图文 99 Broker启动的时候是如何初始化自己的核心配置的?

详情 评论

Broker启动的时候是如何初始化自己的核心配置的?

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：

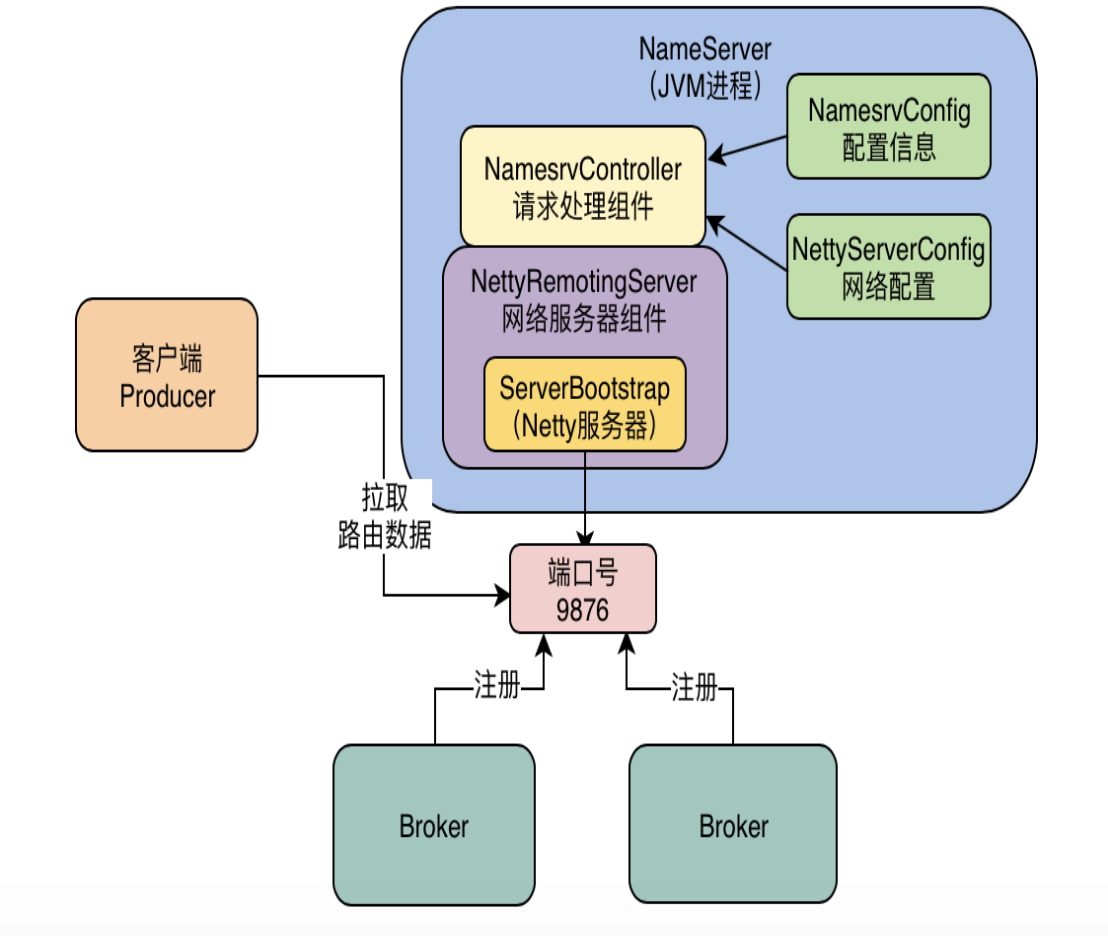
（点击下方蓝字试听）

《从零开始带你成为MySQL实战优化高手》

1、NameServer已经启动后的示意图

之前我们已经用了几讲的内容分析了一下NameServer的启动过程，从他的启动脚本开始讲起，然后一路讲解了他的配置的初始化，以及核心的NamesrvController组件的初始化和启动，最后通过源码一步一步发现，居然底层是基于Netty构建了一个网络服务器，然后监听了9876端口号。

于是我们可以看到下图，当我们的NameServer启动之后，其实他就是有一个Netty服务器监听了9876端口号，此时Broker、客户端这些就可以跟NameServer建立长连接和进行网络通信了！



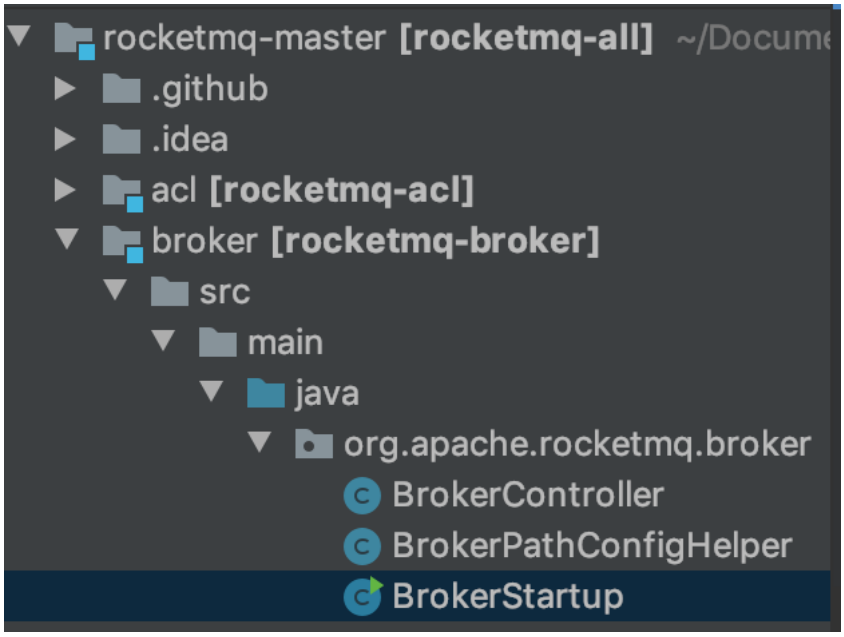
既然NameServer已经启动了，而且我们知道他已经有一个Netty服务器在监听端口号，等待接收别人的连接和请求了，接着我们就应该看看Broker是如何启动的了！

2、BrokerStartup的入口源码分析

其实大家之前看过我们的RocketMQ集群搭建和部署的实操内容，就应该都知道，启动Broker的时候也是通过mqbroker这种脚本来实现的，最终脚本里一定会启动一个JVM进程，开始执行一个main class的代码。

之前我们已经教过大家如何从启动脚本开始分析了，这里就不再重复了，我们在最后的今日源码分析作业里给大家留了作业，让大家自己去从broker的启动脚本开始分析，我们这里就直接从他的main class开始讲起了。

实际上Broker的JVM进程启动之后，会执行BrokerStartup的main()方法，这个BrokerStartup类，就在rocketmq源码中的broker模块里，大家看下图的源码截图，就会看到这个类。



我们进入这个BrokerStartup类，在里面可以看到一个main()方法，如下所示：

```
public static void main(String[] args) {
  start(createBrokerController(args));
}
```

不知道大家看到这段源码有什么感觉没有？是不是发现跟NamesrvStratup里的一段代码是很类似的？同样都是先创建了一个Controller核心组件，然后用start()方法去启动这个Controller组件！

因此，我们今天就重点分析这个createBrokerController()方法是如何创建broker的核心组件BrokerController出来的就可以了。

3、开始分析BrokerController的创建过程

进入了createBrokerContorller()方法之后，首先你会看到下面的一堆代码，很多人看到这里又会出现非常痛苦的心情，因为感觉看不懂啊！

```
System.setProperty(
        RemotingCommand.REMOTING_VERSION_KEY,
        Integer.toString(MQVersion.CURRENT_VERSION));
if (null == System.getProperty(NettySystemConfig.COM_ROCKETMQ_REMOTING_SOCKET_SNDBUF_SIZE)) {
    NettySystemConfig.socketSndbufSize = 131072;
}
if (null == System.getProperty(NettySystemConfig.COM_ROCKETMQ_REMOTING_SOCKET_RCVBUF_SIZE)) {
    NettySystemConfig.socketRcvbufSize = 131072;
}
```

很多人肯定会不知道上面的源码是干什么的，但是其实非常简单，你即使不知道也没什么大不了！因为这些都不是最核心的一些代码，你即使看不懂，也不要有畏难心理，就是接着往下看就是了！

当然，这里可以教大家一些看源码的技巧，当你不停得看各种开源项目的源码，看的多了之后，慢慢的就会摸索出很多看源码的技巧，当你的技巧积累的很多了之后，你会发现你慢慢的什么源码自己都看得懂了！

比如一开始你看到有一个System.setProperty()，他这个明显是在设置一个系统级的变量，至于设置了这个变量干什么用的，你这里留个印象就行了，你不需要知道，暂时也没法知道，毕竟这源码又不是你写的！

然后后续的

```
if (null == System.getProperty()) {
   NettySystemConfig.socketSndbufSize = 131072
}
```

这种代码，很多人可能就有点眉目了，他意思就是，如果某个系统级的变量没有设置，那么就在这里设置，而且明显发现，他设置的是Netty网络通信相关的变量，就是socket的发送缓冲大小。

看到这里，很多人还是一头雾水，为什么要在这里设置Netty的网络通信的参数？

其实要我说，你管他呢！毕竟你又不是RocketMQ的源码作者，人家就是喜欢在这里干这么个事，后续可能你看了别的代码，突然在某个地方会跟这里联想起来。但是你要在这个地方，就直接领悟出来写这段源码的含义，那是不可能的！

毕竟第一你不是RocketMQ源码作者肚子里的蛔虫，第二你又不可能跟人家源码作者有心灵感应！很多源码为什么要在这个地方写，为什么要这么写，写了之后后续派什么用场的，实际上需要你综合看后面很多其他源码，才能综合起来考虑明白。

很多时候，一段源码写在这里，可能只有源码的作者自己才知道是为什么！因为当时就是他这么想的，他才这么写的！

4、又见Broker的核心配置类

我们接着往后看，会发现如下一段奇怪的代码：

```
Options options = ServerUtil.buildCommandlineOptions(new Options());
commandLine = ServerUtil.parseCmdLine(
      "mqbroker",
      args,
      buildCommandlineOptions(options),
      new PosixParser()
);
if (null == commandLine) {
    System.exit(-1);
}
```

这个代码说奇怪，其实也不奇怪，因为之前在NameServer里看到类似的了，其实说白了，他就是用来解析你通过命令行给broker传递的一些参数的，这些参数在main()方法里通过上面的args传递进来，然后在这里他就是通过ServerUtil.parseCmdLine()方法，在解析这些命令行参数罢了！

接着我们继续往后看，会看到一段极为关键的源代码，大家请看我在代码里的注释。

```
// 下面三个，明显是broker的核心配置类
// 看起来分别是broker的配置、netty服务器的配置、netty客户端的配置
final BrokerConfig brokerConfig = new BrokerConfig();
final NettyServerConfig nettyServerConfig = new NettyServerConfig();
final NettyClientConfig nettyClientConfig = new NettyClientConfig();
// 这个地方，你只要知道他设置了netty客户端的一个是否使用TLS的配置就行了
// 至于说TLS是干什么用的，其实你暂时真的也不用知道
// 假如说你真的对Netty的TLS很感兴趣，那么可以网上搜索一下，这是加密机制
nettyClientConfig.setUseTLS(Boolean.parseBoolean(System.getProperty(
        TLS_ENABLE,
        String.valueOf(TlsSystemConfig.tlsMode == TlsMode.ENFORCING)))
);
// 这里就是设置了netty服务器的监听端口号是10911
nettyServerConfig.setListenPort(10911);
// 接着又搞了一个很关键的配置组件出来，一看名字就知道
// 这个明显是broker用来存储消息的一些配置信息
final MessageStoreConfig messageStoreConfig = new MessageStoreConfig();
// 这儿就是说，如果当前这个broker是slave的话，那么这里就要设置一个特殊的参数
// 至于说下面设置的这个accessMessageInMemoryMaxRatio参数是干什么的
// 你可以去RocketMQ的官网里查找一下
if (BrokerRole.SLAVE == messageStoreConfig.getBrokerRole()) {
   int ratio = messageStoreConfig.getAccessMessageInMemoryMaxRatio() - 10;
   messageStoreConfig.setAccessMessageInMemoryMaxRatio(ratio);
}
```

大家看完了上面那段源码的分析，有什么感觉？

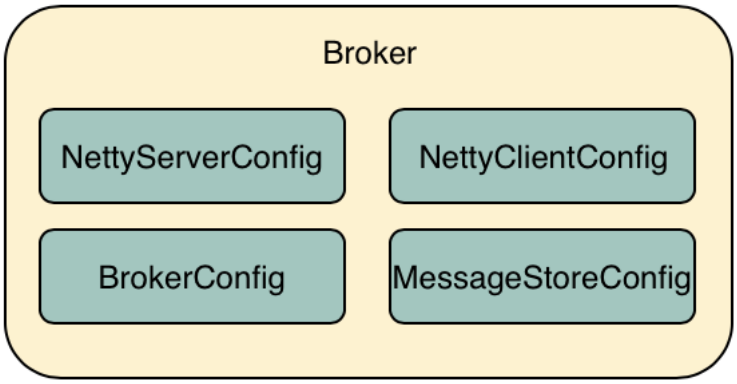
很明显，套路是一样的，broker在这里启动的时候也是先搞了几个核心的配置组件，包括了broker自己的配置、broker作为一个netty服务器的配置、broker作为一个netty客户端的配置、broker的消息存储的配置。

那么为什么broker自己又是netty服务器，又是netty客户端呢？

很简单了，当你的客户端连接到broker上发送消息的时候，那么broker就是一个netty服务器，负责监听客户端的连接请求。

但是当你的broker跟nameserver建立连接的时候，你的broker又是一个netty客户端，他要跟nameserver的netty服务器建立连接。

所以通过上述分析，我们画出了下面的图，包含了Broker的几个核心配置组件。



5、为核心配置类解析和填充信息

接着我们看看他是如何为自己的核心配置类，解析和填充信息的，继续看下面的代码。

```
if (commandLine.hasOption('c')) {
    String file = commandLine.getOptionValue('c');
    if (file != null) {
        configFile = file;
        InputStream in = new BufferedInputStream(new FileInputStream(file));
        properties = new Properties();
        properties.load(in);
        properties2SystemEnv(properties);
        MixAll.properties2Object(properties, brokerConfig);
        MixAll.properties2Object(properties, nettyServerConfig);
        MixAll.properties2Object(properties, nettyClientConfig);
        MixAll.properties2Object(properties, messageStoreConfig);
        BrokerPathConfigHelper.setBrokerConfigPath(file);
        in.close();
    }
}
```

上面代码其实清晰明了，假设说你在启动broker的时候，用了-c选项带了一个配置文件的地址，此时他会读取配置文件里的你自定义的一些配置的信息，然后读取出来覆盖到那4个核心配置类里去。

大家都记得我们之前启动broker的时候，其实都是要自定义一个broker配置文件的，然后用mqbroker启动的时候，都是要用-c选项带上自己的配置文件地址的，就是在上面的代码中，他会读取我们自定义的配置文件，填充到他的配置类里去。

接着我们往后看下面的源代码，我都在注释里写了他是如何解析和填充配置的。

```
// 这段代码就是说，你可能有一些配置是放在命令行参数里的，就在这里解析了
// 然后填充到BrokerConfig配置类里去
MixAll.properties2Object(
    ServerUtil.commandLine2Properties(commandLine), brokerConfig);

// 这个就是在检查ROCKETMQ_HOME这个环境变量了，如果没有的话
// 那么会导致broker启动失败的，他通过System.exit()会直接让JVM进城退出的
if (null == brokerConfig.getRocketmqHome()) {
   System.out.printf(
       "Please set the %s variable in your environment to match the
       MixAll.ROCKETMQ_HOME_ENV);
   System.exit(-2);
}
// 这个就不用多说了把，说白了，就是读取你的nameserver地址列表
// 读取出来之后会进行一定的格式解析，比如按照“;”符号进行字符串的切割
// 因为你的nameserver地址里可能有多个地址
String namesrvAddr = brokerConfig.getNamesrvAddr();
if (null != namesrvAddr) {
   try {
       String[] addrArray = namesrvAddr.split(";");
       for (String addr : addrArray) {
           RemotingUtil.string2SocketAddress(addr);
       }
   } catch (Exception e) {
       ...
   }
}
```

```
// 下面这段代码，就是判断一下broker的角色，针对不同的角色做个处理
switch (messageStoreConfig.getBrokerRole()) {

   case ASYNC_MASTER:
   case SYNC_MASTER:
       brokerConfig.setBrokerId(MixAll.MASTER_ID);
       break;
   case SLAVE:
       if (brokerConfig.getBrokerId() <= 0) {
           System.out.printf("Slave's brokerId must be > 0");
           System.exit(-3);
       }
       break;
   default:
       break;
}
// 这里大家会看到，其实判断是否基于dleger技术来管理主从同步和commitlog
// 如果是的话，就把brokerid设置为-1
if (messageStoreConfig.isEnableDLegerCommitLog()) {
   brokerConfig.setBrokerId(-1);
}
// 下面这段配置，就是设置了HA监听端口号
// 可能有人不知道这HA监听端口号是干什么的，没关系，你现在暂时不用知道
messageStoreConfig.setHaListenPort(
    nettyServerConfig.getListenPort() + 1);

// 下面这段代码其实你也不用关注他，你只要知道他是跟日志相关的就行了
LoggerContext lc = (LoggerContext) LoggerFactory.getILoggerFactory();
JoranConfigurator configurator = new JoranConfigurator();
configurator.setContext(lc);
lc.reset();
configurator.doConfigure(
    brokerConfig.getRocketmqHome() + "/conf/logback_broker.xml");

```

```
// 如果命令行中包含了-p参数
// 其实下面说白了，就是启动broker的时候打印一下所有配置类的启动参数
if (commandLine.hasOption('p')) {
    InternalLogger console = InternalLoggerFactory.
        getLogger(LoggerName.BROKER_CONSOLE_NAME);

    MixAll.printObjectProperties(console, brokerConfig);
    MixAll.printObjectProperties(console, nettyServerConfig);
    MixAll.printObjectProperties(console, nettyClientConfig);
    MixAll.printObjectProperties(console, messageStoreConfig);
    System.exit(0);
}

// 如果命令行包含了-m参数，同样也是打印各种配置参数
else if (commandLine.hasOption('m')) {
    InternalLogger console = InternalLoggerFactory.
        getLogger(LoggerName.BROKER_CONSOLE_NAME);

    MixAll.printObjectProperties(console, brokerConfig, true);
    MixAll.printObjectProperties(console, nettyServerConfig, true);
    MixAll.printObjectProperties(console, nettyClientConfig, true);
    MixAll.printObjectProperties(console, messageStoreConfig, true);
    System.exit(0);
}
// 反正不管如何吧，他都会在这里打印broker的配置参数
log = InternalLoggerFactory.getLogger(LoggerName.BROKER_LOGGER_NAME);
MixAll.printObjectProperties(log, brokerConfig);
MixAll.printObjectProperties(log, nettyServerConfig);
MixAll.printObjectProperties(log, nettyClientConfig);
MixAll.printObjectProperties(log, messageStoreConfig);
```

基本上上面的配置解析和填充的代码，一路看下来就到这里了，大家重点不是去理解他的代码怎么写的，而是去尝试积累这种看源码的经验和技巧，你要明白，任何其他的开源项目，可能都有类似的代码，就是构建配置类，读取配置文件的配置，解析命令行的配置参数，然后做各种配置的校验和设置。

最终他就会在这里得到4个填充完整的配置类了！明天我们就该讲解他有了这些配置之后，是如何构建出来BrokerController的！

6、今日源码分析作业

今天给大家留一个源码分析的小作业，大家可以去看看broker的启动脚本，然后分析一下启动脚本里干的事情，他是如何一步一步的启动Broker的JVM进程的，然后执行的main class是谁，他的默认的JVM参数都是什么。

然后大家顺着main class的源代码，参考本文的源码分析思路，自己再把初始化配置的源码过程自己看一下，找找那种逐步逐步看懂源码的感觉，掌握分析源码的技巧

如果大家看的时候有什么心得体会，可以在评论区里发出来，跟别人一起交流！

End

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**）