**目 录**

[1 Ethernet 1](#_Toc449375430)

[1.1 Enable 1](#_Toc449375431)

[1.1.1 Backgroud 1](#_Toc449375432)

[1.1.2 使能步骤 1](#_Toc449375433)

[1.2 流程分析 2](#_Toc449375434)

[1.2.1 源码分布 2](#_Toc449375435)

[1.2.2 启动流程 3](#_Toc449375436)

[1.2.3 网络连接 4](#_Toc449375437)

[2 Netd 8](#_Toc449375438)

[2.1 概述 8](#_Toc449375439)

[2.2 Netd工作流程 8](#_Toc449375440)

[2.2.1 main 函数分析 8](#_Toc449375441)

[2.2.2 NetlinkManager分析 10](#_Toc449375442)

[2.2.3 CommandListener分析 19](#_Toc449375443)

[2.2.4 Netd框架图 21](#_Toc449375444)

[3 WIFI 22](#_Toc449375445)

[3.1 WIFI Stack 22](#_Toc449375446)

[3.1.1 WIFI整体架构 22](#_Toc449375447)

[3.2 WIFI Framework 24](#_Toc449375448)

[3.2.1 框架 24](#_Toc449375449)

[3.2.2 WifiService 启动 26](#_Toc449375450)

[3.2.3 Wifi Framework层状态机 31](#_Toc449375451)

[3.2.4 WifiNative 32](#_Toc449375452)

[3.2.5 WifiMonitor 34](#_Toc449375453)

[3.3 WIFI HAL 和Wpa\_supplicant 36](#_Toc449375454)

[3.3.1 Wifi HAL 36](#_Toc449375455)

[3.3.2 Wpa\_supplicant 37](#_Toc449375456)

[3.4 WIFI 流程分析 37](#_Toc449375457)

[3.4.1 Wifi 打开 38](#_Toc449375458)

[3.4.2 Wifi 关闭 46](#_Toc449375459)

[3.4.3 Wifi 扫描 48](#_Toc449375460)

[4 Framework 层网络连接管理机制 51](#_Toc449375461)

[4.1 Overview 51](#_Toc449375462)

[4.1.1 ConnectivityManager 51](#_Toc449375463)

[4.1.2 ConnectivityService 51](#_Toc449375464)

[4.1.3 NetworkAgent 52](#_Toc449375465)

[4.1.4 NetworkInfo 53](#_Toc449375466)

[4.2 NetworkAgent注册过程 54](#_Toc449375467)

[4.3 网络的切换 59](#_Toc449375468)

[5 网络评分机制分析 67](#_Toc449375469)

[5.1 Overview 67](#_Toc449375470)

[5.2 NetworkFactory 67](#_Toc449375471)

[5.3 NetworkAgent 69](#_Toc449375472)

[5.3.1 直接更新分值过程 69](#_Toc449375473)

[5.3.2 间接更新分值过程 72](#_Toc449375474)

[5.4 NetworkMonitor 73](#_Toc449375475)

[5.4.1 NetworkMonitor 状态机 74](#_Toc449375476)

[5.4.2 NetworkMonitor 更新分值 76](#_Toc449375477)

# Ethernet

## Enable

### Backgroud

Android Lollipop 默认已经支持Etherent有线网络, 只需要在code中稍微配置，便可以直接使用。

### 使能步骤

1. Linux kernel

Linxu kernel 需包含ethernet 驱动，系统启动后自动生成eth网络设备节点。

Common BSP Ethernet Driver svn path：

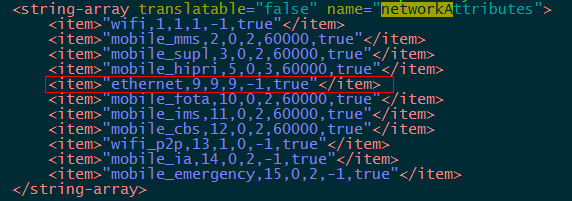
<http://svn.novatek.com.tw/svn/SP_BSP/trunk/nvt-linux/linux-3.10/drivers/net/ethernet/nvt_fasteth_mac>

1. Android Overlay

Android overlay 机制允许在不修改packages中apk的情况下，来自定义 framework和package中的资源文件，实现资源的定制。来达到显示不同的UI得目的。

源码：

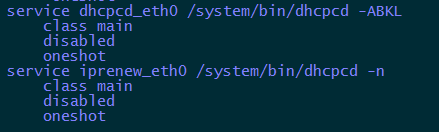
device/novatek/nt72668\_newdriver/atv\_overlay/frameworks/base/core/res/res/values/config.xml



1. Init.rc

Init.novatek-coretexa9.rc 中添加dhcp和ip\_renew service。

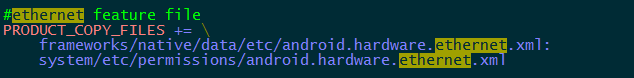
源码： device/novatek/nt72668\_newdriver/init.novatek-coretexa9.rc。



1. dhcpcd\_eth0 service 是启动后利用dhcp 为设备节点eth0 提供IP。
2. iprenew\_eth0 service 是dhcp 分配到的IP 租期到期后用来续租。
3. 两个service 在init.rc 中都声明为disableed，naitive 层会透过property\_set()的方式来启动 。
4. device.mk

拷贝ethernet feature file。

源码：device/novatek/nt72668\_newdriver/device.mk。



## 流程分析

### 源码分布

Android 5.0 (Lollipop)

1. Application

packages/apps/Settings/src/com/android/settings/ethernet

1. Ethernet Service

frameworks/opt/net/ethernet/java/com/android/server/ethernet

1. JNI

frameworks/base/core/jni/android\_net\_NetUtils.cpp

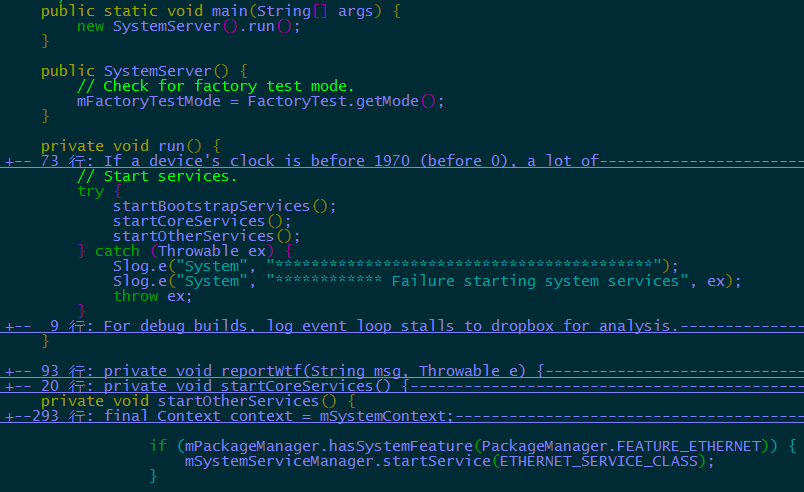
1. dhcp

system/core/libnetutils/dhcp\_utils.c

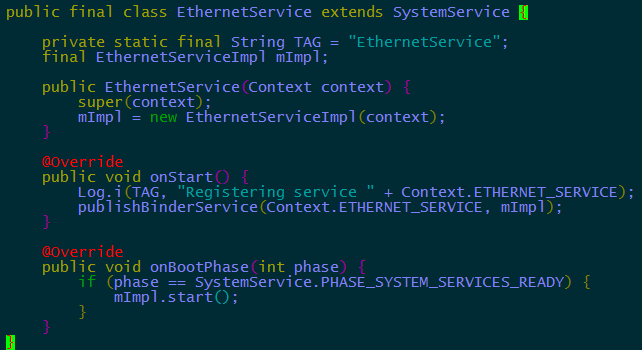
external/dhcpcd

### 启动流程

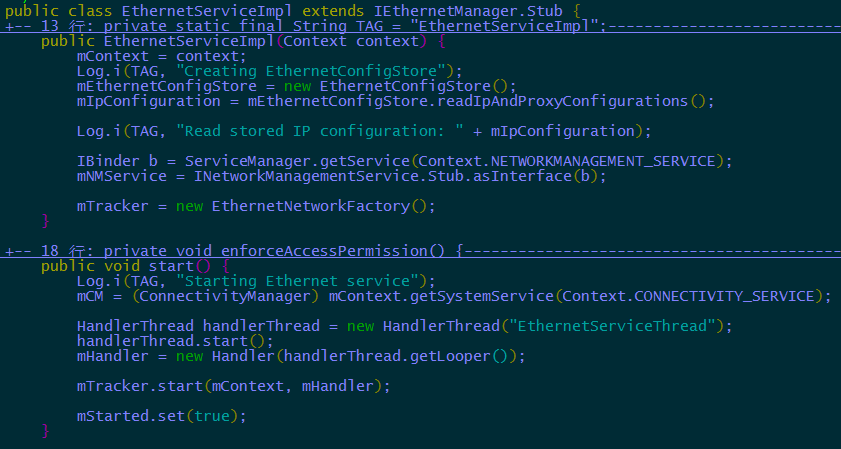
1. frameworks/base/services/java/com/android/server/SystemServer.java



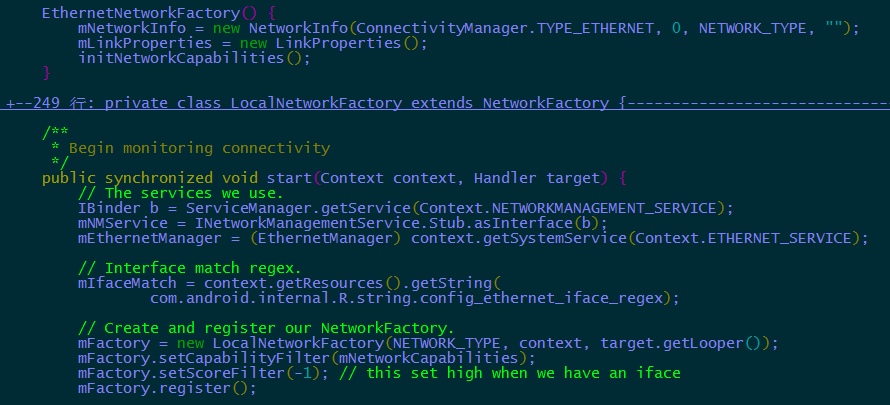
1. frameworks/opt/net/ethernet/java/com/android/server/ethernet/EthernetService.java



1. frameworks/opt/net/ethernet/java/com/android/server/ethernet/EthernetServiceImpl.java

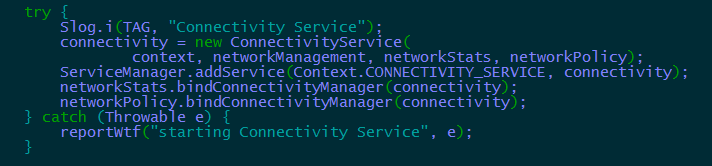


1. frameworks/opt/net/ethernet/java/com/android/server/ethernet/EthernetNetworkFactory.java

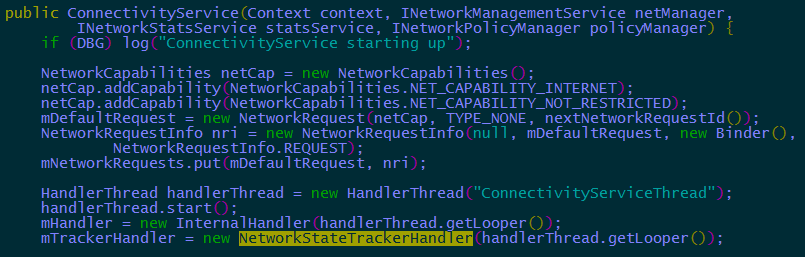


### 网络连接

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



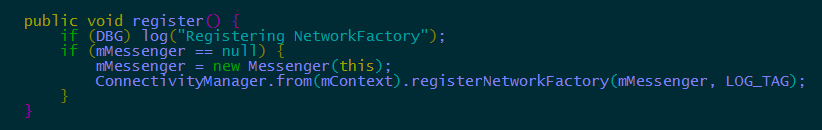
1. frameworks/base/services/core/java/com/android/server/ConnectivityService.java



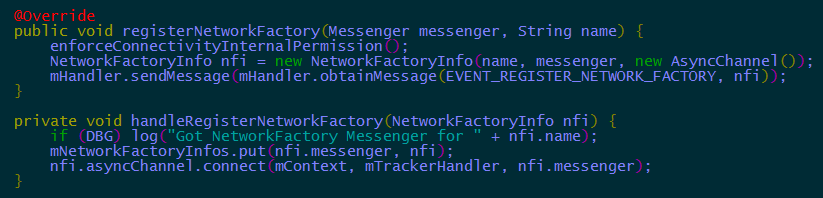
1. connectivityservic的构造函数中，会实例化一个NetworkStateTrackerHandler，

NetworkStateTrackerHandler是connectivityservice中的一个内部类,用于处理各种网络发送的handler间message。

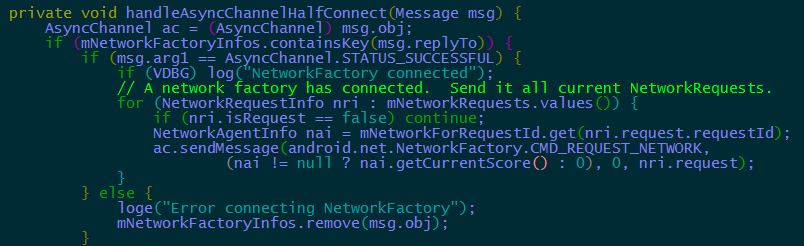
1. 同时，在构造函数中，会生成一个默认的网络连接请求，将其压入栈中。
2. frameworks/base/core/java/android/net/NetworkFactory.java



1. frameworks/base/services/core/java/com/android/server/ConnectivityService.java

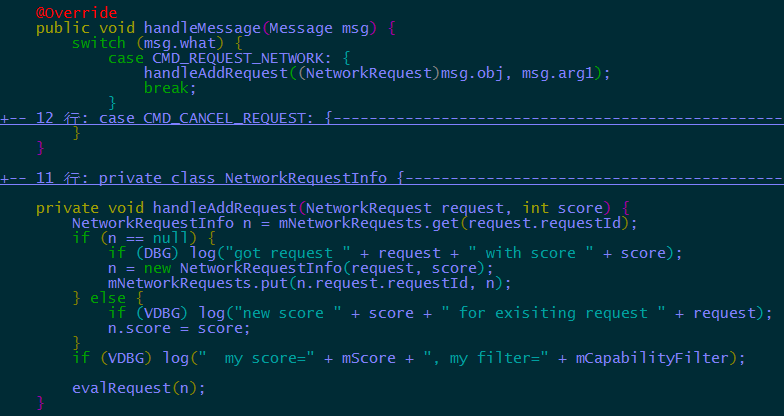


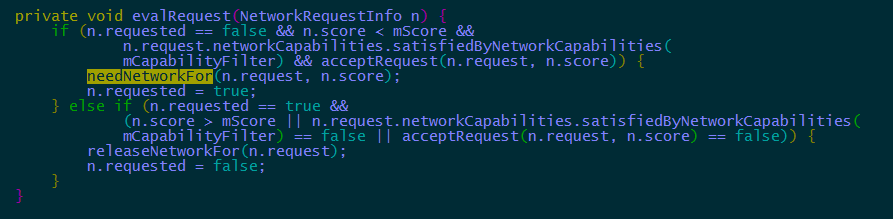
1. frameworks/base/services/core/java/com/android/server/ConnectivityService.java

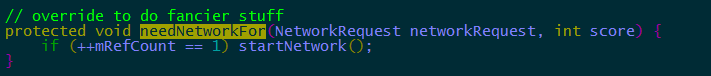


当有一个networkfactory建立连接后，就会向它发出构造函数中生成的默认网络请求。

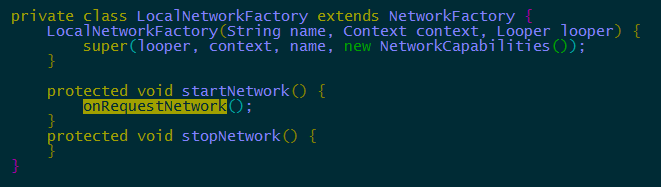
1. framework/base/core/java/android/net/NetworkFactory.java

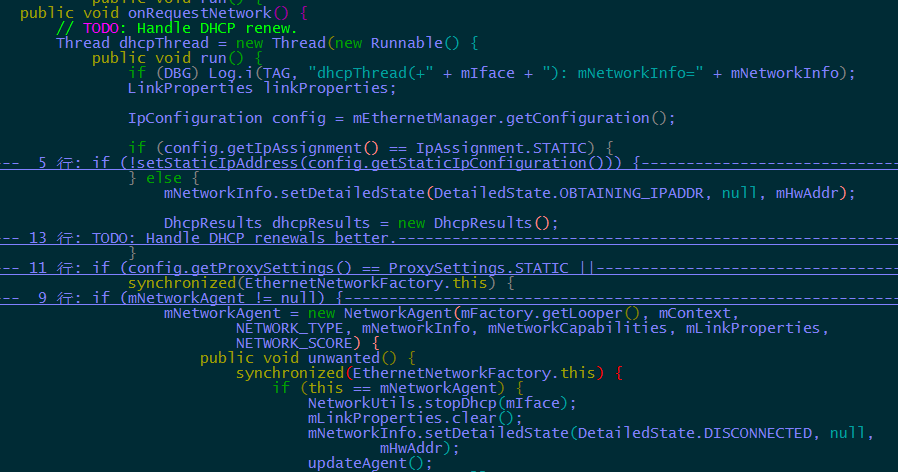






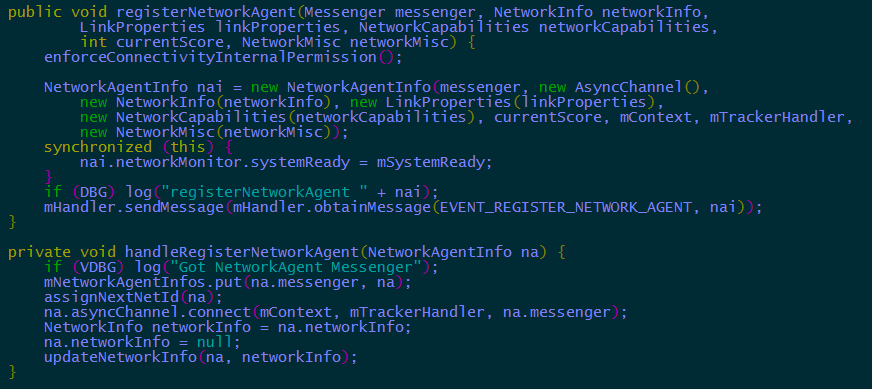
1. framework/opt/net/ethernet/java/com/android/server/ethernet/EthernetNetworkFactory. Java





DHCP成功后， 通过NetworkAgent class通知ConnectivityService网路已经连接。

1. frameworks/base/services/core/java/com/android/server/ConnectivityService.java



后续网络状态的变化就会通过NetworkAgentInfo去通知connectivityservice。

# Netd

## 概述

Netd是Android系统中专门负责网络管理和控制的后台守护进程。NetD位于Framework层和Kernel层之间，它是Android系统中网络相关信息和命令转发及处理的中枢模块。

其功能主要分为三部分：

* 设置防火墙（firewall）、网络地址转换（NAT）、带宽控制、无线网卡软接入点（SoftAP）控制、网络设备绑定（Tether）等。
* Android 系统中DNS信息的缓存和管理。
* 网络服务搜索（Net Service Discovery， NSD）功能，包括服务注册（Service Registration）、服务搜索（Service Browse）和服务解析（Service Resolve）等。

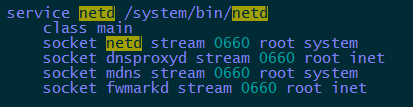
Netd的工作流程和Vold类似，其工作可以分为两部分：

* Netd 接收并处理来自Framework层中NetworkManagementService或NsdService的命令。这些命令交由Netd中对应的command对象去处理。
* Netd 接收并解析来自kernel的Uevent消息，然后再转发给Framework层中对应Service去处理。

## Netd工作流程

Netd作为后台服务进程在Andriod系统启动的init1阶段就被启动了，其在init.rc文件的配置如下：

system/core/rootdir/init.rc



Net启动时将创建四个监听socket，其名称分别为netd、dnsproxyd、mdns、fwmarkd。

### main 函数分析

源码：system/netd/server/main.cpp

int main() {

CommandListener \*cl;

NetlinkManager \*nm;

DnsProxyListener \*dpl;

MDnsSdListener \*mdnsl;

FwmarkServer\* fwmarkServer;

ALOGI("Netd 1.0 starting");

//为Netd进程屏蔽SIGPIPE信号

blockSigpipe();

1. 创建NetlinkManager

if (!(nm = NetlinkManager::Instance())) {

ALOGE("Unable to create NetlinkManager");

exit(1);

};

1. 创建CommandListener，它将创建名为“netd”的监听socket

cl = new CommandListener();

nm->setBroadcaster((SocketListener \*) cl);

if (nm->start()) {

ALOGE("Unable to start NetlinkManager (%s)", strerror(errno));

exit(1);

}

// Set local DNS mode, to prevent bionic from proxying

// back to this service, recursively.

setenv("ANDROID\_DNS\_MODE", "local", 1);

1. 创建DnsProxyListener，它将创建名为“dnsproxyd”的监听socket

dpl = new DnsProxyListener(CommandListener::sNetCtrl);

if (dpl->startListener()) {

ALOGE("Unable to start DnsProxyListener (%s)", strerror(errno));

exit(1);

}

1. 创建MDnsSdListener，它将创建名为“mdns”的监听socket

mdnsl = new MDnsSdListener();

if (mdnsl->startListener()) {

ALOGE("Unable to start MDnsSdListener (%s)", strerror(errno));

exit(1);

}

1. 创建fwmarkserver，它将创建名为“fwmarked”的监听socket

fwmarkServer = new FwmarkServer(CommandListener::sNetCtrl);

if (fwmarkServer->startListener()) {

ALOGE("Unable to start FwmarkServer (%s)", strerror(errno));

exit(1);

}

/\*

\* Now that we're up, we can respond to commands

\*/

if (cl->startListener()) {

ALOGE("Unable to start CommandListener (%s)", strerror(errno));

exit(1);

}

// Eventually we'll become the monitoring thread

while(1) {

sleep(1000);

}

ALOGI("Netd exiting");

exit(0);

}

从上面看Netd 的启动流程并不负责，主要是在main（）函数中启动四个监听socket，后边2.2.3 节的分析会看到每个socket对应一个监听线程，用于监听framework层NetworkManagement下发的command。

### NetlinkManager分析

NetlinkManager主要是负责接收并解析来自Kernel层的Uevent消息。其入口函数在NetlinkManager start（）函数。

源码：system/netd/server/NetlinkManager.cpp

int NetlinkManager::start() {

1. 创建接收NETLINK\_KOBJECT\_UEVENT 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。

if ((mUeventHandler = setupSocket(&mUeventSock, NETLINK\_KOBJECT\_UEVENT,

0xffffffff, NetlinkListener::NETLINK\_FORMAT\_ASCII)) == NULL) {

return -1;

}

1. 创建接收RTMGPR\_LINK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

if ((mRouteHandler = setupSocket(&mRouteSock, NETLINK\_ROUTE,

RTMGRP\_LINK |

RTMGRP\_IPV4\_IFADDR |

RTMGRP\_IPV6\_IFADDR |

RTMGRP\_IPV6\_ROUTE |

(1 << (RTNLGRP\_ND\_USEROPT - 1)),

NetlinkListener::NETLINK\_FORMAT\_BINARY)) == NULL) {

return -1;

}

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

if ((mQuotaHandler = setupSocket(&mQuotaSock, NETLINK\_NFLOG,

NFLOG\_QUOTA\_GROUP, NetlinkListener::NETLINK\_FORMAT\_BINARY)) == NULL) {

ALOGE("Unable to open quota2 logging socket");

// TODO: return -1 once the emulator gets a new kernel.

}

return 0;

}

NM的start（）函数主要是调用setupSocket（）向kernel注册三个用于接收UEvent时间的socket。

NetlinkHandler \*NetlinkManager::setupSocket(int \*sock, int netlinkFamily, int groups, int format) {

struct sockaddr\_nl nladdr;

int sz = 64 \* 1024;

int on = 1;

memset(&nladdr, 0, sizeof(nladdr));

nladdr.nl\_family = AF\_NETLINK;

nladdr.nl\_pid = getpid();

nladdr.nl\_groups = groups;

1. 新建socket，此处socket类型为SOCK\_DGRAM

if ((\*sock = socket(PF\_NETLINK, SOCK\_DGRAM, netlinkFamily)) < 0) {

ALOGE("Unable to create netlink socket: %s", strerror(errno));

return NULL;

}

1. 设置socket属性

if (setsockopt(\*sock, SOL\_SOCKET, SO\_RCVBUFFORCE, &sz, sizeof(sz)) < 0) {

ALOGE("Unable to set uevent socket SO\_RCVBUFFORCE option: %s", strerror(errno));

close(\*sock);

return NULL;

}

if (setsockopt(\*sock, SOL\_SOCKET, SO\_PASSCRED, &on, sizeof(on)) < 0) {

SLOGE("Unable to set uevent socket SO\_PASSCRED option: %s", strerror(errno));

close(\*sock);

return NULL;

}

1. 绑定

if (bind(\*sock, (struct sockaddr \*) &nladdr, sizeof(nladdr)) < 0) {

ALOGE("Unable to bind netlink socket: %s", strerror(errno));

close(\*sock);

return NULL;

}

1. 将socket 封装为NetlinHander，从而在socket收到消息时进行处理

NetlinkHandler \*handler = new NetlinkHandler(this, \*sock, format);

1. 启动监听

if (handler->start()) {

ALOGE("Unable to start NetlinkHandler: %s", strerror(errno));

close(\*sock);

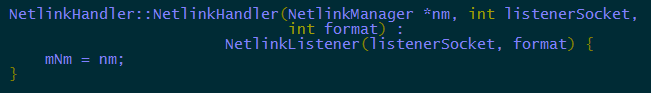
return NULL;

}

return handler;

}

源码：system/netd/server/NetlinkHandler.cpp



源码：system/core/libsysutils/src/NetlinkListener.cpp



需要注意的是，此处SocketListener 对象，listen 初始化为false。

NetlinkHandler的start()函数转调了this-> startListener()，此方法实际上是继承自SocketListener类。这个类是一个比较通用的类，很多与socket的IO复用有关的模块都会调用此类的相关方法。

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

int SocketListener::startListener(int backlog) {

1. 透过成员变量mSocketName获取socket

if (!mSocketName && mSock == -1) {

SLOGE("Failed to start unbound listener");

errno = EINVAL;

return -1;

} else if (mSocketName) {

if ((mSock = android\_get\_control\_socket(mSocketName)) < 0) {

SLOGE("Obtaining file descriptor socket '%s' failed: %s",

mSocketName, strerror(errno));

return -1;

}

SLOGV("got mSock = %d for %s", mSock, mSocketName);

}

1. 对于NetlinkHandler，此处mListen 为false

if (mListen && listen(mSock, backlog) < 0) {

SLOGE("Unable to listen on socket (%s)", strerror(errno));

return -1;

} else if (!mListen)

1. 对于NetlinkManager来说，我们是去监听Kernel 透过socket发送来的UEvent，属于客户端，

所以此处新建一个SocketClient 放入客户端集合中。

mClients->push\_back(new SocketClient(mSock, false, mUseCmdNum));

if (pipe(mCtrlPipe)) {

SLOGE("pipe failed (%s)", strerror(errno));

return -1;

}

1. 创建线程去监听socket。

if (pthread\_create(&mThread, NULL, SocketListener::threadStart, this)) {

SLOGE("pthread\_create (%s)", strerror(errno));

return -1;

}

return 0;

}

上边的监听线程开启后进入SocketListener::runListener()。

void SocketListener::runListener() {

1. 新建一个SocketClientCollection，用来存放活动fd

SocketClientCollection pendingList;

while(1) {

SocketClientCollection::iterator it;

fd\_set read\_fds;

int rc = 0;

int max = -1;

FD\_ZERO(&read\_fds);

if (mListen) {

max = mSock;

FD\_SET(mSock, &read\_fds);

}

FD\_SET(mCtrlPipe[0], &read\_fds);

if (mCtrlPipe[0] > max)

max = mCtrlPipe[0];

pthread\_mutex\_lock(&mClientsLock);

1. 遍历mClients集合

for (it = mClients->begin(); it != mClients->end(); ++it) {

// NB: calling out to an other object with mClientsLock held (safe)

int fd = (\*it)->getSocket();

FD\_SET(fd, &read\_fds);

if (fd > max) {

max = fd;

}

}

pthread\_mutex\_unlock(&mClientsLock);

SLOGV("mListen=%d, max=%d, mSocketName=%s", mListen, max, mSocketName);

if ((rc = select(max + 1, &read\_fds, NULL, NULL, NULL)) < 0) {

if (errno == EINTR)

continue;

SLOGE("select failed (%s) mListen=%d, max=%d", strerror(errno), mListen, max);

sleep(1);

continue;

} else if (!rc)

continue;

if (FD\_ISSET(mCtrlPipe[0], &read\_fds)) {

char c = CtrlPipe\_Shutdown;

TEMP\_FAILURE\_RETRY(read(mCtrlPipe[0], &c, 1));

if (c == CtrlPipe\_Shutdown) {

break;

}

continue;

}

if (mListen && FD\_ISSET(mSock, &read\_fds)) {

struct sockaddr addr;

socklen\_t alen;

int c;

do {

alen = sizeof(addr);

c = accept(mSock, &addr, &alen);

SLOGV("%s got %d from accept", mSocketName, c);

} while (c < 0 && errno == EINTR);

if (c < 0) {

SLOGE("accept failed (%s)", strerror(errno));

sleep(1);

continue;

}

pthread\_mutex\_lock(&mClientsLock);

mClients->push\_back(new SocketClient(c, true, mUseCmdNum));

pthread\_mutex\_unlock(&mClientsLock);

}

/\* Add all active clients to the pending list first \*/

1. 将所有活动的fd 放入到pendinglist，此处只有一个

pendingList.clear();

pthread\_mutex\_lock(&mClientsLock);

for (it = mClients->begin(); it != mClients->end(); ++it) {

SocketClient\* c = \*it;

// NB: calling out to an other object with mClientsLock held (safe)

int fd = c->getSocket();

if (FD\_ISSET(fd, &read\_fds)) {

pendingList.push\_back(c);

c->incRef();

}

}

pthread\_mutex\_unlock(&mClientsLock);

/\* Process the pending list, since it is owned by the thread,

\* there is no need to lock it \*/

1. 处理pendinglist，这里表示kernel 有上报消息，需要上层处理，此处处理函数为

onDataAvailable()

while (!pendingList.empty()) {

/\* Pop the first item from the list \*/

it = pendingList.begin();

SocketClient\* c = \*it;

pendingList.erase(it);

/\* Process it, if false is returned, remove from list \*/

if (!onDataAvailable(c)) {

release(c, false);

}

c->decRef();

}

}

}

当检测到这些socket有可读事件发生时，也就是内核有上层感兴趣的事件发生时。相应的onDataAvailable()被调用，这是一个虚函数。分析可知此时this的具体类型为NetlinkHandler，因此调用的是NetlinkHandler的onDataAvailable()。

源码：system/core/libsysutils/src/NetlinkListener.cpp

bool NetlinkListener::onDataAvailable(SocketClient \*cli)

{

int socket = cli->getSocket();

ssize\_t count;

uid\_t uid = -1;

1. 读取数据

count = TEMP\_FAILURE\_RETRY(uevent\_kernel\_multicast\_uid\_recv(

socket, mBuffer, sizeof(mBuffer), &uid));

if (count < 0) {

if (uid > 0)

LOG\_EVENT\_INT(65537, uid);

SLOGE("recvmsg failed (%s)", strerror(errno));

return false;

}

1. 新建一个NetlinkEvent，解码后，由NetlinkEvent 中的onEvent进行处理

NetlinkEvent \*evt = new NetlinkEvent();

if (evt->decode(mBuffer, count, mFormat)) {

onEvent(evt);

} else if (mFormat != NETLINK\_FORMAT\_BINARY) {

// Don't complain if parseBinaryNetlinkMessage returns false. That can

// just mean that the buffer contained no messages we're interested in.

SLOGE("Error decoding NetlinkEvent");

}

delete evt;

return true;

}

源码：system/netd/server/NetlinkHandler.cpp

void NetlinkHandler::onEvent(NetlinkEvent \*evt) {

const char \*subsys = evt->getSubsystem();

if (!subsys) {

ALOGW("No subsystem found in netlink event");

return;

}

if (!strcmp(subsys, "net")) {

int action = evt->getAction();

const char \*iface = evt->findParam("INTERFACE");

if (action == evt->NlActionAdd) {

notifyInterfaceAdded(iface);

} else if (action == evt->NlActionRemove) {

notifyInterfaceRemoved(iface);

} else if (action == evt->NlActionChange) {

evt->dump();

notifyInterfaceChanged("nana", true);

} else if (action == evt->NlActionLinkUp) {

notifyInterfaceLinkChanged(iface, true);

} else if (action == evt->NlActionLinkDown) {

notifyInterfaceLinkChanged(iface, false);

}

}

}

这里对不同的消息进行处理，所有的notifyXXXXX函数都会调用notify()函数。

void NetlinkHandler::notify(int code, const char \*format, ...) {

char \*msg;

va\_list args;

va\_start(args, format);

if (vasprintf(&msg, format, args) >= 0) {

mNm->getBroadcaster()->sendBroadcast(code, msg, false);

free(msg);

} else {

SLOGE("Failed to send notification: vasprintf: %s", strerror(errno));

}

va\_end(args);

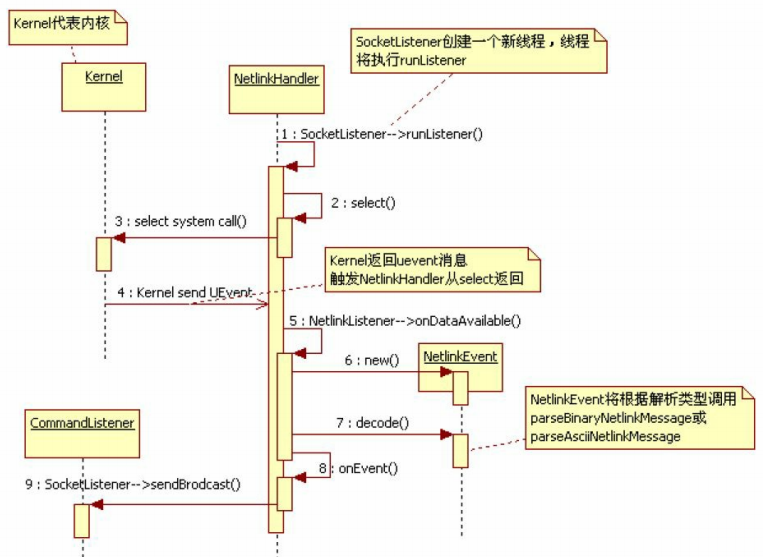
}

|  |
| --- |
|  |

mNm就是之前在main()中新建的NetworkMananger。其BroadCaster已经设置为了cl（CommandListener的一个实例）。CommandListener通过netd向NetworkManagementService发送消息。这里的消息可能有两种：一种是底层主动上报的消息，另一种是上层请求的response。

至此，NetlinkManager的流程分析结束，NetlinkManager新建了4个PF\_NETLINK类型的socket，监听内核发生的uEvent。当内核发生相应的uEvent被对应的 NetlinkManager检测到。NetlinkManager将这个uEvent转化为NetlinkEvent 通过CommandListener广播到更上层。而这里的“更上层”指的是java层。可见，底层C/C++和上层java的联系是通过socket联系在一起的。

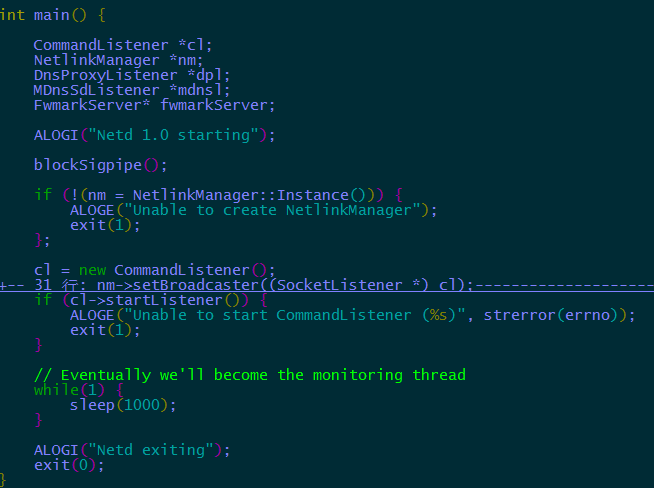
工作流程如下所示：

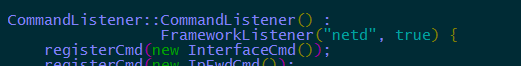


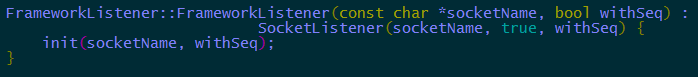
### CommandListener分析

上一节分析了NetlinkManager的流程，这一节分析CommandListener，与NetlinkManager的区别是，NetlinkManager 相当于socket client用于监听kernel 上报的消息，commanListener 则是socket server端，向位于framework 层的socket client 发送event。

回到最初的Netd main() 函数，有透过new CommandListener 实例创建net 监听socket。



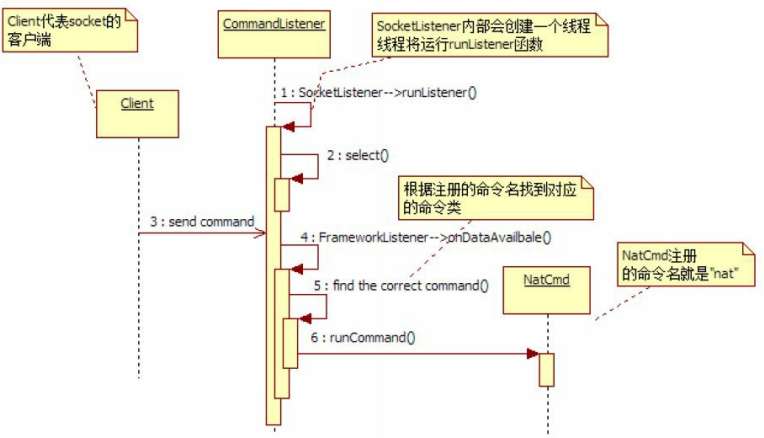




由上面可以看出CommandListener 同样继承于SockerListener，所不同的是此类的mListen被设置为true。

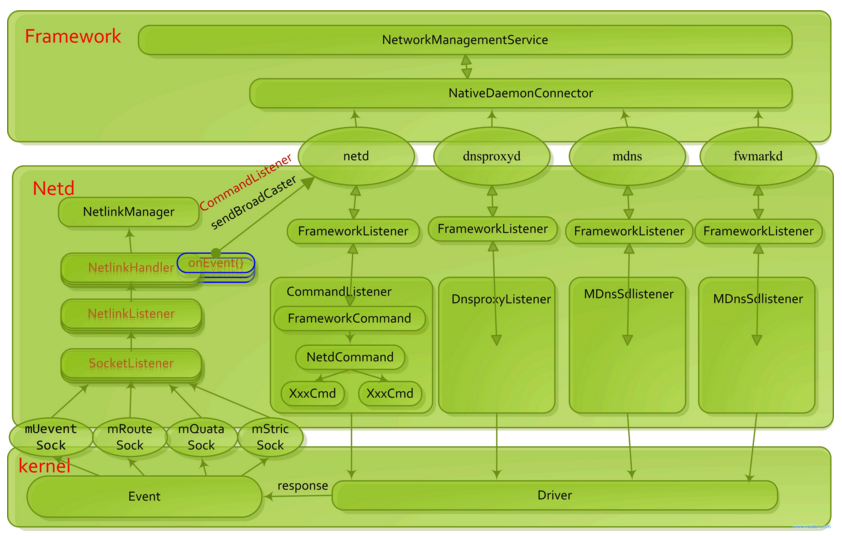
CommandListener在之前的main函数中调用startListener开启监听来自java层的连接。当上层有连接时，select返回，accpet得到一个clientSocket，之后将其封装成SocketClient添加经list，并添加进select的监听队列。（code flow 可参看上节贴出的SocketListener相关code，只是走的mListen 为true 这一路）

当java层下发命令，SocketClient的可读事件被检测到，从而做后续的处理，最后将底层处理结果response回上层，底层主动上报的消息也是通过此clientSocket上发到上层的。



除“netd”外，其他三个在init.rc中配置的socket：dnsproxyd  mdns  fwmarkd也构建了几乎一样的服务端结构。

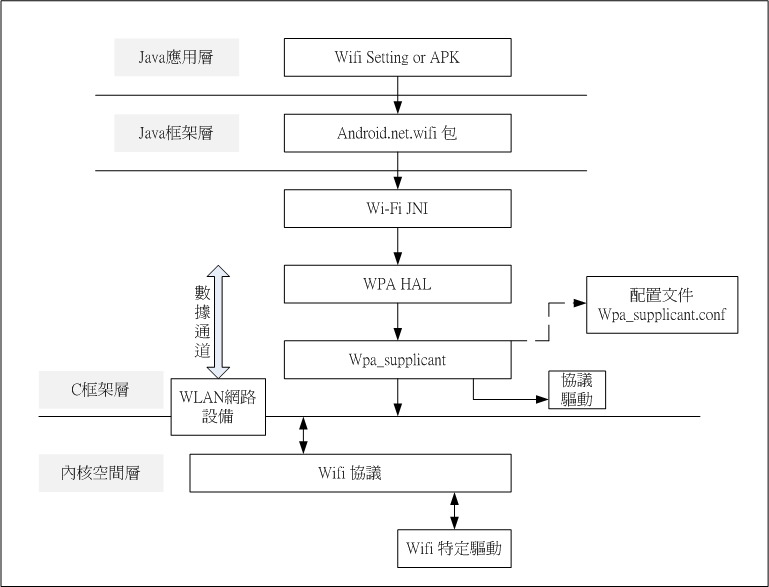
### Netd框架图



# WIFI

## WIFI Stack

### WIFI整体架构



Wifi 的整体架构：

1. wifi setting,UI interface
2. wifi framework
3. wifi jni
4. wifi hal
5. wifi supplicant
6. wifi driver
7. wifi setting
8. 源码：Packages/apps/Settings/src/com/android/settings/wifi。

-WifiSettings.java 负责显示wifi的设置页面

-WifiEnabler.java 负责wifi的开关逻辑

-WifiDialog.java 负责wifi的对话框显示

-AdvancedSettings.ajva 负责wifi高级选项的显示

-AccessPoint.java 表示AP属性的一个类

-WifiInfo.java 主要是wifi的一些配置信息，例如鉴权方式，密码，SSID etc

1. wifi framework
2. 源码：frameworks/base/wifi/java/android/net/wifi。

frameworks/opt/net/wifi/service/java/com/android/server/wifi。

1. wifiManager.java是通过ibinder调用到wifiservice.java的，wifiManager.java只是封装了一些供sdk开发用的api接口，wifiManager.java现实的主要功能都是在wifiservice.java中实现的，所以wifiservice.java是我们研究wifi的主要对象。

wifi 是有两条通信线路的，一条是执行命令的路径，一条是接口事件的路径，wifiservice.java是用来执行命令。而wifiMonitor.java就是用来接收wpa\_supplicant上报的事件。

1. wifi jni
2. 源码：frameworks/opt/net/wifi/service/jni。
3. Java层与native 层通信接口。
4. wifi hal
5. 源码：hardware/libhardware\_legacy/wifi。
6. 实现framework与wpa\_supplicant通信的，wifi hal封装了UNIX域socket，framework通过UNIX域socket与wpa\_supplicant通信。
7. wifi supplicant
8. 源码：external/wpa\_supplicant\_8
9. 该层与上层的wifi.c通过socket 通讯方式来实现控制接口，同样也是通过socket通讯方式实现事件接口，控制接口对应于wifiservice.java命令的发送，事件接口对应于wifiMonitor.java的事件的接收。
10. wifi driver

## WIFI Framework

### 框架

**Application (com.android.Settings)**

**system\_server (framework)**

**WifiService**

**WifiController/WifiStateMachine**

**WifiNavive WifiMonitor**

**WifiManager**

**wpa\_supplicant**

**cmd**

**event**

1. WifiManager
2. 源码：frameworks/base/wifi/java/android/net/wifi/WifiManager.java。
3. This class provides the primary API for managing all aspects of Wi-Fi connectivity.

It deals with several categories of items:

The list of configured networks. The list can be viewed and updated, and attributes of individual entries can be modified.

The currently active Wi-Fi network, if any. Connectivity can be established or torn down, and dynamic information about the state of the network can be queried.

Results of access point scans, containing enough information to make decisions about what access point to connect to.

It defines the names of various Intent actions that are broadcast upon any sort of change in Wi-Fi state.

1. Instance of this class

mWifiManager = (WifiManager) context.getSystemService(Context.WIFI\_SERVICE);

1. WifiService
2. 源码：

frameworks/opt/net/wifi/service/java/com/android/server/wifi/WifiService.java.

1. Implement wifi enable, scan … through WifiController/WifiStateMachine.
2. WifiService start in SystemServer
3. WifiController
4. 源码：

frameworks/opt/net/wifi/service/java/com/android/server/wifi/WifiController.java.

1. Process command from WifiService.
2. Instance of this class and start in WifiServiceImpl.java
3. WifiStateMachine
4. 源码：

frameworks/opt/net/wifi/service/java/com/android/server/wifi/WifiStateMachine.java.

1. Implement API for WifiSevice/WifiController.
2. Use WifiNative for wifi operation.
3. Initial WifiMonitor and receive events.
4. Instance of this class in WifiServiceImpl.java.
5. WifiNative
6. 源码：

frameworks/opt/net/wifi/service/java/com/android/server/wifi/WifiNative.java.

1. Interface to JNI for send cmd to wpa\_supplicant.
2. WifiMonitor
3. 源码：

frameworks/opt/net/wifi/service/java/com/android/server/wifi/WifiMonitor.java.

1. Monitor event from wpa\_supplicant.

### WifiService 启动

1. WifiService 的启动

wifi service的启动是在system server中。源码：

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

frameworks/base/services/core/java/com/android/server/SystemServiceManager.java

在SystemServer中调用了SystemServiceManager的startService()方法，启动WifiService。

private static final String WIFI\_SERVICE\_CLASS =  
            "com.android.server.wifi.WifiService";

mSystemServiceManager.startService(WIFI\_SERVICE\_CLASS);

在SystemServer中调用了SystemServiceManager的startService()方法，启动WifiService。

public <T extends SystemService> T startService(Class<T> serviceClass) {

final String name = serviceClass.getName();

Slog.i(TAG, "Starting " + name);

// Create the service.

if (!SystemService.class.isAssignableFrom(serviceClass)) {

throw new RuntimeException("Failed to create " + name

+ ": service must extend " + SystemService.class.getName());

}

final T service;

try {

1. new WifiService 的一个实例

Constructor<T> constructor = serviceClass.getConstructor(Context.class);

service = constructor.newInstance(mContext);

} catch (InstantiationException ex) {

throw new RuntimeException("Failed to create service " + name

+ ": service could not be instantiated", ex);

} catch (IllegalAccessException ex) {

throw new RuntimeException("Failed to create service " + name

+ ": service must have a public constructor with a Context argument", ex);

} catch (NoSuchMethodException ex) {

throw new RuntimeException("Failed to create service " + name

+ ": service must have a public constructor with a Context argument", ex);

} catch (InvocationTargetException ex) {

throw new RuntimeException("Failed to create service " + name

+ ": service constructor threw an exception", ex);

}

1. 注册

mServices.add(service);

1. Start

try {

service.onStart();

} catch (RuntimeException ex) {

throw new RuntimeException("Failed to start service " + name

+ ": onStart threw an exception", ex);

}

return service;

}

接下来分析WifiService，

public final class WifiService extends SystemService {

private static final String TAG = "WifiService";

final WifiServiceImpl mImpl;

public WifiService(Context context) {

super(context);

mImpl = new WifiServiceImpl(context);

}

@Override

public void onStart() {

Log.i(TAG, "Registering " + Context.WIFI\_SERVICE);

publishBinderService(Context.WIFI\_SERVICE, mImpl);

}

@Override

public void onBootPhase(int phase) {

if (phase == SystemService.PHASE\_SYSTEM\_SERVICES\_READY) {

mImpl.checkAndStartWifi();

}

}

}

new了一个WifiServiceImpl对象实例，并且调用publishBinderService()方法注册该实例对象。publishBinderService该方法是SystemService封装向ServiceManager注册的方法。

 通过上述分析，我们到真正的主角是WifiServiceImpl，我们进一步分析一下WifiServiceImpl的构造方法。

public WifiServiceImpl(Context context) {

mContext = context;

mInterfaceName = SystemProperties.get("wifi.interface", "wlan0");

mTrafficPoller = new WifiTrafficPoller(mContext, mInterfaceName);

mWifiStateMachine = new WifiStateMachine(mContext, mInterfaceName, mTrafficPoller);

mWifiStateMachine.enableRssiPolling(true);

mBatteryStats = BatteryStatsService.getService();

mAppOps = (AppOpsManager)context.getSystemService(Context.APP\_OPS\_SERVICE);

mNotificationController = new WifiNotificationController(mContext, mWifiStateMachine);

mSettingsStore = new WifiSettingsStore(mContext);

HandlerThread wifiThread = new HandlerThread("WifiService");

wifiThread.start();

mClientHandler = new ClientHandler(wifiThread.getLooper());

mWifiStateMachineHandler = new WifiStateMachineHandler(wifiThread.getLooper());

mWifiController = new WifiController(mContext, this, wifiThread.getLooper());

mBatchedScanSupported = mContext.getResources().getBoolean(

R.bool.config\_wifi\_batched\_scan\_supported);

}

这里主要创建一个WifiStateMachine的实例，并且创建一个HandlerThread来处理所有的Message。WifiStateMachine用于监控wifi的开启、关闭、断开、连接等各个状态切换。其中mClientHandler 是接收WifiManager发过来的message。（WifiManager通过binder获取到mClientHandler 向WifiService 的messagequeue 发送message）。

一个线程中只有一个 Looper 实例，一个 MessageQueue 实例，可以有多个 Handler 实例。所以WifiService 线程中的Looper 和MessageQueue 可以处理WifiStateMachine 和WifiControler 发送的message。

1. WifiService 与ConnectivityService 交互

WifiStateMachine 初始状态是mDefaultState，并在其构造函数中发

送CMD\_BOOT\_COMPLETED命令，在DefaultState 的processmessage 中：

case CMD\_BOOT\_COMPLETED:

String countryCode = mPersistedCountryCode;

if (TextUtils.isEmpty(countryCode) == false) {

Settings.Global.putString(mContext.getContentResolver(),

Settings.Global.WIFI\_COUNTRY\_CODE,

countryCode);

// It may be that the state transition that should send this info

// to the driver happened between mPersistedCountryCode getting set

// and now, so simply persisting it here would mean we have sent

// nothing to the driver. Send the cmd so it might be set now.

int sequenceNum = mCountryCodeSequence.incrementAndGet();

sendMessageAtFrontOfQueue(CMD\_SET\_COUNTRY\_CODE,

sequenceNum, 0, countryCode);

}

checkAndSetConnectivityInstance();

mNetworkFactory = new WifiNetworkFactory(getHandler().getLooper(), mContext,

NETWORKTYPE, mNetworkCapabilitiesFilter);

mNetworkFactory.setScoreFilter(60);

mCm.registerNetworkFactory(new Messenger(mNetworkFactory), NETWORKTYPE);

break;

会向ConnectivityService 注册WifiNetworkFactory，这个流程类似于前边在Ethernet的分析，通过AyncChannel建立通络，ConnectivityService向WifiNetworkFactory 发送默认的网络请求，但是因为WifiNetworkFactory 的成员函数startNetwork为空，所以没有后续的网络连接过程。

private class WifiNetworkFactory extends NetworkFactory {

public WifiNetworkFactory(Looper l, Context c, String TAG, NetworkCapabilities f) {

super(l, c, TAG, f);

}

protected void startNetwork() {

// TODO

// Enter association mode.

}

protected void stopNetwork() {

// TODO

// Stop associating.

}

}

ConnectivityService是android中监控所有网络连接的服务。wifi的状态切换是如何通知ConnectvityService，通信的关键类在于前边分析Etherent时提到的NetworkAgent，先进入NetworkAgent的构造函数。

  public NetworkAgent(Looper looper, Context context, String logTag, NetworkInfo ni,

            NetworkCapabilities nc, LinkProperties lp, int score, NetworkMisc misc) {

         super(looper);

        LOG\_TAG = logTag;

        mContext = context;

        if (ni == null || nc == null || lp == null) {

            throw new IllegalArgumentException();

        }

        if (VDBG) log("Registering NetworkAgent");

        ConnectivityManager cm = (ConnectivityManager)mContext.getSystemService(

                Context.CONNECTIVITY\_SERVICE);

        cm.registerNetworkAgent(new Messenger(this), new NetworkInfo(ni),

                new LinkProperties(lp), new NetworkCapabilities(nc), score, misc);

    }

在其构造函数中，我们可以看到，ConnectivityManager的registerNetworkAgent()，

其本质是调用了ConnectivityService的 registerNetworkAgent()的方法，在registerNetworkAgent方法中构建ConnectivitySercvice与NetworkAgent之间的通信， 我们进入registerNetworkAgent方法看看，

    public void registerNetworkAgent(Messenger messenger, NetworkInfo networkInfo,

            LinkProperties linkProperties, NetworkCapabilities networkCapabilities,

            int currentScore, NetworkMisc networkMisc) {

         enforceConnectivityInternalPermission();

        NetworkAgentInfo nai = new NetworkAgentInfo(messenger, new AsyncChannel(),

            new NetworkInfo(networkInfo), new LinkProperties(linkProperties),

            new NetworkCapabilities(networkCapabilities), currentScore, mContext, mTrackerHandler,

            new NetworkMisc(networkMisc));

           synchronized (this) {

            nai.networkMonitor.systemReady = mSystemReady;

           }     if (DBG) log("registerNetworkAgent " + nai);

       mHandler.sendMessage(mHandler.obtainMessage(EVENT\_REGISTER\_NETWORK\_AGENT, nai));

     }

该方法的主要任务是创建NetworkAgentInfo，发送移送消息给mHandler。进入对应消息处理地方，可以看到接收到该消息后的处理函数handleRegisterNetworkAgent()，代码如下：

     private void handleRegisterNetworkAgent(NetworkAgentInfo na) {

        if (VDBG) log("Got NetworkAgent Messenger");

        mNetworkAgentInfos.put(na.messenger, na);

        assignNextNetId(na);

        na.asyncChannel.connect(mContext, mTrackerHandler, na.messenger);

        NetworkInfo networkInfo = na.networkInfo;

        na.networkInfo = null;

        updateNetworkInfo(na, networkInfo);

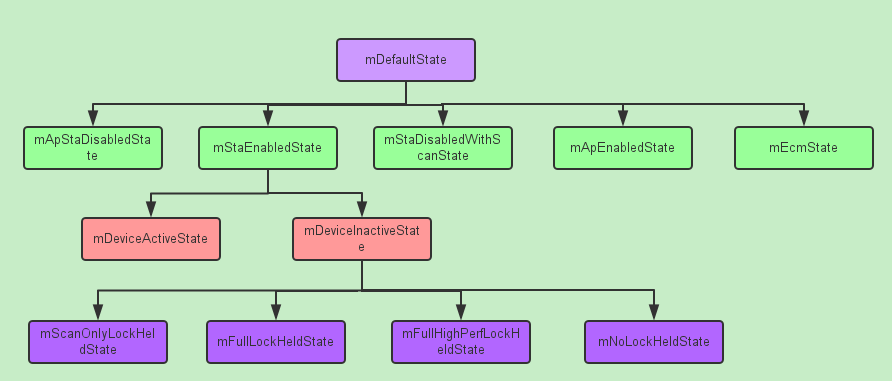
    }

 该方法主要是通过AyncChannel将ConnectivityService与NetworkAgent建立连接，发送同步消息，更新NetworkInfo。而在WifiStateMachine中进行实例化了一个WifiNetworkAgent，WifiNetworkAgent是继承自NetworkAgent，其实就是ConnectivityService与WifiStateMachine的之间建立连接，发送同步消息。

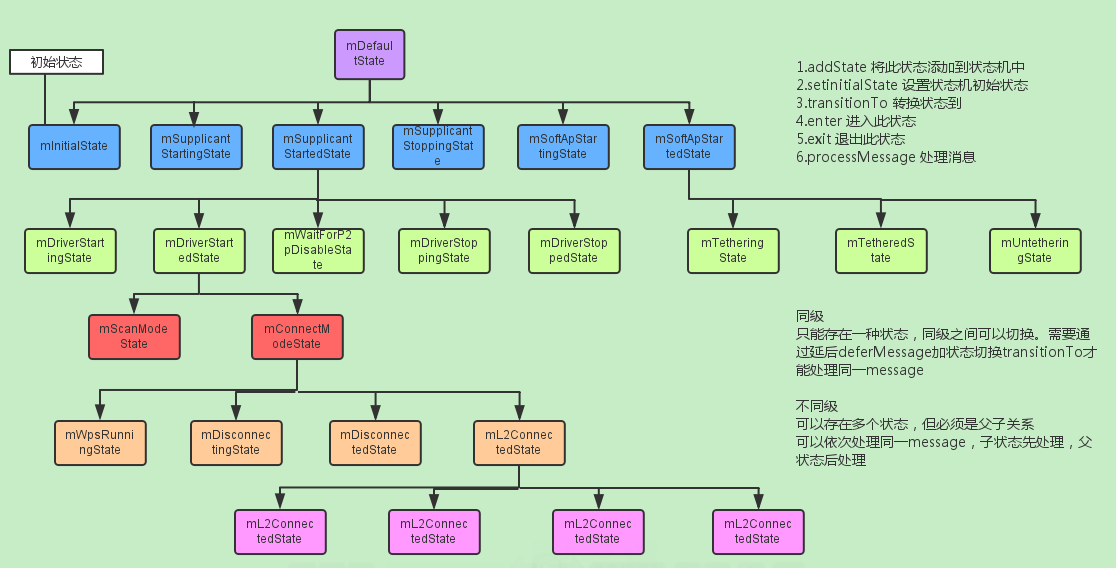
### Wifi Framework层状态机

在Wifi FrameWork 层有两个重要的状态机，分别位于WifiController 和WifiStateMachine 中。

1. WifiController 状态机



1. WifiStateMachine 状态机



WifiStateMachine 是android wifi 功能的枢纽，在WifiService/WifiController 与WifiNative/WifiMonitor 之间承上启下。

WifiStateMachine 在实例化时设置初始状态为mInitialState。然后在WifiServiceImpl 透过mWifiController.start 开启状态机。

系统启动后，WifiController和WifiStateMachine两层状态机就在不同状态间切换，以对应于wifi的开启、关闭、断开、连接等状态。

### WifiNative

从功能上来说，WifiNative是WifiService 和 wpa\_supplicant 进程对话的窗口。

Java层的WifiNative 透过Wifi JNI与Wifi HAL层对应。如下：

/\*

\* JNI registration.

\*/

static JNINativeMethod gWifiMethods[] = {

/\* name, signature, funcPtr \*/

{ "loadDriver", "()Z", (void \*)android\_net\_wifi\_loadDriver },

{ "isDriverLoaded", "()Z", (void \*)android\_net\_wifi\_isDriverLoaded },

{ "unloadDriver", "()Z", (void \*)android\_net\_wifi\_unloadDriver },

{ "startSupplicant", "(Z)Z", (void \*)android\_net\_wifi\_startSupplicant },

{ "killSupplicant", "(Z)Z", (void \*)android\_net\_wifi\_killSupplicant },

{ "connectToSupplicantNative", "()Z", (void \*)android\_net\_wifi\_connectToSupplicant },

{ "closeSupplicantConnectionNative", "()V",

(void \*)android\_net\_wifi\_closeSupplicantConnection },

{ "waitForEventNative", "()Ljava/lang/String;", (void\*)android\_net\_wifi\_waitForEvent },

{ "doBooleanCommandNative", "(Ljava/lang/String;)Z", (void\*)android\_net\_wifi\_doBooleanCommand },

{ "doIntCommandNative", "(Ljava/lang/String;)I", (void\*)android\_net\_wifi\_doIntCommand },

{ "doStringCommandNative", "(Ljava/lang/String;)Ljava/lang/String;",

(void\*) android\_net\_wifi\_doStringCommand },

{ "startHalNative", "()Z", (void\*) android\_net\_wifi\_startHal },

{ "stopHalNative", "()V", (void\*) android\_net\_wifi\_stopHal },

{ "waitForHalEventNative", "()V", (void\*) android\_net\_wifi\_waitForHalEvents },

{ "getInterfacesNative", "()I", (void\*) android\_net\_wifi\_getInterfaces},

{ "getInterfaceNameNative", "(I)Ljava/lang/String;", (void\*) android\_net\_wifi\_getInterfaceName},

{ "getScanCapabilitiesNative", "(ILcom/android/server/wifi/WifiNative$ScanCapabilities;)Z",

(void \*) android\_net\_wifi\_getScanCapabilities},

{ "startScanNative", "(IILcom/android/server/wifi/WifiNative$ScanSettings;)Z",

(void\*) android\_net\_wifi\_startScan},

{ "stopScanNative", "(II)Z", (void\*) android\_net\_wifi\_stopScan},

{ "getScanResultsNative", "(IZ)[Landroid/net/wifi/ScanResult;",

(void \*) android\_net\_wifi\_getScanResults},

{ "setHotlistNative", "(IILandroid/net/wifi/WifiScanner$HotlistSettings;)Z",

(void\*) android\_net\_wifi\_setHotlist},

{ "resetHotlistNative", "(II)Z", (void\*) android\_net\_wifi\_resetHotlist},

{ "trackSignificantWifiChangeNative", "(IILandroid/net/wifi/WifiScanner$WifiChangeSettings;)Z",

(void\*) android\_net\_wifi\_trackSignificantWifiChange},

{ "untrackSignificantWifiChangeNative", "(II)Z",

(void\*) android\_net\_wifi\_untrackSignificantWifiChange},

{ "getWifiLinkLayerStatsNative", "(I)Landroid/net/wifi/WifiLinkLayerStats;",

(void\*) android\_net\_wifi\_getLinkLayerStats},

{ "getSupportedFeatureSetNative", "(I)I",

(void\*) android\_net\_wifi\_getSupportedFeatures},

{ "requestRangeNative", "(II[Landroid/net/wifi/RttManager$RttParams;)Z",

(void\*) android\_net\_wifi\_requestRange},

{ "cancelRangeRequestNative", "(II[Landroid/net/wifi/RttManager$RttParams;)Z",

(void\*) android\_net\_wifi\_cancelRange},

{ "setScanningMacOuiNative", "(I[B)Z", (void\*) android\_net\_wifi\_setScanningMacOui},

{ "getChannelsForBandNative", "(II)[I", (void\*) android\_net\_wifi\_getValidChannels}

};

### WifiMonitor

从功能上来说，WifiMonitor 用于监控wpa\_supplicant 上报的信息，并在WifiService 进行处理。

WifiMonitor 的实例化在WifiStateMachine 的构造函数中，在InitialState 状态下，收到command CMD\_START\_SUPPLICANT 后开始监听。

public boolean processMessage(Message message) {

logStateAndMessage(message, getClass().getSimpleName());

switch (message.what) {

case CMD\_START\_SUPPLICANT:

/\* Stop a running supplicant after a runtime restart

\* Avoids issues with drivers that do not handle interface down

\* on a running supplicant properly.

\*/

mWifiMonitor.killSupplicant(mP2pSupported);

if(mWifiNative.startSupplicant(mP2pSupported)) {

setWifiState(WIFI\_STATE\_ENABLING);

if (DBG) log("Supplicant start successful");

mWifiMonitor.startMonitoring();

transitionTo(mSupplicantStartingState);

} else {

loge("Failed to start supplicant!");

}

} else {

loge("Failed to load driver");

}

break;

default:

return NOT\_HANDLED;

}

return HANDLED;

}

}

在WifiMonitor 中，看startMonitoring()函数

public synchronized void startMonitoring(String iface) {

WifiMonitor m = mIfaceMap.get(iface);

if (m == null) {

Log.e(TAG, "startMonitor called with unknown iface=" + iface);

return;

}

Log.d(TAG, "startMonitoring(" + iface + ") with mConnected = " + mConnected);

if (mConnected) {

m.mMonitoring = true;

m.mStateMachine.sendMessage(SUP\_CONNECTION\_EVENT);

} else {

if (DBG) Log.d(TAG, "connecting to supplicant");

int connectTries = 0;

while (true) {

if (mWifiNative.connectToSupplicant()) {

m.mMonitoring = true;

m.mStateMachine.sendMessage(SUP\_CONNECTION\_EVENT);

new MonitorThread(mWifiNative, this).start();

mConnected = true;

break;

}

if (connectTries++ < 5) {

try {

Thread.sleep(1000);

} catch (InterruptedException ignore) {

}

} else {

mIfaceMap.remove(iface);

m.mStateMachine.sendMessage(SUP\_DISCONNECTION\_EVENT);

Log.e(TAG, "startMonitoring(" + iface + ") failed!");

break;

}

}

}

}

这个方法主要是调用WifiNative的connectToSupplicant()方法去supplicant建立socket连接，然后给StateMachine发送SUP\_CONNECTION\_EVENT消息，并且建立一个MonitorThread实例。进入MonitorThread。

private static class MonitorThread extends Thread {

private final WifiNative mWifiNative;

private final WifiMonitorSingleton mWifiMonitorSingleton;

public MonitorThread(WifiNative wifiNative, WifiMonitorSingleton wifiMonitorSingleton) {

super("WifiMonitor");

mWifiNative = wifiNative;

mWifiMonitorSingleton = wifiMonitorSingleton;

}

public void run() {

//noinspection InfiniteLoopStatement

for (;;) {

String eventStr = mWifiNative.waitForEvent();

// Skip logging the common but mostly uninteresting scan-results event

if (DBG && eventStr.indexOf(SCAN\_RESULTS\_STR) == -1) {

Log.d(TAG, "Event [" + eventStr + "]");

}

if (mWifiMonitorSingleton.dispatchEvent(eventStr)) {

if (DBG) Log.d(TAG, "Disconnecting from the supplicant, no more events");

break;

}

}

}

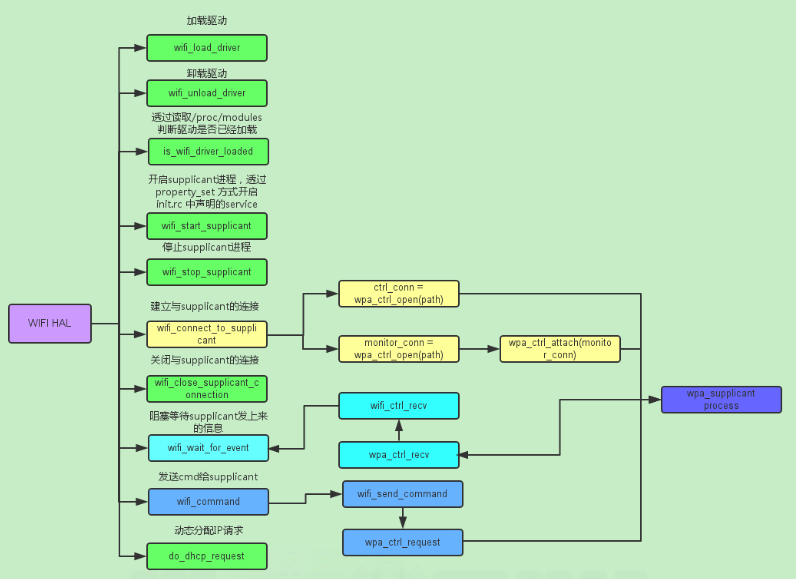
}

通过上述代码可知，开启一个线程，通过调用WifiNative的waitForEvent()方法来监听，supplicant返回的事件，并调用WifiMonitor的dispathchEvent方法将supplicant返回的事件分发给WifiSateMachine。

## WIFI HAL 和Wpa\_supplicant

### Wifi HAL

1. 源码：hardware/libhardware\_legacy/wifi。
2. 主要函数及流程图如下



1. HAL中的wifi.c 与wpa\_supplicant 进程透过socket 来实现进程间通信。wifi\_connect\_to\_supplicant 函数调用的是wpa\_ctrl.c 中的wpa\_ctrl\_open 函数来创建通信socket的，一个是ctrl interface，用于下发控制command。另一个是monitor interface，用于监控从wpa\_supplicant 发送上来的event事件。Monitor socket创建成功后，还需要发送ATTACH到wpa\_supplicant，wpa\_supplicant 在收到后，会将该客户端的信息记录下来，后续有event需要上报时发送上去。

### Wpa\_supplicant

以后补充。

## WIFI 流程分析

### Wifi 打开

1. Wifi打开流程

WifiService是继承自SystemService并且实现了一个onBootPhase()方法（在系统在boot过程调用），在该方法中去调用了WifiServiceImlp的checkAndStartWifi()方法是恢复wifi开机前的状态。即如果系统上次关机时wifi 如果是enabled 则在boot 过程中就会打开wifi 使其enable。

不过我们的分析基于系统启动时wifi 并没有被开启，而是透过Setting 来手动开启，从而分析打开wifi 的整个流程。

开启wifi是通过WifiManager的setWifiEnabled(boolean enabled)来实现，并透过binder调用WifiServiceImpl类的setWifiEnabled（）。

源码： frameworks\opt\net\wifi\service\java\com\android\server\wifi\WifiServiceImpl.java

    public synchronized boolean setWifiEnabled(boolean enable) {

        enforceChangePermission();

        long ident = Binder.clearCallingIdentity();

        try {

// 将wifi enable 状态写入到Setting.db 中，用于下次重新开机时

checkAndStartWifi()读取使用。

            if (! mSettingsStore.handleWifiToggled(enable)) {

                return true;

            }

        } finally {

            Binder.restoreCallingIdentity(ident);

        }

      mWifiController.sendMessage(CMD\_WIFI\_TOGGLED);

       return true;

    }

在上边的[3.2.3](#_Wifi_Framework层状态机) 小节中有提到wifi framework层有包含两个重要的状态机，分别是WifiStateMachine 和WifiController，上边函数的最后就是向WifiController 状态机发送command CMD\_WIFI\_TOGGLED。

源码：frameworks\opt\net\wifi\service\java\com\android\server\wifi\WifiController.java

在WifiController 的构造函数中：

    WifiController(Context context, WifiServiceImpl service, Looper looper) {

        super(TAG, looper);

        mContext = context;

        mWifiStateMachine = service.mWifiStateMachine;

        mSettingsStore = service.mSettingsStore;

        mLocks = service.mLocks;

        mAlarmManager = (AlarmManager)mContext.getSystemService(Context.ALARM\_SERVICE);

        Intent idleIntent = new Intent(ACTION\_DEVICE\_IDLE, null);

        mIdleIntent = PendingIntent.getBroadcast(mContext, IDLE\_REQUEST, idleIntent, 0);

        addState(mDefaultState);

            addState(mApStaDisabledState, mDefaultState);

            addState(mStaEnabledState, mDefaultState);

                addState(mDeviceActiveState, mStaEnabledState);

                addState(mDeviceInactiveState, mStaEnabledState);

                    addState(mScanOnlyLockHeldState, mDeviceInactiveState);

                    addState(mFullLockHeldState, mDeviceInactiveState);

                    addState(mFullHighPerfLockHeldState, mDeviceInactiveState);

                    addState(mNoLockHeldState, mDeviceInactiveState);

            addState(mStaDisabledWithScanState, mDefaultState);

            addState(mApEnabledState, mDefaultState);

            addState(mEcmState, mDefaultState);

        boolean isAirplaneModeOn = mSettingsStore.isAirplaneModeOn();

        boolean isWifiEnabled = mSettingsStore.isWifiToggleEnabled();

        boolean isScanningAlwaysAvailable = mSettingsStore.isScanAlwaysAvailable();

        log("isAirplaneModeOn = " + isAirplaneModeOn +

                ", isWifiEnabled = " + isWifiEnabled +

                ", isScanningAvailable = " + isScanningAlwaysAvailable);

        if (isScanningAlwaysAvailable) {

            setInitialState(mStaDisabledWithScanState);

        } else {

            setInitialState(mApStaDisabledState);

        }

        setLogRecSize(100);

        setLogOnlyTransitions(false);

        IntentFilter filter = new IntentFilter();

        filter.addAction(ACTION\_DEVICE\_IDLE);

        filter.addAction(WifiManager.NETWORK\_STATE\_CHANGED\_ACTION);

        mContext.registerReceiver(

                new BroadcastReceiver() {

                    @Override

                    public void onReceive(Context context, Intent intent) {

                        String action = intent.getAction();

                        if (action.equals(ACTION\_DEVICE\_IDLE)) {

                            sendMessage(CMD\_DEVICE\_IDLE);

                        } else if (action.equals(WifiManager.NETWORK\_STATE\_CHANGED\_ACTION)) {

                            mNetworkInfo = (NetworkInfo) intent.getParcelableExtra(

                                    WifiManager.EXTRA\_NETWORK\_INFO);

                        }

                    }

                },

                new IntentFilter(filter));

        initializeAndRegisterForSettingsChange(looper);

    }

通过上面的构造函数，可以看出在构造函数中构建了一个状态机，并且设置ApStaDisabledState或者StaDisabledWithScanState为初始状态，只有当用户设置了Global.WIFI\_SCAN\_ALWAYS\_AVAILABLE 为1的情况下才将初始化状态设置为StaDisabledWithScanState，但此值在

frameworks/base/packages/SettingsProvider/res/values/defaults.xml

中默认为0，所以是由ApStaDisabledState来处CMD\_WIFI\_TOGGLED信息，接下来分析ApStaDisabledState，我们只看处理CMD\_WIFI\_TOGGLED信息的部分代码。

               case CMD\_WIFI\_TOGGLED:

                case CMD\_AIRPLANE\_TOGGLED:

                    if (mSettingsStore.isWifiToggleEnabled()) {

                        if (doDeferEnable(msg)) {

                            if (mHaveDeferredEnable) {

                                //  have 2 toggles now, inc serial number an ignore both

                                mDeferredEnableSerialNumber++;

                            }

                            mHaveDeferredEnable = !mHaveDeferredEnable;

                            break;

                        }

                        if (mDeviceIdle == false) {

                            transitionTo(mDeviceActiveState);

                        } else {

                            checkLocksAndTransitionWhenDeviceIdle();

                        }

                    } else if (mSettingsStore.isScanAlwaysAvailable()) {

                        transitionTo(mStaDisabledWithScanState);

                    }

                    break;

通过上述代码，可以看到只是将状态transition到DeviceActiveState，DeviceActiveState的父state是StaEnabledState，由StateMachine的机制可知，在状态切换后，首先进入该state到root state的entry()方法。所以在wifiController从ApStaDisabledState状态transition到DeviceActiveState时，激活的状态分支为：DefaultState（entry方法为默认）、StaEnabledState、DeviceActiveState；分别进入其entry方法，其代码如下：

class StaEnabledState extends State {

        public void enter() {

            mWifiStateMachine.setSupplicantRunning(true);

        }

  }

  class DeviceActiveState extends State {

        public void enter() {

            mWifiStateMachine.setOperationalMode(WifiStateMachine.CONNECT\_MODE);

            mWifiStateMachine.setDriverStart(true);

            mWifiStateMachine.setHighPerfModeEnabled(false);

        }

   }

上述State的entry方法分别调用了，WifiSateMachine中的四个方法，setSupplicantRunning(true)、setOperationalMode(WifiStateMachine.CONNECT\_MODE)、setDriverStart(true)、setHighPerfModeEnabled(false)；分别进入WifiSateMachine的方法。

源码：

frameworks\opt\net\wifi\service\java\com\android\server\wifi\WifiStateMachine.java

   public void setSupplicantRunning(boolean enable) {

        if (enable) {

            sendMessage(CMD\_START\_SUPPLICANT);

        } else {

            sendMessage(CMD\_STOP\_SUPPLICANT);

        }

     }

   public void setOperationalMode(int mode) {

        if (DBG) log("setting operational mode to " + String.valueOf(mode));

        sendMessage(CMD\_SET\_OPERATIONAL\_MODE, mode, 0);

    }

   public void setDriverStart(boolean enable) {

        if (enable) {

            sendMessage(CMD\_START\_DRIVER);

        } else {

            sendMessage(CMD\_STOP\_DRIVER);

        }

    }

    public void setHighPerfModeEnabled(boolean enable) {

        sendMessage(CMD\_SET\_HIGH\_PERF\_MODE, enable ? 1 : 0, 0);

    }

通过代码可以看出，只是给WifiSateMachine发送四个消息，分别是CMD\_START\_SUPPLICANT、CMD\_SET\_OPERATIONAL\_MODE、CMD\_START\_DRIVER、CMD\_SET\_HIGH\_PERF\_MODE；

接下来就是看WifiSateMachine是如何处理这些消息。由于WifiStateMachine 状态机在初始化时设置初始状态为InitialState，首先看InitialState是如何处理CMD\_START\_SUPPLICANT消息：

  case CMD\_START\_SUPPLICANT:

                    if (mWifiNative.loadDriver()) {

                        try {

                            mNwService.wifiFirmwareReload(mInterfaceName, "STA");

                        } catch (Exception e) {

                            loge("Failed to reload STA firmware " + e);

                            // Continue

                        }

                        try {

                            mNwService.setInterfaceDown(mInterfaceName);

                            mNwService.clearInterfaceAddresses(mInterfaceName);

                           // Set privacy extensions

                            mNwService.setInterfaceIpv6PrivacyExtensions(mInterfaceName, true);

                            mNwService.disableIpv6(mInterfaceName);

                        } catch (RemoteException re) {

                            loge("Unable to change interface settings: " + re);

                        } catch (IllegalStateException ie) {

                            loge("Unable to change interface settings: " + ie);

                        }

                        mWifiMonitor.killSupplicant(mP2pSupported);

                        if(mWifiNative.startSupplicant(mP2pSupported)) {

                            setWifiState(WIFI\_STATE\_ENABLING);

                            if (DBG) log("Supplicant start successful");

                            mWifiMonitor.startMonitoring();

                            transitionTo(mSupplicantStartingState);

                        } else {

                            loge("Failed to start supplicant!");

                        }

                    } else {

                        loge("Failed to load driver");

                    }

                    break;

这里主要是调用WifiNative的loadDriver()和startSupplicant()，来加载wifi驱动和启动supplicant；当supplicant启动成功后，就会调用WifiMonitor的startMonitoring()方法，startMonitoring()回去和supplicant建立socket连接，并且一直监听supplicant返回的EVENT。WifiMonitor 的启动可以参看[3.2.5](#_WifiMonitor) 小节。

当startMonitoring结束后，WifiSateMachine就跳转到SupplicantStartingState，到目前MessageQueue中的消息分别

为SUP\_CONNECTION\_EVENT（connectToSupplicant中与wpa\_supllicant建立连接后发送给WifiStateMachine）、CMD\_SET\_OPERATIONAL\_MODE、CMD\_START\_DRIVER、CMD\_SET\_HIGH\_PERF\_MODE；

接着来看当WifiSateMachine处理完SUP\_CONNECTION\_EVENT后，马上就会处理CMD\_SET\_OPERATIONAL\_MODE和CMD\_START\_DRIVER消息，被SupplicantStartingState延时处理。（deferMessage会把message保存到一个list里面，直到跳转到新的状态，然后最先被defferd message会被放到StateMachine队列的头部，这些message会被当前状态最先执行）

case CMD\_START\_SUPPLICANT:

case CMD\_STOP\_SUPPLICANT:

case CMD\_START\_AP:

case CMD\_STOP\_AP:

case CMD\_START\_DRIVER:

case CMD\_STOP\_DRIVER:

case CMD\_SET\_OPERATIONAL\_MODE:

case CMD\_SET\_COUNTRY\_CODE:

case CMD\_SET\_FREQUENCY\_BAND:

case CMD\_START\_PACKET\_FILTERING:

case CMD\_STOP\_PACKET\_FILTERING:

messageHandlingStatus = MESSAGE\_HANDLING\_STATUS\_DEFERRED;

deferMessage(message);

break;

SET\_HTGH\_PERF\_MODE会被DefaultState处理。（父子State可以依次处理同一message，子state先处理，父state后处理）

接下来看SupplicantStartingState是如何处理SUP\_CONNECTION\_EVENT，其代码如下：

case WifiMonitor.SUP\_CONNECTION\_EVENT:

                    if (DBG) log("Supplicant connection established");

                    setWifiState(WIFI\_STATE\_ENABLED);

                    mSupplicantRestartCount = 0;

                    /\* Reset the supplicant state to indicate the supplicant

                     \* state is not known at this time \*/

                    mSupplicantStateTracker.sendMessage(CMD\_RESET\_SUPPLICANT\_STATE);

                    /\* Initialize data structures \*/

                    mLastBssid = null;

                    mLastNetworkId = WifiConfiguration.INVALID\_NETWORK\_ID;

                    mLastSignalLevel = -1;

                    mWifiInfo.setMacAddress(mWifiNative.getMacAddress());

                    mWifiNative.enableSaveConfig();

                    mWifiConfigStore.loadAndEnableAllNetworks();

                    if (mWifiConfigStore.enableVerboseLogging > 0) {

                        enableVerboseLogging(mWifiConfigStore.enableVerboseLogging);

                    }

                    if (mWifiConfigStore.associatedPartialScanPeriodMilli < 0) {

                        mWifiConfigStore.associatedPartialScanPeriodMilli = 0;

                    }

                    initializeWpsDetails();

                    sendSupplicantConnectionChangedBroadcast(true);

                    transitionTo(mDriverStartedState);

                    break;

该state对SUP\_CONNECTION\_EVENT处理，主要是通过调用WifiConfigStore的loadAndEnableAllNetworks()，加载并enable用户之前保存过的AP，以及对wps的一些初始化。然后transition到DriverStartedState。再看DriverStartedState的entry函数，其代码如下：

public void enter() {

            mIsRunning = true;

            mInDelayedStop = false;

            mDelayedStopCounter++;

            updateBatteryWorkSource(null);

            mWifiNative.setBluetoothCoexistenceScanMode(mBluetoothConnectionActive);

            /\* set country code \*/

            setCountryCode();

            /\* set frequency band of operation \*/

            setFrequencyBand();

            /\* initialize network state \*/

            setNetworkDetailedState(DetailedState.DISCONNECTED);

            /\* Remove any filtering on Multicast v6 at start \*/

            mWifiNative.stopFilteringMulticastV6Packets();

            /\* Reset Multicast v4 filtering state \*/

            if (mFilteringMulticastV4Packets.get()) {

                mWifiNative.startFilteringMulticastV4Packets();

            } else {

                mWifiNative.stopFilteringMulticastV4Packets();

            }

            mDhcpActive = false;

            startBatchedScan();

            if (mOperationalMode != CONNECT\_MODE) {

                mWifiNative.disconnect();

                mWifiConfigStore.disableAllNetworks();

                if (mOperationalMode == SCAN\_ONLY\_WITH\_WIFI\_OFF\_MODE) {

                    setWifiState(WIFI\_STATE\_DISABLED);

                }

                transitionTo(mScanModeState);

            } else {

                mWifiNative.status();

                transitionTo(mDisconnectedState);

            }

            if (mScreenBroadcastReceived.get() == false) {

                PowerManager powerManager = (PowerManager)mContext.getSystemService(

                        Context.POWER\_SERVICE);

                handleScreenStateChanged(powerManager.isScreenOn(),

            } else {

                mWifiNative.setSuspendOptimizations(mSuspendOptNeedsDisabled == 0

                        && mUserWantsSuspendOpt.get());

            }

            mWifiNative.setPowerSave(true);

            if (mP2pSupported) {

                if (mOperationalMode == CONNECT\_MODE) {

                    mWifiP2pChannel.sendMessage(WifiStateMachine.CMD\_ENABLE\_P2P);

                } else {

                }

            }

            final Intent intent = new Intent(WifiManager.WIFI\_SCAN\_AVAILABLE);

            intent.addFlags(Intent.FLAG\_RECEIVER\_REGISTERED\_ONLY\_BEFORE\_BOOT);

            intent.putExtra(WifiManager.EXTRA\_SCAN\_AVAILABLE, WIFI\_STATE\_ENABLED);

            mContext.sendStickyBroadcastAsUser(intent, UserHandle.ALL);

        }

从上面代码中可以看出，主要分两条分支：

1. mOperationalMode != CONNECT\_MODE ：断开网络，transition到ScanModeState状态。
2. mOperationalMode == CONNECT\_MODE：直接transition到DisconnectedState状态，如果支持p2p，向P2PService发送CMD\_ENABLE\_P2P的消息。

### Wifi 关闭

wifi关闭时，会调用WiFiMananger的setWifiEnabled(false)，接着WifiService会给WifiController发送一个CMD\_WIFI\_TOGGLED消息，通过上述wifi打开的分析可以知道，wifi打开后，transition到DeviceActiveState状态，进行关闭的时候，应该是该DeviceActiveSate来处理CMD\_WIFI\_TOGGLED消息，但是DeivceActiveState状态没有处理该消息，在其父StaEnabledState中处理该消息。

源码：

frameworks\opt\net\wifi\service\java\com\android\server\wifi\WifiController.java

 case CMD\_WIFI\_TOGGLED:

                    if (! mSettingsStore.isWifiToggleEnabled()) {

                        if (mSettingsStore.isScanAlwaysAvailable()) {

                            transitionTo(mStaDisabledWithScanState);

                        } else {

                           transitionTo(mApStaDisabledState);

                        }

                    }

                    break;

StaEnabledState中处理比较简单，只是单纯transition到ApStaDisabledState状态，我们可以看在WifiController这个状态机就回到了最初的初始状态，进入ApStaDisabledState的entry函数，看看如何对WifiSateMachine状态机做处理：

    public void enter() {

            mWifiStateMachine.setSupplicantRunning(false);

            mDisabledTimestamp = SystemClock.elapsedRealtime();

            mDeferredEnableSerialNumber++;

            mHaveDeferredEnable = false;

        }

调用了WifiSateMachine的setSupplicantRunning(false)函数，通过看setSupplicantRunning函数的代码可以知道，这个函数直接给WifiStateMachine发送一个CMD\_STOP\_SUPPLICANT消息，通过对wifi打开的分析，我们可以知道SupplicantStartedState会处理该消息，其代码如下：

      case CMD\_STOP\_SUPPLICANT:   /\* Supplicant stopped by user \*/

                    if (mP2pSupported) {

                        transitionTo(mWaitForP2pDisableState);

                    } else {

                        transitionTo(mSupplicantStoppingState);

                    }

                    break;

SupplicantStoppingState的entry方法，其代码如下：

    public void enter() {

            /\* Send any reset commands to supplicant before shutting it down \*/

            handleNetworkDisconnect();

            if (mDhcpStateMachine != null) {

                mDhcpStateMachine.doQuit();

            }

            if (DBG) log("stopping supplicant");

            mWifiMonitor.stopSupplicant();

            sendMessageDelayed(obtainMessage(CMD\_STOP\_SUPPLICANT\_FAILED,

                    ++mSupplicantStopFailureToken, 0), SUPPLICANT\_RESTART\_INTERVAL\_MSECS);

            setWifiState(WIFI\_STATE\_DISABLING);

            mSupplicantStateTracker.sendMessage(CMD\_RESET\_SUPPLICANT\_STATE);

        }

在SupplicantStoppingState的entry方法中，主要调用handleNetworkDisconnect()和WifiMonitor的stopSupplicant()；handleNetworkDisconnect方法，主要做stopDhcp和清除一些状态信息，WifiMonitor的stopSupplicant用于停掉supplicant，就是向supplicant发送TERMINATE命令；当supplicant接收到TERMINATE命令后，就会给调用者发送TERMINATING这个event，当WifiMonitor接收到这个event后，就会给WifiStateMachine发送一个SUP\_DISCONNECTION\_EVENT消息。回到SupplicantStoppingState，SupplicantStoppingState接收到这个消息后，看如何处理该消息，其代码如下：

  case WifiMonitor.SUP\_DISCONNECTION\_EVENT:

                    if (DBG) log("Supplicant connection lost");

                    handleSupplicantConnectionLoss();

                    transitionTo(mInitialState);

                    break;

进入handleSupplicantConnectionLoss函数，其代码如下：

    private void handleSupplicantConnectionLoss() {

        mWifiMonitor.killSupplicant(mP2pSupported);

        mWifiNative.closeSupplicantConnection();

        sendSupplicantConnectionChangedBroadcast(false);

        setWifiState(WIFI\_STATE\_DISABLED);

    }

 该方法调用了WifiNative的killSupplicant和closeSupplicantConnection函数，去kill supplicant进程及断开与它的socket连接，并且发送SUPPLICANT\_CONNECTION\_CHANGE\_ACTION广播。

### Wifi 扫描

wifi Scan的应用层接口是WifiManager的startScan()函数，其实质就是调用WifiServiceImpl的startScan()函数。

源码：

frameworks/opt/net/wifi/service/java/com/android/server/wifi/WifiServiceImpl.java

     public void startScan(ScanSettings settings, WorkSource workSource) {

        enforceChangePermission();

        if (settings != null) {

            mContext.enforceCallingOrSelfPermission(android.Manifest.permission.LOCATION\_HARDWARE,

                    "LocationHardware");

            settings = new ScanSettings(settings);

            if (!settings.isValid()) {

                return;

            }

        }

        if (workSource != null) {

            enforceWorkSourcePermission();

            workSource.clearNames();

        }

        mWifiStateMachine.startScan(Binder.getCallingUid(), scanRequestCounter++,

                settings, workSource);

}

从上面代码可知，其主要的是调用了WifiStateMachine的startScan函数，进入WifiStateMachine的startScan方法：

  public void startScan(int callingUid, int scanCounter,ScanSettings settings, WorkSource workSource) {

        Bundle bundle = new Bundle();

        bundle.putParcelable(CUSTOMIZED\_SCAN\_SETTING, settings);

        bundle.putParcelable(CUSTOMIZED\_SCAN\_WORKSOURCE, workSource);

        bundle.putLong(SCAN\_REQUEST\_TIME, System.currentTimeMillis());

        sendMessage(CMD\_START\_SCAN, callingUid, scanCounter, bundle);

     }

 给WifiSateMachine发送了一个CMD\_START\_SCAN消息，根据前面的分析可以了解到，这个消息是有DriverStartedState来处理，其代码如下：

  case CMD\_START\_SCAN:

                    handleScanRequest(WifiNative.SCAN\_WITHOUT\_CONNECTION\_SETUP, message);

调用用了handleScanRequest()函数，进入handleScanRequest()函数：

         private void handleScanRequest(int type, Message message) {

        // call wifi native to start the scan

        if (startScanNative(type, freqs)) {

            noteScanStart(message.arg1, workSource);

            if (freqs == null)

                mBufferedScanMsg.clear();

            messageHandlingStatus = MESSAGE\_HANDLING\_STATUS\_OK;

            return;

        }

    }

最主要的调用了startScanNative()函数，startScanNative()函数调用了WifiNative的scan函数，scan函数就是向supplicant发送一个SCAN命令，当supplicant执行完SCAN命令并找到AP后，就会给WifiMonitor发送CTRL-EVENT-SCAN-RESULTS的event，WifiMonitor就会解析到该event，并且向WifiStateMachine发送SCAN\_RESULTS\_EVENT消息。WifiStateMachine的SupplicantStartedState会处理该消息，其代码如下：

      case WifiMonitor.SCAN\_RESULTS\_EVENT:

                    closeRadioScanStats();

                    noteScanEnd();

                    setScanResults();

                    if (mIsFullScanOngoing) {

                        sendScanResultsAvailableBroadcast();

                    }

                    mIsScanOngoing = false;

                    mIsFullScanOngoing = false;

                    if (mBufferedScanMsg.size() > 0)

                        sendMessage(mBufferedScanMsg.remove());

                    break;

该部分代码的主要任务就是获取scanResult和发送SCAN\_RESULTS\_AVAILABLE\_ACTION广播。其中setScanResults()函数是从WifiNative获取到scanreslut，并将其解析为ScanResult类型。wifi的scan流程比较简单，就是向supplicant发送SCAN命令，当supplicant返回CTRL-EVENT-SCAN-RESULTS事件后，通过向supplicant发送BSS RANGE=sid 来获取scan到AP的信息，将其封装成ScanResult类型。

# Framework 层网络连接管理机制

## Overview

Android Framework层网络连接管理主要包含了ConnectivityManager、ConnectivityService、NetworkAgent、NetworkInfo四个部分，下边将一一对其进行分析。

### ConnectivityManager

顾名思义，这个类是用来管理系统的网络连接的，ConnectivityManager 作为客户端对象对应用程序提供对外SDK接口。ConnectivityManager 和 ConnectivityService 分别对应于Binder的client端和server端。

ConnectivityManager的实例化是在ContextImpl中。

源码：

frameworks/base/core/java/android/app/ContextImpl.java

registerService(CONNECTIVITY\_SERVICE, new ServiceFetcher() {

public Object createService(ContextImpl ctx) {

IBinder b = ServiceManager.getService(CONNECTIVITY\_SERVICE);

return new ConnectivityManager(IConnectivityManager.Stub.asInterface(b));

}});

创建ConnectivityManager时传递了一个name为CONNECTIVITY\_SERVICE的服务对象，这个对象是在SystemServer启动的ConnectivityService。

再看ConnectivityManager 的构造函数，

源码：

frameworks/base/core/java/android/net/ConnectivityManager.java

public ConnectivityManager(IConnectivityManager service) {

mService = checkNotNull(service, "missing IConnectivityManager");

}

构造方法里面只是将参数ConnectivityService传递给mService对象。后续应用程序可以利用这个本地客户端访问远程的服务。

### ConnectivityService

ConnectivityService 是网络连接管理的核心，它是以service 的形式在系统初始化时启动的。

源码：

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

try {

Slog.i(TAG, "Connectivity Service");

connectivity = new ConnectivityService(

context, networkManagement, networkStats, networkPolicy);

ServiceManager.addService(Context.CONNECTIVITY\_SERVICE, connectivity);

networkStats.bindConnectivityManager(connectivity);

networkPolicy.bindConnectivityManager(connectivity);

} catch (Throwable e) {

reportWtf("starting Connectivity Service", e);

}

### NetworkAgent

源码：

frameworks/base/core/java/android/net/NetworkAgent.java

在这个类的描述中有如下字段：

\* A Utility class for handling for communicating between bearer-specific

\* code and ConnectivityService.

即此类是一个用于在某个具体网络连接（wifi/Ethernet/BT…….）与ConnectivtiyService之间进行通信的工具类。

从NetworkAgent 类的定义来看，它是Handler 类的子类，并且是一个抽象类，需要在子类中再被实例化。

public abstract class NetworkAgent extends Handler {

其构造函数为：

>> public NetworkAgent(Looper looper, Context context, String logTag, NetworkInfo ni,

>> NetworkCapabilities nc, LinkProperties lp, int score, NetworkMisc misc) {

super(looper);

LOG\_TAG = logTag;

mContext = context;

if (ni == null || nc == null || lp == null) {

throw new IllegalArgumentException();

}

if (VDBG) log("Registering NetworkAgent");

>> ConnectivityManager cm = (ConnectivityManager)mContext.getSystemService(

>> Context.CONNECTIVITY\_SERVICE);

>> cm.registerNetworkAgent(new Messenger(this), new NetworkInfo(ni),

>> new LinkProperties(lp), new NetworkCapabilities(nc), score, misc);

}

主要工作为获取ConnectivityManager对象，并通过registerNetworkAgent方式注册当前的NetworkAgent。

NetwoekAgent 和ConnectivityService 之间的通信链路的建立将在后边的小节进行分析。

### NetworkInfo

源码：

frameworks/base/core/java/android/net/NetworkInfo.java

此类的作用是用于描述一个网络连接的状态，包括网络类型、网络连接状态、网络是否可用等等。

| +getDetailedState()

| +getExtraInfo()

| +getReason()

| +getState()

| +getSubtype()

| +getSubtypeName()

| +getType()

| +getTypeName()

| +isAvailable()

| +isConnected()

| +isConnectedOrConnecting()

| +isConnectedToProvisioningNetwork()

| +isFailover()

| +isRoaming()

NetworkInfo 提供的方法可以获取到一个网络连接的基本信息。

应用程序可以通过ConnectivityManager的getNetworkInfo()方法获取到该对象，并通过该对象查询当前的网络状态，比如可以这样获取当前是否有网络连接。

当网络状态发生变化时，比如当前网络被断开，也需要由具体的网络连接承担者（Wifi/Ethernet/BT……）来更新NetworkInfo，并透过NetWorkAgent 更新到ConnectivityService中。

## NetworkAgent注册过程

前面的[4.1.3](#_NetworkAgent) 小节中有提到，在NetworkAgent 的构造函数中主要工作是向ConnectivityService 注册自己，

>> public NetworkAgent(Looper looper, Context context, String logTag, NetworkInfo ni,

>> NetworkCapabilities nc, LinkProperties lp, int score, NetworkMisc misc) {

super(looper);

LOG\_TAG = logTag;

mContext = context;

if (ni == null || nc == null || lp == null) {

throw new IllegalArgumentException();

}

if (VDBG) log("Registering NetworkAgent");

>> ConnectivityManager cm = (ConnectivityManager)mContext.getSystemService(

>> Context.CONNECTIVITY\_SERVICE);

>> cm.registerNetworkAgent(new Messenger(this), new NetworkInfo(ni),

>> new LinkProperties(lp), new NetworkCapabilities(nc), score, misc);

}

在调用注册方法时传递了六个参数，其中有三个比较重要的参数，都是从构造方法的参数中获取并重新new出来的，其分别是：  
1、new Messenger(this)  
2、new NetworkInfo(ni)  
3、new NetworkCapabilities(nc)

需要注意的是，这里new出来的三个对象的来源，都应该是NetworkAgent子类被初始化时传递给构造方法的。

ConnectivityManager 的registerNetworkAgent方法最终会调用ConnectivityService 中的方法。

public void registerNetworkAgent(Messenger messenger, NetworkInfo networkInfo,

LinkProperties linkProperties, NetworkCapabilities networkCapabilities,

int currentScore, NetworkMisc networkMisc) {

enforceConnectivityInternalPermission();

NetworkAgentInfo nai = new NetworkAgentInfo(messenger, new AsyncChannel(),

new NetworkInfo(networkInfo), new LinkProperties(linkProperties),

new NetworkCapabilities(networkCapabilities), currentScore, mContext, mTrackerHandler,

new NetworkMisc(networkMisc));

synchronized (this) {

nai.networkMonitor.systemReady = mSystemReady;

}

if (DBG) log("registerNetworkAgent " + nai);

mHandler.sendMessage(mHandler.obtainMessage(EVENT\_REGISTER\_NETWORK\_AGENT, nai));

}

在ConnectivityService的registerNetworkAgent中做了两件事情：  
1、创建NetworkAgentInfo对象；  
2、向Handler发送EVENT\_REGISTER\_NETWORK\_AGENT消息；

其中创建NetworkAgentInfo时，传递了九个参数，我们只关注其中三个，分别是：  
1、messenger ----这个参数是registerNetworkAgent的参数，从NetworkAgent传递过来  
2、new AsyncChannel() ----这是现在创建的新对象  
3、 new NetworkCapabilities(NetworkCapabilities) ----这也是用NetworkAgent传递过来的参数创建的对象  
  
 先看ConnectivityService对EVENT\_REGISTER\_NETWORK\_AGENT的处理：

private class InternalHandler extends Handler {

            public void handleMessage(Message msg) {

                NetworkInfo info;

                switch (msg.what) {

                    case EVENT\_REGISTER\_NETWORK\_AGENT: {

                       handleRegisterNetworkAgent((NetworkAgentInfo)msg.obj);

                       break;

                    }

                }

            }

        }

private void handleRegisterNetworkAgent(NetworkAgentInfo na) {

            mNetworkAgentInfos.put(na.messenger, na);

            assignNextNetId(na);

            //向NetworkAgentInfo的asyncChannel对象发起连接请求

            na.asyncChannel.connect(mContext, mTrackerHandler, na.messenger);

            NetworkInfo networkInfo = na.networkInfo;

            na.networkInfo = null;

            updateNetworkInfo(na, networkInfo);

        }

ConnectivityService向NetworkAgentInfo的asyncChannel对象发起connect请求，并且该AsyncChannel的srcHandler是mTrackerHandler，而dstMessenger对象是NetworkAgentInfo的messenger，那么这里的NetworkAgentInfo是哪里来的呢？其实不难看出，这里的NetworkAgentInfo就是在registerNetworkAgent()中创建的，而dstMessenger自然就是NetworkAgent调用registerNetworkAgent()时传递进来的。接下来，ConnectivityService将会利用获取到的NetworkAgent对象创建AsyncChannel双向通道。

前边提到的NetwoekAgent 的构造函数中，

>> cm.registerNetworkAgent(new Messenger(this), new NetworkInfo(ni),

>> new LinkProperties(lp), new NetworkCapabilities(nc), score, misc);

在向ConnectivityService注册时传递的参数是new Messenger(this), 所以AsyncChannel的dstMessenger其实就是NetwoekAgent，或者说是NetworkAgent 子类对象所在thread 的messager。

从AsyncChannel的[机制](#_AsyncChannel分析)我们知道，当利用其发起connect请求时，其将会触发单向连接过程，此时srcHandler(也就是mTrackerHandler)将会收到CMD\_CHANNEL\_HALF\_CONNECTED的消息。

private class NetworkStateTrackerHandler extends Handler {

            public void handleMessage(Message msg) {

                NetworkInfo info;

                switch (msg.what) {

                    case AsyncChannel.CMD\_CHANNEL\_HALF\_CONNECTED: {

                        handleAsyncChannelHalfConnect(msg);

                        break;

                    }

                }

            }

        }

private void handleAsyncChannelHalfConnect(Message msg) {

AsyncChannel ac = (AsyncChannel) msg.obj;

if (mNetworkFactoryInfos.containsKey(msg.replyTo)) {

if (msg.arg1 == AsyncChannel.STATUS\_SUCCESSFUL) {

if (VDBG) log("NetworkFactory connected");

// A network factory has connected. Send it all current NetworkRequests.

for (NetworkRequestInfo nri : mNetworkRequests.values()) {

if (nri.isRequest == false) continue;

NetworkAgentInfo nai = mNetworkForRequestId.get(nri.request.requestId);

ac.sendMessage(android.net.NetworkFactory.CMD\_REQUEST\_NETWORK,

(nai != null ? nai.getCurrentScore() : 0), 0, nri.request);

}

} else {

loge("Error connecting NetworkFactory");

mNetworkFactoryInfos.remove(msg.obj);

}

} else if (mNetworkAgentInfos.containsKey(msg.replyTo)) {

if (msg.arg1 == AsyncChannel.STATUS\_SUCCESSFUL) {

if (VDBG) log("NetworkAgent connected");

// A network agent has requested a connection. Establish the connection.

mNetworkAgentInfos.get(msg.replyTo).asyncChannel.

sendMessage(AsyncChannel.CMD\_CHANNEL\_FULL\_CONNECTION);

} else {

loge("Error connecting NetworkAgent");

NetworkAgentInfo nai = mNetworkAgentInfos.remove(msg.replyTo);

if (nai != null) {

synchronized (mNetworkForNetId) {

mNetworkForNetId.remove(nai.network.netId);

}

// Just in case.

mLegacyTypeTracker.remove(nai);

}

}

}

}

在这个函数中我们可以看到，我们前边章节在分析wifi和etherent时提到的，NetworkFactory在向ConnectivityService 注册时，建立的是单向AsyncChannel，而NetworkAgent 在注册时建立的是双向通道。

ConnectivityService通过AsyncChannel向dstMessenger发送了CMD\_CHANNEL\_FULL\_CONNECTION的消息，从AsyncChannel的机制我们想到，这个消息应该是在向NetworkAgent申请双向通道。

NetworkAgent 中对CMD\_CHANNEL\_FULL\_CONNECTION消息的处理如下：

public void handleMessage(Message msg) {

            switch (msg.what) {

                case AsyncChannel.CMD\_CHANNEL\_FULL\_CONNECTION: {

                   if (mAsyncChannel != null) {

                   } else {

                       //创建WifiNetworkAgent中的AsyncChannel对象

                       AsyncChannel ac = new AsyncChannel();

                       //完成双向通道连接动作

                       ac.connected(null, this, msg.replyTo);

                       ac.replyToMessage(msg, AsyncChannel.CMD\_CHANNEL\_FULLY\_CONNECTED, AsyncChannel.STATUS\_SUCCESSFUL);

                       synchronized (mPreConnectedQueue) {

                           mAsyncChannel = ac;

                           for (Message m : mPreConnectedQueue) {

                               ac.sendMessage(m);

                           }

                           mPreConnectedQueue.clear();

                       }

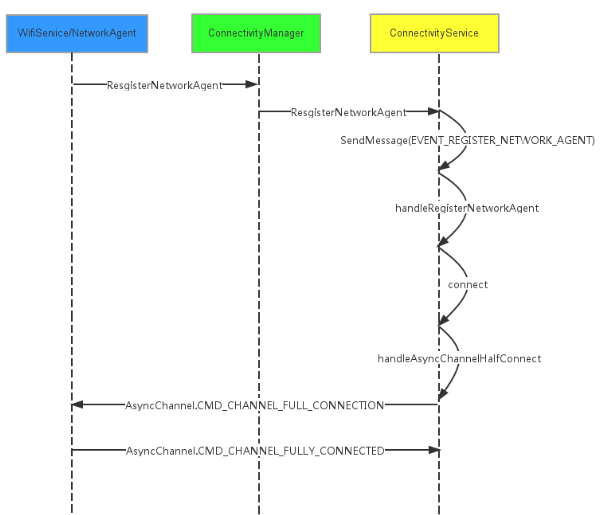
                   }

                   break;

               }

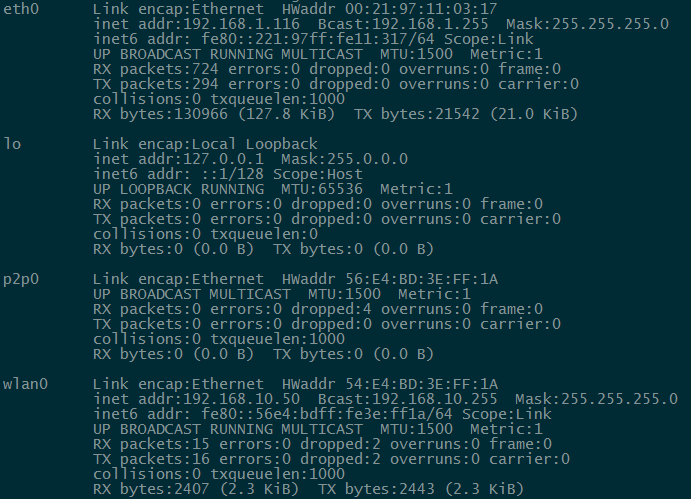
            }

        }



## 网络的切换

我们的TV支持Ethernet和WIFI同时连接网络，只是由于WIFI 的优先级较高，在系统使用网络时NetworkRequest选择的是WIFI 通道。我们设定一个问题，即此时断掉WIFI连接，系统是怎么切换到Ethernet网络连接的。这种场景类似与手机在使用移动数据上网时，如果进入WIFI环境，手机将会自动连上WIFI使用数据，而当WIFI失去覆盖或者关闭WIFI时，手机又会自动连上移动数据，那么这个机制是如何实现的呢？



当WIFI连接断开时，会调用WifiStateMachine 中的handleNetworkDisconnect，在其中将DISCONNECTED的状态传递给setNetworkDetailedState()。

private void handleNetworkDisconnect() {

if (DBG) log("handleNetworkDisconnect: Stopping DHCP and clearing IP"

+ " stack:" + Thread.currentThread().getStackTrace()[2].getMethodName()

+" - "+ Thread.currentThread().getStackTrace()[3].getMethodName()

+" - "+ Thread.currentThread().getStackTrace()[4].getMethodName()

+" - "+ Thread.currentThread().getStackTrace()[5].getMethodName());

clearCurrentConfigBSSID("handleNetworkDisconnect");

stopDhcp();

try {

mNwService.clearInterfaceAddresses(mInterfaceName);

mNwService.disableIpv6(mInterfaceName);

} catch (Exception e) {

loge("Failed to clear addresses or disable ipv6" + e);

}

/\* Reset data structures \*/

mBadLinkspeedcount = 0;

mWifiInfo.reset();

linkDebouncing = false;

/\* Reset roaming parameters \*/

mAutoRoaming = WifiAutoJoinController.AUTO\_JOIN\_IDLE;

fullBandConnectedTimeIntervalMilli = 20 \* 1000; // Start scans at 20 seconds interval

setNetworkDetailedState(DetailedState.DISCONNECTED);

if (mNetworkAgent != null) {

mNetworkAgent.sendNetworkInfo(mNetworkInfo);

mNetworkAgent = null;

}

mWifiConfigStore.updateStatus(mLastNetworkId, DetailedState.DISCONNECTED);

/\* Clear network properties \*/

clearLinkProperties();

/\* Cend event to CM & network change broadcast \*/

sendNetworkStateChangeBroadcast(mLastBssid);

/\* Cancel auto roam requests \*/

autoRoamSetBSSID(mLastNetworkId, "any");

mLastBssid= null;

registerDisconnected();

mLastNetworkId = WifiConfiguration.INVALID\_NETWORK\_ID;

}

再看setNetworkDetailedState，其中会更新当前WIFI的NetworkInfo对象，并将其传递给WifiNetworkAgent。

private boolean setNetworkDetailedState(NetworkInfo.DetailedState state) {

boolean hidden = false;

if (linkDebouncing || isRoaming()) {

hidden = true;

}

if (DBG) {

log("setDetailed state, old ="

+ mNetworkInfo.getDetailedState() + " and new state=" + state

+ " hidden=" + hidden);

}

if (mNetworkInfo.getExtraInfo() != null && mWifiInfo.getSSID() != null) {

// Always indicate that SSID has changed

if (!mNetworkInfo.getExtraInfo().equals(mWifiInfo.getSSID())) {

if (DBG) {

log("setDetailed state send new extra info" + mWifiInfo.getSSID());

}

mNetworkInfo.setExtraInfo(mWifiInfo.getSSID());

sendNetworkStateChangeBroadcast(null);

}

}

if (hidden == true) {

return false;

}

if (state != mNetworkInfo.getDetailedState()) {

mNetworkInfo.setDetailedState(state, null, mWifiInfo.getSSID());

if (mNetworkAgent != null) {

mNetworkAgent.sendNetworkInfo(mNetworkInfo);

}

sendNetworkStateChangeBroadcast(null);

return true;

}

return false;

}

再看NetworkAgent 中是如何将NetworkInfo 传递给ConnectivityService的。

>> public void sendNetworkInfo(NetworkInfo networkInfo) {

>> queueOrSendMessage(EVENT\_NETWORK\_INFO\_CHANGED, new NetworkInfo(networkInfo));

}

private void queueOrSendMessage(int what, Object obj) {

synchronized (mPreConnectedQueue) {

if (mAsyncChannel != null) {

mAsyncChannel.sendMessage(what, obj);

} else {

>> Message msg = Message.obtain();

msg.what = what;

msg.obj = obj;

mPreConnectedQueue.add(msg);

}

}

}

NetworkAgent将最新的NetworkInfo作为一个Object放入一个EVENT\_NETWORK\_INFO\_CHANGED的消息中，然后通过AsyncChannel发送出去。

AsyncChannel 有两种工作模式：

1、单向模式，在该模式下，Client端只能向Server端发起请求，由Server端给出回应。  
2、双向模式，在该模式下，Client端和Server端同时连接上AsyncChannel，Client端可以向Server端发送请求，Server端也可以向Client端发送请求。

具体到ConnectivityService 中，ConnectivityService的角色是Client端，而负责具体网络连接的WifiService 和EthernetService 是Server端。

在[4.2.2](#_NetworkAgent与ConnectivityService通信链) 小节中，分析NetworkAgent向ConnectivityService 的注册过程中，有提到NetworkAgent 与ConnectivityService 之间创建的是双向通道，而NetworkFactory 与ConnectivityService 创建的是单向通道，所以：

1、NetworkFactory 单向通道，Client端ConnectivityService只能向Server端WifiService或者EthernetService中的NetworkFactory发起请求，发送CMD\_REQUEST\_NETWORK。  
2、NetworkAgent 双向通道，Server端WifiService或者EthernetService中的NetworkAgent可以向Client端的ConnectivityService 更新NetworkInfo。

当初是ConnectivityService与WifiNetworkAgent创建了双向的AsyncChannel通道，那么此时的消息当然就会发送给ConnectivityService了，并且该消息将会在其NetworkStateTrackerHandler中被处理。

       private class NetworkStateTrackerHandler extends Handler {

            public void handleMessage(Message msg) {

                NetworkInfo info;

                switch (msg.what) {

                    case NetworkAgent.EVENT\_NETWORK\_INFO\_CHANGED: {

                          //拿到消息中最新的NetworkInfo信息

                          NetworkAgentInfo nai = mNetworkAgentInfos.get(msg.replyTo);

                          if (nai == null) {

                              loge("EVENT\_NETWORK\_INFO\_CHANGED from unknown NetworkAgent");

                              break;

                          }

                          info = (NetworkInfo) msg.obj;

                          //通过updateNetworkInfo来进行更新

                          updateNetworkInfo(nai, info);

                          break;

                      }

                }

            }

        }

private void updateNetworkInfo(NetworkAgentInfo networkAgent, NetworkInfo newInfo) {

            NetworkInfo.State state = newInfo.getState();

            NetworkInfo oldInfo = null;

            synchronized (networkAgent) {

                oldInfo = networkAgent.networkInfo;

                //将最新的networkInfo更新到ConnectivityService

                networkAgent.networkInfo = newInfo;

            }

            if (state == NetworkInfo.State.CONNECTED && !networkAgent.created) {

            } else if (state == NetworkInfo.State.DISCONNECTED || state == NetworkInfo.State.SUSPENDED) {

                //断开WIFI的NetworkAgent中的AsyncChannel

                networkAgent.asyncChannel.disconnect();

            }

        }

在updateNetworkInfo()中，ConnectivityService将最新的NetworkInfo保存在networkAgent中，等待其他应用来获取。然后就向NetworkAgent的AsyncChannel发起disconnect()的请求，该请求将会在ConnectivityService中收到CMD\_CHANNEL\_DISCONNECTED的回应：

private class NetworkStateTrackerHandler extends Handler {

            public void handleMessage(Message msg) {

                NetworkInfo info;

                switch (msg.what) {

                    case AsyncChannel.CMD\_CHANNEL\_DISCONNECTED: {

                        handleAsyncChannelDisconnected(msg);

                        break;

                    }

                }

            }

        }

private void handleAsyncChannelDisconnected(Message msg) {

            NetworkAgentInfo nai = mNetworkAgentInfos.get(msg.replyTo);

            if (nai != null) {

                final ArrayList<NetworkAgentInfo> toActivate = new ArrayList<NetworkAgentInfo>();

                for (int i = 0; i < nai.networkRequests.size(); i++) {

                    //当前网络断开，寻找可替代的网络连接

                    NetworkRequest request = nai.networkRequests.valueAt(i);

                    NetworkAgentInfo currentNetwork = mNetworkForRequestId.get(request.requestId);

                    if (currentNetwork != null && currentNetwork.network.netId == nai.network.netId) {

                        mNetworkForRequestId.remove(request.requestId);

                        sendUpdatedScoreToFactories(request, 0);

                        NetworkAgentInfo alternative = null;

                    }

                }

            }

        }

在这里，将会取出当前断开的网络所能处理的NetworkRequest，然后在当前所有向ConnectivityService注册的列表中查找可替代的连接，并通过sendUpdatedScoreToFactories()向其发起连接请求：

private void sendUpdatedScoreToFactories(NetworkRequest networkRequest, int score) {

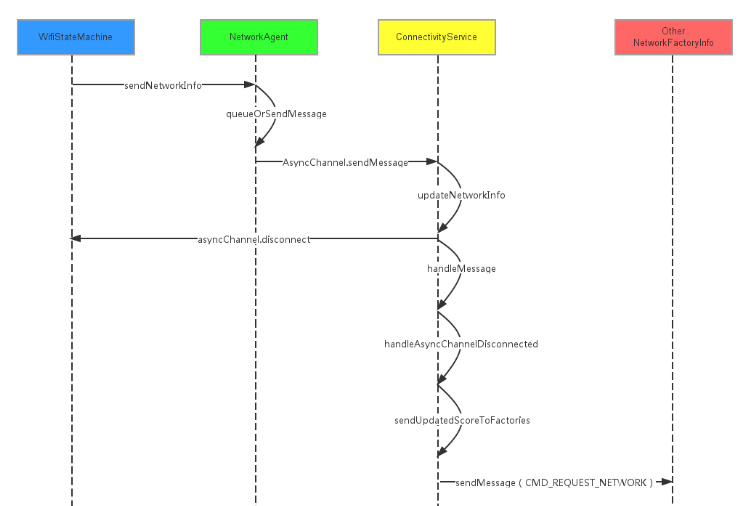
            for (NetworkFactoryInfo nfi : mNetworkFactoryInfos.values()) {

                nfi.asyncChannel.sendMessage(android.net.NetworkFactory.CMD\_REQUEST\_NETWORK, score, 0, networkRequest);

            }

        }

到这里将会向新的连接的NetworkFactoryInfo对象发起CMD\_REQUEST\_NETWORK的请求，之后相应的连接对象就可以建立连接了。至此，从一个WIFI连接转换到Ethernet连接。



# 网络评分机制分析

## Overview

在分析[4.3](#_网络的切换)网络切换有提到，我们的系统是允许Ethernet 和WIFI 同时联网的，但是对于其他应用来说，同一时刻只能选择其中的一种网络进行数据传输。设定系统启动时网络连接类型是Etherrnet，然后打开WIFI连接AP。那么，此时应用程序如果有网络请求，是继续使用原本的Ethernet连接，还是新建的WIFI连接呢？

ConnectivityService中会通过一个评分机制来实现不同连接方式的选择。每一种网络类型在初始化时，都会向ConnectivityService标明自己网络的分值(比如WIFI 60 Etherent 70)，当有更高分数的网络就绪时，就将当前分值低的连接断开。

这套评分机制主要由NetworkFactory、NetworkAgent、NetworkMonitor三个角色参与，下面会一一分析。

## NetworkFactory

前边章节在分析Ethernet和WIFI时，也有提到NetworkFactory，当时没有详细分析，只是有个概念，即为了方便ConnectivityService 对各种网络类型进行管理，利用设计模式中的工厂模式，创建了一个NetworkFactory工厂类，每种网络类型都需要创建一个NetworkFactory的子类对象，并利用该对象向ConnectivityService注册自己。这个注册过程是包含此网络类型分值的。

源码：

frameworks/base/core/java/android/net/NetworkFactory.java

贴一段NetworkFactory类文件的注释内容如下：

\* A NetworkFactory is an entity that creates NetworkAgent objects.

\* The bearers register with ConnectivityService using {@link #register} and

\* their factory will start receiving scored NetworkRequests.

\* Pass a network request to the bearer. If the bearer believes it can

\* satisfy the request it should connect to the network and create a

\* NetworkAgent. Once the NetworkAgent is fully functional it will

\* register itself with ConnectivityService using registerNetworkAgent.

\* If the bearer cannot immediately satisfy the request (no network,

\* user disabled the radio, lower-scored network) it should remember

\* any NetworkRequests it may be able to satisfy in the future. It may

\* disregard any that it will never be able to service, for example

\* those requiring a different bearer.

开宗明义，注释中写到该类是用来创建NetworkAgent对象的。每一种网络类型在初始化的时候都会实例化自己的NetworkFactory对象，并利用NetworkFactory 中的成员函数register（）向ConnectivityService注册自己。

当ConnectivityService 向这种网络类型发送网络请求时，则用成员函数handleAddRequest处理网络请求。

当网络请求完成后，就会在实例化一个NetworkAgent对象，并向ConnectivityService注册，建立通信链路，传递网络状态信息（NetworkInfo）。

在前边分析Ethernet和WIFI的初始化过程中，已经分析了NetworkFactory的初始化和注册过程，下边重点分析下NetWorkFactory 注册过程中的评分机制。

private void evalRequest(NetworkRequestInfo n) {

            if (n.requested == false && n.score < mScore && n.request.networkCapabilities.satisfiedByNetworkCapabilities( mCapabilityFilter) && acceptRequest(n.request, n.score)) {

                needNetworkFor(n.request, n.score);

                n.requested = true;

            } else if (n.requested == true && (n.score > mScore || n.request.networkCapabilities.satisfiedByNetworkCapabilities( mCapabilityFilter) == false || acceptRequest(n.request, n.score) == false)) {

                releaseNetworkFor(n.request);

                n.requested = false;

            }

        }

NetworkFactory在收到CMD\_REQUEST\_NETWORK后，最终会进入到evalRequest（）函数。

该逻辑就是整个网络评分系统最关键的地方，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)，那么就说明，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。  
 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。  
 对于初始化流程来说，由于NetworkRequestInfo是刚才在handleAddRequest新创建的，所以其requested状态必然为false，而且我们前面提到，ConnectivityService发送CMD\_REQUEST\_NETWORK时携带的分值参数为0，并且对于数据网络来说，其mScore=70，因此此时的判定状态将会是：n.requested=false AND n.score < mScore。也就是说，对于Ethernet网络环境初始化过程来说，将会满足第一个if判断，进入needNetworkFor流程，也就是触发数据网络的建立。

[20160421-19:02:09]I/EthernetServiceImpl( 2691): Starting Ethernet service

[20160421-19:02:09]D/Ethernet( 2691): Registering NetworkFactory

[20160421-19:02:09]D/ConnectivityService( 2691): Got NetworkFactory Messenger for Ethernet

[20160421-19:02:09]D/Ethernet( 2691): got request NetworkRequest [ id=1, legacyType=-1, [ Capabilities: INTERNET&NOT\_RESTRICTED&TRUSTED&NOT\_VPN] ] with score 0

[20160421-19:02:09]D/EthernetNetworkFactory( 2691): Started tracking interface eth0

[20160421-19:02:22]D/ConnectivityService( 2691): Got NetworkFactory Messenger for WIFI

[20160421-19:02:22]D/WIFI ( 2691): got request NetworkRequest [ id=1, legacyType=-1, [ Capabilities: INTERNET&NOT\_RESTRICTED&TRUSTED&NOT\_VPN] ] with score 30

由于WifiService启动较慢，当WifiService 中的NetworkFactory收到网络请求时，获取到的Ethernet NetworkAgentinfo 中的score 为30。Ethernet 网络类型的default score为70，此处为什么会变为了30后边会分析到。

## NetworkAgent

NetworkFactory 在被创建时，每种网络类型都会设置一个初始的分值。当有新的网络请求更新过来时，将其所携带的分值与当前NetworkFactory的分值比较，如果低于当前NetworkFactory，将会触发NetworkFactory内部建立连接的流程，反之将会触发NetworkFactory释放连接的流程。

那么网络请求中的分值来自于哪里呢？其实就是来自于当前在使用的网络连接的NetworkAgent。  
 在网络连接过程中，根据网络情况，各个网络的NetworkFactory可以修改当前的NetworkAgent分值，此操作将会把最新的分值广播到系统内所有的NetworkFactory中，从而引发上面描述的评分过程，而更新NetworkAgent分值有两种方式，分别是：直接更新分值、通过sendNetworkInfo更新。下面我们分别来看这两种过程。

### 直接更新分值过程

NetworkAgent 类中有提供一个成员函数sendNetworkScore，所有的NetworkAgent对象，都可以通过这个方法来直接更新当前的NetworkAgent分值。

/\*\*

\* Called by the bearer code when it has a new score for this network.

\*/

public void sendNetworkScore(int score) {

if (score < 0) {

throw new IllegalArgumentException("Score must be >= 0");

}

queueOrSendMessage(EVENT\_NETWORK\_SCORE\_CHANGED, new Integer(score));

}

继续：

private void queueOrSendMessage(int what, Object obj) {

            synchronized (mPreConnectedQueue) {

                if (mAsyncChannel != null) {

                    mAsyncChannel.sendMessage(what, obj);

                } else {

                    Message msg = Message.obtain();

                    msg.what = what;

                    msg.obj = obj;

                    mPreConnectedQueue.add(msg);

                }

            }

        }

我们看到，NetworkAgent将会把最新的分值封装到EVENT\_NETWORK\_SCORE\_CHANGED消息中，通过AsyncChannel发送出去，而这里的AsyncChannel通道就是当初NetworkAgent向ConnectivityService注册时由ConnectivityService发起的双向通道，最终该消息将会被ConnectivityService中的mTrackerHandler处理。

        private class NetworkStateTrackerHandler extends Handler {

            public void handleMessage(Message msg) {

                NetworkInfo info;

                switch (msg.what) {

                    case NetworkAgent.EVENT\_NETWORK\_SCORE\_CHANGED: {

                       NetworkAgentInfo nai = mNetworkAgentInfos.get(msg.replyTo);

                       if (nai == null) {

                           break;

                       }

                       Integer score = (Integer) msg.obj;

                       //更新ConnectivityService中的NetworkAgent分值

                       if (score != null) updateNetworkScore(nai, score.intValue());

                       break;

                   }

                }

            }

        }

在ConnectivityService 内部透过函数updateNetworkScore 来更新NetworkAgent 的分值。

private void updateNetworkScore(NetworkAgentInfo nai, int score) {

            if (score < 0) {

                score = 0;

            }

            final int oldScore = nai.getCurrentScore();

            //1.将分值更新到NetworkAgentInfo中

            nai.setCurrentScore(score);

            //2.触发评分机制

            if (nai.created) rematchAllNetworksAndRequests(nai, oldScore);

            //3.将当前最新分值更新到每个NetworkFactory中

            sendUpdatedScoreToFactories(nai);

        }

这个函数首先会把收到的收到的最新的分值更新到NetworkAgent 中，最后通过sendUpdatedScoreToFactories方法将此评分送达到每个NetworkFactory中。

该函数中还有一个重要的步骤rematchAllNetworksAndRequests。

private void rematchNetworkAndRequests(NetworkAgentInfo newNetwork, boolean nascent) {

            for (NetworkRequestInfo nri : mNetworkRequests.values()) {

                //寻找当前NetworkRequest所使用的NetworkAgent

                if (nri.request.networkCapabilities.satisfiedByNetworkCapabilities( newNetwork.networkCapabilities)) {

                    //如果当前NetworkRequest使用的NetworkAgent分数低于新的NetworkAgent分数，将会用新的NetworkAgent替代

                    if (currentNetwork == null || currentNetwork.getCurrentScore() < newNetwork.getCurrentScore()) {

                        //找到新的NetworkAgent替代方案

                        if (currentNetwork != null) {

                            if (DBG) log("   accepting network in place of " + currentNetwork.name());

                            currentNetwork.networkRequests.remove(nri.request.requestId);

                            currentNetwork.networkLingered.add(nri.request);

                            affectedNetworks.add(currentNetwork);

                        } else {

                            if (DBG) log("   accepting network in place of null");

                        }

                        mNetworkForRequestId.put(nri.request.requestId, newNetwork);

//向新的NetworkAgent 发送NetworkRequest

                        newNetwork.addRequest(nri.request);

                        if (nri.isRequest && nri.request.legacyType != TYPE\_NONE) {

                            mLegacyTypeTracker.add(nri.request.legacyType, newNetwork);

                        }

                        keep = true;

                        //将分数更新到各个NetworkFactory中

                        sendUpdatedScoreToFactories(nri.request, newNetwork.getCurrentScore());

                    }

                }

            }

        }

主要逻辑就是当前的NetworkRequest分配给合适的NetworkAgent，最后把当前最合适的NetworkAgent分数同步给所有的NetworkFactory。

private void sendUpdatedScoreToFactories(NetworkRequest networkRequest, int score) {

            for (NetworkFactoryInfo nfi : mNetworkFactoryInfos.values()) {

                //将分值发送到各个NetworkFactory

                nfi.asyncChannel.sendMessage(android.net.NetworkFactory.CMD\_REQUEST\_NETWORK, score, 0, networkRequest);

            }

        }

将最新的NetworkAgent分值通过AsyncChannel通道发送给各个NetworkFactory，由NetworkFactory来决定自己的网络是否需要建立连接或者释放连接。

### 间接更新分值过程

间接更新过程是指，当前网络的的其他状态发生改变时，可以通过sendNetworkInfo方法通知到ConnectivityService，此过程最终也会触发评分机制。

比如[4.3](#_网络的切换) 节中提到的当WIFI断开时，此时WIFI网络不可用，会透过setNetworkDetailedState来通知ConnectivityService。最终ConnectivityService 在处理断开信息时，会寻找其它可以使用的网络，并进行切换。切换的过程就是将当前网络分值直接设为0，然后把他的0分值通知到各个NetworkFactory中，由NetworkFactory判断是否需要开启自己的网络。

无论是直接更新NetworkAgent中的分数，还是更新NetworkAgent的状态，最终都会触发NetworkFactory中的评分机制。

private void handleAsyncChannelDisconnected(Message msg) {

            NetworkAgentInfo nai = mNetworkAgentInfos.get(msg.replyTo);

            if (nai != null) {

                //删掉当前NetworkAgent对象

                mNetworkAgentInfos.remove(msg.replyTo);

                final ArrayList<NetworkAgentInfo> toActivate = new ArrayList<NetworkAgentInfo>();

                for (int i = 0; i < nai.networkRequests.size(); i++) {

                    NetworkRequest request = nai.networkRequests.valueAt(i);

                    NetworkAgentInfo currentNetwork = mNetworkForRequestId.get(request.requestId);

                    if (currentNetwork != null && currentNetwork.network.netId == nai.network.netId) {

                        mNetworkForRequestId.remove(request.requestId);

                        //将0分更新到各个NetworkFactory中

                        sendUpdatedScoreToFactories(request, 0);

                    }

                }

            }

        }

## NetworkMonitor

当某个NetworkFactory连接上网络时，就会创建NetworkAgent对象，并将其注册到ConnectivityService。在注册过程中，ConnectivityService将会利用NetworkAgent传递过来的NetworkInfo、Messenger、分值等信息创建NetworkAgentInfo对象。该对象的主要作用也就是保存各个向ConnectivityService注册的NetworkAgent，以便于查询或修改某个NetworkAgent对象的相关信息。

我们来看下NetworkAgentInfo的构造函数。

源码：

frameworks/base/services/core/java/com/android/server/connectivity/NetworkAgentInfo.java

>> public NetworkAgentInfo(Messenger messenger, AsyncChannel ac, NetworkInfo info,

>> LinkProperties lp, NetworkCapabilities nc, int score, Context context, Handler handler,

>> NetworkMisc misc) {

this.messenger = messenger;

asyncChannel = ac;

network = null;

networkInfo = info;

linkProperties = lp;

networkCapabilities = nc;

currentScore = score;

>> networkMonitor = new NetworkMonitor(context, handler, this);

networkMisc = misc;

created = false;

validated = false;

}

从NetworkAgentInfo的构造方法中我们看到他创建了一个NetworkMonitor对象，将会创建一个网络监听器NetworkMonitor。

### NetworkMonitor 状态机

源码：

frameworks/base/services/core/java/com/android/server/connectivity/NetworkMonitor.java

        public NetworkMonitor(Context context, Handler handler, NetworkAgentInfo networkAgentInfo) {

            super(TAG + networkAgentInfo.name());

            //初始化各种成员变量

            mContext = context;

            mConnectivityServiceHandler = handler;

            mNetworkAgentInfo = networkAgentInfo;

            mTelephonyManager = (TelephonyManager) context.getSystemService(Context.TELEPHONY\_SERVICE);

            mWifiManager = (WifiManager) context.getSystemService(Context.WIFI\_SERVICE);

            mAlarmManager = (AlarmManager) context.getSystemService(Context.ALARM\_SERVICE);

            //初始化状态机

            addState(mDefaultState);

            addState(mOfflineState, mDefaultState);

            addState(mValidatedState, mDefaultState);

            addState(mEvaluatingState, mDefaultState);

            addState(mUserPromptedState, mDefaultState);

            addState(mCaptivePortalState, mDefaultState);

            addState(mLingeringState, mDefaultState);

            //初始状态机为DefaultState

            setInitialState(mDefaultState);

            mServer = Settings.Global.getString(mContext.getContentResolver(), Settings.Global.CAPTIVE\_PORTAL\_SERVER);

            if (mServer == null) mServer = DEFAULT\_SERVER;

            mLingerDelayMs = SystemProperties.getInt(LINGER\_DELAY\_PROPERTY, DEFAULT\_LINGER\_DELAY\_MS);

            mReevaluateDelayMs = SystemProperties.getInt(REEVALUATE\_DELAY\_PROPERTY, DEFAULT\_REEVALUATE\_DELAY\_MS);

            mIsCaptivePortalCheckEnabled = Settings.Global.getInt(mContext.getContentResolver(), Settings.Global.CAPTIVE\_PORTAL\_DETECTION\_ENABLED, 1) == 1;

            //开始状态机

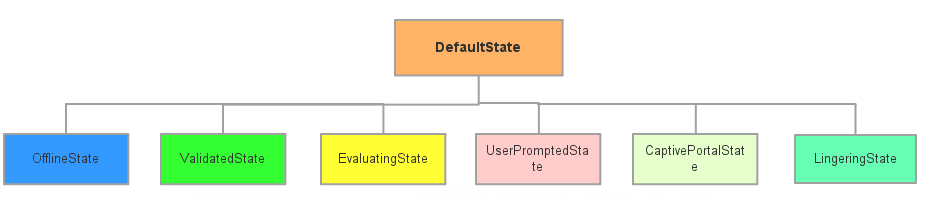
            start();

        }

从NetworkMonitor的构造函数中我们可以看到两个信息：

1、该类内部有七个状态机；

2、初始化的状态机是DefaultState。



DefaultState  
 这是默认的状态机，也是其他所有状态机的父状态，主要处理网络连接的主要状态变化，包括连接、断开、测试、延时等模式。  
EvaluatingState  
 验证状态，网络连上时，将会进入该状态，然后会Ping网络，判断当前网络有效性，并决定下一步是进入ValidatedState还是OfflineState或者UserPromptedState。  
OfflineState  
 脱机状态，当Ping网络时，始终没有任何回应时，就会进入该状态。  
ValidatedState  
 验证通过的状态，当Ping通网络时，说明当前的网络是通畅的，将会进入该状态。  
UserPromptedState  
 验证失败状态，当Ping网络时，网络给出了重定向异常，比如接入中国移动时会跳入移动的帐号认证页面，需要用户进行网络登录后才可以继续上网。此时一般需要在界面上提示用户

### NetworkMonitor 更新分值

NetworkMonitor对象被实例化后，其中的状态机默认状态为DefaultState。并停留在此状态等待处理Event。

当某个网络成功连接后新建NetworkAgent并向ConnectivityService注册，并最终在updateNetworkInfo 中将此网络的状态更新为Connected。

        private void updateNetworkInfo(NetworkAgentInfo networkAgent, NetworkInfo newInfo) {

            if (state == NetworkInfo.State.CONNECTED && !networkAgent.created) {

                networkAgent.created = true;

                updateLinkProperties(networkAgent, null);

                notifyNetworkCallbacks(networkAgent, ConnectivityManager.CALLBACK\_PRECHECK);

                networkAgent.networkMonitor.sendMessage(NetworkMonitor.CMD\_NETWORK\_CONNECTED);

                // Consider network even though it is not yet validated.

                rematchNetworkAndRequests(networkAgent, false);

            } else if (state == NetworkInfo.State.DISCONNECTED || state == NetworkInfo.State.SUSPENDED) {

            }

        }

在更新状态的同时，会向NetworkMonitor 发送CMD\_NETWORK\_CONNECTED。此时NetworkMonitor 中的状态机是在DefaultState，所以其处理函数为：

        private class DefaultState extends State {

            public boolean processMessage(Message message) {

                switch (message.what) {

                    case CMD\_NETWORK\_CONNECTED:

                        transitionTo(mEvaluatingState);

                        return HANDLED;

                }

            }

        }

状态机将会进入EvaluatingState状态

private class EvaluatingState extends State {

            private int mRetries;

            public void enter() {

                mRetries = 0;

                sendMessage(CMD\_REEVALUATE, ++mReevaluateToken, 0);

                if (mUidResponsibleForReeval != INVALID\_UID) {

                    TrafficStats.setThreadStatsUid(mUidResponsibleForReeval);

                    mUidResponsibleForReeval = INVALID\_UID;

                }

            }

            public boolean processMessage(Message message) {

                switch (message.what) {

                    case CMD\_REEVALUATE:

                        if (message.arg1 != mReevaluateToken)

                            return HANDLED;

                        if (mNetworkAgentInfo.isVPN()) {

                            transitionTo(mValidatedState);

                            return HANDLED;

                        }

                        //Ping网络

                        int httpResponseCode = isCaptivePortal();

                        //根据网络回应来进入不同状态

                        if (httpResponseCode == 204) {

                            transitionTo(mValidatedState);

                        } else if (httpResponseCode >= 200 && httpResponseCode <= 399) {

                            transitionTo(mUserPromptedState);

                        } else if (++mRetries > MAX\_RETRIES) {

                            transitionTo(mOfflineState);

                        } else if (mReevaluateDelayMs >= 0) {

                            Message msg = obtainMessage(CMD\_REEVALUATE, ++mReevaluateToken, 0);

                            sendMessageDelayed(msg, mReevaluateDelayMs);

                        }

                        return HANDLED;

                    case CMD\_FORCE\_REEVALUATION:

                        return HANDLED;

                    default:

                        return NOT\_HANDLED;

                }

            }

        }

从这个状态机的初始化过程我们发现，进入该状态时，将会发送一个CMD\_REEVALUATE的消息，然后在该状态机内部收到该消息时，就会通过isCaptivePortal方法来Ping网络。

private int isCaptivePortal() {

            if (!mIsCaptivePortalCheckEnabled) return 204;

            HttpURLConnection urlConnection = null;

            int httpResponseCode = 599;

            try {

                //准备连接的uri

     URL url = new URL("http", mServer, "/generate\_204");

                urlConnection = (HttpURLConnection) mNetworkAgentInfo.network.openConnection(url);

                urlConnection.setInstanceFollowRedirects(false);

                urlConnection.setConnectTimeout(SOCKET\_TIMEOUT\_MS);

                urlConnection.setReadTimeout(SOCKET\_TIMEOUT\_MS);

                urlConnection.setUseCaches(false);

                //发起连接

                long requestTimestamp = SystemClock.elapsedRealtime();

                urlConnection.getInputStream();

                long responseTimestamp = SystemClock.elapsedRealtime();

                //获取服务器回应

                httpResponseCode = urlConnection.getResponseCode();

                //拿到回应

                if (httpResponseCode == 200 && urlConnection.getContentLength() == 0) {

                    httpResponseCode = 204;

                }

                sendNetworkConditionsBroadcast(true /\* response received \*/, httpResponseCode == 204, requestTimestamp, responseTimestamp);

            } catch (IOException e) {

                if (httpResponseCode == 599) {

                }

            } finally {

                if (urlConnection != null) {

                    urlConnection.disconnect();

                }

            }

            return httpResponseCode;

        }

从这里我们看到，其向一个url网址进行Ping操作，而这个url的具体内容为：<http://clients3.google.com/generate_204>。很不幸，是一个google的服务器。

当isCaptivePortal结束之后，EvaluatingState就会对ping的结果进行分类：  
1、httpResponseCode=204，就说明当前网络是可用的，将会进入ValidatedState状态。  
2、httpResponseCode >= 200同时httpResponseCode <= 399，说明当前网络需要进行重定向，当认证完成后才可以有效访问网络。  
3、 如果没有任何回应，将会通过sendMessageDelayed方式延时mReevaluateDelayMs(默 认为5秒)之后再次发送CMD\_REEVALUATE消息，触发循环Ping过程，直到尝试次数超过MAX\_RETRIES(10次)之后，进入OfflineState状态，表示当前网络无法使用。  
 那么当当前网络可用或者不可用时，对NetworkAgent评分有什么影响呢？我们来看当该Ping通过后，进入ValidatedState状态时将会触发什么动作：

首先是ping通过时，NetworkMonitor 中的状态机会转到ValidatedState。

>> private class ValidatedState extends State {

@Override

public void enter() {

if (DBG) log("Validated");

mConnectivityServiceHandler.sendMessage(obtainMessage(EVENT\_NETWORK\_TESTED,

NETWORK\_TEST\_RESULT\_VALID, 0, mNetworkAgentInfo));

}

@Override

>> public boolean processMessage(Message message) {

if (DBG) log(getName() + message.toString());

switch (message.what) {

case CMD\_NETWORK\_CONNECTED:

transitionTo(mValidatedState);

return HANDLED;

default:

return NOT\_HANDLED;

}

}

}

ValidatedState的入口函数中，向ConnectivityService 发送一个EVENT\_NETWORK\_TESTED message， 并携带了当前网络的NetworkAgentInfo对象和测试通过(NETWORK\_TEST\_RESULT\_VALID)的参数。

当Ping不通过时，发生重定向时会进入UserPromptedState状态机。

private class UserPromptedState extends State {

            private static final String ACTION\_SIGN\_IN\_REQUESTED = "android.net.netmon.sign\_in\_requested";

            private CustomIntentReceiver mUserRespondedBroadcastReceiver;

            @Override

            public void enter() {

                mConnectivityServiceHandler.sendMessage(obtainMessage(EVENT\_NETWORK\_TESTED, NETWORK\_TEST\_RESULT\_INVALID, 0, mNetworkAgentInfo));

                mUserRespondedBroadcastReceiver = new CustomIntentReceiver(ACTION\_SIGN\_IN\_REQUESTED, ++mUserPromptedToken, CMD\_USER\_WANTS\_SIGN\_IN);

                Message message = obtainMessage(EVENT\_PROVISIONING\_NOTIFICATION, 1,

                        mNetworkAgentInfo.network.netId,

                        mUserRespondedBroadcastReceiver.getPendingIntent());

                mConnectivityServiceHandler.sendMessage(message);

            }

            @Override

            public boolean processMessage(Message message) {

                switch (message.what) {

                    case CMD\_USER\_WANTS\_SIGN\_IN:

                        if (message.arg1 != mUserPromptedToken)

                            return HANDLED;

                        transitionTo(mCaptivePortalState);

                        return HANDLED;

                    default:

                        return NOT\_HANDLED;

                }

            }

这个状态机与ValidatedState类似，也是在进入时向ConnectivityService发送了同样的EVENT\_NETWORK\_TESTED消息，并携带当前的NetworkAgentInfo对象，不同的是，该消息中携带了一个测试不通过的参数(NETWORK\_TEST\_RESULT\_INVALID)。

ConnectivityService在收到EVENT\_NETWORK\_TESTED message 后的处理是：

        private class NetworkStateTrackerHandler extends Handler {

            public void handleMessage(Message msg) {

                NetworkInfo info;

                switch (msg.what) {

                    case NetworkMonitor.EVENT\_NETWORK\_TESTED: {

                      NetworkAgentInfo nai = (NetworkAgentInfo)msg.obj;

                      if (isLiveNetworkAgent(nai, "EVENT\_NETWORK\_VALIDATED")) {

                          boolean valid = (msg.arg1 == NetworkMonitor.NETWORK\_TEST\_RESULT\_VALID);

                          if (valid) {

                              //验证通过

                              final boolean previouslyValidated = nai.validated;

                              final int previousScore = nai.getCurrentScore();

                              //标记NetworkAgentInfo的validated状态为true

                              nai.validated = true;

                              rematchNetworkAndRequests(nai, !previouslyValidated);

                              if (nai.getCurrentScore() != previousScore) {

                                  //将最新分数更新到其他NetworkFactory

                                  sendUpdatedScoreToFactories(nai);

                              }

                          }

                          updateInetCondition(nai, valid);

                          //通知NetworkAgent

                          nai.asyncChannel.sendMessage( android.net.NetworkAgent.CMD\_REPORT\_NETWORK\_STATUS, (valid ? NetworkAgent.VALID\_NETWORK : NetworkAgent.INVALID\_NETWORK), 0, null);

                      }

                      break;

                  }

                }

            }

        }

在这里我们看到，通过与不通过的差距就在于是否将NetworkAgentInfo的validated标志位设置为true。  
 当NetworkAgent注册到ConnectivityService时，将会创建该NetworkAgent的NetworkAgentInfo，此时该标志位的默认状态是false。因为在NetworkAgent注册时说明网络已经连接，但是并不清楚该网络是否可用，因为还没经过检测。

下边来看validated标志位对NetworkAgentInfo中分值计算的影响。

public int getCurrentScore() {

            //默认是NetworkAgent带过来的

            int score = currentScore;

            //如果没有通过Ping的检测，那么该网络就要被扣掉UNVALIDATED\_SCORE\_PENALTY=40分

            if (!validated) score -= UNVALIDATED\_SCORE\_PENALTY;

            //有效性保护

            if (score < 0) score = 0;

            //如果该网络是用户特意指定的，分值就是 EXPLICITLY\_SELECTED\_NETWORK\_SCORE=100

            if (networkMisc.explicitlySelected) score = EXPLICITLY\_SELECTED\_NETWORK\_SCORE;

            return score;

从这里我们终于找到了Ping网络对评分的影响：无论是通过rematchNetworkAndRequests重新匹配最佳NetworkFactory，还是通过sendUpdatedScoreToFactories广播最新网络的分值，都是需要从NetworkAgentInfo中通过getCurrentScore获取最新分值的，而如果当前网络没有经过有效性检测，那么对其他所有的NetworkFactory来说，当前的网络分值都是被扣掉40分之后的分值。  
 这就完全可能让原本分值高的网络由于没有通过有效性检测，而低于其他网络优先级。比如WIFI最高分值为60分，而Ethernet连接默认70分，但是因为该死的GFW，Ethernet没有通过有效性检测，所以分数只有70-40=30分，由此就会引入一个问题，当我们WIFI 连接外网AP时再插入网线开启Etherent连接， 实际上我们系统用的还是WIFI连接。

还有一点，如果这种网络类型是用户特意指定explicitlySelected的，则它的分值是100，高于任何其他网络，所以也会触发NetworkFactory中的网络连接。

目前只有在WIFI的code 中看到有设置某个连接是explicitlySelected。是在函数：

private void sendConnectedState() {

// Send out a broadcast with the CAPTIVE\_PORTAL\_CHECK to preserve

// existing behaviour. The captive portal check really happens after we

// transition into DetailedState.CONNECTED.

setNetworkDetailedState(DetailedState.CAPTIVE\_PORTAL\_CHECK);

mWifiConfigStore.updateStatus(mLastNetworkId,

DetailedState.CAPTIVE\_PORTAL\_CHECK);

sendNetworkStateChangeBroadcast(mLastBssid);

if (mWifiConfigStore.getLastSelectedConfiguration() != null) {

if (mNetworkAgent != null) mNetworkAgent.explicitlySelected();

}

setNetworkDetailedState(DetailedState.CONNECTED);

mWifiConfigStore.updateStatus(mLastNetworkId, DetailedState.CONNECTED);

sendNetworkStateChangeBroadcast(mLastBssid);

}

这一部分没有仔细看，初步理解是为了在WIFI连接中，当连接了某个AP后，会将其信息保存下来，为了提高这些连接的优先级，会将其分数设为100最高。

这样也会导致一个问题，即开机后WIFI连接过某个AP后，再进行Ethernet连接，

网络不会再切换到Ethernet，因为Ethernet分数最高为70，而此时WIFI的currentScore为100。

这个问题可以透过提高Ethernet默认分数解决，比如修改为110（国内的话还是修改为140吧）。