Netd

**目录**

[1 Netd 3](#_Toc523384677)

[1.1 Netd概述 3](#_Toc523384678)

[1.2 Netd启动 3](#_Toc523384679)

[1.2.1 init.rc 3](#_Toc523384680)

[1.2.2 main 4](#_Toc523384681)

[1.3 NetlinkManager分析 5](#_Toc523384682)

[1.3.1 NetlinkManager的启动 5](#_Toc523384683)

[1.3.2 关系图 7](#_Toc523384684)

[1.3.3 NetlinkHandler的start方法 8](#_Toc523384685)

[1.3.4 线程函数threadStart 9](#_Toc523384686)

[1.3.5 NetlinkListener的onDataAvailable方法 11](#_Toc523384687)

[1.3.6 NetlinkHandler的onEvent函数 12](#_Toc523384688)

[1.3.7 NetlinkManager总结 14](#_Toc523384689)

[1.4 CommandListener分析 14](#_Toc523384690)

[1.4.1 CommandListener的初始化 14](#_Toc523384691)

[1.4.2 CommandListener流程 17](#_Toc523384692)

[1.4.3 CommandListener总结 19](#_Toc523384693)

[1.5 Netd 架构图 20](#_Toc523384694)

[2 NetworkManagementService 20](#_Toc523384695)

[2.1 NetworkManagementService启动 20](#_Toc523384696)

[2.2 NativeDaemonConnector监听 21](#_Toc523384697)

[2.3 NetdCallbackReceiver处理 23](#_Toc523384698)

[2.4 NativeDaemonConnector执行 23](#_Toc523384699)

[3 EthernetService 24](#_Toc523384700)

[3.1 EthernetService启动 25](#_Toc523384701)

[3.2 EthernetServiceImpl 27](#_Toc523384702)

# Netd

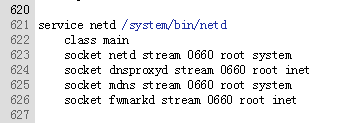
## Netd概述

Netd : Network Daemon 用于管理和控制Android平台网络的后台进程。

## Netd启动

### init.rc

/system/core/rootdir/init.rc



启动Netd,并创建4个TCP监听socket,名称分别为:

”netd”、”dnsproxyd”、”mdns”、”fwmarkd”

Framework层中的

1.NetworkManagementService <=> "netd"

2.NsdService <=> "mdns"

3.调用和域名解析相关的socket API(如getaddrinfo或gethostbyname等)的进程 <=> "dnsproxyd"

建立socket链接、监听并交互。

### main

/system/netd/server/main.cpp



//创建MDnsSdListener并启动监听，它将创建名为"mdns"的监听socket

//创建DnsProxyListener，它将创建名为"dnsproxyd"的监听socket，并启动

//创建CommandListener,它将创建名为”netd”的监听socket

//为本Netd设置环境变量ANDROID\_DNS\_MODE为"local"

//启动NetlinkManager

//设置NetlinkManager的消息发送者（Broadcaster）为CommandListener

//创建NetlinkManager,

//接收并处理Kernel的UEvent消息

//NM会借助Broadcaster(下面为NM设置的CommandListener)发送给Framework层的NetworkManagermentService

//为Netd进程屏蔽SIGPIPE信号

## NetlinkManager分析

### NetlinkManager的启动

/system/netd/server/NetlinkManager.cpp

//创建接收NETLINK\_KOBJECT\_UEVENT(kobject)消息的socket，其值保存在mUeventSock中

//其中，NETLINK\_FORMAT\_ASCII代表UEvent消息的内容为ASCII字符串,一般用来通知内核中某个模块的加载或卸载，



//创建接收NETLINK\_NFLOG消息的socket，其值保存在mQuotaSock中

// UEvent消息的类型为结构体

//带宽控制有关。Netd中的带宽控制可以设置一个预警值，当网络数据超过一定字节数就会触发kernel发送一个警告

//创建接收NETLINK\_ROUTE消息的socket，其值保存在mRouteSock中

//其中,NETLINK\_FORMAT\_BINARY代表UEvent消息的类型为结构体，故需要进行二进制解析

//代表kernel中routing或link改变时对应的消息

NM中，setupSocket方法:



//对该socket执行bind操作

//设置Socket认证，可获取到发送者uid和gid

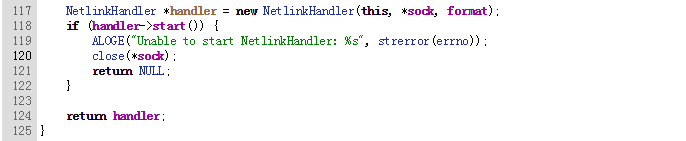
//设置Socket接收缓冲区大小

//PF\_NETLINK使用的socket地址结构是sockaddr\_nl

//而不是一般的sockaddr\_in

//创建PF\_NETLINK地址簇的socket,目前只支持SOCK\_DGRAM类

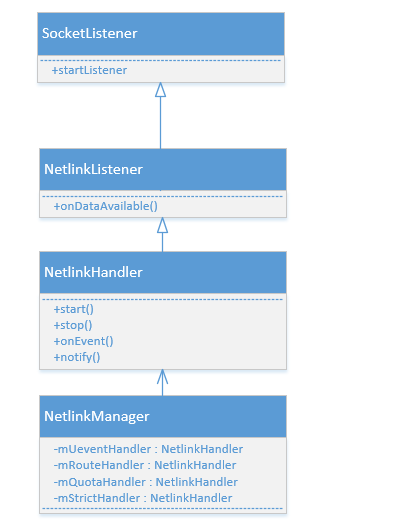
//第三个参数为传入的Uevent事件类型



//最后，将创建好的Socket传入，创建一个NetLINKHandler对象

//调用start方法

### 关系图

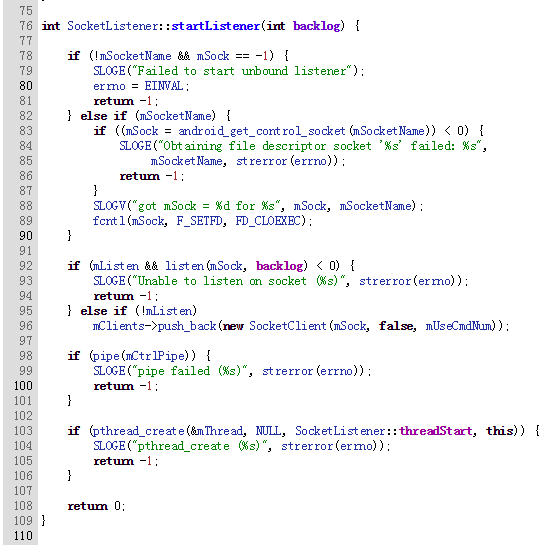


### NetlinkHandler的start方法



//startListener由SocketListener实现

/system/core/libsysutils/src/SocketListener.cpp



//创建一个工作线程,线程函数是threadStart

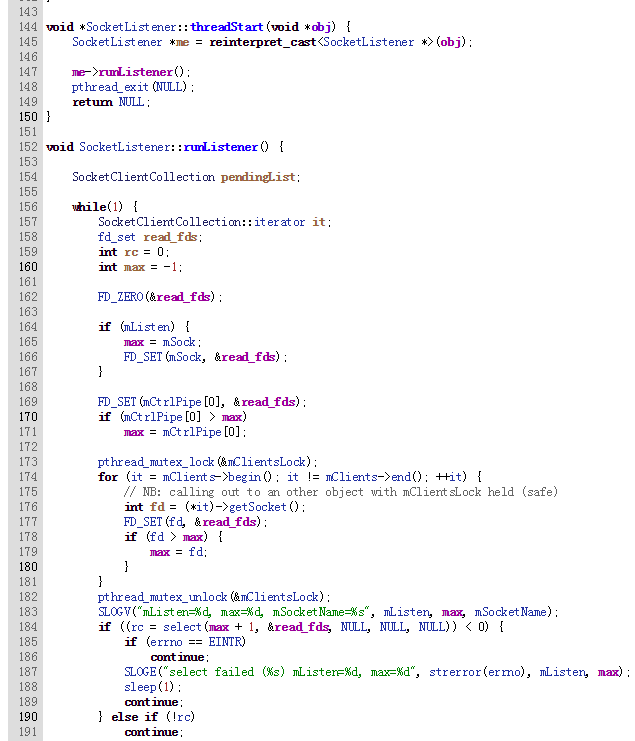
//pipe系统调用将创建一个匿名管道,mCtrlPipe是一个int二元数组

mCtrlPipe[0]用于从管道读数据,mCtrlPipe[1]用于往管道写数据

//mSock为参数构造SocketClient对象,并加入到对应的List

//mListen为false(NetlinkListener.cpp构造函数,传入)

### 线程函数threadStart



//0超时

//-1,返回出错

//管道句柄、以及遍历SocketClien对象

//得到最大的描述符，传入select函数

//select函数第一个参数，必现为它所监视的文件描述符集合中最大的文件描述符+1

//mListen为true时,listen端,此处为false

//fd\_set先置零



//有数据通过Socket发送过来，调用onDateAvailable处理

//如果onDataAvailable返回false,则表示需要关闭该连接

//根据返回的客户端Socket描述符

//构造一个SocketClient对象，并加入到list

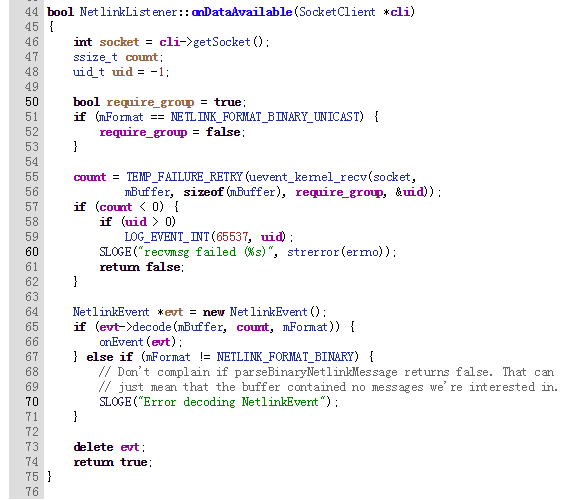
//调用accept接受客户端的连接,返回用于和客户端通信的

//Socket描述符

//如果是listen端,mSock可读表示有客户端connect上

//返回准备就绪的描述符数,继续往下

### NetlinkListener的onDataAvailable方法



//调用uevent\_kernel\_recv来接收数据,放在mBuffer中

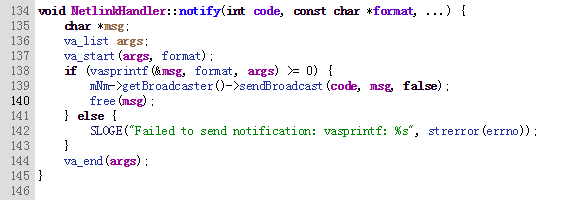
//创建NetlinkEvent,调用decode来解析收到的Uevent数据(根据指定的类型)

//调用onEvent,传入NetlinkEvent对象

### NetlinkHandler的onEvent函数



//根据不同的字段，调用不同的notifyXXXX方法,不过最后都是调用notify方法



//调用NetlinkManager的消息发送者（Broadcaster）—CommandListener

//的sendBroadcast方法,发送消息给发送给Framework层的NetworkManagermentService

### NetlinkManager总结

1.NetlinkManager创建socket,根据socket创建NetlinkHandler,并且启动start

2.NetlinkHandler继承 NetlinkListener 继承 SocketListener,最后start调用的是SocketListener的startListener

3. startListener启动一个线程，接收内核消息，调用onDataAvailable来处理, onDataAvailable方法在NetlinkListener实现

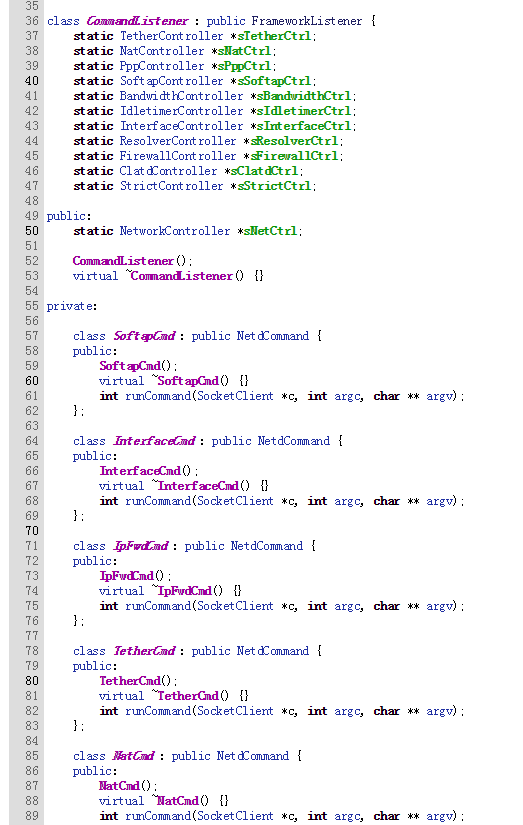
4. NetlinkListener的onDataAvailable,调用onEvent来处理消息，onEvent方法在NetlinkHandler实现

5. NetlinkHandler的onEvent方法中处理消息，最后调用notify函数，调用NetlinkManager的Broadcaster也就是CommandListener的sendBroadcast方法,发送消息给发送给Framework层的NetworkManagermentServic

## CommandListener分析

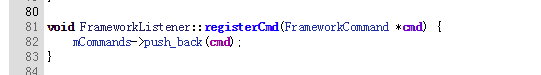
### CommandListener的初始化





CommandListener的初始化时注册了14个命令类对象, 这些类的定义在CommandListener.h中，都从NetdCommand继承, NetdCommand从FrameworkCommand继承，只写了构造函数和析构函数

registerCmd方法在CommandListener的基类FrameworkListener实现,就是加入到list中

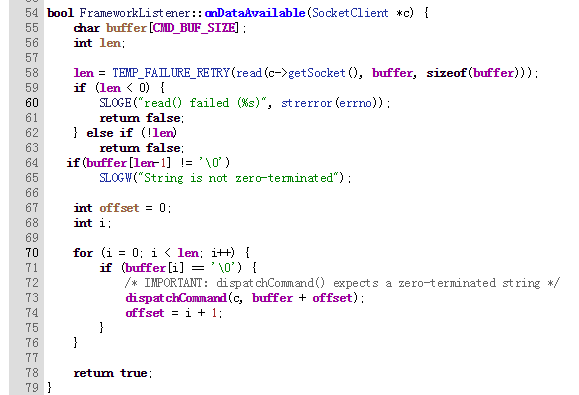


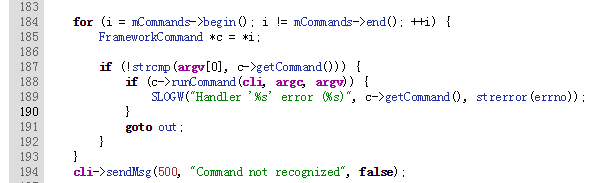


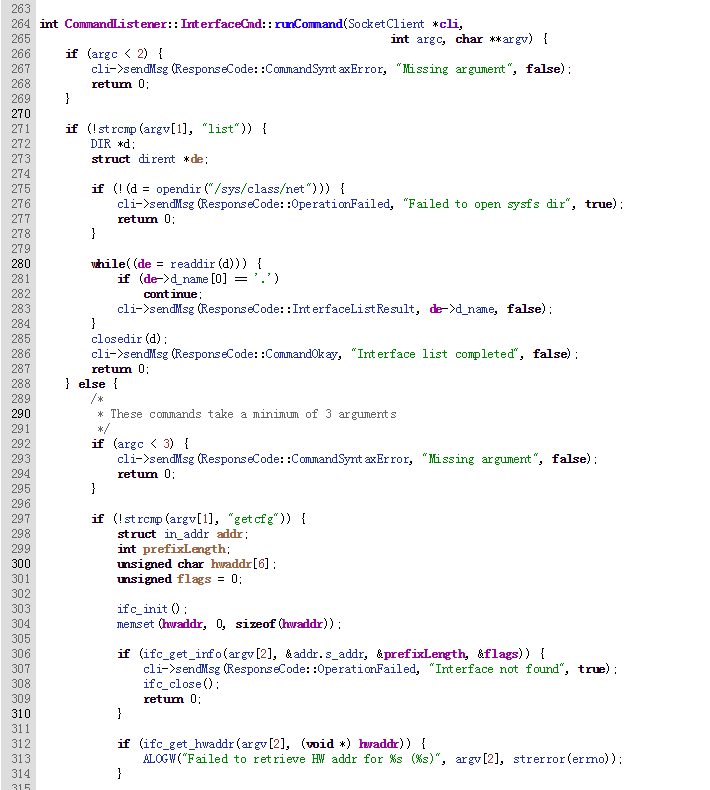
### CommandListener流程

CommandListener的startListener, 同上面NetlinkListerner一样，CommandListener继承于FrameworkListener，FrameworkListener继承于SocketListene, startListener最后SocketListener启动一个线程。执行runListener方法，来处理接收到的消息

不同的是FrameworkListener实现的onDataAvailable方法,最后调用dispatchCommand处理命令,最后调用runCommand来处理，最后消息在刚开始注册入的，不同类型的命令处理里执行对应的runCommand方法







### CommandListener总结

1. CommandListener注册入不同的命令处理类,并且启动start

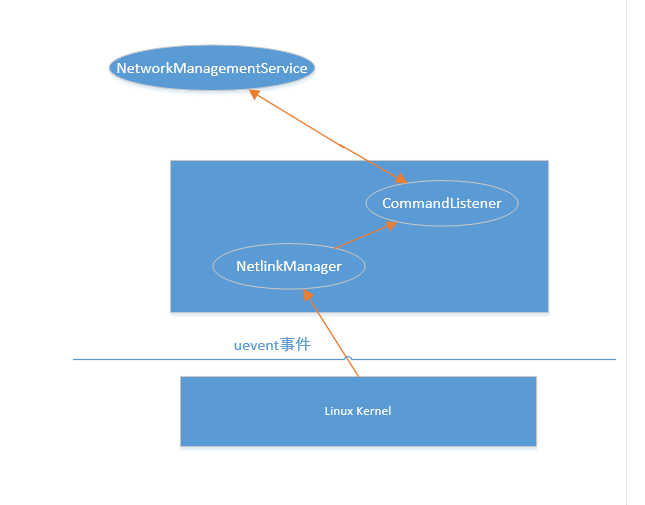
2. CommandListener继承 FrameworkListener 继承 SocketListener,最后start调用的是SocketListener的startListener

3. startListener启动一个线程，接收内核消息，调用onDataAvailable来处理, onDataAvailable方法在FrameworkListener实现

4. FrameworkListener的onDataAvailable,调用dispatchCommand来处理消息，最后调用在runCommand

5. runCommand不同的命令处理里，不同的实现来处理

## Netd 架构图

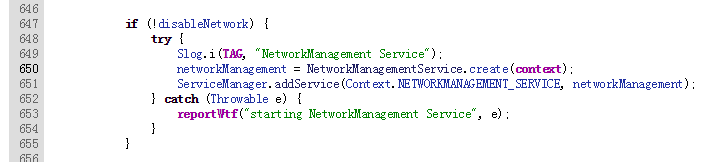


# NetworkManagementService

## NetworkManagementService启动

/frameworks/base/services/java/com/android/server/SystemServer.java

在SystemServer的startOtherServices()中启动



//创建NetworkManagementService对象，并添加到ServiceManager中



//NativeDaemonConnector和”netd”建立联系

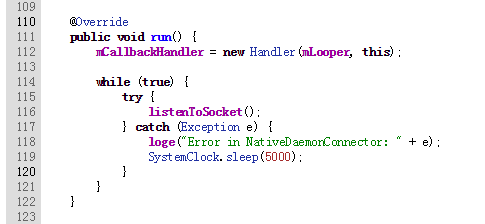
//回调在NetdCallbackReceiver中

//并放在名为: NetdConnector的Thread中执行，NativeDaemonConnector实现了Runnable

//socket name 为”netd”

## NativeDaemonConnector监听

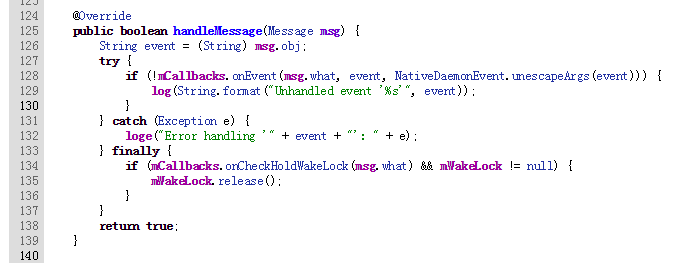
NativeDaemonConnector实现了run方法，循环执行listenToSocket





//sendMessage处理消息

//循环监听socket netd过来的消息



//最后调用mCallbacks(也就是NetdCallbackReceiver)的onEvent方法

## NetdCallbackReceiver处理

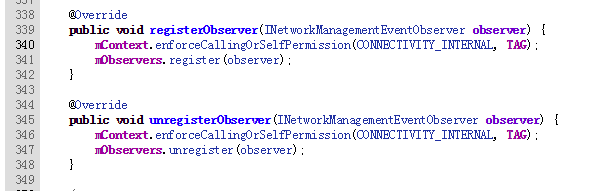
NetdCallbackReceiver的onEvent方法,处理各种事件,调用对应的notifyXXXX方法

各notifyXXXX方法最后调用mObservers实现的方法

private final RemoteCallbackList<INetworkManagementEventObserver> mObservers =

new RemoteCallbackList<INetworkManagementEventObserver>();

mObservers通过registerObserver注入



## NativeDaemonConnector执行

NMS发送命令给netd,通过mConnector.execute方法



//写入命令

# EthernetService

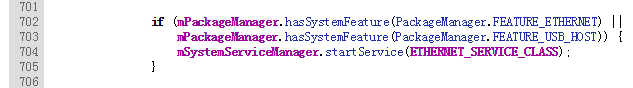
INetworkManagementEventObserver处理netd发送过来的消息

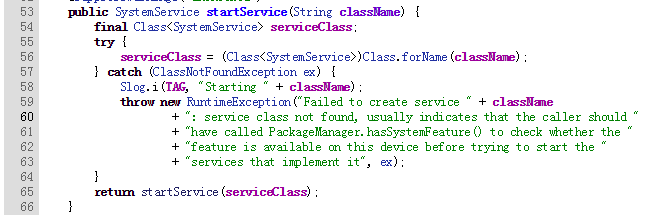
INetworkManagementEventObserver中定义了接口名称,不同的服务实现对应的接口功能，注册到NMS中,下面以EthernetService为例，简述下流程



## EthernetService启动

同样在SystemServer的startOtherServices()中启动





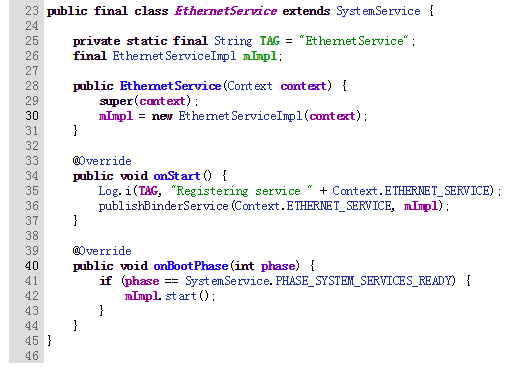
//根据类名,获取Class



//启动服务

//添加到list中

//根据Class获取构造方法,创建对象



//EthernetServiceImpl start

//onBootPhase在SystemServer中

//mSystemServiceManager.startBootPhase(SystemService.PHASE\_WAIT\_FOR\_DEFAULT\_DISPLAY);

//会遍历执行,list中所有的onBootPhase方法

//注册到Systemservice中

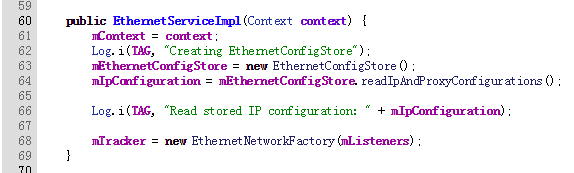
//创建了一个EthernetServiceImpl对象

## EthernetServiceImpl

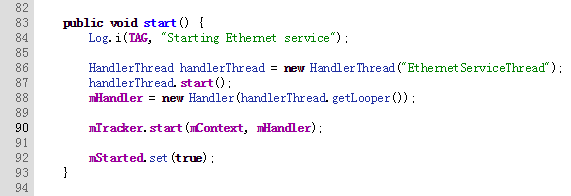
EthernetService的实质是,创建了一个EthernetServiceImpl对象，并调用其start方法

//创建一个EthernetConfigStore对象

从/misc/ethernet/ipconfig.txt"读取获取Ethernet配置信息得到IpConfiguration对象



//创建了一个EthernetNetworkFactory对象



//创建一个名字叫ernetServiceThread 线程 取出它的Looper ，创建一个handler 传入到 EthernetNetworkFactory的start 函数



//注册回调到NMS,至此NMS可以回调ES,ES也可以调用NMS的方法

//创建一个NetworkFactory并且register,target.getLooper

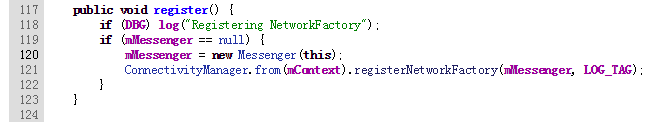
//就是EthernetServiceImpl的start中创建的HandlerThread线程的looper

//LocalNetworkFactory 继承->NetworkFactory->Handler, NetworkFactory的创建，实质是传入一个looper创建一个Handler

//获取到NetworkManagementService以及EthernetManager

## NetworkFactory

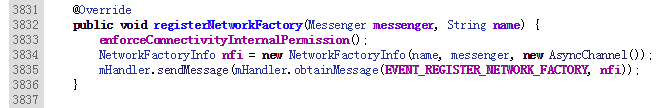
ConnectivityService服务是系统中管理，各个类型网络连接，路由配置信息的服务，这部分介绍下NetworkFactory, 每一个具备提供网络服务的对象都需要创建一个NetworkFactory的子类对象，并利用该对象注册自己，以及提供自己的分值



//register中,根据自身NetworkFactory也就是Handler,创建一个Messenger对象,LOG\_TAG值为” Ethernet”

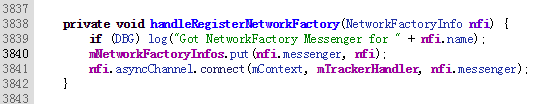
// ConnectivityManager.from(mContext)返回的是ConnectivityService

//最后实际调用的是调用ConnectivityService的registerNetworkFactory方法



//根据传入的TAG、Messenger以及新建的AsyncChannel对象创建一个 NetworkFactoryInfo

然后发送一个EVENT\_REGISTER\_NETWORK\_FACTORY消息，带上NetworkFactoryInfo对象

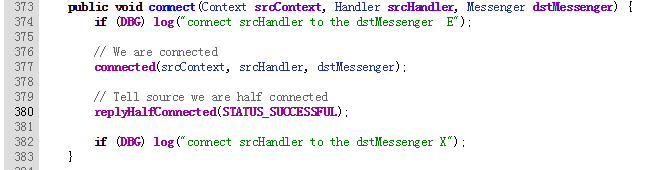


EVENT\_REGISTER\_NETWORK\_FACTORY消息最后在handleRegisterNetworkAgent函数中处理

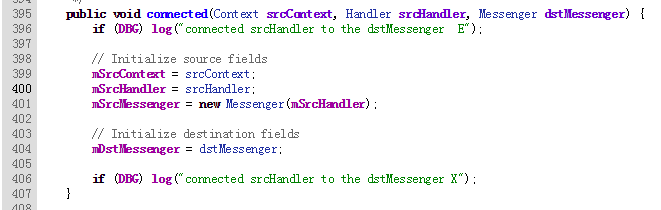
mTrackerHandler为ConnectivityServiceThread线程的Looper创建的Handler

na.messenger 为ernetServiceThread线程的Looper创建的Handler(NetworkFactory)创建的Messenger

传入到AsyncChannel里面connect



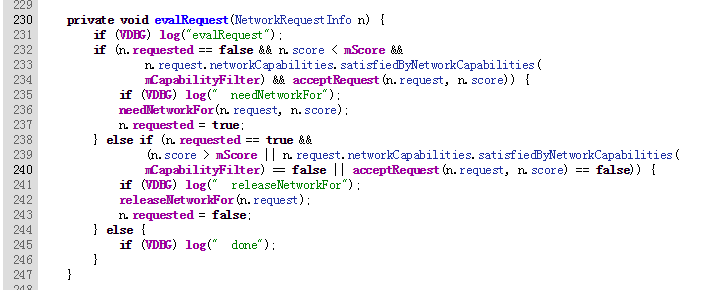
//这边会发送一个CMD\_CHANNEL\_HALF\_CONNECTED消息,在ConnectivityService中处理这个消息, handleAsyncChannelHalfConnect通过AsyncChannel的mDstMessenger发送CMD\_REQUEST\_NETWORK,在NetworkFactory处理handleAddRequest



这个mSrcHandler就是为ConnectivityServiceThread线程的Looper创建的Handler,是ConnectivityService中

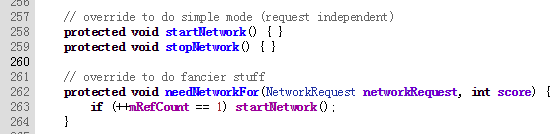
根据mSrcHandler创建一个mSrcMessenger对象

mDstMessenger就是前面那个传入的NetworkService线程的Looper创建的Handler创建的Messenger



//当NetworkRequestInfo被requested过(也就是当前网络被needNetworkFor过)，此时如果再次收到请求，并且携带的新score大于当前NetworkFactory所处网络的mScore，那么就说明当前NetworkFactory所在网络优先级已经不是最高，需要将其releaseNetworkFor掉，并标记NetworkRequestInfo.requested=false

//如果NetworkRequestInfo没有被requested过，并且其分值(n.score)小于当前NetworkFactory自己的分值(mScore), 说明当前NetworkFactory所处的网络优先级高于其他网络的优先级，就会触发当前NetworkFactory所在网络的needNetworkFor()流程，也就是连接建立流程，并将标记NetworkRequestInfo.requested=true,



//建立以太网连接,startNetwork在EthernetNetworkFactory实现

总结:

这部分流程就是EthernetService创建一个NetworkFactory注册到ConnectivityService,通过AsyncChannel建立ConnectivityService和EthernetService的单向连接, ConnectivityService来管理控制EthernetService的连接和断开