
60 String oldAddr = brokerData.getBrokerAddrs().put(brokerId, brokerAddr);
61 registerFirst = registerFirst || (null == oldAddr);
62
63 if (null != topicConfigWrapper
64     && MixAll.MASTER_ID == brokerId) {
65     if (this.isBrokerTopicConfigChanged(brokerAddr, topicConfigWrapper.getDataVersion())
66         || registerFirst) {
67         ConcurrentMap<String, TopicConfig> tcTable =
68             topicConfigWrapper.getTopicConfigTable();
69         if (tcTable != null) {
70             for (Map.Entry<String, TopicConfig> entry : tcTable.entrySet()) {
71                 this.createAndUpdateQueueData(brokerName, entry.getValue());
72             }
73         }
74     }
75 }
76
77 // 其实比较核心的在这里，这就是每隔30s发送注册请求作为心跳的时候
78 // 他最核心的处理逻辑
79 // 说白了，他就是每隔30s都会封装一个新的BrokerLiveInfo放入Map
80 // 所以每隔30s，最新的BrokerLiveInfo都会覆盖之前一次的BrokerLiveInfo
81 // 这个BrokerLiveInfo里，就有一个当前时间戳，代表你最近一次心跳的时间
82 // 这就是Broker每隔30s发送注册请求作为心跳的处理逻辑
83 BrokerLiveInfo prevBrokerLiveInfo = this.brokerLiveTable.put(brokerAddr,
84     new BrokerLiveInfo(
85         System.currentTimeMillis(),
86         topicConfigWrapper.getDataVersion(),
87         channel,
88         haServerAddr));
89 if (null == prevBrokerLiveInfo) {
90     log.info("new broker registered, {} HAServer: {}", brokerAddr, haServerAddr);
91 }

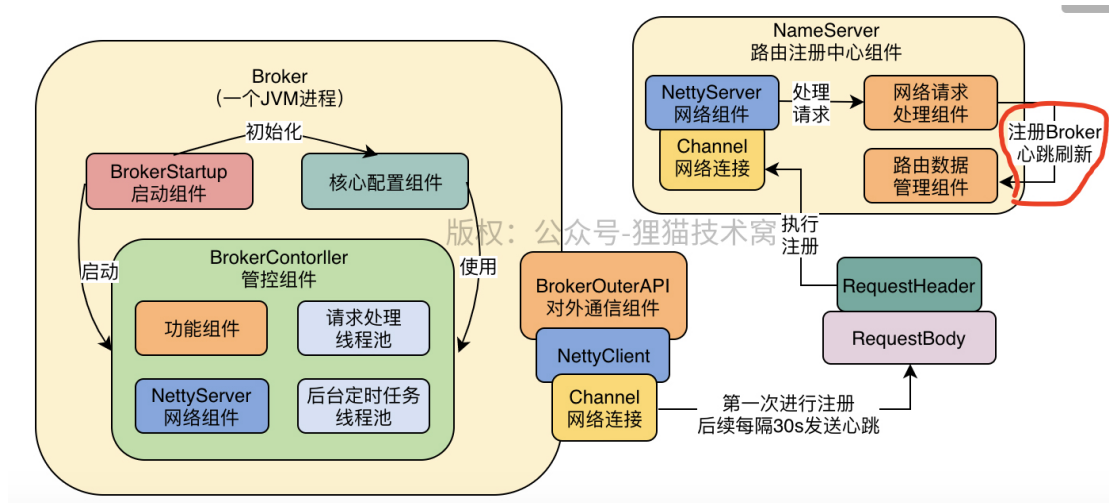
```

```

93 // 下面的代码暂时先不用管了
94 if (filterServerList != null) {
95     if (filterServerList.isEmpty()) {
96         this.filterServerTable.remove(brokerAddr);
97     } else {
98         this.filterServerTable.put(brokerAddr, filterServerList);
99     }
100 }
101
102 if (MixAll.MASTER_ID != brokerId) {
103     String masterAddr = brokerData.getBrokerAddrs().get(MixAll.MASTER_ID);
104     if (masterAddr != null) {
105         BrokerLiveInfo brokerLiveInfo = this.brokerLiveTable.get(masterAddr);
106         if (brokerLiveInfo != null) {
107             result.setHaServerAddr(brokerLiveInfo.getHaServerAddr());
108             result.setMasterAddr(masterAddr);
109         }
110     }
111 }
112 } finally {
113     this.lock.writeLock().unlock();
114 }
115 } catch (Exception e) {
116     log.error("registerBroker Exception", e);
117 }
118
119 return result;
120 }
121

```

所以我们看下图，有一个红色圈圈示意了，每隔30s你发送注册请求作为心跳的时候，RouteInfoManager里会进行心跳时间刷新处理。



接着我们来思考最后一个问题，那么假设Broker已经挂了，或者故障了，隔了很久都没有发送那个每隔30s一次的注册请求作为心跳，那么此时NameServer是如何感知到这个Broker已经挂掉的呢？

我们重新回到NamesrvController的initialize()方法里去，里面有一个代码是启动了RouteInfoManager中的一个定时扫描不活跃Broker的线程。

```

1 this.scheduledExecutorService.scheduleAtFixedRate(new Runnable() {
2
3     @Override
4     public void run() {
5         NamesrvController.this.routeInfoManager.scanNotActiveBroker();
6     }
7
8 }, 5, 10, TimeUnit.SECONDS);
9

```

上面这段代码，就是启动一个定时调度线程，每隔10s扫描一次目前不活跃的Broker，使用的是RouteInfoManager中的scanNotActiveBroker()方法，我们去看看那个方法的逻辑，就知道他如何感知到一个Broker挂掉了。

```
1 public void scanNotActiveBroker() {
2     // 大家可以看到，他这里就是扫描BrokerLiveInfo这个心跳数据结构
3     // 他里面遍历一下，就可以拿到每个Broker最近一次心跳刷新的BrokerLiveInfo
4     // 也就知道一个Broker最近一次发送心跳是什么时候
5     Iterator<Entry<String, BrokerLiveInfo>> it = this.brokerLiveTable.entrySet().iterator();
6     while (it.hasNext()) {
7         Entry<String, BrokerLiveInfo> next = it.next();
8         long last = next.getValue().getLastUpdateTimestamp();
9         // 下面这就是核心判断逻辑
10        // 如果说当前时间距离上一次心跳时间，超过了broker心跳超时时间，默认是120s
11        // 也就是说，如果一个Broker两分钟没发送心跳，就认为他已经死掉了
12        if ((last + BROKER_CHANNEL_EXPIRED_TIME) < System.currentTimeMillis()) {
13            RemotingUtil.closeChannel(next.getValue().getChannel());
14            it.remove();
15            log.warn("The broker channel expired, {} {}ms", next.getKey(), BROKER_CHANNEL_EXPIRED_TIME);
16            // 此时在这里就会把这个Broker从路由数据表里都剔除出去
17            this.onChannelDestroy(next.getKey(), next.getValue().getChannel());
18        }
19    }
20 }
```

今天给大家留一个源码分析的小作业，就是把Broker的注册、心跳以及故障发现的相关源码都看一遍，同时结合我们画的图，深刻的理解和记忆Broker跟NameServer的交互流程，核心组件。

End

专栏版权归公众号**狸猫技术窝**所有

未经许可不得传播，如有侵权将追究法律责任

狸猫技术窝精品专栏及课程推荐：

[《从零开始带你成为JVM实战高手》](#)
[《21天互联网Java进阶面试训练营》（分布式篇）](#)
[《互联网Java工程师面试突击》（第1季）](#)
[《互联网Java工程师面试突击》（第3季）](#)

重要说明：

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：《付费用户如何加群》（[购买后可见](#)）

