# TODO

WifiApConfigStore

**private static final** String DEFAULT\_AP\_CONFIG\_FILE =  
 Environment.getDataDirectory() + **"/misc/wifi/softap.conf"**;

原理修改

# 架构

Wifi



ConnectivityService提供数据连接管理服务，NetworkPolicyManagerService提供网络策略管理服务，NetworkStatsService提供网络传输数据统计服务，NetworkManagementService提供对物理网络接口的管理服务，connectivityService也包括VPN、Tethering对象提供虚拟连接及共享连接管理。

ConnectivityService、NetworkPolicyManagerService、NetworkStatsService三个服务都通过INetworkManagementService接口跨进程访问NetworkManagementService服务，实现与网络接口的交互及信息读取。

NetworkStatsService、NetworkPolicyManagerService两个服务还通过IConnectivityManager接口与connectivityService服务通讯，从connectivityService读取网络连接的信息及打开数据连接的策略控制。

ConnectivityService服务也通过INetworkPolicyManager接口调用NetworkPolicyManagerService的API，读取网络限额信息，登记监听对象。connectivityService服务通过NetworkPolicyManagerService服务的registerListener函数向NetworkPolicyManagerService服务注册一个INetworkPolicyListener.Stub监听桩对象。 NetworkPolicyManagerService通过该监听对象的远程代理接口向ConnectivityService服务传送规则变化通知。

另外ConnectivityService服务的Tethering、VPN对象及NetworkPolicyManagerService、NetworkStatsService服务的内部NetworkAlertObserver类型的对象都直接或间接派生自INetworkManagementEventObserver.Stub，且四个对象都登记为NetworkManagementService的监听对象，NetworkManagementService服务通过INetworkManagementEventObserver接口向这些对象传送网络接口事件通知。

NetworkPolicyManagerService维护网络使用策略，策略可以从一个策略文件读取（策略文件保存在系统目录下的netpolicy.xml文件中）。也可以通过NetworkPolicyManager对外提供的设置策略接口（setNetworkPolicies及setUidPolicy）进行设置，NetworkPolicyManagerService能够根据这些设置或从策略文件中读取的策略控制网络连接。另外NetworkPolicyManagerService还具有动态调节网络连接限额及动态设置网络连接的功能，动态调节网络连接限额机制是通过INetworkStatsService访问NetworkStatsService服务获得上面设置或读取的策略匹配的网络连接类型的传输统计信息（NetworkPolicyManagerService采用NetworkTemplate进行网络连接类型的匹配），并根据这些信息生成有效的规则，并提交给ConnectivityService服务，并调用NetworkManagementService的setInterfaceQuota函数对网络连接的带宽限额进行控制。

动态设置网络连接规则的机制是NetworkPolicyManagerService服务通过检测系统发出的一些相关事件（在NetworkPolicyManagerService的启动systemReady函数中注册），包括ActivityManager服务中IProcessObserver的onForegroundActivitiesChanged及onProcessDied回调事件，NetworkManager服务中INetworkManagementEventObserver的limitReached回调事件，以及ACTION\_SCREEN\_ON 、CONNECTIVITY\_ACTION\_IMMEDIATE、ACTION\_PACKAGE\_ADDED、ACTION\_UID\_REMOVED、ACTION\_NETWORK\_STATS\_UPDATED、ACTION\_ALLOW\_BACKGROUND等INTENT事件，当这些事件发生时，根据事件不同对网络规则进行不同设置，如与应用程序相关的事件调用updateRulesForUidLocked函数对uid涉及的NetworkRule进行更新，其它事件通过updateNetworkEnabledLocked函数调用connectivityService的setPolicyDataEnable函数对特定网络连接类型的数据连接进行设置。

NetworkStatsService服务定期调用performPoll函数获得网络传输统计信息，performPoll函数通过调用NetworkManagementService服务的getNetworkStatsUidDetail、getNetworkStatsSummary及getNetworkStatsTethering函数从/proc/目录下的包含网络传输统计数据的文件中读取网络统计信息，并转换为NetworkStatsHistory数据结构，保存到以网络接口名称对应的NetworkIdentitySet类型和UID对应的UidStatsKey类型的变量为key的NetworkStatsService的三个HashMap变量中。然后根据performPoll传进来的参数标志信息（指示不同的PERSIST方法）调用writeNetworkDevStatsLocked、writeNetworkXtStatsLocked、writeUidStatsLocked函数把HashMap变量中的统计信息分别写入系统目录下的三个相应的BIN文件（netstats.bin、netstats\_xt.bin、netstats\_uid.bin）中。

NetworkManagementService的一个重要功能是与本地netd进程进行通讯，完成对网络物理接口的操作。NetworkManagementService通过NativeDaemonConnector与本地netd进程通过LocalSocket建立连接进行双向通讯，发送命令，读取事件和命令应答消息，对网络接口的实际操作由netd进程完成。NativeDaemonConnector对象是一个实现Runnable接口的对象，NativeDaemonConnector对象在NetworkManagementService创建的线程中运行。

NativeDaemonConnector通过实例化时从NetworkManagementService传进来的回调函数向NetworkManagementService传送从netd进程读取的事件，主要事件有接口增加、接口移出、接口状态变化、LINK状态变化等接口改变事件以及带宽控制事件。

## 各个服务的启动

在

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79765186) [copy](https://blog.csdn.net/qq1028850792/article/details/79765186)

1. **if** (!disableNetwork) {
2. **try** {
3. Slog.i(TAG, "NetworkManagement Service");
4. networkManagement = NetworkManagementService.create(context);
5. ServiceManager.addService(Context.NETWORKMANAGEMENT\_SERVICE, networkManagement);
6. } **catch** (Throwable e) {
7. reportWtf("starting NetworkManagement Service", e);
8. }
9. }
10. **if** (!disableNetwork) {
11. **try** {
12. Slog.i(TAG, "Network Score Service");
13. networkScore = **new** NetworkScoreService(context);
14. ServiceManager.addService(Context.NETWORK\_SCORE\_SERVICE, networkScore);
15. } **catch** (Throwable e) {
16. reportWtf("starting Network Score Service", e);
17. }
19. **try** {
20. Slog.i(TAG, "NetworkStats Service");
21. networkStats = **new** NetworkStatsService(context, networkManagement, alarm);
22. ServiceManager.addService(Context.NETWORK\_STATS\_SERVICE, networkStats);
23. } **catch** (Throwable e) {
24. reportWtf("starting NetworkStats Service", e);
25. }
27. **try** {
28. Slog.i(TAG, "NetworkPolicy Service");
29. networkPolicy = **new** NetworkPolicyManagerService(
30. context, mActivityManagerService,
31. (IPowerManager)ServiceManager.getService(Context.POWER\_SERVICE),
32. networkStats, networkManagement);
33. ServiceManager.addService(Context.NETWORK\_POLICY\_SERVICE, networkPolicy);
34. } **catch** (Throwable e) {
35. reportWtf("starting NetworkPolicy Service", e);
36. }
38. mSystemServiceManager.startService(WIFI\_P2P\_SERVICE\_CLASS);
39. mSystemServiceManager.startService(WIFI\_SERVICE\_CLASS);
40. mSystemServiceManager.startService(
41. "com.android.server.wifi.WifiScanningService");
43. mSystemServiceManager.startService("com.android.server.wifi.RttService");
45. **if** (mPackageManager.hasSystemFeature(PackageManager.FEATURE\_ETHERNET)) {
46. mSystemServiceManager.startService(ETHERNET\_SERVICE\_CLASS);
47. }
49. **try** {
50. Slog.i(TAG, "Connectivity Service");
51. connectivity = **new** ConnectivityService(
52. context, networkManagement, networkStats, networkPolicy);
53. ServiceManager.addService(Context.CONNECTIVITY\_SERVICE, connectivity);
54. networkStats.bindConnectivityManager(connectivity);
55. networkPolicy.bindConnectivityManager(connectivity);
56. } **catch** (Throwable e) {
57. reportWtf("starting Connectivity Service", e);
58. }
60. **try** {
61. Slog.i(TAG, "Network Service Discovery Service");
62. serviceDiscovery = NsdService.create(context);
63. ServiceManager.addService(
64. Context.NSD\_SERVICE, serviceDiscovery);
65. } **catch** (Throwable e) {
66. reportWtf("starting Service Discovery Service", e);
67. }
68. }
70. **if** (!disableNonCoreServices) {
71. **try** {
72. Slog.i(TAG, "UpdateLock Service");
73. ServiceManager.addService(Context.UPDATE\_LOCK\_SERVICE,
74. **new** UpdateLockService(context));
75. } **catch** (Throwable e) {
76. reportWtf("starting UpdateLockService", e);
77. }
78. }

## 配置

<!-- Regex of wired ethernet ifaces -->

<string translatable="false" name="config\_ethernet\_iface\_regex">eth\\d</string>

# ConnectivityService

系统对网络的判断大多都是在 ConnectivityService.java中处理的，用户操作的类是 ConnectivityManager.java 通过aidl访问

ConnectivityService.java提供的服务。而启动在SystemServer.java中

## ConnectivityService()

Config.xml (base\core\res\res\values):

<string-array translatable="false" name="networkAttributes">

<item>"wifi,1,1,1,-1,true"</item>

<item>"mobile,0,0,0,-1,true"</item>

<item>"mobile\_mms,2,0,2,60000,true"</item>

<item>"mobile\_supl,3,0,2,60000,true"</item>

<item>"mobile\_hipri,5,0,3,60000,true"</item>

<item>"mobile\_fota,10,0,2,60000,true"</item>

<item>"mobile\_ims,11,0,2,60000,true"</item>

<item>"mobile\_cbs,12,0,2,60000,true"</item>

<item>"wifi\_p2p,13,1,0,-1,true"</item>

<item>"mobile\_ia,14,0,2,-1,true"</item>

<item>"mobile\_emergency,15,0,2,-1,true"</item>

</string-array>

## NetworkStateTrackerHandler

# NetworkStatsService

# netd

Netd（Network Daemon ），表示Network守护进程，类似的命名还有很多，例如 Vold（Volumn Deamon）---磁盘管理，Rild（Radio Interface Layer Deamon）--- 电话的基本数据功能……类似的还有好多，遍及Android各类服务，各个层次~

Netd负责跟一些涉及物理端口的网络操作相关的功能实现，例如带宽控制（Bandwidth），网络地址转换（NAT），个人局域网（pan），PPP链接，soft-ap，共享上网（Tether）等等……都是按照模块（.cpp+.h）组织在netd文件目录下的~

基本框架的四大部分：

(1)Linux Kernel 用于检测：network 相关的所有 event 事件。

(2)Netd 作为 Kernel 与：Framework 之间通信的桥梁。

(3)Framework 层操作：Netd,向 Netd 发送操作命令。

(4)UI 与 Framework：交互,用于用户进行网络的操控。

主要源码位置：

Netd:

/System/netd

/system/core/libsysutils/src

/system/core/include/sysutils

Framework:

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

**框架图**





NetworkmanagementService

此模块运行在SystemService中，负责Java层的实现机制，提供对上层的一些运行接口，当然，上层是通过一些抽象类实现进程间通讯进行访问的。

NetD

此模块是C++的Daemon，负责底层部分对于一些关键网络服务的管理。对上面Java服务提供接口，采用进程间通讯的方式。

Wpa\_supplicant

此模块是提供WIFI支持的模块，不做详细描述了。在Android中是一个关键的底层服务。

Dnsmasq

此服务实现了DHCP Server，用于辅助Hostapd，实现IP的管理。

Hosted

此服务实现了WIFI AP的关键服务，直接控制底层设备，此服务正常运行后，其他终端可以搜索到AP，并连接。

**从init.rc文件中可以看到，是在启动就开始运行的一个系统级的守护进程。而且同 Vold 基本并列~ 对于init.rc文件的意义请参考附录2**

****

## 如何开区一个softap

Soft AP代表通过软件实现Access Point的功能,

从功能角度来看，AP作为基站设备，起着连接其他无线设备到有线网的作用，相当于有线网络中的HUB与交换机。在日常工作和家庭中经常使用的无线路由器就是一个AP。一般情况下，它一端接着有线网络，另一端连接其他无线设备。

Station代表配备无线网络接口的设备，如手机、笔记本等。

虽然AP和Station是两个不同的设备，但实际上在Station中用软件也能实现AP拥有的功能，如桥接、路由等。在基本功能上，Soft AP与AP并没有太大的差别，只是Soft AP设备的接入能力和覆盖范围不如AP。

在Android系统中使用Soft AP功能还得借助另一个开源软件“hostapd”，这是一个运行在用户空间的用于AP和认证服务器的守护进程。它实现了IEEE 802.11相关的接入管理、IEEE 802.1X/WPA/WPA2/EAP 认证、RADIUS客户端、EAP服务器和RADIUS认证服务器。

以下是启动WIFI AP 的流程：



### 设置并开启 softap(便携式 WLAN 热点)

在 Setting 选项中进行设置,该设置功能的开启将涉及相关的UI 路径下相关的文件,packages /apps/Settings/src/com/android/settings/wifi 路径下,程序中的 WifiApEnabler. OnPreferenceChange 里, 设置 soft ap 在 wifiApDialog.onClick 里。

### 调用相应的处理函数,通过 socket 向 netd 下发命令。

对于 app 层的函数调用关系不做详细介绍,最终会调用到 Framework 层的 NetworkManagementService.startAccessPoint 函数。

frameworks\base\services\core\java\com\android\server

**[java]** [view plain](http://blog.csdn.net/xiaokeweng/article/details/8130218) [copy](http://blog.csdn.net/xiaokeweng/article/details/8130218)

1. //NetworkManagementService.java
2. @Override
3. **public** **void** startAccessPoint(
4. WifiConfiguration wifiConfig, String wlanIface, String softapIface) {
5. mContext.enforceCallingOrSelfPermission(CONNECTIVITY\_INTERNAL, TAG);
6. **try** {
7. wifiFirmwareReload(wlanIface, "AP");
8. **if** (wifiConfig == **null**) {
9. mConnector.execute("softap", "set", wlanIface, softapIface);
10. } **else** {
11. mConnector.execute("softap", "set", wlanIface, softapIface, wifiConfig.SSID,
12. getSecurityType(wifiConfig), wifiConfig.preSharedKey);
13. }
14. mConnector.execute("softap", "startap");
15. } **catch** (NativeDaemonConnectorException e) {
16. **throw** e.rethrowAsParcelableException();
17. }
18. }

### **netd处理,并将反馈给 Framework**

Netd 中 softap 控制的功能在/system/netd/SoftapController.{h,cpp}里, 具体的执行是通过调用网卡驱动的 ap 功能。NetworkManagementService 通过 NativeDaemonConnector 向下通过 socket 向下 softap 相关的字符串命令,NativeDaemonConnector 中维护着与 Netd 中 CommandListener 相关联的内部socket 线程。两者可以通过它相互通信。

**[cpp]** [view plain](http://blog.csdn.net/xiaokeweng/article/details/8130218) [copy](http://blog.csdn.net/xiaokeweng/article/details/8130218)

1. //CommandListener.cpp
2. int CommandListener::SoftapCmd::runCommand(SocketClient \*cli,
3. int argc, char \*\*argv) {
4. int rc = ResponseCode::SoftapStatusResult;
5. char \*retbuf = NULL;
6. if (sSoftapCtrl == NULL) {
7. cli->sendMsg(ResponseCode::ServiceStartFailed, "SoftAP is not available", false);
8. return -1;
9. }
10. if (argc < 2) {
11. cli->sendMsg(ResponseCode::CommandSyntaxError,
12. "Missing argument in a SoftAP command", false);
13. return 0;
14. }
15. if (!strcmp(argv[1], "startap")) {
16. rc = sSoftapCtrl->startSoftap();
17. } else if (!strcmp(argv[1], "stopap")) {
18. rc = sSoftapCtrl->stopSoftap();
19. } else if (!strcmp(argv[1], "fwreload")) {
20. rc = sSoftapCtrl->fwReloadSoftap(argc, argv);
21. } else if (!strcmp(argv[1], "status")) {
22. asprintf(&retbuf, "Softap service %s running",
23. (sSoftapCtrl->isSoftapStarted() ? "is" : "is not"));
24. cli->sendMsg(rc, retbuf, false);
25. free(retbuf);
26. return 0;
27. } else if (!strcmp(argv[1], "set")) {
28. **rc = sSoftapCtrl->setSoftap(argc, argv);**
29. } else {
30. cli->sendMsg(ResponseCode::CommandSyntaxError, "Unrecognized SoftAP command", false);
31. return 0;
32. }
33. if (rc >= 400 && rc < 600)
34. cli->sendMsg(rc, "SoftAP command has failed", false);
35. else
36. cli->sendMsg(rc, "Ok", false);
37. return 0;
38. }……

**sSoftapCtrl = new SoftapController();**

## SoftAP

### （2）SoftapCmd命令使用

和TetherCmd类似，开启[Android](https://www.2cto.com/kf/yidong/Android/)中手机的Soft AP功能将涉及大量Framework层中的操作，本节仅关注和Netd相关的三个步骤。

1）首先为Wi-Fi加载不同的固件（Firmware），这是通过SoftapController的fwReloadSoftap函数完成的，代码如下所示。

#### SoftapController->fwReloadSoftap

int SoftapController::fwReloadSoftap(int argc, char \*argv[])  
{  
    int ret, i = 0;  
    char \*iface;  
    char \*fwpath;

   ......// 参数检测  
    iface = argv[2];  
    if (strcmp(argv[3], "AP") == 0) {  
        fwpath = (char \*)wifi\_get\_fw\_path(WIFI\_GET\_FW\_PATH\_AP);  
    } else if (strcmp(argv[3], "P2P") == 0) {  
        fwpath = (char \*)wifi\_get\_fw\_path(WIFI\_GET\_FW\_PATH\_P2P);  
    } else {  
        fwpath = (char \*)wifi\_get\_fw\_path(WIFI\_GET\_FW\_PATH\_STA);  
    }  
    // 通过往/sys/module/wlan/parameters/fwpath文件中写入固件名  
    // 触发驱动去加载对应的固件  
    ret = wifi\_change\_fw\_path((const char \*)fwpath);  
    ......  
    return ret;  
}

上面这段代码表示在Android中，如果要让Wi-Fi无线设备扮演不同的角色，得为它们加载不同的固件（Firmware），具体说明如下。

WIFI\_GET\_FW\_PATH\_AP：代表Soft AP功能的固件，其对应的文件位置由WIFI\_DRIVER\_FW\_PATH\_AP宏表达。三星Tuna平台中，该文件位置为/vendor/firmware/fw\_bcmdhd\_apsta.bin。

WIFI\_GET\_FW\_PATH\_P2P：代表P2P功能的固件，其对应的文件位置由WIFI\_DRIVER\_FW\_PATH\_P2P宏表达。三星Tuna平台中，该文件位置为/vendor/firmware/ fw\_bcmdp2p.bin。

WIFI\_GET\_FW\_PATH\_STA：代表Station功能的固件，其对应的文件位置由WIFI\_DRIVER\_FW\_PATH\_STA宏表达。三星Tuna平台中，该文件位置为/vendor/firmware/fw\_bcmdhd.bin。

提示　三星Tuna平台对应的配置文件在Android 4.2[源码](https://www.2cto.com/ym/)根目录/device/samsung/tuna目录中。从上面的固件文件名来看，它用的Wi-Fi无线芯片是博通（Broadcom）公司生产的。通过加载不同固件的方式来启用无线芯片硬件的不同功能可能和Wi-Fi驱动及芯片的设计有关。

另外，根据审稿专家的反馈，在Android 4.2中，STA和P2P可同时运行（即所谓的共存模式），这样STA和P2P实际对应的固件相同，但可能文件名不同。而SoftAP的固件与STA/P2P就不一样了。

#### SoftapController.cpp::setSoftap

2）加载完指定的Wi-Fi固件后，下一步将对Soft AP功能进行一些配置，配置信息最终将写到一个配置文件。这部分功能由SoftapController的setSoftap函数完成，代码如下所示。

int SoftapController::setSoftap(int argc, char \*argv[]) {  
    char psk\_str[2\*SHA256\_DIGEST\_LENGTH+1];  
    int ret = 0, i = 0, fd;  
    char \*ssid, \*iface;

    ......// 参数检查

    iface = argv[2];

    char \*wbuf = NULL;  
    char \*fbuf = NULL;

    if (argc > 3) {  
        ssid = argv[3];  
    } else {  
        ssid = (char \*)"AndroidAP"; // SSID即接入点的名称  
    }

[asp](https://www.2cto.com/kf/web/asp/)rintf(&wbuf, "interface=%s\ndriver=nl80211\nctrl\_interface="  
            "/data/misc/wifi/hostapd\nssid=%s\nchannel=6\nieee80211n=1\n",  
            iface, ssid);

    if (argc > 4) { // 判断AP的[加密](https://www.2cto.com/article/jiami/)类型  
        if (!strcmp(argv[4], "wpa-psk")) {  
            generatePsk(ssid, argv[5], psk\_str);  
            [asp](https://www.2cto.com/kf/web/asp/)rintf(&fbuf, "%swpa=1\nwpa\_pairwise=TKIP CCMP\nwpa\_psk=%s\n",  
                      wbuf, psk\_str);  
        } else if (!strcmp(argv[4], "wpa2-psk")) {  
            generatePsk(ssid, argv[5], psk\_str);  
            asprintf(&fbuf, "%swpa=2\nrsn\_pairwise=CCMP\nwpa\_psk=%s\n",  
                      wbuf, psk\_str);  
        } else if (!strcmp(argv[4], "open")) {  
            asprintf(&fbuf, "%s", wbuf);  
        }  
    }  ......  
    // HOSTAPD\_CONF\_FILE指向/data/misc/wifi/hostapd.conf文件  
    fd = open(HOSTAPD\_CONF\_FILE, O\_CREAT | O\_TRUNC | O\_WRONLY, 0660);  
    ......  
    if (write(fd, fbuf, strlen(fbuf)) < 0) {  
        ALOGE("Cannot write to \"%s\": %s", HOSTAPD\_CONF\_FILE, strerror(errno));  
        ret = -1;  
    }  
    ......// 修改该文件的读写权限等  
    return ret;  
}

上面代码中涉及Wi-Fi技术的很多概念，将在后续章节统一介绍。从功能上来说，setSoftap函数无非就是把一些配置信息写到一个hostapd.conf文件中。可以通过一个例子文件来了解此文件的内容。

Android4.2/hardware/ti/wlan/mac80211/config目录中有一个hostapd.conf文件，其内容如下所示。  
[-->hostapd.conf]  
driver=nl80211   #指定Wi-Fi驱动的名称  
......#略去部分内容  
ssid=AndroidAP   #设置接入点名称为AndroidAP  
country\_code=US  
wep\_rekey\_period=0  
eap\_server=0  
own\_ip\_addr=127.0.0.1  
wpa\_group\_rekey=0  
wpa\_gmk\_rekey=0   #[加密](https://www.2cto.com/article/jiami/)方式等设置  
wpa\_ptk\_rekey=0  
interface=wlan1   #网络设备接口  
......#略去部分内容

由上边示例的hostapd.conf可知，当使用该配置文件后，其他Station搜索到由这台手机设置的Soft AP的名称将会是“AndroidAP”。

#### SoftapController的startap

3）最后，SoftapController的startap函数被调用，它将启动hostapd进程。重点关注hostapd启动的参数信息，如下所示。  
hostapd \  
-e /data/misc/wifi/entropy.bin \和Wi-Fi协议中的信息加密有关  
/data/misc/wifi/hostapd.conf \hostapd的配置文件

不同的wifi芯片需要不同的驱动支持。请参考external/wpa\_supplicant/README

1：例如通用的Broadcom wl.0 driver，它用于，Broadcom IEEE 802.11a/g cards，同时支持wifi热点和无线路由，即service端和client端。可以这样说，wifi热点和无线wifi是互斥的，两者不能共存。

关于Broadcom wl.0 driver，他不需要通过hostap的方式实现热点与driver的通信。而是直接在system/netd/SoftapController.cpp中与驱动通信。而system/netd/SoftapController.cpp与framework的通信则是通过socket。framework中NetworkManagementService.java中的startAccessPoint方法调用doCommand方法像SoftapController通过socket发送请求，并在NativeDaemonConnector.java中调用listenToSocket方法监听上报的事件，并向事件广播给app层。 之前在NetworkManagementService.java中的startAccessPoint中通过向下发送fwreload的请求来关闭wifi，打开wifi热点，但这样做会引起后期多次切换无线wifi和wifi热点的过程，打开无线wifi和wifi热点出错的Bug。之后改进的方法是在hardware/wifi/wifi.c中显式的load firmware。

2：另外madwifi driver 基于Atheros chip，这个Atheros在wifi.c中随处可见，但没用到。

3：ATMEL AT76C5XXx driver 用于USB和PCMCIA

4：其实移植android2.3。4的源码时，如果需要增加wifi热点功能，只需把softap的interface改成wl0.1就ok了。除非修改Bug，否则不需要大改，因为android2.2之后的版本本身就支持wifi热点。   
5：另外如果出现以下Bug：”保存添加的网络后重新开机，不显示添加的网络“或者“设备重启后WIFI不会自动连接上次保存好的ap”时，则需要考虑是否wpa\_supplicant.conf有可写的权限。可考虑修改init.rc。将service wpa\_supplicant /system/bin/logwrapper /system/bin/wpa\_supplicant -Dwext -ieth0 -c/system/etc/wifi/wpa\_supplicant.conf 改成   
service wpa\_supplicant /system/bin/logwrapper /system/bin/wpa\_supplicant -Dwext -ieth0 -c/data/misc/wifi/wpa\_supplicant.conf   
6：当wifi启用静态IP，并且设置的静态IP，网关，网络掩码都正确时，即使连接的ap本身不能上网，也不影响启用静态IP后能正常访问网络。

## 实战分析

### Softap热点原理分析

# Tethering

# Ethernet

## 启动流程

### SystemServer

1. **public** **final** **class** SystemServer {
2. **private** **static** **final** String ETHERNET\_SERVICE\_CLASS =
3. "com.android.server.ethernet.EthernetService";
4. **public** **static** **void** main(String[] args) {
5. **new** SystemServer().run();
6. }
7. **private** **void** run() {
8. startOtherServices();
9. }
10. **private** **void** startOtherServices() {
11. **if** (mPackageManager.hasSystemFeature(PackageManager.FEATURE\_ETHERNET)) {
12. mSystemServiceManager.startService(ETHERNET\_SERVICE\_CLASS);
13. }
14. }
15. }

E

### EthernetService()

**[java]** [view plain](https://blog.csdn.net/tankai19880619/article/details/44559419) [copy](https://blog.csdn.net/tankai19880619/article/details/44559419)

1. **public** **final** **class** EthernetService **extends** SystemService {
2. **private** **static** **final** String TAG = "EthernetService";
3. **final** EthernetServiceImpl mImpl;
4. **public** EthernetService(Context context) {
5. **super**(context);
6. mImpl = **new** EthernetServiceImpl(context);
7. }
8. @Override
9. **public** **void** onStart() {
10. Log.i(TAG, "Registering service " + Context.ETHERNET\_SERVICE);
11. publishBinderService(Context.ETHERNET\_SERVICE, mImpl);
12. }
13. @Override
14. **public** **void** onBootPhase(**int** phase) {
15. **if** (phase == SystemService.PHASE\_SYSTEM\_SERVICES\_READY) {
16. mImpl.start();
17. }
18. }
19. }

### EthernetServiceImpl

打发斯蒂芬

**[java]** [view plain](https://blog.csdn.net/tankai19880619/article/details/44559419) [copy](https://blog.csdn.net/tankai19880619/article/details/44559419)

1. **public** **class** EthernetServiceImpl **extends** IEthernetManager.Stub {
2. **public** EthernetServiceImpl(Context context) {
3. mContext = context;
4. Log.i(TAG, "Creating EthernetConfigStore");
5. mEthernetConfigStore = **new** EthernetConfigStore();
6. mIpConfiguration = mEthernetConfigStore.readIpAndProxyConfigurations();
7. Log.i(TAG, "Read stored IP configuration: " + mIpConfiguration);
8. IBinder b = ServiceManager.getService(Context.NETWORKMANAGEMENT\_SERVICE);
9. mNMService = INetworkManagementService.Stub.asInterface(b);
10. mTracker = **new** EthernetNetworkFactory();
11. }
12. **public** **void** start() {
13. mCM = (ConnectivityManager) mContext.getSystemService(Context.CONNECTIVITY\_SERVICE);
14. HandlerThread handlerThread = **new** HandlerThread("EthernetServiceThread");
15. handlerThread.start();
16. mHandler = **new** Handler(handlerThread.getLooper());
17. mEnabled = getPersistedState();
18. Log.i(TAG, "Ethernet Persisted Enabled " + mEnabled);
19. setState(mEnabled);  //重要
20. }
21. **public** **synchronized** **void** setState(**int** state) {
22. enforceChangePermission();
23. Log.i(TAG, "setState from mState=" + mState + " to state=" + state);
24. **if** (mState != state) {
25. mState = state;
26. **if** (state == EthernetManager.ETHERNET\_STATE\_DISABLED) {
27. setPersistedState(EthernetManager.ETHERNET\_STATE\_DISABLED);
28. mTracker.stopInterface();
29. mStarted.set(**false**);
30. } **else** {
31. setPersistedState(EthernetManager.ETHERNET\_STATE\_ENABLED);
32. mTracker.stop();
33. mTracker.start(mContext, mHandler);
34. mStarted.set(**true**);
35. }
36. }
37. }
38. }

private static final String ipConfigFile = Environment.getDataDirectory() +

"/misc/ethernet/ipconfig.txt";

### EthernetNetworkFactory

发广播s

1. **class** EthernetNetworkFactory {
2. EthernetNetworkFactory() {
3. mNetworkInfo = **new** NetworkInfo(ConnectivityManager.TYPE\_ETHERNET, 0, NETWORK\_TYPE, "");
4. mLinkProperties = **new** LinkProperties();
5. initNetworkCapabilities();
6. }
7. **public** **synchronized** **void** start(Context context, Handler target) {
8. IBinder b = ServiceManager.getService(Context.NETWORKMANAGEMENT\_SERVICE);
9. mNMService = INetworkManagementService.Stub.asInterface(b);
10. mEthernetManager = (EthernetManager) context.getSystemService(Context.ETHERNET\_SERVICE);
11. // Interface match regex.
12. mIfaceMatch = context.getResources().getString(
13. com.android.internal.R.string.config\_ethernet\_iface\_regex);
14. mFactory = **new** LocalNetworkFactory(NETWORK\_TYPE, context, target.getLooper());
15. mFactory.setCapabilityFilter(mNetworkCapabilities);
16. mFactory.setScoreFilter(-1); // this set high when we have an iface
17. mFactory.register();
18. /\*
19. public void register() {
20. if (DBG) log("Registering NetworkFactory");
21. if (mMessenger == null) {
22. mMessenger = new Messenger(this);
23. ConnectivityManager.from(mContext).registerNetworkFactory(mMessenger, LOG\_TAG);
24. }
25. }
26. frameworks/base/services/core/java/com/android/server/ConnectivityService.java
27. public void registerNetworkFactory(Messenger messenger, String name) {
28. NetworkFactoryInfo nfi = new NetworkFactoryInfo(name, messenger, new AsyncChannel());
29. mHandler.sendMessage(mHandler.obtainMessage(EVENT\_REGISTER\_NETWORK\_FACTORY, nfi));
30. }
31. private class InternalHandler extends Handler {
32. public void handleMessage(Message msg) {
33. case EVENT\_REGISTER\_NETWORK\_FACTORY: {
34. handleRegisterNetworkFactory((NetworkFactoryInfo)msg.obj);
35. break;
36. }
37. }
38. }
39. private void handleRegisterNetworkFactory(NetworkFactoryInfo nfi) {
40. if (DBG) log("Got NetworkFactory Messenger for " + nfi.name);
41. mNetworkFactoryInfos.put(nfi.messenger, nfi);
42. nfi.asyncChannel.connect(mContext, mTrackerHandler, nfi.messenger);
43. }
44. \*/
45. mInterfaceObserver = **new** InterfaceObserver();
46. **try** {
47. mNMService.registerObserver(mInterfaceObserver);
48. } **catch** (RemoteException e) {
49. Log.e(TAG, "Could not register InterfaceObserver " + e);
50. }
51. updateInterfaceState(iface, **true**); //注册
52. }
53. **private** **void** updateInterfaceState(String iface, **boolean** up) {
54. updateAgent();
55. mFactory.setScoreFilter(up ? NETWORK\_SCORE : -1); //设置scroe值；这个是网络优先级判断依据
56. }
57. **public** **void** updateAgent() {
58. mNetworkAgent.sendNetworkInfo(mNetworkInfo);
59. }
60. }

Ds

## 2.ETHERNET监听NETD进程的socket

Deas

init.rc

**[plain]** [view plain](https://blog.csdn.net/tankai19880619/article/details/44559419) [copy](https://blog.csdn.net/tankai19880619/article/details/44559419)

1. service netd /system/bin/netd
2. class main
3. socket netd stream 0660 root system
4. socket dnsproxyd stream 0660 root inet
5. socket mdns stream 0660 root system
6. socket fwmarkd stream 0660 root inet

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

**[java]** [view plain](https://blog.csdn.net/tankai19880619/article/details/44559419) [copy](https://blog.csdn.net/tankai19880619/article/details/44559419)

1. **public** **static** NetworkManagementService create(Context context) **throws** InterruptedException {
2. **return** create(context, NETD\_SOCKET\_NAME);
3. /\*
4. private static final String NETD\_SOCKET\_NAME = "netd";
5. \*/
6. }
8. **private** NetworkManagementService(Context context, String socket) {
9. mConnector = **new** NativeDaemonConnector(
10. **new** NetdCallbackReceiver(), socket, 10, NETD\_TAG, 160, wl,
11. FgThread.get().getLooper());
12. /\*
13. frameworks/base/services/core/java/com/android/server/NativeDaemonConnector.java
14. public void run() {
15. mCallbackHandler = new Handler(mLooper, this);
16. while (true) {
17. try {
18. listenToSocket();
19. } catch (Exception e) {
20. loge("Error in NativeDaemonConnector: " + e);
21. SystemClock.sleep(5000);
22. }
23. }
24. }
25. private void listenToSocket() throws IOException {
26. LocalSocketAddress address = determineSocketAddress();
27. mCallbackHandler.sendMessage();
28. }
29. public boolean handleMessage(Message msg) {
30. mCallbacks.onEvent
31. }
32. \*/
33. }
35. **private** **class** NetdCallbackReceiver **implements** INativeDaemonConnectorCallbacks {
36. **public** **boolean** onEvent(**int** code, String raw, String[] cooked) {
37. notifyInterfaceAdded(cooked[3]);
38. ......
39. notifyInterfaceRemoved(cooked[3]);
40. ......
41. notifyInterfaceStatusChanged(cooked[3], cooked[4].equals("up"));
42. ......
43. notifyInterfaceLinkStateChanged(cooked[3], cooked[4].equals("up"));
44. }
45. **private** **void** notifyInterfaceLinkStateChanged(String iface, **boolean** up) {
46. mObservers.getBroadcastItem(i).interfaceLinkStateChanged(iface, up);
47. }
48. }

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

**[java]** [view plain](https://blog.csdn.net/tankai19880619/article/details/44559419) [copy](https://blog.csdn.net/tankai19880619/article/details/44559419)

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

**[java]** [view plain](https://blog.csdn.net/tankai19880619/article/details/44559419) [copy](https://blog.csdn.net/tankai19880619/article/details/44559419)

1. **private** **class** InterfaceObserver **extends** BaseNetworkObserver {
2. **public** **void** interfaceLinkStateChanged(String iface, **boolean** up) {
3. updateInterfaceState(iface, up);
4. }
5. }
7. **private** **void** updateInterfaceState(String iface, **boolean** up) {
8. mFactory.setScoreFilter(up ? NETWORK\_SCORE : -1);
9. }

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

**[java]** [view plain](https://blog.csdn.net/tankai19880619/article/details/44559419) [copy](https://blog.csdn.net/tankai19880619/article/details/44559419)

1. **public** **void** setScoreFilter(**int** score) {
2. sendMessage(obtainMessage(CMD\_SET\_SCORE, score, 0));
3. }
5. **public** **void** handleMessage(Message msg) {
6. **case** CMD\_SET\_SCORE: {
7. handleSetScore(msg.arg1);
8. **break**;
9. }
10. }
12. **private** **void** handleSetScore(**int** score) {
13. mScore = score;
14. evalRequests();
15. }
17. **private** **void** evalRequests() {
18. **for** (**int** i = 0; i < mNetworkRequests.size(); i++) {
19. NetworkRequestInfo n = mNetworkRequests.valueAt(i);
20. evalRequest(n);
21. }
22. }
24. **private** **void** evalRequest(NetworkRequestInfo n) {
25. needNetworkFor(n.request, n.score);
26. }
28. **protected** **void** needNetworkFor(NetworkRequest networkRequest, **int** score) {
29. **if** (++mRefCount == 1) startNetwork();
30. }

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

**[java]** [view plain](https://blog.csdn.net/tankai19880619/article/details/44559419) [copy](https://blog.csdn.net/tankai19880619/article/details/44559419)

1. **private** **class** LocalNetworkFactory **extends** NetworkFactory {
2. LocalNetworkFactory(String name, Context context, Looper looper) {
3. **super**(looper, context, name, **new** NetworkCapabilities());
4. }
5. **protected** **void** startNetwork() {
6. onRequestNetwork();
7. }
8. **protected** **void** stopNetwork() {
9. }
10. }
12. **public** **void** onRequestNetwork() {
13. Thread dhcpThread = **new** Thread(**new** Runnable() {
14. **public** **void** run() {
15. DhcpResults dhcpResults = **new** DhcpResults(); //DHCP相关
16. **if** (!NetworkUtils.runDhcp(mIface, dhcpResults)) {
17. /\*
18. frameworks/base/core/java/android/net/NetworkUtils.java
19. public native static boolean runDhcp(String interfaceName, DhcpResults dhcpResults);
20. frameworks/base/core/jni/android\_net\_NetUtils.cp
21. static jboolean android\_net\_utils\_runDhcp(JNIEnv\* env, jobject clazz, jstring ifname, jobject info)
22. {
23. return android\_net\_utils\_runDhcpCommon(env, clazz, ifname, info, false);
24. }
25. static jboolean android\_net\_utils\_runDhcpCommon(JNIEnv\* env, jobject clazz, jstring ifname,
26. jobject dhcpResults, bool renew)
27. {
28. if (renew) {
29. result = ::dhcp\_do\_request\_renew(nameStr, ipaddr, gateway, &prefixLength,
30. dns, server, &lease, vendorInfo, domains, mtu);
31. } else {
32. result = ::dhcp\_do\_request(nameStr, ipaddr, gateway, &prefixLength,
33. dns, server, &lease, vendorInfo, domains, mtu);
34. }
35. }
36. system/core/libnetutils/dhcp\_utils.c
37. 见博文：《Android系统DHCP问题》
38. 上处DHCP Client和DHCP server（system/bin/dhcpd进程）通过property\_get/set 共享内存来共享信息
39. \*/
40. Log.e(TAG, "DHCP request error:" + NetworkUtils.getDhcpError());
41. // set our score lower than any network could go
42. // so we get dropped.
43. mFactory.setScoreFilter(-1);
44. **return**;
45. }
46. mNetworkAgent = **new** NetworkAgent(mFactory.getLooper(), mContext,
47. NETWORK\_TYPE, mNetworkInfo, mNetworkCapabilities, mLinkProperties,
48. NETWORK\_SCORE)
49. }
50. });
51. dhcpThread.start();
52. }

应用程序调用关键API

**[java]** [view plain](https://blog.csdn.net/tankai19880619/article/details/44559419) [copy](https://blog.csdn.net/tankai19880619/article/details/44559419)

1. **public** **void** updateDevInfo(EthernetDevInfo info);

配置完成以后ConnectivityService向EthernetManager发送CONNECTIVITY\_ACTION\_IMMEDIATE的广播；EthernetManager接收到该广播以后向应用程序发送ETHERNET\_INTERFACE\_CONF\_CHANGED广播。否则；应用程序将TIMEOUT。

**[java]** [view plain](https://blog.csdn.net/tankai19880619/article/details/44559419) [copy](https://blog.csdn.net/tankai19880619/article/details/44559419)

1. **public** **static** **final** String EXTRA\_ETHERNET\_STATE = "ETHERNET\_state";
2. **public** **static** **final** String ETHERNET\_INTERFACE\_CONF\_CHANGED =
3. "android.net.ethernet.ETHERNET\_INTERFACE\_CONF\_CHANGED"; //add by tank
4. **private** **void** sendEthBroadcast(String action, **boolean** state) {
5. String bootStr = SystemProperties.get("sys.boot\_completed");
6. Log.d(TAG, "sendEthBroadcast -->: " + bootStr);
7. **if**(bootStr.equals("1")) { //boot complete
8. Intent intent = **new** Intent(action);
9. intent.putExtra(EXTRA\_ETHERNET\_STATE, state);
11. Log.d(TAG, "sendEthBroadcast --> action= " + action + " state=" + state);
12. mContext.sendBroadcast(intent);
13. }
14. }
15. //连接成功调用如下：
16. sendEthBroadcast(ETHERNET\_INTERFACE\_CONF\_CHANGED, **true**);
17. //连接失败调用如下：
18. sendEthBroadcast(ETHERNET\_INTERFACE\_CONF\_CHANGED, **false**);

## 相关源文件

### Settings

**Setting中添加选项代码**

packages/apps/Settings/src/com/android/settings/ethernet/EthernetSettings.java

packages/apps/Settings/src/com/android/settings/ethernet/EthernetEnabler.java

packages/apps/Settings/src/com/android/settings/ethernet/EthernetConfigDialog.java

### frameworks/base/ SystemUI:

SystemUI:  **//状态栏（status\_bar）显示部分代码**  
frameworks/base/packages/SystemUI/src/com/android/systemui/statusbar/policy/NetworkController.java  
frameworks/base/packages/SystemUI/src/com/android/systemui/statusbar/SignalClusterView.java    //现实statusbar

### frameworks/base/services

ConnectivityService:  
frameworks/base/services/java/com/android/server/ConnectivityService.java   **//这里是ethernet部分程序的起始点**

本来就有的ethernet：  
frameworks/base/services/java/com/android/server/EthernetService.java  
frameworks/base/services/java/com/android/server/NetworkManagementService.java  
frameworks/base/core/java/android/net/NetworkStats.java   
  
新添加的ethernet：  
frameworks/base/ethernet/\* **// 这是主要ethernet部分，java api 代码。**  
frameworks/base/ethernet/java/android/net/ethernet/EthernetManager.java

### jni:

frameworks/base/core/jni/android\_net\_ethernet.cpp  **//新加的一些jni**

## 常用命令

### netcfg   //查看ip情况

源码：system\core\netcfg

设计模式！pingni确实知道不少哦

struct

{

const char \*name;

int nargs;

void \*func;

} CMDS[] = {

{ "dhcp", 1, do\_dhcp },

{ "up", 1, ifc\_up },

{ "down", 1, ifc\_down },

{ "deldefault", 1, ifc\_remove\_default\_route },

{ "hwaddr", 2, set\_hwaddr },

{ 0, 0, 0 },

};

#### 使用

netcfg //查看ip情况

netcfg eth0 up dhcp //通过dhcp 自动获取ip和网关

root@gl300k:/ # netcfg

wlan0 UP 192.168.43.44/24 0x00001043 00:03:7f:20:52:81

lo UP 127.0.0.1/8 0x00000049 00:00:00:00:00:00

usb0 UP 192.168.42.2/24 0x00001043 ce:a8:20:9c:1a:cd

sit0 DOWN 0.0.0.0/0 0x00000080 00:00:00:00:00:00

p2p0 UP 0.0.0.0/0 0x00001003 06:03:7f:20:52:81

### ifconfig

ifconfig usb0 192.168.42.2 up

ifconfig usb0 192.168.42.2 netmask 255.255.255.0 up

### gateway 配置

route add default gw 192.168.42.2 dev usb0

route add default gw 192.168.1.3 dev waln0

### dns 配置

echo "nameserver 8.8.8.8" > resolv.conf

nameserver 8.8.8.8

setprop net.dns1 8.8.8.8

setprop net.dns2 8.8.4.4

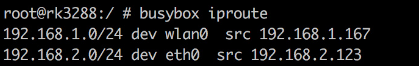
setprop net.dns1 0.0.0.0

setprop net.dns2 0.0.0.0

### mac adddr

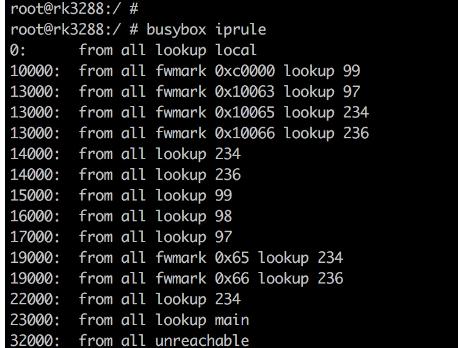
ifconfig eth0 hw ether 00:11:22:33:44:55

### iproute



Ip route

### Iprole



### VPN-地址绑定

由于某些原因，可能需要指定域名对应的IP地址。Android是基于Linux的系统，与Linux类似，通过hosts文件来设置。

在Android下，/etc是link到/system/etc的，我们需要修改/system/etc/hosts来实现

127.0.0.1 localhost

192.168.1.94 [www.leadding.com](http://www.leadding.com/) >

注意换行问题，最好用vi编辑器来或者AS来初六

## [2个辅助模块 libnetutils 和 dhcpcd](http://blog.csdn.net/lamdoc/article/details/7646265)

### Libnetutils

1. 一个是 system/core/libnetutils/\* , 即libnetutils.so 库. 这里面有两个.c经常会调用到ifc\_utils.c ,

dhcp\_utils.c

java会 通过 JNI （CPP）再调用到 C代码

ifc\_utils

/\*

\* Clears IPv4 addresses on the specified interface.

\*/

void ifc\_clear\_ipv4\_addresses(const char \*name) {

unsigned count, addr;

ifc\_init();

for (count=0, addr=1;((addr != 0) && (count < 255)); count++) {

if (ifc\_get\_addr(name, &addr) < 0)

break;

if (addr)

ifc\_set\_addr(name, 0);

}

ifc\_close();

}

int ifc\_set\_addr(const char \*name, in\_addr\_t addr)

{

struct ifreq ifr;

int ret;

ifc\_init\_ifr(name, &ifr);

init\_sockaddr\_in(&ifr.ifr\_addr, addr);

ret = ioctl(ifc\_ctl\_sock, SIOCSIFADDR, &ifr);

if (DBG) printerr("ifc\_set\_addr(%s, xx) = %d", name, ret);

return ret;

}

static const char DAEMON\_NAME[] = "dhcpcd";

static const char DAEMON\_PROP\_NAME[] = "init.svc.dhcpcd";

static const char HOSTNAME\_PROP\_NAME[] = "net.hostname";

static const char DHCP\_PROP\_NAME\_PREFIX[] = "dhcp";

static const char DHCP\_CONFIG\_PATH[] = "/system/etc/dhcpcd/dhcpcd.conf";

#### java 接口：ＮｅｔｗｏｒｋＵｔｉｌｓ

Native methods for managing network interfaces

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

NetWorkUtils 类中的一些函数， 这些函数，会调用JNI：android\_net\_NetUtils.cpp，然后调到　ｌｉｂｎｅｔｕｔｉｌｓ　库中去。

NetworkUtils.disableInterface(ifname);

NetworkUtils.runDhcp(mInterfaceName, mDhcpInfo)；

NetworkUtils.stopDhcp(mInterfaceName)；

NetworkUtils.resetConnections(mInterfaceName, NetworkUtils.RESET\_ALL\_ADDRESSES);

NetworkUtils.configureInterface(info.getIfName(), mDhcpInfo1)；

NetworkUtils.removeDefaultRoute(ifname);

#### android\_net\_NetUtils.cpp

JNI映射方法

**[cpp]** [view plaincopy](http://blog.csdn.net/lamdoc/article/details/7653476)

1. /\*
2. \* JNI registration.
3. \*/
4. **static** JNINativeMethod gNetworkUtilMethods[] = {
5. /\* name, signature, funcPtr \*/
7. { "enableInterface", "(Ljava/lang/String;)I",  (**void** \*)android\_net\_utils\_enableInterface },
8. { "disableInterface", "(Ljava/lang/String;)I",  (**void** \*)android\_net\_utils\_disableInterface },
9. { "resetConnections", "(Ljava/lang/String;I)I",  (**void** \*)android\_net\_utils\_resetConnections },
10. { "removeDefaultRoute", "(Ljava/lang/String;)I",  (**void** \*)android\_net\_utils\_removeDefaultRoute },
11. { "runDhcp", "(Ljava/lang/String;Landroid/net/DhcpInfoInternal;)Z",  (**void** \*)android\_net\_utils\_runDhcp },
12. { "runDhcpRenew", "(Ljava/lang/String;Landroid/net/DhcpInfoInternal;)Z",  (**void** \*)android\_net\_utils\_runDhcpRenew },
13. { "stopDhcp", "(Ljava/lang/String;)Z",  (**void** \*)android\_net\_utils\_stopDhcp },
14. { "releaseDhcpLease", "(Ljava/lang/String;)Z",  (**void** \*)android\_net\_utils\_releaseDhcpLease },
15. { "getDhcpError", "()Ljava/lang/String;", (**void**\*) android\_net\_utils\_getDhcpError },
16. { "configureNative", "(Ljava/lang/String;IIIII)Z",  (**void** \*)android\_net\_utils\_configureInterface },
17. };

### dhcpcd

2. 第二个是 /external/dhcpcd/\*, 生成/system/bin/dhcpcd.

这个是 守护进程 dhcpcd\_eth0 会用到的工具。

最近在调android ethernet功能，android本身不带 ethernet 功能，需要打patch。这个patch可以在setting里出来 ethernet configuration 选项。即添加了用户配置IP的功能。

我打上patch之后，点击选上DHCP功能，结果路由器一直不能自动分配IP。

经检测，命令行里运行 netcfg eth0 up dhcp 时，ethernet能被正常启动，DHCP能分配到IP。

但是Setting里选上时，dhcp却不能正常分配IP。 这很费解，我先后查看了，

/system/core/libnetutils/\*

/externel/dhcpcd/\*

发现都没什么问题。

在Setting中点击turn on ethernet选项时，从log看到能调到

E/EthernetStateTracker( 185): DhcpHandler: DHCP request failed: Timed out waiting for dhcpcd to start

D/EthernetStateTracker( 185): DhcpHandler: DHCP request started

说明patch是好的能正常工作，能正常掉用dhcp，只是DHCP运行不成功。

然后我查了 getprop: 显示 init.svc.dhcpcd\_eth0 = stop

正常应该是running的，这样DHCP 才能运行成功。

最后调试了两天，才搞明白，原来是 init.rc 中 dhcpcd\_eth0 守护进程的问题：

改成：

on property:init.svc.dhcpcd\_eth0=stopped

start dhcpcd\_eth0

service dhcpcd\_eth0 /system/bin/dhcpcd -ABKL -f /system/etc/dhcpcd/dhcpcd.conf -d eth0

class main

disabled

oneshot

这样就可以了。

这时init.svc.dhcpcd\_eth0 就会是 running 了。这时再点击Setting -> ethernet configuration, DHCP就能正常分配IP了。

1. libnetutils 调用过程：

jni

=>runDhcp

=>android\_net\_utils\_runDhcp

libs/netutils/dhcp\_utils.c

=>dhcp\_do\_request

=>

static const char DAEMON\_NAME[] = "dhcpcd";

static const char DAEMON\_PROP\_NAME[] = "init.svc.dhcpcd";

static const char DHCP\_PROP\_NAME\_PREFIX[] = "dhcp";

const char \*ctrl\_prop = "ctl.start";

const char \*desired\_status = "running";

snprintf(result\_prop\_name, sizeof(result\_prop\_name), "%s.%s.result",

DHCP\_PROP\_NAME\_PREFIX,

interface);

property\_set(result\_prop\_name, "");//设置dhcp.eth0.result="";等到成功完成dhcp之后,

property\_set(ctrl\_prop, DAEMON\_NAME);//向名字为dhcpcd的service,发送"ctrl.start"启动命令字,该service在init.rc中

//init.rc中dhcpcd服务进程命令字

//service dhcpcd /system/bin/dhcpcd eth0

// disabled

// oneshot

wait\_for\_property(DAEMON\_PROP\_NAME, desired\_status, 10);

//init.c=>init进程

//=>handle\_property\_set\_fd因为是"ctrl.start"命令字,所以调用handle\_control\_message处理控制信息

//=>handle\_control\_message

//=>msg\_start

//=>

// struct service \*svc = service\_find\_by\_name(name);

// service\_start(svc);//启动svc,即执行：/system/bin/dhcpcd eth0

//=>service\_start

//=>pid = fork();

// if(pid == 0)execve(svc->args[0], (char\*\*) svc->args, (char\*\*) ENV);子进程执行execve运行/system/bin/dhcpcd,参数为eth0

//=>否则父进程,即init进程将

//=>notify\_service\_state(svc->name, "running");设置该svc的状态prop

// snprintf(pname, sizeof(pname), "init.svc.%s", name);

// property\_set(pname, state);//所以这样上面wait\_for\_property(DAEMON\_PROP\_NAME, desired\_status, 10);也才能够正常pass[luther.gliethttp].

wait\_for\_property(result\_prop\_name, NULL, 15);//等待dhcp.eth0.result=非空

2. dhcpcd 调用过程：

system/extra/dhcpcd-4.0.0-beta9/dhcpcd.c

dhcpcd

=>main

# define SYSCONFDIR "/system/etc/dhcpcd"

#define PACKAGE "dhcpcd"

# define CONFIG SYSCONFDIR "/" PACKAGE ".conf"

# define LIBEXECDIR "/system/etc/dhcpcd"

# define SCRIPT LIBEXECDIR "/" PACKAGE "-run-hooks"

=>strlcpy(options->script, SCRIPT, sizeof(options->script));//默认的options->script="/system/etc/dhcpcd /dhcpcd-run-hooks"

=>f = fopen(cf ? cf : CONFIG, "r");//如果没有指定.conf文件,那么使用默认.conf文件

=>parse\_config\_line//解析"/system/etc/dhcpcd/dhcpcd.conf"默认配置文件

=>parse\_option

=>如果在"/system/etc/dhcpcd/dhcpcd.conf"有"script"这个节

=>那么执行strlcpy(options->script, oarg, sizeof(options->script));直接拷贝

/\*

{"script", required\_argument, NULL, 'c'},

{"option", required\_argument, NULL, 'o'},

"/system/etc/dhcpcd/dhcpcd.conf"中的部分内容如下：

...

option domain\_name\_servers, domain\_name, domain\_search, host\_name

...

\*/

=>dhcp\_run

=>handle\_dhcp\_packet

=>handle\_dhcp

=>bind\_dhcp

reason = "TIMEOUT";reason = "BOUND";reason = "REBIND";reason = "RENEW";

system/extra/dhcpcd-4.0.0-beta9/configure.c

=> configure(iface, reason, state->new, state->old, &state->lease, options, 1);

//如果dhcp超时或者dhcp成功,都会调用exec\_script来执行脚本,

//执行setprop dhcp.${interface}.result "failed"或者

//执行setprop dhcp.${interface}.result "ok"

=>exec\_script(options, iface->name, reason, NULL, old);

=>然后configure\_env通过环境变量将reason传递到脚本中

int exec\_script(const struct options \*options, const char \*iface, const char \*reason,

const struct dhcp\_message \*dhcpn, const struct dhcp\_message \*dhcpo)

=>pid = fork();

=>if(pid == 0)execve(options->script, argv, env);//子进程执行脚本,默认"/system/etc/dhcpcd/dhcpcd-run-hooks"

//dhcpcd-run-hooks脚本会根据level值,决定是否执行system/etc/dhcpcd/dhcpcd-hook/\*目录下的相应文件

//我们的系统在该system/etc/dhcpcd/dhcpcd-hook/\*目录下有如下3个文件

//95-configured

//20-dns.conf

//01-test

=>父进程返回while (waitpid(pid, &status, 0) == -1)等待子进程脚本执行完成

system/extra/dhcpcd-4.0.0-beta9/dhcpcd-hooks/20-dns.conf

system/extra/dhcpcd-4.0.0-beta9/dhcpcd-hooks/95-configured

...

setprop dhcp.${interface}.ipaddress "${new\_ip\_address}"

setprop dhcp.${interface}.result "ok"//设置属性为ok

setprop dhcp.${interface}.result "failed"

## [ConnectivityService 中调用 EthernetStateTracker 和 EthernetService](http://blog.csdn.net/lamdoc/article/details/7648379)

整个 android 系统的mobile, wifi，wimax 和 bluetooth 都是通过ConnectivitySerivice 来提供服务的。

android本身不自带ethernet服务，需要新加 android-x86 里拿的ics-ethernet patch。

接下来分析 ConnectivityService 是怎么提供 ethernet 服务的。

### 创建 EthernetStateTracker 和 EthernetService

### EthernetMonitor

### ethernet-service

frameworks\opt\net\ethernet

# ConnectivityManager

其中网络的配置在frameworks\base\core\res\res\values\config.xml中。

系统对网络的判断大多都是在 ConnectivityService.java中处理的，用户操作的类是 ConnectivityManager.java 通过aidl访问

ConnectivityService.java提供的服务。而启动在SystemServer.java中：

## TASK

[Android5.0以太网流程源码情景分析](https://blog.csdn.net/tankai19880619/article/details/44559419)

<https://blog.csdn.net/martingang/article/details/8170950>

<https://my.oschina.net/blackylin/blog/89591>

android Ethernet

<https://blog.csdn.net/kangear/article/details/14446527>

<https://blog.csdn.net/yeqishi/article/details/48037499>

<http://blog.51cto.com/huamm/1554742>

# 流量统计源码分析

## 流量统计

在没有Root的情况下，Android应用流量统计在6.0之前一直没有太好的办法，官方虽然提供了TrafficStats，但其主要功能是设备启动以来流量的统计信息，和时间信息无法很好的配合。最近再看TrafficStats类时，发现说明中提到，为获取更具鲁棒性的网络历史数据，建议使用NetworkStatsManager。

本文首先简单对比下TrafficStats和NetworkStatsManager各自的限制和优缺点，然后详细说明NetworkStatsManager的用法，并给出主要代码。

### TrafficStats

Android API8提供了android.net.TrafficStats类。 通过此类能获取设备重启以来网络信息，部分函数如下所示：

1. **static** long  getMobileRxBytes()  //获取通过移动数据网络收到的字节总数static long  getMobileTxBytes()  //通过移动数据网发送的总字节数  static long  getTotalRxBytes()  //获取设备总的接收字节数 static long  getTotalTxBytes()  //获取设备总的发送字节数static long  getUidRxBytes(int uid)  //获取指定uid的接收字节数  static long  getUidTxBytes(int uid) //获取指定uid的发送字节数

通过文档及上述函数可以知道，TrafficStats能够获取设备的数据流量和总的网络流量消耗（一般情况下也就得到Wi-Fi下的流量信息）；可以查询uid对应的流量信息，而uid可以通过应用的包名查询到，因此能够查询某个应用的流量统计信息（不考虑shareuid）。非常方便的是，它的使用不需要特别的权限。另一方面它也一些限制：

（1）无法获取应用的数据流量消耗

从文档中仅能获取到指定uid的流量，但无法区分不同网络类型下的消耗

间接方法是通过监听网络切换，做好流量记录（但是要保证你的应用一直存活，且一定准确接收到网络切换信息），基本不可用。

（2）无法获取某个时间段内的流量消耗

从API文档中看，函数参数没有与时间相关的信息。而且重要的一点是，TrafficStats类中记录的是设备重启以来的流量统计信息。因为TrafficStats 类，底层还是读取/proc/net/xt\_qtaguid/stats 对内容进行解析，将得到对应的结果返回上层。

### NetworkStatsManager

在Android 6.0（API23）中新增加的类，提供网络使用历史统计信息，同时特别强调了可查询指定时间间隔内的统计信息。看看部分函数（非静态）：

**[源码]** [view plain](http://www.php361.com/index.php?c=index&a=view&id=1695)

1. //查询指定网络类型在某时间间隔内的总的流量统计信息NetworkStats.Bucket querySummaryForDevice(int networkType, String subscriberId, long startTime, long endTime) //查询某uid在指定网络类型和时间间隔内的流量统计信息NetworkStats queryDetailsForUid(int networkType, String subscriberId, long startTime, long endTime, int uid)
3. //查询指定网络类型在某时间间隔内的详细的流量统计信息（包括每个uid）NetworkStats queryDetails(int networkType, String subscriberId, long startTime, long endTime)

从上述函数和文档看，NetworkStatsManager类克服了TrafficStats的查询限制，而且统计信息也不再是设备重启以来的数据。但它也有自己的限制和缺点。   
（1）权限限制   
NetworkStatsManager的使用需要额外的权限，”android.permission.PACKAGE\_USAGE\_STATS”是系统权限，需要主动引导用户开启应用的“有权查看使用情况的应用”（使用记录访问权限）权限，后面会有代码示例。   
（2）文档不完善   
不好说是文档不全，还是我没找对。首先文档中没有给出类的实例对象的构造方法，一开始还是反射获取的，后来才发现可以通过获取系统服务方式得到。另外queryDetailsForUid函数中设置的时间间隔不太有用，没能及时的获取流量统计信息，而是有两个小时的时间间隔。还好可以在querySummary函数中获得。

代码示例

下面说说具体的使用和代码，使用前必须明确的是这里的统计信息都是在网络层以上的数据。   
1.权限设置   
（1）AndroidManifest中添加权限声明

**[源码]** [view plain](http://www.php361.com/index.php?c=index&a=view&id=1695)

1. <uses-permission android:name="android.permission.READ\_PHONE\_STATE" /><uses-permission android:name="android.permission.PACKAGE\_USAGE\_STATS" tools:ignore="ProtectedPermissions"/>

（2）代码中主动引导用户开启权限   
这里没有说明READ\_PHONE\_STATE的主动获取，大家根据自己的targetSdkVersion设置

**[源码]** [view plain](http://www.php361.com/index.php?c=index&a=view&id=1695)

1. **private** boolean hasPermissionToReadNetworkStats() {**if** (Build.VERSION.SDK\_INT < Build.VERSION\_CODES.M) {**return** true;
2. }final AppOpsManager appOps = (AppOpsManager) getSystemService(Context.APP\_OPS\_SERVICE);int mode = appOps.checkOpNoThrow(AppOpsManager.OPSTR\_GET\_USAGE\_STATS,
3. android.os.Process.myUid(), getPackageName());**if** (mode == AppOpsManager.MODE\_ALLOWED) {**return** true;
4. }
6. requestReadNetworkStats();**return** false;
7. }// 打开“有权查看使用情况的应用”页面private void requestReadNetworkStats() {
8. Intent intent = **new** Intent(Settings.ACTION\_USAGE\_ACCESS\_SETTINGS);
9. startActivity(intent);
10. }

2.查看设备和某应用的流量统计   
（1）获取NetworkStatsManager示例对象

**[源码]** [view plain](http://www.php361.com/index.php?c=index&a=view&id=1695)

1. NetworkStatsManager networkStatsManager = (NetworkStatsManager) getSystemService(NETWORK\_STATS\_SERVICE);

（2）查询设备总的流量统计信息

**[源码]** [view plain](http://www.php361.com/index.php?c=index&a=view&id=1695)

1. NetworkStats.Bucket bucket = null;// 获取到目前为止设备的Wi-Fi流量统计
2. bucket = networkStatsManager.querySummaryForDevice(ConnectivityManager.TYPE\_WIFI, "", 0, System.currentTimeMillis());Log.i("Info", "Total: " + (bucket.getRxBytes() + bucket.getTxBytes()));

（3）查询某应用（uid）的数据流量统计信息

**[源码]** [view plain](http://www.php361.com/index.php?c=index&a=view&id=1695)

1. // 获取subscriberId
2. TelephonyManager tm = (TelephonyManager) getSystemService(TELEPHONY\_SERVICE);String subId = tm.getSubscriberId();NetworkStats summaryStats;long summaryRx = 0;long summaryTx = 0;NetworkStats.Bucket summaryBucket = **new** NetworkStats.Bucket();long summaryTotal = 0;summaryStats = networkStatsManager.querySummary(ConnectivityManager.TYPE\_MOBILE, subId, getTimesMonthmorning(), System.currentTimeMillis());**do** {
3. summaryStats.getNextBucket(summaryBucket);int summaryUid = summaryBucket.getUid();**if** (uid == summaryUid) {
4. summaryRx += summaryBucket.getRxBytes();summaryTx += summaryBucket.getTxBytes();}
5. Log.i(MainActivity.**class**.getSimpleName(), "uid:" + summaryBucket.getUid() + " rx:" + summaryBucket.getRxBytes() +" tx:" + summaryBucket.getTxBytes());summaryTotal += summaryBucket.getRxBytes() + summaryBucket.getTxBytes();} **while** (summaryStats.hasNextBucket());

3.附赠实用函数   
（1）应用包名查uid

**[源码]** [view plain](http://www.php361.com/index.php?c=index&a=view&id=1695)

1. NetworkStats summaryStats;long summaryRx = 0;long summaryTx = 0;NetworkStats.Bucket summaryBucket = **new** NetworkStats.Bucket();long summaryTotal = 0;summaryStats = networkStatsManager.querySummary(ConnectivityManager.TYPE\_MOBILE, subId, getTimesMonthmorning(), System.currentTimeMillis());**do** {
2. summaryStats.getNextBucket(summaryBucket);int summaryUid = summaryBucket.getUid();**if** (uid == summaryUid) {
3. summaryRx += summaryBucket.getRxBytes();summaryTx += summaryBucket.getTxBytes();}
4. Log.i(MainActivity.**class**.getSimpleName(), "uid:" + summaryBucket.getUid() + " rx:" + summaryBucket.getRxBytes() +" tx:" + summaryBucket.getTxBytes());summaryTotal += summaryBucket.getRxBytes() + summaryBucket.getTxBytes();} **while** (summaryStats.hasNextBucket());

（2）获得本月第一天0点时间

**[源码]** [view plain](http://www.php361.com/index.php?c=index&a=view&id=1695)

1. **public** **static** long getTimesMonthMorning() {
2. Calendar cal = Calendar.getInstance();cal.set(cal.get(Calendar.YEAR), cal.get(Calendar.MONTH), cal.get(Calendar.DAY\_OF\_MONTH), 0, 0, 0);cal.set(Calendar.DAY\_OF\_MONTH, cal.getActualMinimum(Calendar.DAY\_OF\_MONTH));**return** cal.getTimeInMillis();}

4.提示无权限信息

**[源码]** [view plain](http://www.php361.com/index.php?c=index&a=view&id=1695)

1. 15:39:06.531 5276-5276/cn.arainfo.test.android.testapp1 E/AndroidRuntime: FATAL EXCEPTION: mainProcess: cn.arainfo.test.android.testapp1, PID: 5276java.lang.SecurityException: Network stats history of uid 10145 is forbidden **for** caller 10144at android.os.Parcel.readException(Parcel.java:1665)
2. at android.os.Parcel.readException(Parcel.java:1618)
3. at android.net.INetworkStatsSession$Stub$Proxy.getHistoryIntervalForUid(INetworkStatsSession.java:425)
4. at android.app.usage.NetworkStats.startHistoryEnumeration(NetworkStats.java:433)
5. at android.app.usage.NetworkStatsManager.queryDetailsForUidTag(NetworkStatsManager.java:254)
6. at android.app.usage.NetworkStatsManager.queryDetailsForUid(NetworkStatsManager.java:219)

统计测试

（1）测试设备   
小米5S Plus  Android 6.0  和 华为Mate9 Android 7.0设备上实际测试   
（2）流量差距   
实际测试流量有30M左右，和运营商流量统计相差2M左右

### REF

[Android应用流量统计——NetworkStatsManager使用](http://www.php361.com/index.php?c=index&a=view&id=1695)

## TrafficStats

Framework/base/core/java/android/net/TrafficStats.java

最后都是调用 系统 Native

核心方法：

TrafficStats 系统封装的流量统计类，适配版本 Android 2.2 以上，主要用于流量统计的方法有：

getMobileRxBytes -- 获得 mobile 的接受流量

getMobileTxBytes -- 获得 mobile 的发送流量

getTotalRxBytes -- 获得总共的接受流量（ mobile + wifi ）

getTotalTxBytes -- 获得总共的发送流量（ mobile + wifi ）

getUidRxBytes -- 获得指定 uid 的接受流量（ mobile + wifi ）

getUidTxBytes -- 获得指定 uid 的发送流量（ mobile + wifi ）

NetworkStatsManager 一个强大的流量统计工具，适配版本 android 6.0，需要系统权限才能使用，不实用

查看源码

public static long getUidRxBytes(int uid) {

// This isn't actually enforcing any security; it just returns the

// unsupported value. The real filtering is done at the kernel level.

final int callingUid = android.os.Process.myUid();

if (callingUid == android.os.Process.SYSTEM\_UID || callingUid == uid) {

return nativeGetUidStat(uid, TYPE\_RX\_BYTES);

} else {

return UNSUPPORTED;

}

}

可以看到在获取流量时，其实是在调用 nativeGetUidStat(uid, TYPE\_RX\_BYTES)，这个方法是系统 Native 方法，其实 TrafficStats 这个类所有的方法都是在调用三个 Native 方法：

private static native long nativeGetTotalStat(int type);

private static native long nativeGetIfaceStat(String iface, int type);

private static native long nativeGetUidStat(int uid, int type);

Native 方法就需要去下载 Android 源码进行查看了，在源码中找到

## android\_net\_TrafficStats.cpp

framework/base/core/jni

libandroid\_runtime.so

调试的时候只能reboot

在 android\_net\_TrafficStats.cpp 查看 gMethods 内 nativeGetUidStat 在源码中对应的方法为 getUidStat

static const JNINativeMethod gMethods[] = {

{"nativeGetTotalStat", "(I)J", (void\*) getTotalStat},

{"nativeGetIfaceStat", "(Ljava/lang/String;I)J", (void\*) getIfaceStat},

{"nativeGetUidStat", "(II)J", (void\*) getUidStat},

};

### getUidStat

查看 getUidStat 的源码，其实通过 parseUidStats(uid, &stats) 解析数据，通过 getStatsType(&stats, (StatsType) type) 读取数据

static jlong getUidStat(JNIEnv\* env, jclass clazz, jint uid, jint type) {

struct Stats stats;

memset(&stats, 0, sizeof(Stats));

if (parseUidStats(uid, &stats) == 0) {

return getStatsType(&stats, (StatsType) type);

} else {

return UNKNOWN;

}

}

### parseUidStats(uid, &stats)

解析数据源码：

static const char\* QTAGUID\_UID\_STATS = "/proc/net/xt\_qtaguid/stats";

static int parseUidStats(const uint32\_t uid, struct Stats\* stats) {

FILE \*fp = fopen(QTAGUID\_UID\_STATS, "r");

if (fp == NULL) {

return -1;

}

char buffer[384];

char iface[32];

uint32\_t idx, cur\_uid, set;

uint64\_t tag, rxBytes, rxPackets, txBytes, txPackets;

while (fgets(buffer, sizeof(buffer), fp) != NULL) {

if (sscanf(buffer,

"%" SCNu32 " %31s 0x%" SCNx64 " %u %u %" SCNu64 " %" SCNu64

" %" SCNu64 " %" SCNu64 "",

&idx, iface, &tag, &cur\_uid, &set, &rxBytes, &rxPackets,

&txBytes, &txPackets) == 9) {

if (uid == cur\_uid && tag == 0L) {

stats->rxBytes += rxBytes;

stats->rxPackets += rxPackets;

stats->txBytes += txBytes;

stats->txPackets += txPackets;

}

}

}

if (fclose(fp) != 0) {

return -1;

}

return 0;

}

可以看到源码其实就是打开 "/proc/net/xt\_qtaguid/stats" 按照你需要的字段进行读取数据，看来最重要的就是这个 "/proc/net/xt\_qtaguid/stats" 文件了，那就在读取一下这个文件看一下内容是什么，直接读取文件内容如下：

idx iface acct\_tag\_hex uid\_tag\_int cnt\_set rx\_bytes rx\_packets tx\_bytes tx\_packets rx\_tcp\_bytes rx\_tcp\_packets rx\_udp\_bytes rx\_udp\_packets rx\_other\_bytes rx\_other\_packets tx\_tcp\_bytes tx\_tcp\_packets tx\_udp\_bytes tx\_udp\_packets tx\_other\_bytes tx\_other\_packets

2 rmnet\_data0 0x0 10224 0 98175 208 33887 266 98175 208 0 0 0 0 33887 266 0 0 0 0

3 rmnet\_data0 0x0 10224 1 78165 148 29214 143 78165 148 0 0 0 0 29214 143 0 0 0 0

4 wlan0 0x0 10224 0 0 0 1560 26 0 0 0 0 0 0 1560 26 0 0 0 0,

5 wlan0 0x0 10224 1 573000 629 56692 499 573000 629 0 0 0 0 56692 499 0 0 0 0

可以看到，第一行数据是表头数据，剩下的就是对应的流量数据，还可以通过 iface 进行区分 net 还是 wlan ，这正是我们需要的数据。

# Shell命令

## 查看网络

255|root@gl300k:/ # busybox ifconfig

lo Link encap:Local Loopback

inet addr:127.0.0.1 Mask:255.0.0.0

UP LOOPBACK RUNNING MTU:65536 Metric:1

RX packets:374338 errors:0 dropped:0 overruns:0 frame:0

TX packets:374338 errors:0 dropped:0 overruns:0 carrier

collisions:0 txqueuelen:0

RX bytes:52957848 (50.5 MiB) TX bytes:52957848 (50.5 M

usb0 Link encap:Ethernet HWaddr 7E:5B:B8:0A:19:13

inet addr:192.168.42.8 Bcast:192.168.42.255 Mask:255.

UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1

RX packets:42233 errors:0 dropped:0 overruns:0 frame:0

TX packets:47576 errors:0 dropped:0 overruns:0 carrier:

collisions:0 txqueuelen:1000

RX bytes:3967845 (3.7 MiB) TX bytes:5348769 (5.1 MiB)

## 修改android系统的dns

iptables -t nat -A OUTPUT -p udp --dport 53 -j DNAT --to-destination 10.81.0.158

setprop net.dns1 10.81.0.158

setprop net.dns2 10.81.0.158

setprop dhcp.wlan0.dns1 10.81.0.158

setprop dhcp.wlan0.dns2 10.81.0.158

root@ag406:/ # tcpdump

# 实战

## 隐藏ethernet

需求描述：内部芯片之间用了ethernet通信，不想把这个开放给用户看到，造成误解

解决方案：

|  |  |  |
| --- | --- | --- |
| 系统应用层 | QS/Settings流量统计 |  |
| 脚本命令层 | 查询网口，这个其实可以不用处理 |  |
| Framework层 | 应用程序api调用 |  |

parseUidStats

### Framework层方案

ConnectivityManager connectMgr = (ConnectivityManager) context.getSystemService(Context.CONNECTIVITY\_SERVICE);

NetworkInfo mobNetInfo = connectMgr.getNetworkInfo(ConnectivityManager.TYPE\_MOBILE);

public static final int TYPE\_ETHERNET = 9;

ConnectivityManager getActiveNetworkInfo isConnected

系统应用层方案

### 快捷菜单方案

Ui显示是否有个hide\_in\_qs

### 状态栏方案

Hide\_in\_stusbar

Wifi流量统计原理

## 多网络配置

关键字：ap网速低，ethernet

指定路由：api指定路由

Iptable

[net.change]: [net.qtaguid\_enabled]

[net.dns1]: [8.8.8.8]

[net.dns2]: [8.8.4.4]

改为

[net.change]: [net.dns2]

[net.dns1]: [2409:8899:85e0:bc6b::31]

[net.dns2]: [192.168.43.1]

直接设置不work的。。

setprop net.dns1 2409:8899:85e0:bc6b::31

setprop net.dns2 192.168.43.1

学习

1860连上了网络，安全性如何保证？

不支持网卡

### AP/thernet 适配android 以太网和wifi共存

首先必须解决一个网络起来后另一个网络被踢掉的问题，在网络的核心类ConnectivityService.java找到了调用

tcpdump -i wlan0 –v

把nai.asyncChannel.disconnect()函数注释后， 另一个网络不会被踢掉

两个网络可以共存，通过命令dumpsys connectivity，可以看到当前正在使用的网络

通过命令 ip ru 查看当前的路由表,

也可以通过netcfg来查看，通过这个方法有时不准确，最好不要使用

可以看到 wlan0 和以太网是同时存在的，功能已经基本上实现

但是这样存在问题,当wifi先打开的时候，以太网不能连接，通过跟代码发现。EthernetNetworkFactory.java是核心类，这里不会被调用

[Android中设置Ethernet为默认网络类型](https://blog.csdn.net/kangear/article/details/14446527)

[android 以太网和wifi共存](https://blog.csdn.net/qq_32072451/article/details/73826030)

# android网络的评分机制

android下可以有多种网络存在，如：wifi、mobile network、ethernet、bt-pan。而对于上层应用来说，只会看到一个连通的网络，在多个网络同时存在的情况下，android就需要一套评分机制来选择一个当前使用的网络，当那个网络的分值高时，就优先使用那个网络。

Android下各种网络的分值在NetworkAgentInfo.java中管理，保存在currentScore中，各种网络初始化时会设置自己的分值。

Wifi初始分值为60（WifiStateMachine.java）；   
Ethernet初始分值为70（EthernetNetworkFactory.java）；   
Mobile network初始分值为50（DataConnection.java）；   
bt-pan初始分值为69（BluetoothTetheringNetworkFactory.java）：   
bt-pan的分值比wifi还高，这比较奇怪，已知的bt-pan网速都比较慢，google出于什么原因设计成这样？就不清楚了。

在实际运行中，还会根据网络的实时状态调整分值。

## 概括

**[html]** [view plain](https://blog.csdn.net/u010164190/article/details/78920552) [copy](https://blog.csdn.net/u010164190/article/details/78920552)

1. Android可以支持：移动数据网络、WIFI、蓝牙、网线等，这些连接本身都可以独立使用，连接管理通过一个评分机制来实现不同接入方式的选择。具体来说就是，每一种上网方式在初始化时，都向ConnectivityService标明自己网络的分值(比如数据连接50，WIFI60，蓝牙69，网线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。而当当前网络被断开时，就寻找当前就绪的其他网络连接，选取分值高的进行接入。并且，每一个网络接入时，都会进行有效性检测，如果检测不通过，将会被扣掉一定分数，此时该网络的优先级也会相应的下降。以下是Ethernet和Wifi连接管理评分的更新流程。
3. 【1】frameworks/opt/net/ethernet/java/com/android/server/ethernet/EthernetNetworkFactory.java
4. private static final int NETWORK\_SCORE = 70;//以太网
5. 【2】frameworks/base/core/java/android/net/NetworkAgent.java
6. public static final int WIFI\_BASE\_SCORE = 60;
7. 【3】packages/apps/Bluetooth/src/com/android/bluetooth/pan/BluetoothTetheringNetworkFactory.java
8. private static final int NETWORK\_SCORE = 69;//蓝牙
9. 【4】frameworks/opt/telephony/src/java/com/android/internal/telephony/dataconnection/TelephonyNetworkFactory.java
10. private final static int TELEPHONY\_NETWORK\_SCORE = 50;//移动数据
11. public TelephonyNetworkFactory(){
12. setScoreFilter(TELEPHONY\_NETWORK\_SCORE);
13. }
15. 1.Ethernet的连接评分
16. **<1>**.frameworks/opt/net/ethernet/java/com/android/server/ethernet/EthernetNetworkFactory.java
17. public void updateAgent() {
18. private static final int NETWORK\_SCORE = 70;//以太网的初始评分,分数越高，优先连接
19. // never set the network score below 0.
20. mNetworkAgent.sendNetworkScore(mLinkUp? NETWORK\_SCORE : 0);
21. }
23. **<2>**.Wifi的连接评分
24. frameworks/opt/net/wifi/service/java/com/android/server/wifi/WifiScoreReport.java
25. public static WifiScoreReport calculateScore(){
26. public static final int WIFI\_BASE\_SCORE = 60;//Wifi的初始评分
27. networkAgent.sendNetworkScore(score);
28. }
30. 2.frameworks/base/core/java/android/net/NetworkAgent.java
31. public void sendNetworkScore(int score) {
32. queueOrSendMessage(EVENT\_NETWORK\_SCORE\_CHANGED, new Integer(score));
33. }
35. 3.frameworks/base/services/core/java/com/android/server/ConnectivityService.java
36. **<1>**.case NetworkAgent.EVENT\_NETWORK\_SCORE\_CHANGED: {
37. updateNetworkScore(nai, score.intValue());
38. }
40. **<2>**.private void updateNetworkScore(NetworkAgentInfo nai, int score) {
41. final int oldScore = nai.getCurrentScore();
42. log("OldScore = " + OldScore + " newScore = " + String.valueOf(score) );
43. }

## [NetworkFactory](https://blog.csdn.net/u014386544/article/details/53706828)

我们先来看一下该对象的属性：

**[java]** [view plain](http://blog.csdn.net/u010961631/article/details/48971431) [copy](http://blog.csdn.net/u010961631/article/details/48971431)

1. @NetworkFactory.java
2. /\*\*
3. \* A NetworkFactory is an entity that creates NetworkAgent objects.
4. \* The bearers register with ConnectivityService using {@link #register} and
5. \* their factory will start receiving scored NetworkRequests.  NetworkRequests
6. \* can be filtered 3 ways: by NetworkCapabilities, by score and more complexly by
7. \* overridden function.  All of these can be dynamic - changing NetworkCapabilities
8. \* or score forces re-evaluation of all current requests.
9. \* @hide
10. \*\*/
11. **public** **class** NetworkFactory **extends** Handler {}

他的本质是一个Handler类。  
        然后我们来简单介绍一下该类提供的几个重要方法：

**[java]** [view plain](http://blog.csdn.net/u010961631/article/details/48971431) [copy](http://blog.csdn.net/u010961631/article/details/48971431)

1. //将当前网络注册到ConnectivityService
2. **public** **void** register() { }
3. //处理网络请求，用于打开或者释放当前连接
4. **private** **void** handleAddRequest(NetworkRequest request, **int** score) {}
5. //更新当前网络的分值
6. **public** **void** setScoreFilter(**int** score) {}

        以上三个是最重要的方法，在接下来的分析中将会多次看到他们的调用。还有几个比较特殊的方法：

**[java]** [view plain](http://blog.csdn.net/u010961631/article/details/48971431) [copy](http://blog.csdn.net/u010961631/article/details/48971431)

1. **protected** **void** startNetwork() { }
2. **protected** **void** stopNetwork() { }
3. **protected** **void** needNetworkFor(NetworkRequest networkRequest, **int** score) { }
4. **protected** **void** releaseNetworkFor(NetworkRequest networkRequest) { }

        这些方法都是protected属性，他们的作用就是在评分后，决定当前网络被激活或者释放，因此**一般都会在子类中被覆盖**。

        属性和方法介绍到这里，下面介绍该对象的使用

### NetworkFactory在WIFI连接中的初始化过程

### NetworkFactory的注册过程

上面两个连接在初始化NetworkFactory过程中，最后都要将其注册到ConnectivityService中，下面我们就详细介绍一下该注册过程。  
        虽然注册的过程是向ConnectivityManager申请的，但是从前一节《[Framework中的连接管理机制](http://blog.csdn.net/u010961631/article/details/48629601)》中我们知道，ConnectivityManager将会把相应请求交给ConnectivityService来处理，对于注册的申请，也是传递到ConnectivityService中的registerNetworkFactory()来处理的：

**[java]** [view plain](http://blog.csdn.net/u010961631/article/details/48971431) [copy](http://blog.csdn.net/u010961631/article/details/48971431)

1. @ConnectivityService.java
2. **public** **void** registerNetworkFactory(Messenger messenger, String name) {
3. enforceConnectivityInternalPermission();
4. //NetworkFactory在ConnectivityService内部是以NetworkFactoryInfo形式存在的
5. NetworkFactoryInfo nfi = **new** NetworkFactoryInfo(name, messenger, **new** AsyncChannel());
6. mHandler.sendMessage(mHandler.obtainMessage(EVENT\_REGISTER\_NETWORK\_FACTORY, nfi));
7. }

        从这里我们看到，在ConnectivityService中，利用当前的NetworkFactory创建NetworkFactoryInfo对象，该对象保存了当前网络连接的name、Messenger对象，并且为其创建了AsyncChannel。  
        接下来继续看注册过程：

**[java]** [view plain](http://blog.csdn.net/u010961631/article/details/48971431) [copy](http://blog.csdn.net/u010961631/article/details/48971431)

1. **private** **class** InternalHandler **extends** Handler {
2. **public** **void** handleMessage(Message msg) {
3. NetworkInfo info;
4. **switch** (msg.what) {
5. **case** EVENT\_REGISTER\_NETWORK\_FACTORY: {
6. handleRegisterNetworkFactory((NetworkFactoryInfo)msg.obj);
7. **break**;
8. }
9. }
10. }
11. }

        继续：

**[java]** [view plain](http://blog.csdn.net/u010961631/article/details/48971431) [copy](http://blog.csdn.net/u010961631/article/details/48971431)

1. **private** **void** handleRegisterNetworkFactory(NetworkFactoryInfo nfi) {
2. mNetworkFactoryInfos.put(nfi.messenger, nfi);
3. nfi.asyncChannel.connect(mContext, mTrackerHandler, nfi.messenger);
4. }

        到这里我们发现，**ConnectivityService将当前的NetworkFactoryInfo保存到mNetworkFactoryInfos**中，然后向当前的NetworkFactory发起AsyncChannel**单向通道**申请([AsyncChannel相关介绍见这里](http://blog.csdn.net/u010961631/article/details/48179305))。  
        申请成功时，将会向当前的mTrackerHandler发送CMD\_CHANNEL\_HALF\_CONNECTED消息：

**[java]** [view plain](http://blog.csdn.net/u010961631/article/details/48971431) [copy](http://blog.csdn.net/u010961631/article/details/48971431)

1. @ConnectivityService.java
2. **private** **class** NetworkStateTrackerHandler **extends** Handler {
3. **public** **void** handleMessage(Message msg) {
4. NetworkInfo info;
5. **switch** (msg.what) {
6. **case** AsyncChannel.CMD\_CHANNEL\_HALF\_CONNECTED: {
7. handleAsyncChannelHalfConnect(msg);
8. **break**;
9. }
10. }
11. }
12. }

        处理单向连接：

**[java]** [view plain](http://blog.csdn.net/u010961631/article/details/48971431) [copy](http://blog.csdn.net/u010961631/article/details/48971431)

1. **private** **void** handleAsyncChannelHalfConnect(Message msg) {
2. AsyncChannel ac = (AsyncChannel) msg.obj;
3. **if** (mNetworkFactoryInfos.containsKey(msg.replyTo)) {
4. **if** (msg.arg1 == AsyncChannel.STATUS\_SUCCESSFUL) {
5. **for** (NetworkRequestInfo nri : mNetworkRequests.values()) {
6. **if** (nri.isRequest == **false**) **continue**;
7. NetworkAgentInfo nai = mNetworkForRequestId.get(nri.request.requestId);
8. ac.sendMessage(android.net.NetworkFactory.CMD\_REQUEST\_NETWORK, (nai != **null** ? nai.getCurrentScore() : 0), 0, nri.request);
9. }
10. } **else** {
11. }
12. } **else** **if** (mNetworkAgentInfos.containsKey(msg.replyTo)) {
13. }
14. }

        在这里，ConnectivityService通过AsyncChannel通道向当前的NetworkFactory发起CMD\_REQUEST\_NETWORK的请求，需要注意的是，该请求所附带的第二个参数选择，由于当前处于初始化阶段，因此当前的mNetworkForRequestId中为空，也就是说此时传递的第二个参数必然为0。  
        我们接下来看NetworkFactory收到该请求时的处理：

**[java]** [view plain](http://blog.csdn.net/u010961631/article/details/48971431) [copy](http://blog.csdn.net/u010961631/article/details/48971431)

1. @NetworkFactory.java
2. **public** **void** handleMessage(Message msg) {
3. **switch** (msg.what) {
4. **case** CMD\_REQUEST\_NETWORK: {
5. handleAddRequest((NetworkRequest)msg.obj, msg.arg1);
6. **break**;
7. }
8. }
9. }

        继续：

**[java]** [view plain](http://blog.csdn.net/u010961631/article/details/48971431) [copy](http://blog.csdn.net/u010961631/article/details/48971431)

1. **private** **void** handleAddRequest(NetworkRequest request, **int** score) {
2. NetworkRequestInfo n = mNetworkRequests.get(request.requestId);
3. **if** (n == **null**) {
4. n = **new** NetworkRequestInfo(request, score);
5. mNetworkRequests.put(n.request.requestId, n);
6. } **else** {
7. n.score = score;
8. }
9. evalRequest(n);
10. }

        在这里由于是第一次接收到CMD\_REQUEST\_NETWORK的请求，因此将会在NetworkFactory中创建NetworkRequestInfo的对象，然后进入网络评价过程：

**[java]** [view plain](http://blog.csdn.net/u010961631/article/details/48971431) [copy](http://blog.csdn.net/u010961631/article/details/48971431)

1. **private** **void** evalRequest(NetworkRequestInfo n) {
2. **if** (n.requested == **false** && n.score < mScore && n.request.networkCapabilities.satisfiedByNetworkCapabilities( mCapabilityFilter) && acceptRequest(n.request, n.score)) {
3. needNetworkFor(n.request, n.score);
4. n.requested = **true**;
5. } **else** **if** (n.requested == **true** && (n.score > mScore || n.request.networkCapabilities.satisfiedByNetworkCapabilities( mCapabilityFilter) == **false** || acceptRequest(n.request, n.score) == **false**)) {
6. releaseNetworkFor(n.request);
7. n.requested = **false**;
8. }
9. }

        该逻辑就是整个网络评价系统最关键的地方，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=50，因此此时的判定状态将会是：n.requested=false AND n.score < mScore。  
        也就是说，对于数据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。  
        至此，NetworkFactory注册流程结束。

在WIFI检测到系统初始化完毕的时候，将会创建自己的NetworkFactory并向ConnectivityService注册：

**[java]** [view plain](http://blog.csdn.net/u010961631/article/details/48971431) [copy](http://blog.csdn.net/u010961631/article/details/48971431)

1. @WifiStateMachine.java
2. **class** DefaultState **extends** State {
3. @Override
4. **public** **boolean** processMessage(Message message) {
5. **switch** (message.what) {
6. **case** CMD\_BOOT\_COMPLETED:
7. String countryCode = mPersistedCountryCode;
8. checkAndSetConnectivityInstance();
9. //创建WIFI的NetworkFactory子类
10. mNetworkFactory = **new** WifiNetworkFactory(getHandler().getLooper(), mContext, NETWORKTYPE, mNetworkCapabilitiesFilter);
11. //设置WIFI网络分数为60
12. mNetworkFactory.setScoreFilter(60);
13. //注册
14. mCm.registerNetworkFactory(**new** Messenger(mNetworkFactory), NETWORKTYPE);
15. **break**;
16. **default**:
17. **break**;
18. }
19. **return** HANDLED;
20. }
21. }

在开机时，各个提供网络连接的对象需要向ConnectivityService注册自己，并把自己所提供的网络的分值告诉ConnectivityService。

为了ConnectivityService便于统一管理，每一个具备提供网络服务的对象都需要创建一个NetworkFactory的子类对象，并利用该对象注册自己，以及提供自己的分值。

ethernet根据网卡的up和down状态，把分值设置为70（NETWORK\_SCORE）或0。

ethernet根据网卡的up和down状态，把分值设置为70（NETWORK\_SCORE）或0。

（EthernetNetworkFactory.java）

mNetworkAgent.sendNetworkScore(mLinkUp? NETWORK\_SCORE : 0);

而wifi的分值还跟信号状态、当前数据速率等一系列因素有关：   
Wifi的分值计算在WifiStateMachine.java的calculateWifiScore函数中进行，初始计算的基础分值为：int score = 56;根据wifi网络的状态，进行小的加减，最后，如果分值大于60（NetworkAgent.WIFI\_BASE\_SCORE），就把分值设置为60。   
上面设置的分值计算，只考虑网络是否连接好，至于连接的网络是否能连接上internet，还没加入考虑。如wifi已经连接上ap，而该ap是否能连接上internet，就没在这里考虑。   
上面设置的网络分值，是最终保存在NetworkAgentInfo类中的分值，而在获取网络分值时，还会根据网络是否连接上internet，是否用户指定使用的网络，返回经过计算后的分值。

public int getCurrentScore() {

// TODO: We may want to refactor this into a NetworkScore class that takes a base score from

// the NetworkAgent and signals from the NetworkAgent and uses those signals to modify the

// score. The NetworkScore class would provide a nice place to centralize score constants

// so they are not scattered about the transports.

int score = currentScore;

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

if (score < 0) score = 0;

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

return score;

}

如果需要根据网络是否连通internet，就进行if (!everValidated && !pretendValidated) score -= UNVALIDATED\_SCORE\_PENALTY（40）;处理，当网络与internet不通时，分值减去40。如果是用户指定使用的网络，直接返回分值if (networkMisc.explicitlySelected) score = EXPLICITLY\_SELECTED\_NETWORK\_SCORE;（100）。

pretendValidated参数确定是否认为当前网络就是与internet连通的。everValidated表示当前网络与internet是否连通的标志。networkMisc.explicitlySelected为用户是否指定使用当前网络的标志，在用户手动连接ap的时候，该标志就会被设置，所以这时候的分值比ethernet还高，就会优先选择wifi作为首选网络。但在开关wifi后，自动连接上ap时，该标志就不会设置。

最后，分析一下everValidated标志是由哪里设置的，这里以wifi作为例子分析。

在连接wifi的过程中，当WifiStateMachine进入L2ConnectedState时，就会创建：

mNetworkAgent = new WifiNetworkAgent(getHandler().getLooper(), mContext,

"WifiNetworkAgent", mNetworkInfo, mNetworkCapabilitiesFilter,

mLinkProperties, 60);

在WifiNetworkAgent初始化时，把everValidated设置为false，而当网络断开连接时，就会注销WifiNetworkAgent：

if (mNetworkAgent != null) {

mNetworkAgent.sendNetworkInfo(mNetworkInfo);

mNetworkAgent = null;

}

所以在每次连接网络后，都会重新设置everValidated，断开网络时就会清除。

创建WifiNetworkAgent时，在WifiNetworkAgent内创建了NetworkMonitor， NetworkMonitor就是一个检测网络的状态机，状态机包含下面状态，初始状态为mDefaultState，检测网络是否与internet连通就是在该状态机中实现。

addState(mDefaultState);

addState(mOfflineState, mDefaultState);

addState(mValidatedState, mDefaultState);

addState(mMaybeNotifyState, mDefaultState);

addState(mEvaluatingState, mMaybeNotifyState);

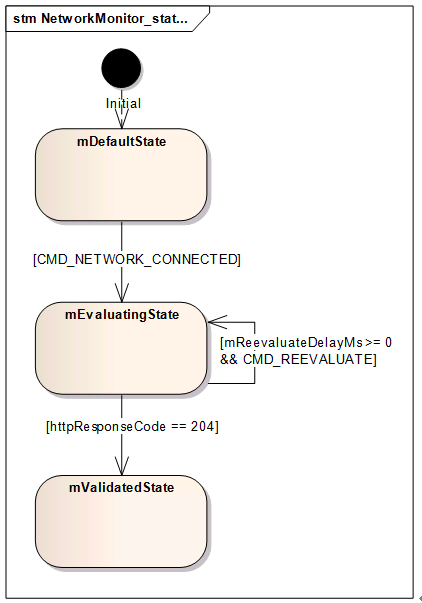
addState(mCaptivePortalState, mMaybeNotifyState);

addState(mLingeringState, mDefaultState);

setInitialState(mDefaultState);

## 当网络与internet连通时

NetworkMonitor所走的状态机如下图：



当进入mValidatedState时，就会给connectivity发送消息：

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

NETWORK\_TEST\_RESULT\_VALID, mNetwork3gTestResultIsFake,   
mNetworkAgentInfo))；   
connectivity对NETWORK\_TEST\_RESULT\_VALID消息进行处理时，就会设置everValidated为true

检测网络是否连通的代码如下：

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

public boolean processMessage(Message message) {

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

switch (message.what) {

case CMD\_REEVALUATE:

if (message.arg1 != mReevaluateToken)

return HANDLED;

// Don't bother validating networks that don't satisify the default request.

// This includes:

// - VPNs which can be considered explicitly desired by the user and the

// user's desire trumps whether the network validates.

// - Networks that don't provide internet access. It's unclear how to

// validate such networks.

// - Untrusted networks. It's unsafe to prompt the user to sign-in to

// such networks and the user didn't express interest in connecting to

// such networks (an app did) so the user may be unhappily surprised when

// asked to sign-in to a network they didn't want to connect to in the

// first place. Validation could be done to adjust the network scores

// however these networks are app-requested and may not be intended for

// general usage, in which case general validation may not be an accurate

// measure of the network's quality. Only the app knows how to evaluate

// the network so don't bother validating here. Furthermore sending HTTP

// packets over the network may be undesirable, for example an extremely

// expensive metered network, or unwanted leaking of the User Agent string.

if (!mDefaultRequest.networkCapabilities.satisfiedByNetworkCapabilities(

mNetworkAgentInfo.networkCapabilities)) {

transitionTo(mValidatedState);

return HANDLED;

}

// Note: This call to isCaptivePortal() could take up to a minute. Resolving the

// server's IP addresses could hit the DNS timeout, and attempting connections

// to each of the server's several IP addresses (currently one IPv4 and one

// IPv6) could each take SOCKET\_TIMEOUT\_MS. During this time this StateMachine

// will be unresponsive. isCaptivePortal() could be executed on another Thread

// if this is found to cause problems.

int httpResponseCode = isCaptivePortal();

if (httpResponseCode == 204) {

transitionTo(mValidatedState);

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

transitionTo(mCaptivePortalState);

} else if (++mAttempt > mMaxAttempts) {

transitionTo(mOfflineState);

} else if (mReevaluateDelayMs >= 0) {

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

sendMessageDelayed(msg, mReevaluateDelayMs);

}

return HANDLED;

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

/\*\*

\* Do a URL fetch on a known server to see if we get the data we expect.

\* Returns HTTP response code.

\*/

private int isCaptivePortal() {

HttpURLConnection urlConnection = null;

int httpResponseCode = 599;

try {

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

// On networks with a PAC instead of fetching a URL that should result in a 204

// reponse, we instead simply fetch the PAC script. This is done for a few reasons:

// 1. At present our PAC code does not yet handle multiple PACs on multiple networks

// until something like https://android-review.googlesource.com/#/c/115180/ lands.

// Network.openConnection() will ignore network-specific PACs and instead fetch

// using NO\_PROXY. If a PAC is in place, the only fetch we know will succeed with

// NO\_PROXY is the fetch of the PAC itself.

// 2. To proxy the generate\_204 fetch through a PAC would require a number of things

// happen before the fetch can commence, namely:

// a) the PAC script be fetched

// b) a PAC script resolver service be fired up and resolve mServer

// Network validation could be delayed until these prerequisities are satisifed or

// could simply be left to race them. Neither is an optimal solution.

// 3. PAC scripts are sometimes used to block or restrict Internet access and may in

// fact block fetching of the generate\_204 URL which would lead to false negative

// results for network validation.

boolean fetchPac = false;

{

final ProxyInfo proxyInfo = mNetworkAgentInfo.linkProperties.getHttpProxy();

if (proxyInfo != null && !Uri.EMPTY.equals(proxyInfo.getPacFileUrl())) {

url = new URL(proxyInfo.getPacFileUrl().toString());

fetchPac = true;

}

}

if (DBG) {

log("Checking " + url.toString() + " on " +

mNetworkAgentInfo.networkInfo.getExtraInfo());

}

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

urlConnection.setInstanceFollowRedirects(fetchPac);

urlConnection.setConnectTimeout(SOCKET\_TIMEOUT\_MS);

urlConnection.setReadTimeout(SOCKET\_TIMEOUT\_MS);

urlConnection.setUseCaches(false);

// Time how long it takes to get a response to our request

long requestTimestamp = SystemClock.elapsedRealtime();

urlConnection.getInputStream();

// Time how long it takes to get a response to our request

long responseTimestamp = SystemClock.elapsedRealtime();

httpResponseCode = urlConnection.getResponseCode();

if (DBG) {

log("isCaptivePortal: ret=" + httpResponseCode +

" headers=" + urlConnection.getHeaderFields());

}

// NOTE: We may want to consider an "HTTP/1.0 204" response to be a captive

// portal. The only example of this seen so far was a captive portal. For

// the time being go with prior behavior of assuming it's not a captive

// portal. If it is considered a captive portal, a different sign-in URL

// is needed (i.e. can't browse a 204). This could be the result of an HTTP

// proxy server.

// Consider 200 response with "Content-length=0" to not be a captive portal.

// There's no point in considering this a captive portal as the user cannot

// sign-in to an empty page. Probably the result of a broken transparent proxy.

// See http://b/9972012.

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

if (DBG) log("Empty 200 response interpreted as 204 response.");

httpResponseCode = 204;

}

if (httpResponseCode == 200 && fetchPac) {

if (DBG) log("PAC fetch 200 response interpreted as 204 response.");

httpResponseCode = 204;

}

sendNetworkConditionsBroadcast(true /\* response received \*/,

httpResponseCode != 204 /\* isCaptivePortal \*/,

requestTimestamp, responseTimestamp);

} catch (IOException e) {

if (DBG) log("Probably not a portal: exception " + e);

if (httpResponseCode == 599) {

// TODO: Ping gateway and DNS server and log results.

}

} finally {

if (urlConnection != null) {

urlConnection.disconnect();

}

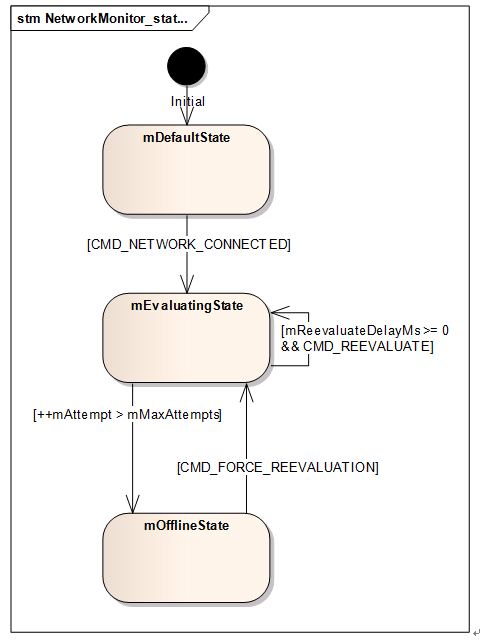
}

return httpResponseCode;

}

## 当网络与internet不连通时，

NetworkMonitor所走状态机如下：



当与internet不通时，开始进行间隔5秒（每次检测超时为10(SOCKET\_TIMEOUT\_MS)秒，总的来说就是15秒检测一次）的检测，当大于3（INITIAL\_ATTEMPTS）次检测都不通时，就会转到mOfflineState状态，并发送一个10分钟后的CMD\_FORCE\_REEVALUATION消息，10分钟后再进入mEvaluatingState状态进行检测，这次检测如果失败，就会再次进入mOfflineState状态，不停的循环检测。但有时上层会发送一个CMD\_FORCE\_REEVALUATIO消息过来，这时马上就进入mEvaluatingState状态进行检测，如果失败就继续进入mOfflineState状态并发送一个10分钟的CMD\_FORCE\_REEVALUATION消息。   
从android代码看，在进入mOfflineState状态时，会给connectivity发送NETWORK\_TEST\_RESULT\_INVALID消息，但connectivity中没有对该消息进行处理，也就是说，只要有一次检测到网络与internet连通后，everValidated设置为true，即使后面因为其他原因与internet不通了（与ap连接还是正常的），即使用户发送CMD\_FORCE\_REEVALUATIO消息进行网络检测，everValidated也不会被设置为false。   
在android中，判断网络是否连接，是通过连接google服务器是否有回应判断的，在isCaptivePortal函数中实现，所以在大陆连接wifi的时候，由于无法与google服务器通信，所以在机子看来网络也是不通的。一个明显的标志是连接大陆wifi，在状态栏的wifi图标旁边有一个“感叹号”，如果连接的wifi能与google服务器通信，wifi图标旁就不会有“感叹号”。Wifi在检测到多次不能连接google服务器后，会在/data/misc/wifi/ networkHistory.txt 中记录：   
!NO\_INTERNET\_ACCESS\_REPORTS : 1   
“VALIDATED\_INTERNET\_ACCESS: false   
当VALIDATED\_INTERNET\_ACCESS为false，numNoInternetAccessReports大于0时，wifi AP名的下面就会出现提示“未检测到任何互联网连接，因此不会自动重新连接”，这就是连接大陆wifi时，会遇到的不会回连ap的情况。

这个问题，可以通过修改代码来规避：   
如修改下面代码：

private int isCaptivePortal() {

/\* 增加代码，不配置该属性，默认为1，直接返回检测网络成功 \*/

if (SystemProperties.get("ro.isCaptivePortal", ”1”).equals("1")) return 204;

* 1
* 2
* 3

上面的修改，对于原生的检查代码，还缺少一个发送网络检测时间的广播，下面的修改，增加上广播网络回应时间。

protected int isCaptivePortal() {

if (SystemProperties.get("ro.isCaptivePortal", ”1”).equals("1")) {

/\* make fake return \*/

int fakehttpResponseCode = 204;

// Time how long it takes to get a response to our request

long fakerequestTimestamp = SystemClock.elapsedRealtime();

try

{

Thread.currentThread().sleep(200);

}

catch(Exception e){}

// Time how long it takes to get a response to our request

long fakeresponseTimestamp = SystemClock.elapsedRealtime();

sendNetworkConditionsBroadcast(true /\* response received \*/,

fakehttpResponseCode != 204 /\* isCaptivePortal \*/,

fakerequestTimestamp, fakeresponseTimestamp);

return fakehttpResponseCode;

}

## 甚至我们可以重定向检测网络的地址。

private int isCaptivePortal() {

HttpURLConnection urlConnection = null;

int httpResponseCode = 599;

try {

URL url = new URL("http", "www.baidu.com", "/more/index.html"); /\* 确保能成功获取该内容 \*/

//URL url = new URL("http", "www.baidu.com", "/img/baidu\_jgylogo3.gif"); /\* 确保能成功获取该内容 \*/

// On networks with a PAC instead of fetching a URL that should result in a 204

// reponse, we instead simply fetch the PAC script. This is done for a few reasons:

// 1. At present our PAC code does not yet handle multiple PACs on multiple networks

// until something like https://android-review.googlesource.com/#/c/115180/ lands.

// Network.openConnection() will ignore network-specific PACs and instead fetch

// using NO\_PROXY. If a PAC is in place, the only fetch we know will succeed with

// NO\_PROXY is the fetch of the PAC itself.

// 2. To proxy the generate\_204 fetch through a PAC would require a number of things

// happen before the fetch can commence, namely:

// a) the PAC script be fetched

// b) a PAC script resolver service be fired up and resolve mServer

// Network validation could be delayed until these prerequisities are satisifed or

// could simply be left to race them. Neither is an optimal solution.

// 3. PAC scripts are sometimes used to block or restrict Internet access and may in

// fact block fetching of the generate\_204 URL which would lead to false negative

// results for network validation.

boolean fetchPac = false;

{

final ProxyInfo proxyInfo = mNetworkAgentInfo.linkProperties.getHttpProxy();

if (proxyInfo != null && !Uri.EMPTY.equals(proxyInfo.getPacFileUrl())) {

url = new URL(proxyInfo.getPacFileUrl().toString());

fetchPac = true;

}

}

if (DBG) {

log("Checking " + url.toString() + " on " +

mNetworkAgentInfo.networkInfo.getExtraInfo());

}

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

urlConnection.setInstanceFollowRedirects(fetchPac);

urlConnection.setConnectTimeout(SOCKET\_TIMEOUT\_MS);

urlConnection.setReadTimeout(SOCKET\_TIMEOUT\_MS);

urlConnection.setUseCaches(false);

// Time how long it takes to get a response to our request

long requestTimestamp = SystemClock.elapsedRealtime();

urlConnection.getInputStream();

// Time how long it takes to get a response to our request

long responseTimestamp = SystemClock.elapsedRealtime();

httpResponseCode = urlConnection.getResponseCode();

if (DBG) {

log("isCaptivePortal: ret=" + httpResponseCode +

" headers=" + urlConnection.getHeaderFields());

}

// NOTE: We may want to consider an "HTTP/1.0 204" response to be a captive

// portal. The only example of this seen so far was a captive portal. For

// the time being go with prior behavior of assuming it's not a captive

// portal. If it is considered a captive portal, a different sign-in URL

// is needed (i.e. can't browse a 204). This could be the result of an HTTP

// proxy server.

// Consider 200 response with "Content-length=0" to not be a captive portal.

// There's no point in considering this a captive portal as the user cannot

// sign-in to an empty page. Probably the result of a broken transparent proxy.

// See http://b/9972012.

if (httpResponseCode == 200) {

if (DBG) log("www.baidu.com 200 response interpreted as 204 response.");

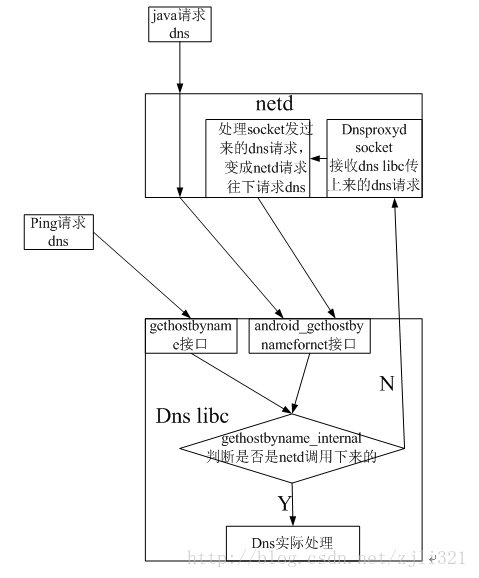
httpResponseCode = 204;

}

[**网络连接评分机制之NetworkFactory**](https://blog.csdn.net/u014386544/article/details/53706828)

# DNS原理

Android重写了libc库的dns实现   
部分，重写的dns代码在android\bionic\libc\dns目录。重写后的dns，请求dns的接口，   
有gethostbyname（兼容以前的c库接口）及android\_gethostbynamefornet（给   
android使用）接口，但根据不同的dns请求发起者，走的流程有点不一样，其中还涉及到   
了netd的处理。Android java层请求dns时，是通过netd进行，netd调用   
android\_gethostbynamefornet接口。而Android下编译的ping程序是不通过netd而直   
接调用gethostbyname接口，但最终还是要走到netd，由netd调用   
android\_gethostbynamefornet接口。Android下dns的请求流程如下图：



ethostbyname\_internal中判断是否为netd调用下来的依据是

getenv(“ANDROID\_DNS\_MODE”)返回是否不为NULL并且值为“local”，而netd启动   
时会设置setenv(“ANDROID\_DNS\_MODE”, “local”, 1)，而ping程序不会

## InetAddress

InetAddress.getHostFromNameService

java.net.InetAddress类是Java对IP地址（包括IPv4和IPv6）的高层表示。大多数其他网络类都要用到这个类，包括Socket，ServerSocket，URL，DatagramSocket，DatagramPacket等。一般地讲，它包括一个主机名和一个IP地址。

### 主机名解析

主机名到 IP 地址的解析 通过使用本地机器配置信息和网络命名服务（如域名系统（Domain Name System，DNS）和网络信息服务（Network Information Service，NIS））来实现。  
反向名称解析 意味着对于任何 IP 地址，都返回与 IP 地址关联的主机。   
InetAddress 类提供将主机名解析为其 IP 地址（或反之）的方法。

**[java]** [view plain](https://blog.csdn.net/zhangquanit/article/details/52995171) [copy](https://blog.csdn.net/zhangquanit/article/details/52995171)

1. **static** InetAddress[] getAllByName(String host)
2. 在给定主机名的情况下，根据系统上配置的名称服务返回所有的 IP 地址。
3. **static** InetAddress getByAddress(**byte**[] addr)
4. 在给定原始 IP 地址的情况下，返回 InetAddress 对象。
5. **static** InetAddress getByAddress(String host, **byte**[] addr)
6. 根据提供的主机名和 IP 地址创建 InetAddress。
7. **static** InetAddress getByName(String host)
8. 在给定主机名的情况下确定主机的 IP 地址。
9. String getCanonicalHostName()
10. 获取此 IP 地址的完全限定域名。
11. String getHostAddress()
12. 返回 IP 地址字符串，比如192.168.1.1
13. **String getHostName()**
14. **获取此 IP 地址的主机名。 比如www.baidu.com**
15. **static** InetAddress getLocalHost()
16. 返回本地主机。
17. **boolean** isReachable(**int** timeout)
18. 测试是否可以达到该地址。

### InetAddress 缓存

对于相同的域名查找，InetAddress 类具有一个缓存，用于存储成功及不成功的主机名解析。  
默认情况下，如果安装了安全管理器(security manager)，为了防止DNS欺骗攻击，会永久缓存正确的域名解析。  
如果没有安装安全管理器，默认行为将缓存一段有限（与实现相关）时间，不成功主机名解析的结果缓存非常短的时间（10 秒）以提高性能。   
如果不需要默认行为，则可以将 Java 安全属性设置为另外的 Time-to-live (TTL) 值来进行缓存。  
两个 Java 安全属性控制着用于正负主机名解析缓存的 TTL 值：

**networkaddress.cache.ttl**  
  用来设置正确的域名解析缓存时间，单位：秒， -1表示“cache forever”  
**networkaddress.cache.negative.ttl** (default: 10)  
  用来设置错误的域名解析缓存时间，单位：秒，-1表示“cache forever”，0表示“never cache”

我们来看看InetAddress缓存策略

**[java]** [view plain](https://blog.csdn.net/zhangquanit/article/details/52995171) [copy](https://blog.csdn.net/zhangquanit/article/details/52995171)

1. /\*\*
2. \* InetAddress缓存策略
3. \*/
4. **public** **final** **class** InetAddressCachePolicy {
5. **private** **static** **final** String cachePolicyProp = "networkaddress.cache.ttl";
6. **private** **static** **final** String cachePolicyPropFallback = "sun.net.inetaddr.ttl";
7. **private** **static** **final** String negativeCachePolicyProp = "networkaddress.cache.negative.ttl";
8. **private** **static** **final** String negativeCachePolicyPropFallback = "sun.net.inetaddr.negative.ttl";
9. **public** **static** **final** **int** FOREVER = -1;
10. **public** **static** **final** **int** NEVER = 0;
11. **public** **static** **final** **int** DEFAULT\_POSITIVE = 30; //正确解析 默认30秒
12. **private** **static** **int** cachePolicy = -1;
13. **private** **static** **int** negativeCachePolicy = 0; //错误解析，默认“never cache”
14. **private** **static** **boolean** propertySet;  //是否设置了正确解析的缓存时间
15. **private** **static** **boolean** propertyNegativeSet; //是否设置了错误解析的缓存时间
17. **public** InetAddressCachePolicy() {
18. }
20. //正确解析的缓存时间
21. **public** **static** **synchronized** **int** get() {
22. **return** cachePolicy;
23. }
25. //错误解析的缓存时间
26. **public** **static** **synchronized** **int** getNegative() {
27. **return** negativeCachePolicy;
28. }
30. **public** **static** **synchronized** **void** setIfNotSet(**int** var0) {
31. **if**(!propertySet) {
32. checkValue(var0, cachePolicy);
33. cachePolicy = var0;
34. }
36. }
38. **public** **static** **synchronized** **void** setNegativeIfNotSet(**int** var0) {
39. **if**(!propertyNegativeSet) {
40. negativeCachePolicy = var0;
41. }
43. }
45. **private** **static** **void** checkValue(**int** var0, **int** var1) {
46. **if**(var0 != -1) {
47. **if**(var1 == -1 || var0 < var1 || var0 < -1) {
48. **throw** **new** SecurityException("can\'t make InetAddress cache more lax");
49. }
50. }
51. }
53. **static** {
54. //获取解析成功的缓存时间
55. Integer var0 = (Integer)AccessController.doPrivileged(**new** PrivilegedAction() {
56. **public** Integer run() {
57. String var1;
58. **try** {
59. var1 = Security.getProperty("networkaddress.cache.ttl");
60. **if**(var1 != **null**) {
61. **return** Integer.valueOf(var1);
62. }
63. } **catch** (NumberFormatException var3) {
64. ;
65. }
67. **try** {
68. var1 = System.getProperty("sun.net.inetaddr.ttl");
69. **if**(var1 != **null**) {
70. **return** Integer.decode(var1);
71. }
72. } **catch** (NumberFormatException var2) {
73. ;
74. }
76. **return** **null**;
77. }
78. });
79. **if**(var0 != **null**) {
80. cachePolicy = var0.intValue();
81. **if**(cachePolicy < 0) {
82. cachePolicy = -1;
83. }
85. propertySet = **true**;
86. } **else** **if**(System.getSecurityManager() == **null**) {
87. cachePolicy = 30;
88. }
90. var0 = (Integer)AccessController.doPrivileged(**new** PrivilegedAction() {
91. **public** Integer run() {
92. String var1;
93. **try** {
94. var1 = Security.getProperty("networkaddress.cache.negative.ttl");
95. //默认10秒
96. **if**(var1 != **null**) {
97. **return** Integer.valueOf(var1);
98. }
99. } **catch** (NumberFormatException var3) {
100. ;
101. }
103. **try** {
104. var1 = System.getProperty("sun.net.inetaddr.negative.ttl");
105. **if**(var1 != **null**) {
106. **return** Integer.decode(var1);
107. }
108. } **catch** (NumberFormatException var2) {
109. ;
110. }
112. **return** **null**;
113. }
114. });
115. **if**(var0 != **null**) {
116. negativeCachePolicy = var0.intValue();
117. **if**(negativeCachePolicy < 0) {
118. negativeCachePolicy = -1;
119. }
121. propertyNegativeSet = **true**;
122. }
124. }
125. }

##### getHostName

### InetAddressImpl

//libcore/luni/

InetAddress实现有Inet4AddressImpl和Inet6AddressImpl

**[java]** [view plain](https://blog.csdn.net/zhangquanit/article/details/52995171) [copy](https://blog.csdn.net/zhangquanit/article/details/52995171)

1. **public** **class** Inet4AddressImpl **implements** InetAddressImpl{
2. **public** **native** String getLocalHostName() **throws** UnknownHostException;
3. **public** **native** java.net.InetAddress[]
4. lookupAllHostAddr(String hostname) **throws** UnknownHostException;
5. **public** **native** String getHostByAddr(**byte**[] addr) **throws** UnknownHostException;
6. **private** **native** **boolean** isReachable0(**byte**[] addr, **int** timeout, **byte**[] ifaddr, **int** ttl) **throws** IOException;
8. /\*\*
9. \* 返回  0.0.0.0/0.0.0.0
10. \*/
11. **public** **synchronized** java.net.InetAddress anyLocalAddress() {
12. **if** (anyLocalAddress == **null**) {
13. anyLocalAddress = **new** Inet4Address(); // {0x00,0x00,0x00,0x00}
14. anyLocalAddress.holder().hostName = "0.0.0.0";
15. }
16. **return** anyLocalAddress;
17. }
19. /\*\*
20. \* 返回  localhost/127.0.0.1
21. \*/
22. **public** **synchronized** java.net.InetAddress loopbackAddress() {
23. **if** (loopbackAddress == **null**) {
24. **byte**[] loopback = {0x7f,0x00,0x00,0x01}; //127.0.0.1
25. loopbackAddress = **new** Inet4Address("localhost", loopback);
26. }
27. **return** loopbackAddress;
28. }
30. **public** **boolean** isReachable(java.net.InetAddress addr, **int** timeout, NetworkInterface netif, **int** ttl) **throws** IOException {
31. **byte**[] ifaddr = **null**;
32. **if** (netif != **null**) {
33. /\*
34. \* Let's make sure we use an address of the proper family
35. \*/
36. java.util.Enumeration<java.net.InetAddress> it = netif.getInetAddresses();
37. java.net.InetAddress inetaddr = **null**;
38. **while** (!(inetaddr **instanceof** Inet4Address) &&
39. it.hasMoreElements())
40. inetaddr = it.nextElement();
41. **if** (inetaddr **instanceof** Inet4Address)
42. ifaddr = inetaddr.getAddress();
43. }
44. **return** isReachable0(addr.getAddress(), timeout, ifaddr, ttl);
45. }
46. **private** java.net.InetAddress anyLocalAddress;
47. **private** java.net.InetAddress loopbackAddress;
48. }

#### InetAddress源码

Sdf

1. **class** InetAddress **implements** java.io.Serializable {
2. **static** **final** **int** IPv4 = 1;
3. **static** **final** **int** IPv6 = 2;
4. **static** **transient** **boolean** preferIPv6Address = **false**;
6. **static** **class** InetAddressHolder {
8. String originalHostName;
10. InetAddressHolder() {}
12. InetAddressHolder(String hostName, **int** address, **int** family) {
13. **this**.originalHostName = hostName;
14. **this**.hostName = hostName;
15. **this**.address = address;
16. **this**.family = family;
17. }
19. **void** init(String hostName, **int** family) {
20. **this**.originalHostName = hostName;
21. **this**.hostName = hostName;
22. **if** (family != -1) {
23. **this**.family = family;
24. }
25. }
27. String hostName;
29. String getHostName() {
30. **return** hostName;
31. }
33. String getOriginalHostName() {
34. **return** originalHostName;
35. }
37. /\*\*
38. \* Holds a 32-bit IPv4 address.
39. \*/
40. **int** address;
42. **int** getAddress() {
43. **return** address;
44. }
46. **int** family; //IPv4=1,IPv6=2
48. **int** getFamily() {
49. **return** family;
50. }
51. }
53. /\* Used to store the serializable fields of InetAddress \*/
54. **final** **transient** java.net.InetAddress.InetAddressHolder holder;
56. java.net.InetAddress.InetAddressHolder holder() {
57. **return** holder;
58. }
60. /\* Used to store the name service provider \*/
61. **private** **static** List<NameService> nameServices = **null**;
63. /\* Used to store the best available hostname \*/
64. **private** **transient** String canonicalHostName = **null**;
66. /\*\* use serialVersionUID from JDK 1.0.2 for interoperability \*/
67. **private** **static** **final** **long** serialVersionUID = 3286316764910316507L;
69. /\*
70. \* Load net library into runtime, and perform initializations.
71. \*/
72. **static** {
73. preferIPv6Address = java.security.AccessController.doPrivileged(
74. **new** GetBooleanAction("java.net.preferIPv6Addresses")).booleanValue();
75. AccessController.doPrivileged(
76. **new** java.security.PrivilegedAction<Void>() {
77. **public** Void run() {
78. System.loadLibrary("net");
79. **return** **null**;
80. }
81. });
82. init();
83. }
85. InetAddress() {
86. holder = **new** java.net.InetAddress.InetAddressHolder();
87. }

90. **private** Object readResolve() **throws** ObjectStreamException {
91. // will replace the deserialized 'this' object
92. **return** **new** Inet4Address(holder().getHostName(), holder().getAddress());
93. }
94. **public** **boolean** isMulticastAddress() {
95. **return** **false**;
96. }
97. **public** **boolean** isAnyLocalAddress() {
98. **return** **false**;
99. }
100. **public** **boolean** isLoopbackAddress() {
101. **return** **false**;
102. }
103. **public** **boolean** isLinkLocalAddress() {
104. **return** **false**;
105. }
106. **public** **boolean** isSiteLocalAddress() {
107. **return** **false**;
108. }
109. **public** **boolean** isMCGlobal() {
110. **return** **false**;
111. }
112. **public** **boolean** isMCNodeLocal() {
113. **return** **false**;
114. }
115. **public** **boolean** isMCLinkLocal() {
116. **return** **false**;
117. }
118. **public** **boolean** isMCSiteLocal() {
119. **return** **false**;
120. }
121. **public** **boolean** isMCOrgLocal() {
122. **return** **false**;
123. }
125. **public** **boolean** isReachable(**int** timeout) **throws** IOException {
126. **return** isReachable(**null**, 0 , timeout);
127. }
129. /\*\*
130. \*  测试是否可以达到该地址。
131. \*  实现尽最大努力试图到达主机，但防火墙和服务器配置可能阻塞请求，使其在某些特定的端口可以访问时处于不可到达状态。
132. \* @param   netif   用于完成测试的 NetworkInterface；或者用于任何接口的 null
133. \* @param   ttl     要尝试的最大跳数或默认值 0
134. \* @param   timeout 调用中止前的时间（以毫秒为单位）
135. \*/
136. **public** **boolean** isReachable(NetworkInterface netif, **int** ttl,
137. **int** timeout) **throws** IOException {
138. **if** (ttl < 0)
139. **throw** **new** IllegalArgumentException("ttl can't be negative");
140. **if** (timeout < 0)
141. **throw** **new** IllegalArgumentException("timeout can't be negative");
143. **return** impl.isReachable(**this**, timeout, netif, ttl);
144. }
146. /\*\*
147. \* 获取此 IP 地址的主机名。
148. \* 如果此 InetAddress 是用host name创建的，则记忆并返回主机名；
149. \* 否则，将执行反向名称查找并基于系统配置的名称查找服务返回结果。
150. \* 如果需要查找名称服务，则调用 getCanonicalHostName。
151. \* 如果有安全管理器，则首先使用host name和 -1 作为参数来调用其 checkConnect 方法，
152. \* 以查看是否允许该操作。如果不允许该操作，则其返回 IP 地址的文本表示形式。
153. \*
154. \* @return  此 IP 地址的主机名；如果安全检查不允许操作，则返回 IP 地址的文本表示形式。
155. \*
156. \* @see java.net.InetAddress#getCanonicalHostName
157. \* @see SecurityManager#checkConnect
158. \*/
159. **public** String getHostName() {
160. **return** getHostName(**true**);
161. }
162. String getHostName(**boolean** check) {
163. //如果hostName=null，则查找并缓存hostName
164. **if** (holder().getHostName() == **null**) {
165. holder().hostName = java.net.InetAddress.getHostFromNameService(**this**, check);
166. }
167. **return** holder().getHostName();
168. }
170. /\*\*
171. \* 获取此 IP 地址的完全限定域名。根据底层系统配置可能不能返回 FQDN。
172. \*/
173. **public** String getCanonicalHostName() {
174. //canonicalHostName=null，则查找并缓存canonicalHostName
175. **if** (canonicalHostName == **null**) {
176. canonicalHostName =
177. java.net.InetAddress.getHostFromNameService(**this**, **true**);
178. }
179. **return** canonicalHostName;
180. }
181. **private** **static** String getHostFromNameService(java.net.InetAddress addr, **boolean** check) {
182. String host = **null**;
183. **for** (NameService nameService : nameServices) {
184. **try** {
185. // first lookup the hostname
186. host = nameService.getHostByAddr(addr.getAddress());
188. /\* check to see if calling code is allowed to know
189. \* the hostname for this IP address, ie, connect to the host
190. \*/
191. **if** (check) {
192. SecurityManager sec = System.getSecurityManager();
193. **if** (sec != **null**) {
194. sec.checkConnect(host, -1);
195. }
196. }
198. /\* now get all the IP addresses for this hostname,
199. \* and make sure one of them matches the original IP
200. \* address. We do this to try and prevent spoofing.
201. \*/
203. java.net.InetAddress[] arr = java.net.InetAddress.getAllByName0(host, check);
204. **boolean** ok = **false**;
206. **if**(arr != **null**) {
207. **for**(**int** i = 0; !ok && i < arr.length; i++) {
208. ok = addr.equals(arr[i]);
209. }
210. }
212. //XXX: if it looks a spoof just return the address?
213. **if** (!ok) {
214. host = addr.getHostAddress();
215. **return** host;
216. }
218. **break**;
220. } **catch** (SecurityException e) {
221. host = addr.getHostAddress();
222. **break**;
223. } **catch** (UnknownHostException e) {
224. host = addr.getHostAddress();
225. // let next provider resolve the hostname
226. }
227. }
229. **return** host;
230. }
232. /\*\*
233. \* Returns the raw IP address of this {@code InetAddress}
234. \* object. The result is in network byte order: the highest order
235. \* byte of the address is in {@code getAddress()[0]}.
236. \*
237. \* @return  the raw IP address of this object.
238. \*/
239. **public** **byte**[] getAddress() {
240. **return** **null**;
241. }
243. **public** String getHostAddress() {
244. **return** **null**;
245. }
246. **public** **int** hashCode() {
247. **return** -1;
248. }
249. **public** **boolean** equals(Object obj) {
250. **return** **false**;
251. }
253. **public** String toString() {
254. String hostName = holder().getHostName();
255. **return** ((hostName != **null**) ? hostName : "")
256. + "/" + getHostAddress();
257. }
259. //成功解析的缓存
260. **private** **static** java.net.InetAddress.Cache addressCache = **new** java.net.InetAddress.Cache(java.net.InetAddress.Cache.Type.Positive);
261. //失败解析的缓存
262. **private** **static** java.net.InetAddress.Cache negativeCache = **new** java.net.InetAddress.Cache(java.net.InetAddress.Cache.Type.Negative);
263. **private** **static** **boolean** addressCacheInit = **false**;
264. **static** java.net.InetAddress[]    unknown\_array; // put THIS in cache
265. **static** InetAddressImpl impl;
266. **private** **static** **final** HashMap<String, Void> lookupTable = **new** HashMap<>();
268. /\*\*
269. \* Represents a cache entry
270. \*/
271. **static** **final** **class** CacheEntry {
273. CacheEntry(java.net.InetAddress[] addresses, **long** expiration) {
274. **this**.addresses = addresses;
275. **this**.expiration = expiration;
276. }
278. java.net.InetAddress[] addresses;
279. **long** expiration;
280. }
282. /\*\*
283. \* A cache that manages entries based on a policy specified
284. \* at creation time.
285. \*/
286. **static** **final** **class** Cache {
287. **private** LinkedHashMap<String, java.net.InetAddress.CacheEntry> cache;
288. **private** java.net.InetAddress.Cache.Type type;
290. **enum** Type {Positive, Negative};
292. /\*\*
293. \* Create cache
294. \*/
295. **public** Cache(java.net.InetAddress.Cache.Type type) {
296. **this**.type = type;
297. cache = **new** LinkedHashMap<String, java.net.InetAddress.CacheEntry>();
298. }
300. **private** **int** getPolicy() {
301. **if** (type == java.net.InetAddress.Cache.Type.Positive) {
302. **return** InetAddressCachePolicy.get();
303. } **else** {
304. **return** InetAddressCachePolicy.getNegative();
305. }
306. }
308. /\*\*
309. \* Add an entry to the cache. If there's already an
310. \* entry then for this host then the entry will be
311. \* replaced.
312. \*/
313. **public** java.net.InetAddress.Cache put(String host, java.net.InetAddress[] addresses) {
314. **int** policy = getPolicy();
315. **if** (policy == InetAddressCachePolicy.NEVER) {
316. **return** **this**;
317. }
319. // purge any expired entries
321. **if** (policy != InetAddressCachePolicy.FOREVER) {
323. // As we iterate in insertion order we can
324. // terminate when a non-expired entry is found.
325. LinkedList<String> expired = **new** LinkedList<>();
326. **long** now = System.currentTimeMillis();
327. **for** (String key : cache.keySet()) {
328. java.net.InetAddress.CacheEntry entry = cache.get(key);
330. **if** (entry.expiration >= 0 && entry.expiration < now) {
331. expired.add(key);
332. } **else** {
333. **break**;
334. }
335. }
337. **for** (String key : expired) {
338. cache.remove(key);
339. }
340. }
342. // create new entry and add it to the cache
343. // -- as a HashMap replaces existing entries we
344. //    don't need to explicitly check if there is
345. //    already an entry for this host.
346. **long** expiration;
347. **if** (policy == InetAddressCachePolicy.FOREVER) {
348. expiration = -1;
349. } **else** {
350. expiration = System.currentTimeMillis() + (policy \* 1000);
351. }
352. java.net.InetAddress.CacheEntry entry = **new** java.net.InetAddress.CacheEntry(addresses, expiration);
353. cache.put(host, entry);
354. **return** **this**;
355. }
357. /\*\*
358. \* Query the cache for the specific host. If found then
359. \* return its CacheEntry, or null if not found.
360. \*/
361. **public** java.net.InetAddress.CacheEntry get(String host) {
362. **int** policy = getPolicy();
363. **if** (policy == InetAddressCachePolicy.NEVER) {
364. **return** **null**;
365. }
366. java.net.InetAddress.CacheEntry entry = cache.get(host);
368. // check if entry has expired
369. **if** (entry != **null** && policy != InetAddressCachePolicy.FOREVER) {
370. **if** (entry.expiration >= 0 &&
371. entry.expiration < System.currentTimeMillis()) {
372. cache.remove(host);
373. entry = **null**;
374. }
375. }
377. **return** entry;
378. }
379. }
381. /\*\*
382. \* 初始化缓存
383. \*/
384. **private** **static** **void** cacheInitIfNeeded() {
385. **assert** Thread.holdsLock(addressCache);
386. **if** (addressCacheInit) {
387. //如果已初始化缓存，则直接返回
388. **return**;
389. }
390. unknown\_array = **new** java.net.InetAddress[1];
391. unknown\_array[0] = impl.anyLocalAddress(); // 0.0.0.0/0.0.0.0
392. // 0.0.0.0 <->unknown\_array
393. addressCache.put(impl.anyLocalAddress().getHostName(),
394. unknown\_array);
396. addressCacheInit = **true**;
397. }
399. /\*\*
400. \* 添加缓存
401. \*/
402. **private** **static** **void** cacheAddresses(String hostname,
403. java.net.InetAddress[] addresses,
404. **boolean** success) {
405. hostname = hostname.toLowerCase();
406. **synchronized** (addressCache) {
407. cacheInitIfNeeded();
408. **if** (success) { //成功解析
409. addressCache.put(hostname, addresses);
410. } **else** { //失败解析
411. negativeCache.put(hostname, addresses);
412. }
413. }
414. }
416. /\*\*
417. \* 获取缓存
418. \*/
419. **private** **static** java.net.InetAddress[] getCachedAddresses(String hostname) {
420. hostname = hostname.toLowerCase();
422. // 同时查找成功和失败的解析缓存
424. **synchronized** (addressCache) {
425. cacheInitIfNeeded();
426. //先查找成功解析的缓存
427. java.net.InetAddress.CacheEntry entry = addressCache.get(hostname);
428. **if** (entry == **null**) {
429. //再查找失败解析的缓存
430. entry = negativeCache.get(hostname);
431. }
433. **if** (entry != **null**) {
434. **return** entry.addresses;
435. }
436. }
437. **return** **null**;
438. }
440. //创建NameServiceProvider
441. **private** **static** NameService createNSProvider(String provider) {
442. **if** (provider == **null**)
443. **return** **null**;
445. NameService nameService = **null**;
446. **if** (provider.equals("default")) {
447. // initialize the default name service
448. nameService = **new** NameService() {
449. **public** java.net.InetAddress[] lookupAllHostAddr(String host)
450. **throws** UnknownHostException {
451. //根据host查找ip
452. **return** impl.lookupAllHostAddr(host);
453. }
454. **public** String getHostByAddr(**byte**[] addr)
455. **throws** UnknownHostException {
456. //根据ip查找host
457. **return** impl.getHostByAddr(addr);
458. }
459. };
460. } **else** {
461. **final** String providerName = provider;
462. **try** {
463. nameService = java.security.AccessController.doPrivileged(
464. **new** java.security.PrivilegedExceptionAction<NameService>() {
465. **public** NameService run() {
466. Iterator<NameServiceDescriptor> itr =
467. ServiceLoader.load(NameServiceDescriptor.**class**)
468. .iterator();
469. **while** (itr.hasNext()) {
470. NameServiceDescriptor nsd = itr.next();
471. **if** (providerName.
472. equalsIgnoreCase(nsd.getType()+","
473. +nsd.getProviderName())) {
474. **try** {
475. **return** nsd.createNameService();
476. } **catch** (Exception e) {
477. e.printStackTrace();
478. System.err.println(
479. "Cannot create name service:"
480. +providerName+": " + e);
481. }
482. }
483. }
485. **return** **null**;
486. }
487. }
488. );
489. } **catch** (java.security.PrivilegedActionException e) {
490. }
491. }
493. **return** nameService;
494. }
496. /\*\*
497. \* 静态初始化
498. \*/
499. **static** {
500. // 创建InetAddress实现，Inet4AddressImpl还是Inet6AddressImpl
501. impl = java.net.InetAddressImplFactory.create();
503. // get name service if provided and requested
504. String provider = **null**;;
505. String propPrefix = "sun.net.spi.nameservice.provider.";
506. **int** n = 1;
507. nameServices = **new** ArrayList<NameService>();
508. provider = AccessController.doPrivileged(
509. **new** GetPropertyAction(propPrefix + n));
510. // provider = System.getProperty(propPrefix + n);
512. **while** (provider != **null**) {
513. NameService ns = createNSProvider(provider);
514. **if** (ns != **null**)
515. nameServices.add(ns);
517. n++;
518. provider = AccessController.doPrivileged(
519. **new** GetPropertyAction(propPrefix + n));
520. }
522. // if not designate any name services provider,
523. // create a default one
524. **if** (nameServices.size() == 0) {
525. NameService ns = createNSProvider("default");
526. nameServices.add(ns);
527. }
528. }
529. **public** **static** java.net.InetAddress getByAddress(String host, **byte**[] addr)
530. **throws** UnknownHostException {
531. **if** (host != **null** && host.length() > 0 && host.charAt(0) == '[') {
532. **if** (host.charAt(host.length()-1) == ']') {
533. host = host.substring(1, host.length() -1);
534. }
535. }
536. **if** (addr != **null**) {
537. **if** (addr.length == Inet4Address.INADDRSZ) {
538. **return** **new** Inet4Address(host, addr);
539. } **else** **if** (addr.length == Inet6Address.INADDRSZ) {
540. **byte**[] newAddr
541. = IPAddressUtil.convertFromIPv4MappedAddress(addr);
542. **if** (newAddr != **null**) {
543. **return** **new** Inet4Address(host, newAddr);
544. } **else** {
545. **return** **new** Inet6Address(host, addr);
546. }
547. }
548. }
549. **throw** **new** UnknownHostException("addr is of illegal length");
550. }
552. **public** **static** java.net.InetAddress getByName(String host)
553. **throws** UnknownHostException {
554. **return** java.net.InetAddress.getAllByName(host)[0];
555. }
557. // called from deployment cache manager
558. **private** **static** java.net.InetAddress getByName(String host, java.net.InetAddress reqAddr)
559. **throws** UnknownHostException {
560. **return** java.net.InetAddress.getAllByName(host, reqAddr)[0];
561. }
563. **public** **static** java.net.InetAddress[] getAllByName(String host)
564. **throws** UnknownHostException {
565. **return** getAllByName(host, **null**);
566. }
568. **private** **static** java.net.InetAddress[] getAllByName(String host, java.net.InetAddress reqAddr)
569. **throws** UnknownHostException {
571. **if** (host == **null** || host.length() == 0) {
572. java.net.InetAddress[] ret = **new** java.net.InetAddress[1];
573. ret[0] = impl.loopbackAddress(); // localhost/127.0.0.1
574. **return** ret;
575. }
577. **boolean** ipv6Expected = **false**;
578. **if** (host.charAt(0) == '[') {
579. // This is supposed to be an IPv6 literal
580. **if** (host.length() > 2 && host.charAt(host.length()-1) == ']') {
581. host = host.substring(1, host.length() -1);
582. ipv6Expected = **true**;
583. } **else** {
584. // This was supposed to be a IPv6 address, but it's not!
585. **throw** **new** UnknownHostException(host + ": invalid IPv6 address");
586. }
587. }
589. // if host is an IP address, we won't do further lookup
590. **if** (Character.digit(host.charAt(0), 16) != -1
591. || (host.charAt(0) == ':')) {
592. **byte**[] addr = **null**;
593. **int** numericZone = -1;
594. String ifname = **null**;
595. // see if it is IPv4 address
596. addr = IPAddressUtil.textToNumericFormatV4(host);
597. **if** (addr == **null**) {
598. // This is supposed to be an IPv6 literal
599. // Check if a numeric or string zone id is present
600. **int** pos;
601. **if** ((pos=host.indexOf ("%")) != -1) {
602. numericZone = checkNumericZone (host);
603. **if** (numericZone == -1) { /\* remainder of string must be an ifname \*/
604. ifname = host.substring (pos+1);
605. }
606. }
607. **if** ((addr = IPAddressUtil.textToNumericFormatV6(host)) == **null** && host.contains(":")) {
608. **throw** **new** UnknownHostException(host + ": invalid IPv6 address");
609. }
610. } **else** **if** (ipv6Expected) {
611. // Means an IPv4 litteral between brackets!
612. **throw** **new** UnknownHostException("["+host+"]");
613. }
614. java.net.InetAddress[] ret = **new** java.net.InetAddress[1];
615. **if**(addr != **null**) {
616. **if** (addr.length == Inet4Address.INADDRSZ) {
617. ret[0] = **new** Inet4Address(**null**, addr);
618. } **else** {
619. **if** (ifname != **null**) {
620. ret[0] = **new** Inet6Address(**null**, addr, ifname);
621. } **else** {
622. ret[0] = **new** Inet6Address(**null**, addr, numericZone);
623. }
624. }
625. **return** ret;
626. }
627. } **else** **if** (ipv6Expected) {
628. // We were expecting an IPv6 Litteral, but got something else
629. **throw** **new** UnknownHostException("["+host+"]");
630. }
631. **return** getAllByName0(host, reqAddr, **true**);
632. }
634. /\*\*
635. \* Returns the loopback address.
636. \* <p>
637. \* The InetAddress returned will represent the IPv4
638. \* loopback address, 127.0.0.1, or the IPv6 loopback
639. \* address, ::1. The IPv4 loopback address returned
640. \* is only one of many in the form 127.\*.\*.\*
641. \*
642. \* @return  the InetAddress loopback instance.
643. \* @since 1.7
644. \*/
645. **public** **static** java.net.InetAddress getLoopbackAddress() {
646. **return** impl.loopbackAddress();
647. }

650. /\*\*
651. \* check if the literal address string has %nn appended
652. \* returns -1 if not, or the numeric value otherwise.
653. \*
654. \* %nn may also be a string that represents the displayName of
655. \* a currently available NetworkInterface.
656. \*/
657. **private** **static** **int** checkNumericZone (String s) **throws** UnknownHostException {
658. **int** percent = s.indexOf ('%');
659. **int** slen = s.length();
660. **int** digit, zone=0;
661. **if** (percent == -1) {
662. **return** -1;
663. }
664. **for** (**int** i=percent+1; i<slen; i++) {
665. **char** c = s.charAt(i);
666. **if** (c == ']') {
667. **if** (i == percent+1) {
668. /\* empty per-cent field \*/
669. **return** -1;
670. }
671. **break**;
672. }
673. **if** ((digit = Character.digit (c, 10)) < 0) {
674. **return** -1;
675. }
676. zone = (zone \* 10) + digit;
677. }
678. **return** zone;
679. }
681. **private** **static** java.net.InetAddress[] getAllByName0 (String host)
682. **throws** UnknownHostException
683. {
684. **return** getAllByName0(host, **true**);
685. }
687. /\*\*
688. \* package private so SocketPermission can call it
689. \*/
690. **static** java.net.InetAddress[] getAllByName0 (String host, **boolean** check)
691. **throws** UnknownHostException  {
692. **return** getAllByName0 (host, **null**, check);
693. }
695. **private** **static** java.net.InetAddress[] getAllByName0 (String host, java.net.InetAddress reqAddr, **boolean** check)
696. **throws** UnknownHostException  {

699. /\*
700. \* 检测该主机是否允许连接
701. \*/
702. **if** (check) {
703. SecurityManager security = System.getSecurityManager();
704. **if** (security != **null**) {
705. security.checkConnect(host, -1);
706. }
707. }
709. //获取缓存
710. java.net.InetAddress[] addresses = getCachedAddresses(host);
712. **if** (addresses == **null**) {
713. //查找
714. addresses = getAddressesFromNameService(host, reqAddr);
715. }
717. **if** (addresses == unknown\_array)
718. **throw** **new** UnknownHostException(host);
720. **return** addresses.clone();
721. }
723. **private** **static** java.net.InetAddress[] getAddressesFromNameService(String host, java.net.InetAddress reqAddr)
724. **throws** UnknownHostException
725. {
726. java.net.InetAddress[] addresses = **null**;
727. **boolean** success = **false**;
728. UnknownHostException ex = **null**;
730. // Check whether the host is in the lookupTable.
731. // 1) If the host isn't in the lookupTable when
732. //    checkLookupTable() is called, checkLookupTable()
733. //    would add the host in the lookupTable and
734. //    return null. So we will do the lookup.
735. // 2) If the host is in the lookupTable when
736. //    checkLookupTable() is called, the current thread
737. //    would be blocked until the host is removed
738. //    from the lookupTable. Then this thread
739. //    should try to look up the addressCache.
740. //     i) if it found the addresses in the
741. //        addressCache, checkLookupTable()  would
742. //        return the addresses.
743. //     ii) if it didn't find the addresses in the
744. //         addressCache for any reason,
745. //         it should add the host in the
746. //         lookupTable and return null so the
747. //         following code would do  a lookup itself.
748. **if** ((addresses = checkLookupTable(host)) == **null**) {
749. **try** {
750. **for** (NameService nameService : nameServices) {
751. **try** {
752. addresses = nameService.lookupAllHostAddr(host);
753. success = **true**;
754. **break**;
755. } **catch** (UnknownHostException uhe) {
756. **if** (host.equalsIgnoreCase("localhost")) {
757. java.net.InetAddress[] local = **new** java.net.InetAddress[] { impl.loopbackAddress() };
758. addresses = local;
759. success = **true**;
760. **break**;
761. }
762. **else** {
763. addresses = unknown\_array;
764. success = **false**;
765. ex = uhe;
766. }
767. }
768. }
770. // More to do?
771. **if** (reqAddr != **null** && addresses.length > 1 && !addresses[0].equals(reqAddr)) {
772. // Find it?
773. **int** i = 1;
774. **for** (; i < addresses.length; i++) {
775. **if** (addresses[i].equals(reqAddr)) {
776. **break**;
777. }
778. }
779. // Rotate
780. **if** (i < addresses.length) {
781. java.net.InetAddress tmp, tmp2 = reqAddr;
782. **for** (**int** j = 0; j < i; j++) {
783. tmp = addresses[j];
784. addresses[j] = tmp2;
785. tmp2 = tmp;
786. }
787. addresses[i] = tmp2;
788. }
789. }
791. // 加入缓存中
792. cacheAddresses(host, addresses, success);
794. **if** (!success && ex != **null**)
795. **throw** ex;
797. } **finally** {
798. updateLookupTable(host);
799. }
800. }
802. **return** addresses;
803. }

806. **private** **static** java.net.InetAddress[] checkLookupTable(String host) {
807. **synchronized** (lookupTable) {
808. // If the host isn't in the lookupTable, add it in the
809. // lookuptable and return null. The caller should do
810. // the lookup.
811. **if** (lookupTable.containsKey(host) == **false**) {
812. lookupTable.put(host, **null**);
813. **return** **null**;
814. }
816. // If the host is in the lookupTable, it means that another
817. // thread is trying to look up the addresses of this host.
818. // This thread should wait.
819. **while** (lookupTable.containsKey(host)) {
820. **try** {
821. lookupTable.wait();
822. } **catch** (InterruptedException e) {
823. }
824. }
825. }
827. // The other thread has finished looking up the addresses of
828. // the host. This thread should retry to get the addresses
829. // from the addressCache. If it doesn't get the addresses from
830. // the cache, it will try to look up the addresses itself.
831. java.net.InetAddress[] addresses = getCachedAddresses(host);
832. **if** (addresses == **null**) {
833. **synchronized** (lookupTable) {
834. lookupTable.put(host, **null**);
835. **return** **null**;
836. }
837. }
839. **return** addresses;
840. }
842. **private** **static** **void** updateLookupTable(String host) {
843. **synchronized** (lookupTable) {
844. lookupTable.remove(host);
845. lookupTable.notifyAll();
846. }
847. }
849. **public** **static** java.net.InetAddress getByAddress(**byte**[] addr)
850. **throws** UnknownHostException {
851. **return** getByAddress(**null**, addr);
852. }
854. **private** **static** java.net.InetAddress cachedLocalHost = **null**;
855. **private** **static** **long** cacheTime = 0;
856. **private** **static** **final** **long** maxCacheTime = 5000L;
857. **private** **static** **final** Object cacheLock = **new** Object();
859. **public** **static** java.net.InetAddress getLocalHost() **throws** UnknownHostException {
861. SecurityManager security = System.getSecurityManager();
862. **try** {
863. String local = impl.getLocalHostName();
865. **if** (security != **null**) {
866. security.checkConnect(local, -1);
867. }
869. **if** (local.equals("localhost")) {
870. **return** impl.loopbackAddress(); //返回  localhost/127.0.0.1
871. }
873. java.net.InetAddress ret = **null**;
874. **synchronized** (cacheLock) {
875. **long** now = System.currentTimeMillis();
876. **if** (cachedLocalHost != **null**) {
877. **if** ((now - cacheTime) < maxCacheTime) // Less than 5s old?
878. ret = cachedLocalHost;
879. **else**
880. cachedLocalHost = **null**;
881. }
883. **if** (ret == **null**) {
884. java.net.InetAddress[] localAddrs;
885. **try** {
886. localAddrs =
887. java.net.InetAddress.getAddressesFromNameService(local, **null**);
888. } **catch** (UnknownHostException uhe) {
889. // Rethrow with a more informative error message.
890. UnknownHostException uhe2 =
891. **new** UnknownHostException(local + ": " +
892. uhe.getMessage());
893. uhe2.initCause(uhe);
894. **throw** uhe2;
895. }
896. cachedLocalHost = localAddrs[0];
897. cacheTime = now;
898. ret = localAddrs[0];
899. }
900. }
901. **return** ret;
902. } **catch** (java.lang.SecurityException e) {
903. **return** impl.loopbackAddress();
904. }
905. }
907. /\*\*
908. \* Perform class load-time initializations.
909. \*/
910. **private** **static** **native** **void** init();

913. /\*
914. \* Returns the InetAddress representing anyLocalAddress
915. \* (typically 0.0.0.0 or ::0)
916. \*/
917. **static** java.net.InetAddress anyLocalAddress() {
918. **return** impl.anyLocalAddress();
919. }
921. /\*
922. \* Load and instantiate an underlying impl class
923. \*/
924. **static** InetAddressImpl loadImpl(String implName) {
925. Object impl = **null**;
927. /\*
928. \* Property "impl.prefix" will be prepended to the classname
929. \* of the implementation object we instantiate, to which we
930. \* delegate the real work (like native methods).  This
931. \* property can vary across implementations of the java.
932. \* classes.  The default is an empty String "".
933. \*/
934. String prefix = AccessController.doPrivileged(
935. **new** GetPropertyAction("impl.prefix", ""));
936. **try** {
937. impl = Class.forName("java.net." + prefix + implName).newInstance();
938. } **catch** (ClassNotFoundException e) {
939. System.err.println("Class not found: java.net." + prefix +
940. implName + ":\ncheck impl.prefix property " +
941. "in your properties file.");
942. } **catch** (InstantiationException e) {
943. System.err.println("Could not instantiate: java.net." + prefix +
944. implName + ":\ncheck impl.prefix property " +
945. "in your properties file.");
946. } **catch** (IllegalAccessException e) {
947. System.err.println("Cannot access class: java.net." + prefix +
948. implName + ":\ncheck impl.prefix property " +
949. "in your properties file.");
950. }
952. **if** (impl == **null**) {
953. **try** {
954. impl = Class.forName(implName).newInstance();
955. } **catch** (Exception e) {
956. **throw** **new** Error("System property impl.prefix incorrect");
957. }
958. }
960. **return** (InetAddressImpl) impl;
961. }
963. **private** **void** readObjectNoData (ObjectInputStream s) **throws**
964. IOException, ClassNotFoundException {
965. **if** (getClass().getClassLoader() != **null**) {
966. **throw** **new** SecurityException ("invalid address type");
967. }
968. }
970. **private** **static** **final** **long** FIELDS\_OFFSET;
971. **private** **static** **final** sun.misc.Unsafe UNSAFE;
973. **static** {
974. **try** {
975. sun.misc.Unsafe unsafe = sun.misc.Unsafe.getUnsafe();
976. FIELDS\_OFFSET = unsafe.objectFieldOffset(
977. java.net.InetAddress.**class**.getDeclaredField("holder")
978. );
979. UNSAFE = unsafe;
980. } **catch** (ReflectiveOperationException e) {
981. **throw** **new** Error(e);
982. }
983. }
985. **private** **void** readObject (ObjectInputStream s) **throws**
986. IOException, ClassNotFoundException {
987. **if** (getClass().getClassLoader() != **null**) {
988. **throw** **new** SecurityException ("invalid address type");
989. }
990. ObjectInputStream.GetField gf = s.readFields();
991. String host = (String)gf.get("hostName", **null**);
992. **int** address= gf.get("address", 0);
993. **int** family= gf.get("family", 0);
994. java.net.InetAddress.InetAddressHolder h = **new** java.net.InetAddress.InetAddressHolder(host, address, family);
995. UNSAFE.putObject(**this**, FIELDS\_OFFSET, h);
996. }
998. /\* needed because the serializable fields no longer exist \*/
1000. /\*\*
1001. \* @serialField hostName String
1002. \* @serialField address int
1003. \* @serialField family int
1004. \*/
1005. **private** **static** **final** ObjectStreamField[] serialPersistentFields = {
1006. **new** ObjectStreamField("hostName", String.**class**),
1007. **new** ObjectStreamField("address", **int**.**class**),
1008. **new** ObjectStreamField("family", **int**.**class**),
1009. };
1011. **private** **void** writeObject (ObjectOutputStream s) **throws**
1012. IOException {
1013. **if** (getClass().getClassLoader() != **null**) {
1014. **throw** **new** SecurityException ("invalid address type");
1015. }
1016. ObjectOutputStream.PutField pf = s.putFields();
1017. pf.put("hostName", holder().getHostName());
1018. pf.put("address", holder().getAddress());
1019. pf.put("family", holder().getFamily());
1020. s.writeFields();
1021. }
1022. }
1024. /\*
1025. \* Simple factory to create the impl
1026. \*/
1027. **class** InetAddressImplFactory {
1029. **static** InetAddressImpl create() {
1030. **return** java.net.InetAddress.loadImpl(isIPv6Supported() ?
1031. "Inet6AddressImpl" : "Inet4AddressImpl");
1032. }
1034. **static** **native** **boolean** isIPv6Supported();
1035. }

## 实战

05-06 22:12:35.232 19267-19419/dji.go.v4 W/System.err: at dji.thirdparty.afinal.FinalHttp.postSync(FinalHttp.java:429)

at dji.pilot2.flymonitor.service.FlyMonitorService.getRemoteCountryCode(FlyMonitorService.java:558)

at dji.pilot2.flymonitor.service.FlyMonitorService.access$600(FlyMonitorService.java:140)

at dji.pilot2.flymonitor.service.FlyMonitorService$FlyMonitorHandler.doFetchRemoteCountryCode(FlyMonitorService.java:728)

at dji.pilot2.flymonitor.service.FlyMonitorService$FlyMonitorHandler.handleMessage(FlyMonitorService.java:655)

at android.os.Handler.dispatchMessage(Handler.java:102)

at android.os.Looper.loop(Looper.java:154)

at android.os.HandlerThread.run(HandlerThread.java:61)

05-06 22:12:45.297 19267-19419/dji.go.v4 W/System.err: java.net.UnknownHostException: Unable to resolve host "dict.djiservice.org": No address associated with hostname

05-06 22:12:45.299 19267-19419/dji.go.v4 W/System.err: at java.net.Inet6AddressImpl.lookupHostByName(Inet6AddressImpl.java:125)

05-06 22:12:45.300 19267-19419/dji.go.v4 W/System.err: at java.net.Inet6AddressImpl.lookupAllHostAddr(Inet6AddressImpl.java:74)

at java.net.InetAddress.getAllByName(InetAddress.java:752)

at org.apache.http.impl.conn.DefaultClientConnectionOperator.openConnection(DefaultClientConnectionOperator.java:142)

05-06 22:12:45.301 19267-19419/dji.go.v4 W/System.err: at org.apache.http.impl.conn.AbstractPoolEntry.open(AbstractPoolEntry.java:169)

at org.apache.http.impl.conn.AbstractPooledConnAdapter.open(AbstractPooledConnAdapter.java:124)

at org.apache.http.impl.client.DefaultRequestDirector.execute(DefaultRequestDirector.java:366)

05-06 22:12:45.302 19267-19419/dji.go.v4 W/System.err: at org.apache.http.impl.client.AbstractHttpClient.execute(AbstractHttpClient.java:560)

at org.apache.http.impl.client.AbstractHttpClient.execute(AbstractHttpClient.java:492)

at dji.thirdparty.afinal.http.SyncRequestHandler.makeRequestWithRetries(SyncRequestHandler.java:51)

05-06 22:12:45.303 19267-19419/dji.go.v4 W/System.err: at dji.thirdparty.afinal.http.SyncRequestHandler.sendRequest(SyncRequestHandler.java:78)

at dji.thirdparty.afinal.FinalHttp.sendSyncRequest(FinalHttp.java:559)

at dji.thirdparty.afinal.FinalHttp.postSync(FinalHttp.java:429)

05-06 22:12:45.304 19267-19419/dji.go.v4 W/System.err: at dji.pilot2.flymonitor.service.FlyMonitorService.getRemoteCountryCode(FlyMonitorService.java:558)

at dji.pilot2.flymonitor.service.FlyMonitorService.access$600(FlyMonitorService.java:140)

at dji.pilot2.flymonitor.service.FlyMonitorService$FlyMonitorHandler.doFetchRemoteCountryCode(FlyMonitorService.java:728)

05-06 22:12:45.305 19267-19419/dji.go.v4 W/System.err: at dji.pilot2.flymonitor.service.FlyMonitorService$FlyMonitorHandler.handleMessage(FlyMonitorService.java:655)

at android.os.Handler.dispatchMessage(Handler.java:102)

at android.os.Looper.loop(Looper.java:154)

05-06 22:12:45.306 19267-19419/dji.go.v4 W/System.err: at android.os.HandlerThread.run(HandlerThread.java:61)

05-06 22:12:45.307 19267-19419/dji.go.v4 W/System.err: Caused by: android.system.GaiException: android\_getaddrinfo failed: EAI\_NODATA (No address associated with hostname)

at libcore.io.Posix.android\_getaddrinfo(Native Method)

05-06 22:12:45.308 19267-19419/dji.go.v4 W/System.err: at libcore.io.ForwardingOs.android\_getaddrinfo(ForwardingOs.java:55)

at java.net.Inet6AddressImpl.lookupHostByName(Inet6AddressImpl.java:106)

... 19 more

05-06 22:12:45.309 19267-19419/dji.go.v4 W/System.err: java.net.UnknownHostException: Unable to resolve host "dict.djiservice.org": No address associated with hostname

at java.net.Inet6AddressImpl.lookupHostByName(Inet6AddressImpl.java:125)

at java.net.Inet6AddressImpl.lookupAllHostAddr(Inet6AddressImpl.java:74)

# 客制化

修改热点名字

*<!-- Do not translate. Default access point SSID used for tethering -->*<**string name="wifi\_tether\_configure\_ssid\_default" translatable="false"**>AndroidAP</**string**>

rm500:/data/misc/wifi # cat softap.conf

修改之后，务必更新framwork-res.apk和framework.jar。如果值有偏移的话

如何兼容之前的呢，恢复出厂设置就好了！

# 核心

systemUI:(getTotalTxPackets)

Settings:

## Ref

[SoftapCmd命令](https://book.2cto.com/201405/43241.html)

[**wifi热点移植总结**](http://huangqinqin.iteye.com/blog/1286084)

[Android 4.1 Netd详细分析（一）概述与应用实例](http://blog.csdn.net/xiaokeweng/article/details/8130218)

<http://gaozhipeng.me/posts/nativedaemonconnector_source_code/>

[android网络的评分机制、连接国内ap wifi不回连问题](https://blog.csdn.net/zjli321/article/details/52424434)

# Task

[WiFi Tethering & Usb Tethering](https://blog.csdn.net/census/article/details/46639303)

[Android8.0 WIFI ap Tethering 相关知识](https://blog.csdn.net/Aaron121314/article/details/78538852)

<https://www.google.com.hk/search?safe=strict&ei=aFddWtDTNomp0AS0j4Vo&q=android+TrafficStats+%E6%BA%90%E7%A0%81&oq=android+TrafficStats+%E6%BA%90%E7%A0%81&gs_l=psy-ab.3...900205.900205.0.900580.1.1.0.0.0.0.109.109.0j1.1.0....0...1c.1.64.psy-ab..0.0.0....0.bWU8TyS4qfs>

<http://blog.csdn.net/goohong/article/details/7937148>

<http://blog.csdn.net/DaHeMaDeGeGe/article/details/51259129>

[Android系统svc命令](https://www.jianshu.com/p/d6e5cee13a83)

关机前

<http://blog.51cto.com/george0052/147365>

<https://blog.csdn.net/Fighting4344/article/details/49128645>

<https://blog.csdn.net/kangear/article/details/14446527>

<https://blog.csdn.net/qq_32072451/article/details/73826030>

<https://blog.csdn.net/tankai19880619/article/details/45099385>

<https://blog.csdn.net/wszonline/article/details/53740667>

<https://blog.csdn.net/moyu123456789/article/details/50002099>

<https://blog.csdn.net/qwertyuiop159158/article/details/52900099>

<https://blog.csdn.net/jrunw/article/details/68490969>

<http://blog.51cto.com/george0052/147365>

<https://www.cnblogs.com/ggjucheng/archive/2012/01/14/2322659.html>

https://blog.csdn.net/martingang/article/details/8170950