图文 96 NameServer在启动的时候都会解析哪些配置信息?



详情 评论

**NameServer在启动的时候都会解析哪些配置信息?**


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、猜猜NamesrvController到底是个什么东西?**

我们现在来正式开始看NameServer的启动流程的源码，首先我们昨天已经讲到，NamesrvStartup这个类的main()方法会被执行，然后执行的时候实际上会执行一个main0()这么个方法，如下所示。

```java
public static NamesrvController main0(String[] args) {

    try {
        NamesrvController controller = createNamesrvController(args);
        start(controller);

        String tip = "The Name Server boot success. serializeType="
            + RemotingCommand.getSerializeTypeConfigInThisServer();

        log.info(tip);
        System.out.printf("%s%n", tip);
        return controller;
    } catch (Throwable e) {
        e.printStackTrace();
        System.exit(-1);
    }
    return null;
}
```

在上面的源码中，我们会注意到这么一行代码：

NamesrvController controller = createNamesrvController(args);

这行代码很明显，就是在创建一个NamesrvController类，这个类似乎是NameServer中的一个核心组件。

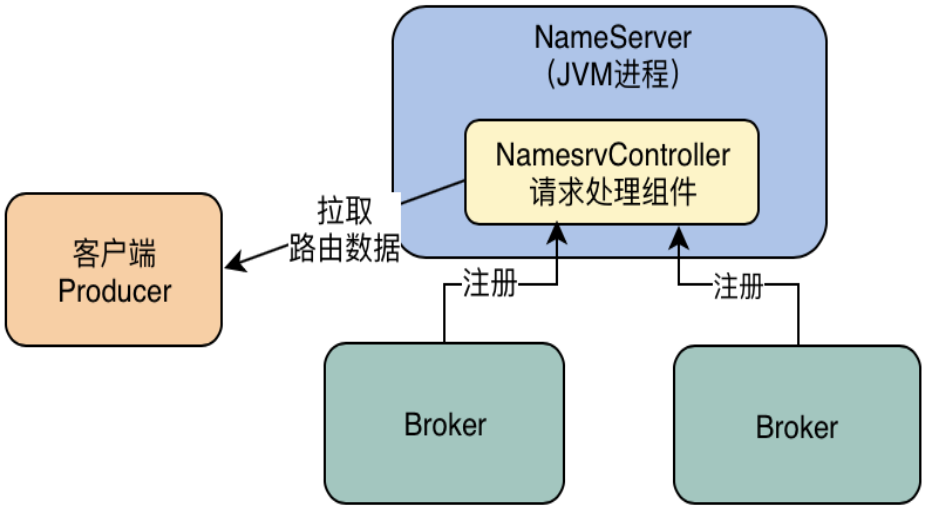
那么大家觉得这个类可能会是用来干什么的呢?

我们可以大胆的推测一下，NameServer启动之后，是不是需要接受Broker的请求？因为Broker都要把自己注册到NameServer上去。

然后Producer这些客户端是不是也要从NameServer拉取元数据？因为他们需要知道一个Topic的MessageQueue都在哪些Broker上。

所以我们完全可以猜想一下，NamesrvController这个组件，很可能就是NameServer中专门用来接受Broker和客户端的网络请求的一个组件！因为平时我们写Java Web系统的时候，大家都喜欢用Spring MVC框架，在Spring MVC框架中，用于接受HTTP请求的，就是Controlller组件！

所以我们看下面的图，大家可以先推测一下，NamesrvController组件，实际上就是NameServer中的核心组件，用来负责接受网络请求的！




**2、NamesrvController是如何被创建出来的?**

接着我们来看一下，NamesrvController是如何被创建出来的？还是回到那行代码：

NamesrvController controller = createNamesrvController(args)

这里明显调用了一个createNamesrvController()方法，创建出来了NamesrvController这个关键组件！

所以我们可以初步看一下，createNamesrvController()这个方法中大概是在干什么呢？我们继续往下看。

```
public static NamesrvController createNamesrvController(String[] args)
    throws IOException, JoranException {

    System.setProperty(RemotingCommand.REMOTING_VERSION_KEY,
                       Integer.toString(MQVersion.CURRENT_VERSION));

   //PackageConflictDetect.detectFastjson();
   Options options = ServerUtil.buildCommandlineOptions(new Options());
   commandLine = ServerUtil.parseCmdLine("mqnamesrv",
                                         args,
                                         buildCommandlineOptions(options),
                                         new PosixParser());
   if (null == commandLine) {
       System.exit(-1);
       return null;
   }
   final NamesrvConfig namesrvConfig = new NamesrvConfig();
   final NettyServerConfig nettyServerConfig = new NettyServerConfig();
   nettyServerConfig.setListenPort(9876);

   if (commandLine.hasOption('c')) {
       String file = commandLine.getOptionValue('c');
       if (file != null) {
           InputStream in = new BufferedInputStream(new FileInputStream(file));
           properties = new Properties();
           properties.load(in);
           MixAll.properties2Object(properties, namesrvConfig);
           MixAll.properties2Object(properties, nettyServerConfig);
           namesrvConfig.setConfigStorePath(file);
           System.out.printf("load config properties file OK, %s%n", file);
           in.close();
       }
   }
```

```
34    if (commandLine.hasOption('p')) {
35
36        InternalLogger console = InternalLoggerFactory.getLogger(LoggerName.NAMESRV_CONSOLE_NAME);
37        MixAll.printObjectProperties(console, namesrvConfig);
38        MixAll.printObjectProperties(console, nettyServerConfig);
39        System.exit(0);
40    }
41    MixAll.properties2Object(ServerUtil.commandLine2Properties(commandLine), namesrvConfig);
42
43    if (null == namesrvConfig.getRocketmqHome()) {
44
45        System.out.printf(
46            "Please set the %s variable in your environment to match the location of the RocketMQ installation%n",
47            MixAll.ROCKETMQ_HOME_ENV);
48
49        System.exit(-2);
50    }
51
52    LoggerContext lc = (LoggerContext) LoggerFactory.getILoggerFactory();
53    JoranConfigurator configurator = new JoranConfigurator();
54    configurator.setContext(lc);
55    lc.reset();
56    configurator.doConfigure(namesrvConfig.getRocketmqHome() + "/conf/logback_namesrv.xml");
57
58    log = InternalLoggerFactory.getLogger(LoggerName.NAMESRV_LOGGER_NAME);
59
60    MixAll.printObjectProperties(log, namesrvConfig);
61    MixAll.printObjectProperties(log, nettyServerConfig);
62
63    final NamesrvController controller = new NamesrvController(namesrvConfig, nettyServerConfig);
64    // remember all configs to prevent discard
65    controller.getConfiguration().registerConfig(properties);
66    return controller;
67 }
```

上面那段代码是不是看着让人感觉特别的痛苦？是不是大家开始初步的感觉到阅读源码的痛苦了？往往看一些开源项目源码的时候，很多人就是初步看一看，看到类似上面这种代码的时候，就感觉看不下去了，因为实在是看不懂他在干什么！

这个时候大家不要着急，我们来慢慢的给大家解释一下，分一个一个小的代码片段，来给大家拆解一下上面的代码在干什么。

**3、阅读源码的一个技巧：哪些需要细看，哪些可以暂时先跳过**

这里我们结合上面的源码，来给大家讲解一下阅读源码的一个小技巧，简单来说，就是在阅读源码的时候，有些源码是要细看的，但是有些源码你可以大致猜测一下他的作用，就直接略过去了，抓住真正的重点去看！

比如说上面的createNamesrvController()方法，进入之后，刚开始就有一段让人看不太懂的代码，我们看看下面。

```
1 System.setProperty(
2     RemotingCommand.REMOTING_VERSION_KEY, Integer.toString(MQVersion.CURRENT_VERSION));
3
4 //PackageConflictDetect.detectFastjson();
5 Options options = ServerUtil.buildCommandlineOptions(new Options());
6 commandLine = ServerUtil.parseCmdLine("mqnamesrv",
7                                       args,
8                                       buildCommandlineOptions(options),
9                                       new PosixParser()
10 );
11 if (null == commandLine) {
12     System.exit(-1);
13     return null;
14 }
```

上面这段源码，大家看了有什么感受？其实估计大部分人都没什么感受，就是不知道这段代码是在干什么！

如果这个时候，有的人喜欢钻牛角尖的，直接去分析上面代码中的一些细节，比如看看ServerUtil.buildCommandlineOptions(new Options())是在干什么，或者看看ServerUtil.parseCmdLine()是在干什么，那你就误入迷途了

因为很明显上面的代码并不存在什么核心逻辑，你从他的代码的字面意思就可以大致猜测出来，他里面包含了很多CommandLine相关的字眼，那么顾名思义，这就是一段跟命令行参数相关的代码！

你其实大致推测一下都知道，我们在启动NameServer的时候，是使用mqnamesrv命令来启动的，启动的时候可能会在命令行里给他带入一些参数，所以很可能就是在这个地方，上面那块代码，就是解析一下我们传递进去的一些命令行参数而已！

所以这个地方你大致猜测一下，就可以直接略过去了，其实并没有必要陷入解析命令行参数的各种细节里去。

**4、非常核心的两个NameServer的配置类**

接着我们继续分析上述的代码片段，你略过刚才那段一看就是在解析命令行参数的代码，继续往下走，可以看到很关键的三行代码：

```
final NamesrvConfig namesrvConfig = new NamesrvConfig();
final NettyServerConfig nettyServerConfig = new NettyServerConfig();
nettyServerConfig.setListenPort(9876);
```

上面三行代码才是你真正要关注的，你会看到他创建了NamesrvConfig和NettyServerConfig两个关键的配置类！

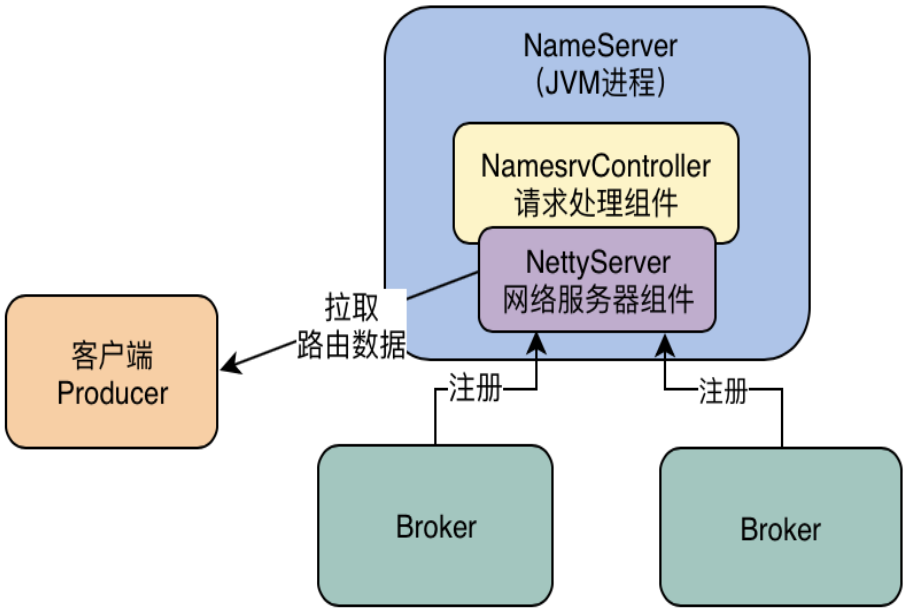
从他的类名，我们就可以推测出来，NamesrvConfig包含的是NameServer自身运行的一些配置参数，NettyServerConfig包含的是用于接收网络请求的Netty服务器的配置参数。

在这里也能明确感觉到，NameServer对外接收Broker和客户端的网络请求的时候，底层应该是基于Netty实现的网络服务器！

如果有朋友不知道Netty是什么，建议可以上网查一些Netty入门的博客和资料看看。

而且我们通过nettyServerConfig.setListenPort(9876)这行代码就可以发现，NameServer他默认固定的监听请求的端口号就是9876，因为他直接在代码里写死了这个端口号了，所以NettyServer应该就是监听了9876这个端口号，来接收Broker和客户端的请求的！

我们看下面的图，在图里我示意了基于Netty实现的服务器用于接收网络请求。




**5、看看NameServer的两个核心配置类里都包含了什么？**

接着我们看看NameServer的那两个核心配置类里都包含了什么东西，我们直接看下面的两个类的代码片段以及我写的注释就可以了。

```
public class NamesrvConfig {

    // 这个顾名思义，就是RocketMQ的home主目录地址
    // 其实大家会发现，他就是去尝试获取了ROCKETMQ_HOME这个环境变量的值
    private String rocketmqHome = System.getProperty(
        MixAll.ROCKETMQ_HOME_PROPERTY,
        System.getenv(MixAll.ROCKETMQ_HOME_ENV));


    // 这个看起来大家猜一下就是NameServer存放kv配置属性的路径
    private String kvConfigPath = System.getProperty("user.home")
        + File.separator
        + "namesrv"
        + File.separator
        + "kvConfig.json";

    // 这个是NameServer自己的配置存储路径
    private String configStorePath = System.getProperty("user.home")
        + File.separator
        + "namesrv"
        + File.separator
        + "namesrv.properties";


    // 这个一看就是说生产环境的名称，他是默认的center
    private String productEnvName = "center";

    // 这个就是说，是否启动了clusterTest测试集群，默认是false
    private boolean clusterTest = false;

    // 这个就是说，是否支持有序消息，默认就是false，不支持的
    private boolean orderMessageEnable = false;
}
```

其实看完了上面的NamesrvConfig，你会发现里面并没有什么特别关键的NameServer的配置信息。

```java
public class NettyServerConfig implements Cloneable {

    // 这个是NettyServer默认的监听端口号，是8888，但是其实上面看到了
    // 这个端口号被他在代码里设置成9876了
    private int listenPort = 8888;

    // 这个是NettyServer的工作线程的数量，默认是8
    private int serverWorkerThreads = 8;

    // 下面这个是Netty的public线程池的线程数量，默认是0
    private int serverCallbackExecutorThreads = 0;

    // 这是Netty的IO线程池的线程数量，默认是3，这里的线程是负责解析网络请求的
    // 他这里的线程解析完网络请求之后，就会把请求转发给work线程来处理
    private int serverSelectorThreads = 3;


    // 下面两个是broker端的参数
    // 就是broker端在基于netty构建网络服务器的时候，会使用下面两个参数
    private int serverOnewaySemaphoreValue = 256;
    private int serverAsyncSemaphoreValue = 64;

    // 如果一个网络连接空闲超过120s，就会被关闭
    private int serverChannelMaxIdleTimeSeconds = 120;

    // socket send buffer缓冲区以及receive buffer缓冲区的大小
    private int serverSocketSndBufSize = NettySystemConfig.socketSndbufSize;
    private int serverSocketRcvBufSize = NettySystemConfig.socketRcvbufSize;

    // ByteBuffer是否开启缓存，默认是开启的
    private boolean serverPooledByteBufAllocatorEnable = true;

    // 是否启动epoll IO模型，默认是不开启的
    private boolean useEpollNativeSelector = false;
}
```

其实上面的NettyServerConfig一看就很明确了，那里的参数就是用来配置NettyServer的，配置好NettyServer之后，就可以监听9876端口号，然后Broker和客户端有请求过来，他就可以处理了。

**6、NameServer的核心配置到底是如何进行解析的?**

看明白上面两个核心配置类之后，接着我们就可以继续往下看代码，看看那两个核心配置类的配置都是如何解析的。

```
// 这段代码意思就是说，如果你用mqnamesrv启动的时候，带上了“-c”这个选项
// 那么“-c”这个选型意思就是带上一个配置文件的地址
// 接着他就可以读取那个配置文件里的内容了
if (commandLine.hasOption('c')) {

    String file = commandLine.getOptionValue('c');
    if (file != null) {
        // 大家通过这段代码可以看到，他就是基于输入流从配置文件里读取了配置
        // 读取的配置会放入一个Properties里去
        InputStream in = new BufferedInputStream(new FileInputStream(file));
        properties = new Properties();
        properties.load(in);

        // 然后他就可以基于工具类，把读取到的配置都放入到两个核心配置类里去了
        MixAll.properties2Object(properties, namesrvConfig);
        MixAll.properties2Object(properties, nettyServerConfig);

        namesrvConfig.setConfigStorePath(file);
        System.out.printf("load config properties file OK, %s%n", file);
        in.close();
    }
}
```

上面的代码如果看懂了，我来给大家举个例子，比如说你在启动NameServer的时候，用-c选项带上了一个配置文件的地址，然后此时他启动的时候，运行到上面的代码，就会把你配置文件里的配置，放入两个核心配置类里去。

比如你有一个配置文件是：nameserver.properties，里面有一个配置是serverWorkerThreads=16，那么上面的代码就会读取出来这个配置，然后覆盖到NettyServerConfig里去！

接着我们来解释剩余的配置相关的代码。

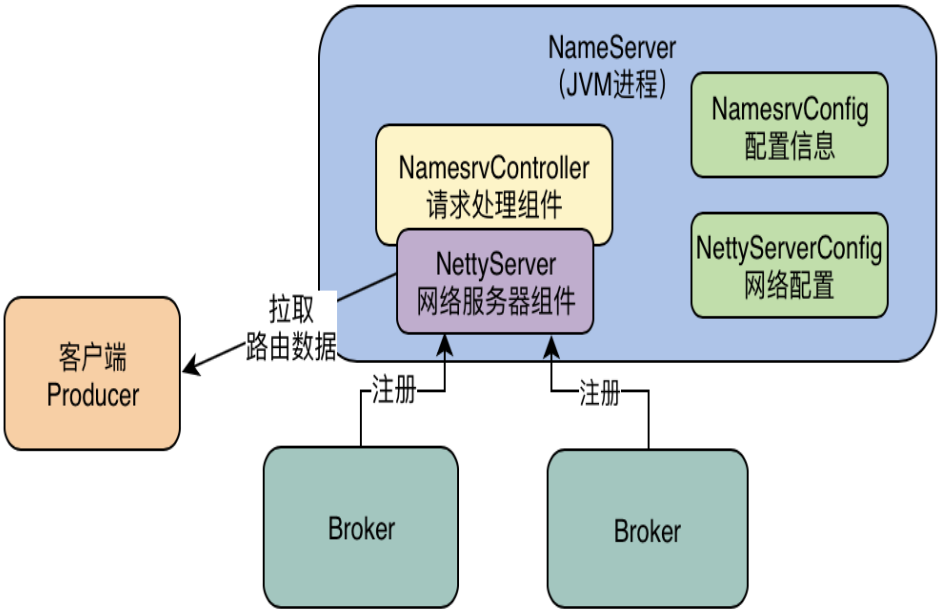
```
// 下面这段代码，其实就是说，你的mqnamesrv如果带了“-p”的选项
// 那么他的意思就是print，让你打印出来你的NameServer的所有的配置信息
if (commandLine.hasOption('p')) {
    InternalLogger console = InternalLoggerFactory.getLogger(LoggerName.NAMESRV_CONSOLE_NAME);
    MixAll.printObjectProperties(console, namesrvConfig);
    MixAll.printObjectProperties(console, nettyServerConfig);
    System.exit(0);
}
// 这个代码也很好理解了，其实他意思就是，把你在mqnamesrv命令行中
// 带上的配置选项，都读取出来，然后覆盖到NamesrvConfig里去
MixAll.properties2Object(ServerUtil.commandLine2Properties(commandLine), namesrvConfig);

// 这个也好懂了，他意思就是如果你的ROCKETMQ_HOME发现是空的
// 那么就会输出一个异常日志，说让你设置一下ROCKETMQ_HOME这个环境变量
if (null == namesrvConfig.getRocketmqHome()) {

    System.out.printf(
        "Please set the %s variable in your environment to match the location of the RocketMQ installation%n",
        MixAll.ROCKETMQ_HOME_ENV);
    System.exit(-2);
}
// 下面这段代码估计很多人是不理解的，但是不理解也没关系
// 你大致可以推测出来，都是Logger、Configurator相关的
// 所以也是日志、配置相关的
LoggerContext lc = (LoggerContext) LoggerFactory.getILoggerFactory();
JoranConfigurator configurator = new JoranConfigurator();
configurator.setContext(lc);
lc.reset();
configurator.doConfigure(namesrvConfig.getRocketmqHome() + "/conf/logback_namesrv.xml");

// 最后就是在这里他会打印一下你的NameServer的所有配置信息了
log = InternalLoggerFactory.getLogger(LoggerName.NAMESRV_LOGGER_NAME);
MixAll.printObjectProperties(log, namesrvConfig);
MixAll.printObjectProperties(log, nettyServerConfig);
```

其实在下面的图里，我直接展示出来了，NameServer启动的时候后，刚开始就是在初始化和解析NameServerConfig、NettyServerConfig相关的配置信息，但是一般情况下，我们其实不会特意设置什么配置，所以他这里一般都是用默认配置的！




**7、跟NameServer启动日志配合起来看**

其实我们知道NameServer刚启动就会初始化和解析一些核心配置信息，尤其是NettyServer的一些网络配置信息，然后初始化完毕配置信息之后，他就会打印这些配置信息，我们此时可以看一下之前讲解源码环境搭建的时候，不是指定了NameServer的启动日志么？

实际上翻看一下NameServer的启动日志，会看到如下的内容：

```
2020-02-05 15:10:05 INFO main - rocketmqHome=rocketmq-nameserver
2020-02-05 15:10:05 INFO main - kvConfigPath=namesrv/kvConfig.json
2020-02-05 15:10:05 INFO main - configStorePath=namesrv/namesrv.properties
2020-02-05 15:10:05 INFO main - productEnvName=center
2020-02-05 15:10:05 INFO main - clusterTest=false
2020-02-05 15:10:05 INFO main - orderMessageEnable=false
2020-02-05 15:10:05 INFO main - listenPort=9876
2020-02-05 15:10:05 INFO main - serverWorkerThreads=8
2020-02-05 15:10:05 INFO main - serverCallbackExecutorThreads=0
2020-02-05 15:10:05 INFO main - serverSelectorThreads=3
2020-02-05 15:10:05 INFO main - serverOnewaySemaphoreValue=256
2020-02-05 15:10:05 INFO main - serverAsyncSemaphoreValue=64
2020-02-05 15:10:05 INFO main - serverChannelMaxIdleTimeSeconds=120
2020-02-05 15:10:05 INFO main - serverSocketSndBufSize=65535
2020-02-05 15:10:05 INFO main - serverSocketRcvBufSize=65535
2020-02-05 15:10:05 INFO main - serverPooledByteBufAllocatorEnable=true
2020-02-05 15:10:05 INFO main - useEpollNativeSelector=false
```

不知道大家有何感觉？是不是感觉通过分析源码以及其中的日志打印，可以初步把源码运行的过程和日志文件的打印结合起来了？

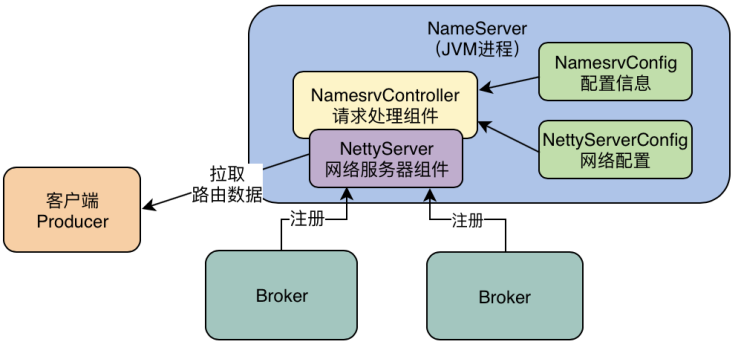
**8、完成NamesrvController组件的创建**

在今天最后要讲解的内容，就是初步看一下NamesrvController是如何创建出来的，继续看下面的代码。

这里非常明确，就是直接构造了NamesrvController这个组件，同时传递了NamesrvConfig和NettyServerConfig两个核心配置类给他。

```
1 final NamesrvController controller = new NamesrvController(namesrvConfig,
2                                                            nettyServerConfig);
3 // remember all configs to prevent discard
4 controller.getConfiguration().registerConfig(properties);
```

我们看下面的图示，我们可以看到箭头的指向，两个核心配置类在初始化完毕之后，都是交给了NamesrvController这个核心的组件的。




**9、今天的源码分析作业**

今天我们其实着重给大家分析了NameServer启动过程中的createNamesrvController()方法的流程，讲解了他是如何初始化化两个核心配置类，然后基于核心配置类构造了NamesrvController这个核心组件的。

同时在源码分析的过程中还给大家讲解了一些小技巧，所以希望大家可以今天自己在RocketMQ源码环境中，自己阅读和分析一下createNamesrvController()这个方法，去体会一下里面的源码逻辑。

**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**）