Android Wifi 的工作流程

一、WIFI 工作相关部分

Wifi 网卡状态

1. WIFI_STATE_DISABLED: WIFI 网卡不可用
2. WIFI_STATE_DISABLING: WIFI 正在关闭
3. WIFI_STATE_ENABLED:WIFI 网卡可用
4. WIFI_STATE_ENABLING:WIFI 网卡正在打开
5. WIFI_STATE_UNKNOWN:未知网卡状态

WIFI 访问网络需要的权限

"android.permission.CHANGE_NETWORK_STATE">

修改网络状态的权限

android:name="android.permission.CHANGE_WIFI_STATE">

修改 WIFI 状态的权限

"android.permission.ACCESS_NETWORK_STATE">

访问网络权限

"android.permission.ACCESS_WIFI_STATE">

访问 WIFI 权限

WIFI 核心模块

n WifiService

由 SystemServer 启动的时候生成的 ConnecttivityService 创建，负责启动关闭 wpa_supplicant,启动和关闭 WifiMonitor 线程，把命令下发给 wpa_supplicant 以及跟新 WIFI 的状态

n WifiMonitor

负责从 wpa_supplicant 接收事件通知

n Wpa_supplicant

1、读取配置文件

2、初始化配置参数，驱动函数

3、让驱动 scan 当前所有的 bssid

4、检查扫描的参数是否和用户设置的想否

5、如果相符，通知驱动进行权限 认证操作

6、连上 AP

n Wifi 驱动模块

厂商提供的 source,主要进行 load firmware 和 kernel 的 wireless 进行通信

n Wifi 电源管理模块

主要控制硬件的 GPIO 和上下电，让 CPU 和 Wifi 模组之间通过 sdio 接口通信

二、Wifi 工作步骤

1 Wifi 模块初期化

2 Wifi 启动

3 查找热点（AP）

4 配置 AP

5 配置 AP 参数

6 Wifi 连接

7 IP 地址配置

三、WIFI 的架构和流程

1、WIFI 的基本架构

1、wifi 用户空间的程序和库:

external/wpa_supplicant/

生成库 libwpaclient.so 和守护进程 wpa_supplicant

2、hardware/libhardware_legary/wifi/是 wifi 管理库

3、JNI 部分：

frameworks/base/core/jni/android_net_wifi_Wifi.cpp

4、JAVA 部分：

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

frameworks/base/wifi/java/android/net/wifi/

5、WIFI Settings 应用程序位于：

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

6、WIFI 驱动模块 wlan.ko

wpa_supplicant 通过 wireless_ext 接口和驱动通信

7、WIFI 硬件模块

2、WIFI 在 Android 中如何工作

Android 使用一个修改版 wpa_supplicant 作为 daemon 来控制 WIFI，代码位于 external/wpa_supplicant。wpa_supplicant 是通过 socket 与 hardware/libhardware_legacy/wifi/wifi.c 通信。UI 通过 android.net.wifi package（frameworks/base/wifi/java/android/net/wifi/）发送命令给 wifi.c。相应的 JNI 实现位于 frameworks/base/core/jni/android_net_wifi_Wifi.cpp。更高一级的网络管理位于 frameworks/base/core/java/android/net。

3、配置 Android 支持 WIFI

在 BoardConfig.mk 中添加：

BOARD_HAVE_WIFI := true

BOARD_WPA_SUPPLICANT_DRIVER := WEXT

这将在 external/wpa_supplicant/Android.mk 设置 WPA_BUILD_SUPPLICANT 为 true，默认使用驱动 driver_wext.c。

如果使用定制的 wpa_supplicant 驱动(例如 wlan0)，可以设置:

BOARD_WPA_SUPPLICANT_DRIVER := wlan0

4、使能 wpa_supplicant 调试信息

默认 wpa_supplicant 设置为 MSG_INFO，为了输出更多信息，可修改：

1、在 common.c 中设置 wpa_debug_level = MSG_DEBUG;

2、在 common.c 中把#define wpa_printf 宏中的

if ((level) >= MSG_INFO)

改为

if ((level) >= MSG_DEBUG)

5、配置 wpa_supplicant.conf

wpa_supplicant 是通过 wpa_supplicant.conf 中的 ctrl_interface=来指定控制 socket 的，应该在 AndroidBoard.mk 中配置好复制到 $(TARGET_OUT_ETC)/wifi（也就是/system/etc/wifi/wpa_supplicant.conf）这个位置会在 init.rc 中再次检测的。

一般的 wpa_supplicant.conf 配置为：

```
ctrl_interface=DIR=/data/system/wpa_supplicant GROUP=wifi
update_config=1
fast_reauth=1
```

有时，驱动需要增加：

```
ap_scan=1
```

如果遇到 AP 连接问题，需要修改 ap_scan=0 来让驱动连接，代替 wpa_supplicant。

如果要连接到 non-WPA or open wireless networks，要增加：

```
network={
        key_mgmt=NONE
}
```

6、配置路径和权限

Google 修改的 wpa_supplicant 要运行在 wifi 用户和组下的。代码可见 wpa_supplicant/os_unix.c 中的 os_program_init()函数。

如果配置不对，会出现下面错误：

```
E/WifiHW  (  ): Unable to open connection to supplicant on
"/data/system/wpa_supplicant/wlan0": No such file or directory will appear.
```

确认 init.rc 中有如下配置：

```
mkdir /system/etc/wifi 0770 wifi wifi
chmod 0770 /system/etc/wifi
chmod 0660 /system/etc/wifi/wpa_supplicant.conf
chown wifi wifi /system/etc/wifi/wpa_supplicant.conf
# wpa_supplicant socket
mkdir /data/system/wpa_supplicant 0771 wifi wifi
chmod 0771 /data/system/wpa_supplicant
#wpa_supplicant control socket for android wifi.c
mkdir /data/misc/wifi 0770 wifi wifi
mkdir /data/misc/wifi/sockets 0770 wifi wifi
chmod 0770 /data/misc/wifi
chmod 0660 /data/misc/wifi/wpa_supplicant.conf
```

如果系统的/system 目录为只读，那应该使用路径/data/misc/wifi/wpa_supplicant.conf。

7、运行 wpa_supplicant 和 dhcpcd

在 init.rc 中确保有如下语句：

```
service wpa_supplicant /system/bin/logwrapper /system/bin/wpa_supplicant -dd
        -Dwext -iwlan0 -c /data/misc/wifi/wpa_supplicant.conf
  user root
  group wifi inet
socket wpa_wlan0 dgram 660 wifi wifi
  oneshot
service dhcpcd /system/bin/logwrapper /system/bin/dhcpcd -d -B wlan0
  disabled
```

oneshot

根据所用的 WIFI 驱动名字，修改 wlan0 为自己驱动的名字。

8、编译 WIFI 驱动为 module 或 kernel built in

1、编译为 module

在 BoardConfig.mk 中添加：

WIFI_DRIVER_MODULE_PATH := "/system/lib/modules/xxxx.ko"

WIFI_DRIVER_MODULE_ARG := "" #for example nohwcrypt

WIFI_DRIVER_MODULE_NAME := "xxxx" #for example wlan0

WIFI_FIRMWARE_LOADER := ""

2、编译为 kernel built in

1）在 hardware/libhardware_legacy/wifi/wifi.c 要修改 interface 名字，

2）在 init.rc 中添加：

setprop wifi.interface "wlan0"

3）在 hardware/libhardware_legacy/wifi/wifi.c 中当 insmod/rmmod 时，

直接 return 0。

9、WIFI 需要的 firmware

Android 不使用标准的 hotplug binary，WIFI 需要的 firmware 要复制到/etc/firmware。

或者复制到 WIFI 驱动指定的位置，然后 WIFI 驱动会自动加载。

10、修改 WIFI 驱动适合 Android

Google 修改的 wpa_supplicant 要求 SIOCSIWPRIV ioctl 发送命令到驱动，及接收信息，例如 signal strength, mac address of the AP, link speed 等。所以要正确实现 WIFI 驱动，需要从 SIOCSIWPRIV ioctl 返回 RSSI (signal strength)和 MACADDR 信息。

如果没实现这个 ioctl，会出现如下错误：

E/wpa_supplicant( ): wpa_driver_priv_driver_cmd failed

wpa_driver_priv_driver_cmd RSSI len = 4096

E/wpa_supplicant( ): wpa_driver_priv_driver_cmd failed

D/wpa_supplicant( ): wpa_driver_priv_driver_cmd LINKSPEED len = 4096

E/wpa_supplicant( ): wpa_driver_priv_driver_cmd failed

I/wpa_supplicant( ): CTRL-EVENT-DRIVER-STATE HANGED

10、设置 dhcpcd.conf

一般/system/etc/dhcpcd/dhcpcd.conf 的配置为：

interface wlan0

option subnet_mask, routers, domain_name_server

四、WIFI 模块分析

Wifi 模块学习流程

最近研究 Wifi 模块，查了不少的相关资料，但发现基本上是基于 android2.0 版本的的分析，而现在研发的 android 移动平台基本上都是 2.3 的版本，跟 2.0 版本的差别，在 Wifi 模块上也是显而易见的。2.3 版本 Wifi 模块没有了 WifiLayer，之前的 WifiLayer 主要负责一些复杂的 Wifi 功能，如 AP 选择等以提供给用户自定义，而新的版本里面的这块内容基本上被 WifiSettings 所代替

本文就是基于 android2.3 版本的 Wifi 分析，主要分为两部分来分别说明：

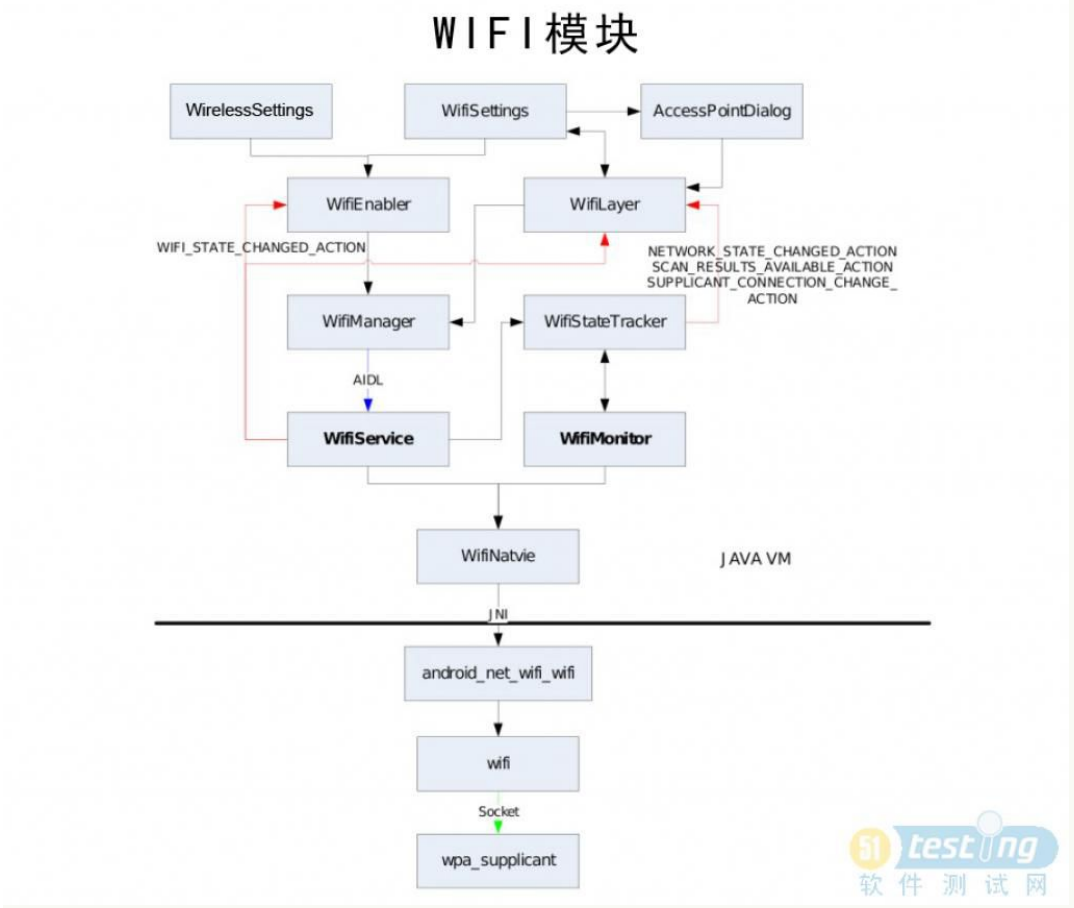
（a） Wifi 的启动流程（有代码供参考分析）

（b） Wifi 模块相关文件的解析

（c） Wpa_supplicant 解析

Awifi 的基本运行流程（针对代码而言）

首先给一张我网上 down 下来的图，针对 2.3 版本之前的，由于不怎么擅长画这些，大家也就将就点，只要能助理解就可以了



（一）初始化

a.流程

1.在 SystemServer 启动的时候会生成一个 ConnectivityService 的实例

2.ConnectivityService 的构造函数会创建 WifiService

3.WifiStateTracker 会创建 WifiMonitor 接受来自底层的事件，WifiService 和 WifiMonitor 是整个 wifi 模块的核心，WifiService 负责启动和关闭 wpa_supplicant，启动和关闭 WifiMonitor 监视线程和把命令下方给 wpa_supplicant，而 WifiMonitor 则负责从 wpa_supplicant 接受事件通知

b.代码分析

要想使用 Wifi 模块，必须首先使能 Wifi，当你第一次按下 Wifi 使能按钮时，WirelessSettings 会实例化一个 WifiEnabler 对象，实例化代码如下：

packages/apps/settings/src/com/android/settings/WirelessSettings.java

[java] view plaincopy

```java
protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);
……
                CheckBoxPreferencewifi = (CheckBoxPreference) findPreference(KEY_TOGGLE_WIFI);

                mWifiEnabler= new WifiEnabler(this, wifi);
……
}
```

WifiEnabler 类的定义大致如下，它实现了一个监听接口，当 WifiEnabler 对象被初始化后，它监听到你按键的动作，会调用响应函数 onPreferenceChange（），这个函数会调用 WifiManager 的 setWifiEnabled（）函数。

[java] view plaincopy

```java
public class WifiEnabler implementsPreference.OnPreferenceChangeListener {
……
public boolean onPreferenceChange(Preference preference,Object value) {
        booleanenable = (Boolean) value;
……
if (mWifiManager.setWifiEnabled(enable)) {
                mCheckBox.setEnabled(false);
……
}
……
}
```

我们都知道 Wifimanager 只是个服务代理，所以它会调用 WifiService 的 setWifiEnabled（）函数，而这个函数会调用 sendEnableMessage（）函数，了解 android 消息处理机制的都知道，这个函数最终会给自己发送一个 MESSAGE_ENABLE_WIFI 的消息，被 WifiService 里面定义的 handlermessage()函数处理，会调用 setWifiEnabledBlocking（）函数。下面是调用流程：

mWifiEnabler.onpreferencechange()===>mWifiManage.setWifienabled()===>mWifiService.setWifiEnabled()===>mWifiService.sendEnableMessage()
===>mWifiService.handleMessage()===>mWifiService.setWifiEnabledBlocking().
在 setWifiEnabledBlocking()函数中主要做如下工作：加载 Wifi 驱动，启动 wpa_supplicant，注册广播接收器，启动 WifiThread 监听线程。代码如下：

```
1. ……
2. if (enable) {
3.             if (!mWifiStateTracker.loadDriver()) {
4.                 Slog.e(TAG, "Failed toload Wi-Fi driver.");
5.                 setWifiEnabledState(WIFI_STATE_UNKNOWN, uid);
6.                  return false;
7.             }
8.             if (!mWifiStateTracker.startSupplicant()) {
9.                  mWifiStateTracker.unloadDriver();
10.                 Slog.e(TAG, "Failed tostart supplicant daemon.");
11.                setWifiEnabledState(WIFI_STATE_UNKNOWN, uid);
12.                 return false;
13.            }
14.            registerForBroadcasts();
15.            mWifiStateTracker.startEventLoop();
16. ……
```

至此，Wifi 使能（开启）结束，自动进入扫描阶段。

（二）扫描 AP

当驱动加载成功后,如果配置文件的 AP_SCAN = 1,扫描会自动开始,WifiMonitor 将会从 supplicant 收到一个消息 EVENT_DRIVER_STATE_CHANGED,调用 handleDriverEvent（），然后调用 mWifiStateTracker.notifyDriverStarted()，该函数向消息队列添加 EVENT_DRIVER_STATE_CHANGED,handlermessage()函数处理消息时调用 scan()函数，并通过 WifiNative 将扫描命令发送到 wpa_supplicant。
看代码 Frameworks/base/wifi/java/android/net/wifi/WifiMonitor.java
[java] view plaincopy

```
1.  private void handleDriverEvent(Stringstate) {
2.             if (state == null) {
3.                  return;
4.             }
5.             if (state.equals("STOPPED")) {
6.                 mWifiStateTracker.notifyDriverStopped();
7.             } else if (state.equals("STARTED")) {
8.                  mWifiStateTracker.notifyDriverStarted();
9.             } else if (state.equals("HANGED")) {
10.                 mWifiStateTracker.notifyDriverHung();
11.            }
12. }
```

Frameworks/base/wifi/java/android/net/wifi/WifiStateTracker.java
[java] view plaincopy

```
1. ...
2. case EVENT_DRIVER_STATE_CHANGED:
3.                 switch(msg.arg1) {
4.                 case DRIVER_STARTED:
5.
6.                     setNumAllowedChannels();
7.                    synchronized (this) {
8.                        if (mRunState == RUN_STATE_STARTING) {
9.                            mRunState = RUN_STATE_RUNNING;
10.                           if (!mIsScanOnly) {
11.                               reconnectCommand();
12.                           } else {
13.                               // In somesituations, supplicant needs to be kickstarted to
```

```
                                // start thebackground scanning
                                scan(true);
                            }
                        }
                    }
                    break;
...
```

上面是启动 Wifi 时，自动进行的 AP 的扫描，用户当然也可以手动扫描 AP，这部分实现在 WifiService 里面，WifiService 通过 startScan()接口函数发送扫描命令到 supplicant。

看代码：Frameworks/base/wifi/java/android/net/wifi/WifiStateTracker.java

[java] view plaincopy

```
public boolean startScan(booleanforceActive) {
        enforceChangePermission();
        switch (mWifiStateTracker.getSupplicantState()) {
            case DISCONNECTED:
            case INACTIVE:
            case SCANNING:
            case DORMANT:
                 break;
            default:
                mWifiStateTracker.setScanResultHandling(
                        WifiStateTracker.SUPPL_SCAN_HANDLING_LIST_ONLY);
                 break;
        }
        return mWifiStateTracker.scan(forceActive);
}
```

然后下面的流程同上面的自动扫描，我们来分析一下手动扫描从哪里开始的。我们应该知道手动扫描是通过菜单键的扫描键来响应的，而响应该动作的应该是 WifiSettings 类中 Scanner 类的 handlerMessage()函数，它调用 WifiManager 的 startScanActive()，这才调用 WifiService 的 startScan()。

如代码:packages/apps/Settings/src/com/android/settings/wifiwifisettings.java

[java] view plaincopy

```
public boolean onCreateOptionsMenu(Menu menu) {
        menu.add(Menu.NONE, MENU_ID_SCAN, 0, R.string.wifi_menu_scan)
                .setIcon(R.drawable.ic_menu_scan_network);
        menu.add(Menu.NONE, MENU_ID_ADVANCED, 0, R.string.wifi_menu_advanced)
                .setIcon(android.R.drawable.ic_menu_manage);
        return super.onCreateOptionsMenu(menu);
}
```

当按下菜单键时，WifiSettings 就会调用这个函数绘制菜单。如果选择扫描按钮，WifiSettings 会调用 onOptionsItemSelected()。

packages/apps/Settings/src/com/android/settings/wifiwifisettings.java

```
public booleanonOptionsItemSelected(MenuItem item) {
        switch (item.getItemId()) {
            case MENU_ID_SCAN:
                 if(mWifiManager.isWifiEnabled()) {
                     mScanner.resume();
                 }
                 return true;
            case MENU_ID_ADVANCED:
                 startActivity(new Intent(this,AdvancedSettings.class));
                 return true;
        }
        return super.onOptionsItemSelected(item);
}
```

Handler 类：

```
private class Scanner extends Handler {
        private int mRetry = 0;
        void resume() {
            if (!hasMessages(0)) {
                 sendEmptyMessage(0);
```

```
            }
        }
        void pause() {
            mRetry = 0;
             mAccessPoints.setProgress(false);
            removeMessages(0);
        }
        @Override
        public void handleMessage(Message message) {
            if (mWifiManager.startScanActive()){
                 mRetry = 0;
            } else if (++mRetry >= 3) {
                 mRetry = 0;
                Toast.makeText(WifiSettings.this, R.string.wifi_fail_to_scan,
                        Toast.LENGTH_LONG).show();
                 return;
            }
            mAccessPoints.setProgress(mRetry != 0);
            sendEmptyMessageDelayed(0, 6000);
        }
}
```

这里的 mWifiManager.startScanActive()就会调用 WifiService 里的 startScan（）函数，下面的流程和上面的一样，这里不赘述。

当 supplicant 完成了这个扫描命令后，它会发送一个消息给上层，提醒他们扫描已经完成，WifiMonitor 会接收到这消息，然后再发送给 WifiStateTracker。

Frameworks/base/wifi/java/android/net/wifi/WifiMonitor.java

```
 void handleEvent(int event, String remainder) {
             switch (event) {
                 caseDISCONNECTED:
                    handleNetworkStateChange(NetworkInfo.DetailedState.DISCONNECTED,remainder);
                     break;
                 case CONNECTED:
                    handleNetworkStateChange(NetworkInfo.DetailedState.CONNECTED,remainder);
                     break;
                 case SCAN_RESULTS:
                     mWifiStateTracker.notifyScanResultsAvailable();
                     break;
                 case UNKNOWN:
                     break;
             }
}
```

WifiStateTracker 将会广播 SCAN_RESULTS_AVAILABLE_ACTION 消息：

代码如：Frameworks/base/wifi/java/android/net/wifi/WifiStateTracker.java

```
public voidhandleMessage(Message msg) {
        Intent intent;
……
case EVENT_SCAN_RESULTS_AVAILABLE:
                if(ActivityManagerNative.isSystemReady()) {
                    mContext.sendBroadcast(new Intent(WifiManager.SCAN_RESULTS_AVAILABLE_ACTION));
                }
                sendScanResultsAvailable();

                setScanMode(false);
                break;
……
}
```

由于 WifiSettings 类注册了 intent，能够处理 SCAN_RESULTS_AVAILABLE_ACTION 消息，它会调用 handleEvent()，调用流程如下所示。

WifiSettings.handleEvent() ====>WifiSettings.updateAccessPoints() ====> mWifiManager.getScanResults()====> mService.getScanResults()====>mWifiStateTracker.scanResults() ====> WifiNative.scanResultsCommand()……

将获取 AP 列表的命令发送到 supplicant，然后 supplicant 通过 Socket 发送扫描结果，由上层接收并显示。这和前面的消息获取流程基本相同。

（三）配置，连接 AP

当用户选择一个活跃的 AP 时，WifiSettings 响应打开一个对话框来配置 AP，比如加密方法和连接 AP 的验证模式。配置好 AP 后，WifiService 添加或更新网络连接到特定的 AP。
代码如：packages/apps/settings/src/com/android/settings/wifi/WifiSetttings.java

```java
public booleanonPreferenceTreeClick(PreferenceScreen screen, Preference preference) {
        if (preference instanceof AccessPoint) {
            mSelected = (AccessPoint) preference;
            showDialog(mSelected, false);
        } else if (preference == mAddNetwork) {
            mSelected = null;
            showDialog(null, true);
        } else if (preference == mNotifyOpenNetworks) {
            Secure.putInt(getContentResolver(),
                    Secure.WIFI_NETWORKS_AVAILABLE_NOTIFICATION_ON,
                    mNotifyOpenNetworks.isChecked() ? 1 : 0);
        } else {
            return super.onPreferenceTreeClick(screen, preference);
        }
        return true;
}
```

配置好以后，当按下“Connect Press”时，WifiSettings 通过发送 LIST_NETWORK 命令到 supplicant 来检查该网络是否配置。如果没有该网络或没有配置它，WifiService 调用 addorUpdateNetwork（）函数来添加或更新网络，然后发送命令给 supplicant，连接到这个网络。下面是从响应连接按钮到 WifiService 发送连接命令的代码：

packages/apps/settings/src/com/android/settings/wifi/WifiSetttings.java
[java] view plaincopy

```java
public void onClick(DialogInterfacedialogInterface, int button) {
        if (button == WifiDialog.BUTTON_FORGET && mSelected != null) {
            forget(mSelected.networkId);
        } else if (button == WifiDialog.BUTTON_SUBMIT && mDialog !=null) {
            WifiConfiguration config = mDialog.getConfig();
            if (config == null) {
                 if (mSelected != null&& !requireKeyStore(mSelected.getConfig())) {
                     connect(mSelected.networkId);
                 }
            } else if (config.networkId != -1) {
                 if (mSelected != null) {
                     mWifiManager.updateNetwork(config);

                     saveNetworks();
                 }
            } else {
                 int networkId =mWifiManager.addNetwork(config);
                 if (networkId != -1) {
                    mWifiManager.enableNetwork(networkId, false);
                     config.networkId =networkId;
                     if (mDialog.edit || requireKeyStore(config)){
                         saveNetworks();
                     } else {
                         connect(networkId);
                     }
                 }
            }
        }
}
```

Frameworks\base\wifi\java\android\net\wifi\WifiManager.java

```java
public intupdateNetwork(WifiConfiguration config) {
         if(config == null || config.networkId < 0) {
            return -1;
         }
         return addOrUpdateNetwork(config);
```

```
}
private intaddOrUpdateNetwork(WifiConfiguration config) {
        try {
            return mService.addOrUpdateNetwork(config);
        } catch (RemoteException e) {
            return -1;
        }
}
```

WifiService.addOrUpdateNetwork()通过调用 mWifiStateTracker.setNetworkVariable()将连接命令发送到 Wpa_supplicant。

（四）获取 IP 地址

当连接到 supplicant 后，WifiMonitor 就会通知 WifiStateTracker。
代码如：Frameworks/base/wifi/java/android/net/wifi/WifiMonitor.java

```
Public void Run(){
if (connectToSupplicant()) {
                // Send a message indicatingthat it is now possible to send commands
                // to the supplicant
                mWifiStateTracker.notifySupplicantConnection();
            } else {
                mWifiStateTracker.notifySupplicantLost();
                return;
            }
……
}
```

WifiStateTracker 发送 EVENT_SUPPLICANT_CONNECTION 消息到消息队列，这个消息有自己的 handlermessage()函数处理，它会启动一个 DHCP 线程，而这个线程会一直等待一个消息事件，来启动 DHCP 协议分配 IP 地址。
frameworks/base/wifi/java/android/net/wifi/WifiStateTracker.java

```
void notifySupplicantConnection() {
        sendEmptyMessage(EVENT_SUPPLICANT_CONNECTION);
}
public void handleMessage(Message msg) {
        Intent intent;
    switch (msg.what) {
            case EVENT_SUPPLICANT_CONNECTION:
              ……
              HandlerThread dhcpThread = newHandlerThread("DHCP Handler Thread");
                 dhcpThread.start();
                 mDhcpTarget = newDhcpHandler(dhcpThread.getLooper(), this);
        ……
        ……
           "code" class="java">           case EVENT_NETWORK_STATE_CHANGED:

              ……
                configureInterface();
              ……

        }}

"code" class="java">private void configureInterface() {
        checkPollTimer();
         mLastSignalLevel = -1;
         if(!mUseStaticIp) {          //使用 DHCP 线程动态 IP
             if(!mHaveIpAddress && !mObtainingIpAddress) {
                mObtainingIpAddress = true;
                 //发送启动 DHCP 线程获取 IP
                 mDhcpTarget.sendEmptyMessage(EVENT_DHCP_START);
             }
         } else {        //使用静态 IP，IP 信息从 mDhcpInfo 中获取
             intevent;
             if(NetworkUtils.configureInterface(mInterfaceName, mDhcpInfo)) {
                mHaveIpAddress = true;
                event = EVENT_INTERFACE_CONFIGURATION_SUCCEEDED;
```

```
                if (LOCAL_LOGD) Log.v(TAG, "Static IP configurationsucceeded");
             }else {
                mHaveIpAddress = false;
                event = EVENT_INTERFACE_CONFIGURATION_FAILED;
                if (LOCAL_LOGD) Log.v(TAG, "Static IP configuration failed");
             }
            sendEmptyMessage(event);           //发送 IP 获得成功消息事件
         }
```

当 Wpa_supplicant 连接到 AP 后，它会发送一个消息给上层来通知连接成功，WifiMonitor 会接受到这个消息并上报给 WifiStateTracker。

Frameworks/base/wifi/java/android/net/wifi/WifiMonitor.java

```
"code" class="java">void handleEvent(int event, String remainder) {
            switch (event) {
                case DISCONNECTED:
                    handleNetworkStateChange(NetworkInfo.DetailedState.DISCONNECTED,remainder);
                    break;
                case CONNECTED:
                    handleNetworkStateChange(NetworkInfo.DetailedState.CONNECTED,remainder);
                    break;
                 ……
    }
private void handleNetworkStateChange(NetworkInfo.DetailedState newState, String data) {
        StringBSSID = null;
        intnetworkId = -1;
        if(newState == NetworkInfo.DetailedState.CONNECTED) {
           Matcher match = mConnectedEventPattern.matcher(data);
            if(!match.find()) {
               if (Config.LOGD) Log.d(TAG, "Could not find BSSID in CONNECTEDevent string");
            }else {
               BSSID = match.group(1);
               try {
                   networkId = Integer.parseInt(match.group(2));
               } catch (NumberFormatException e) {
                   networkId = -1;
                }
            }
        }
        mWifiStateTracker.notifyStateChange(newState,BSSID, networkId);
}
void notifyStateChange(DetailedState newState, StringBSSID, int networkId) {
        Messagemsg = Message.obtain(
           this, EVENT_NETWORK_STATE_CHANGED,
            newNetworkStateChangeResult(newState, BSSID, networkId));
       msg.sendToTarget();
}
```

DhcpThread 获取 EVENT_DHCP_START 消息事件后，调用 handleMessage（）函数，启动 DHCP 获取 IP 地址的服务。

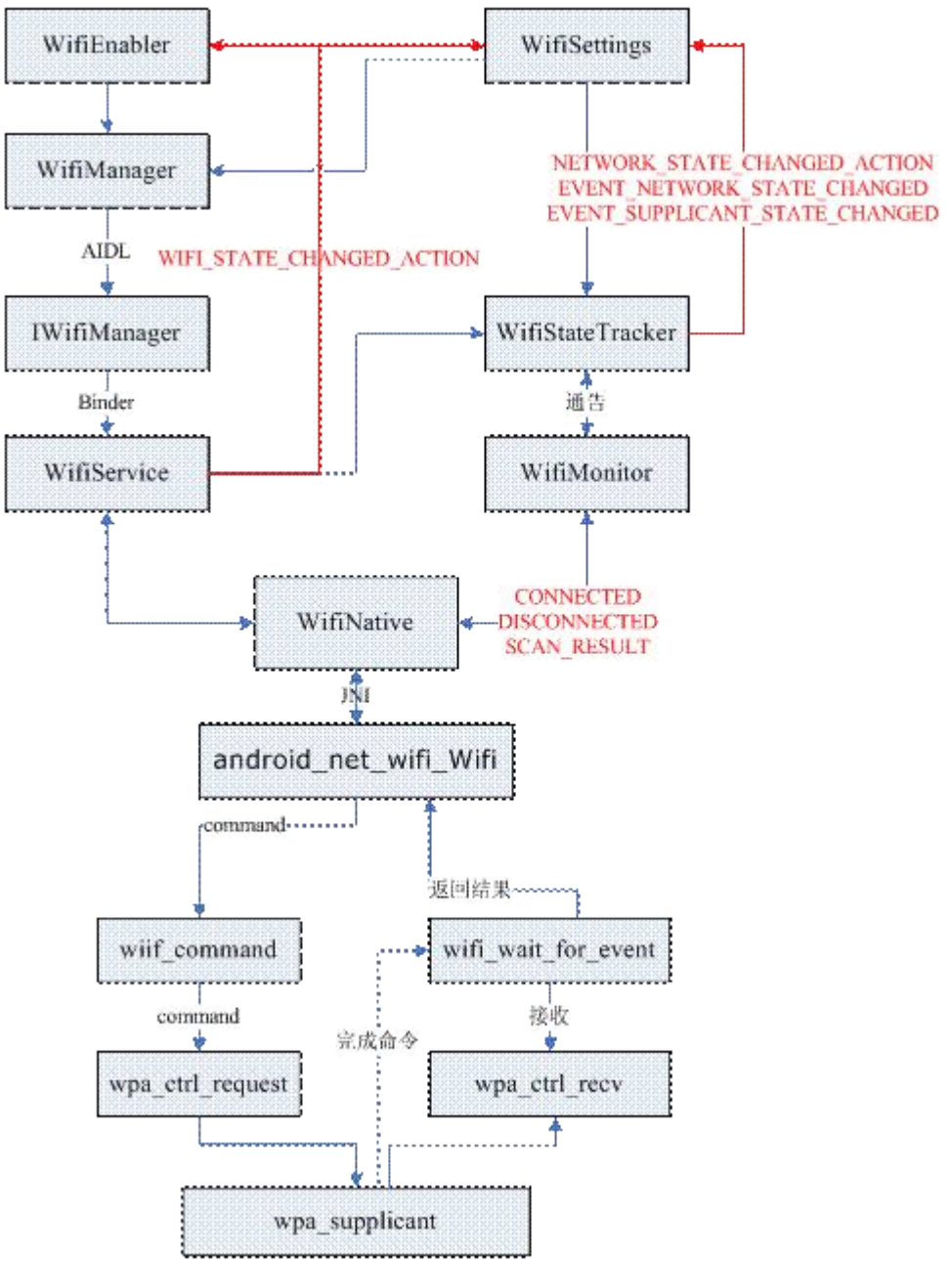
frameworks/base/wifi/java/android/net/wifi/WifiStateTracker.java

```

"code" class="java">public void handleMessage(Message msg) {
            intevent;
    switch (msg.what) {
               case EVENT_DHCP_START:
    ……
    Log.d(TAG, "DhcpHandler: DHCP requeststarted");
    //启动一个 DHCPclient 的精灵进程，为 mInterfaceName 请求分配一个 IP 地//址
        if (NetworkUtils.runDhcp(mInterfaceName, mDhcpInfo)) {
         event= EVENT_INTERFACE_CONFIGURATION_SUCCEEDED;
         if(LOCAL_LOGD) Log.v(TAG, "DhcpHandler: DHCP request succeeded");
        } else {
           event= EVENT_INTERFACE_CONFIGURATION_FAILED;
```

五、WIFI 连接流程分析

Wifi 连接部分



当用户选择一个 AP 时会弹出一个 AP 参数配置对话框，此对话框会显示当前选择的 AP 信号强度，若此 AP 设置了密码则需要用户输入密码才能登录。WifiSettings 中的 onPreferenceTreeClick 会被调用　　@Override

```
public boolean onPreferenceTreeClick(PreferenceScreen screen, Preference preference) {
    //点击 AP 响应函数
    if (preference instanceof AccessPoint) {
        mSelected = (AccessPoint) preference;
        showDialog(mSelected, false);
    } else if (preference == mAddNetwork) {
        mSelected = null;
        showDialog(null, true);
    } else if (preference == mNotifyOpenNetworks) {
        Secure.putInt(getContentResolver(),
                Secure.WIFI_NETWORKS_AVAILABLE_NOTIFICATION_ON,
                mNotifyOpenNetworks.isChecked() ? 1 : 0);
    } else {
        return super.onPreferenceTreeClick(screen, preference);
    }
    return true;
}
```

用户配置好之后点击连接按钮，onClick 函数会被调用。

```
public void onClick(DialogInterface dialogInterface, int button) {
    //点击连接按钮的响应函数
```

```
if (button == WifiDialog.BUTTON_FORGET && mSelected != null) {
    forget(mSelected.networkId);
} else if (button == WifiDialog.BUTTON_SUBMIT && mDialog != null) {
    WifiConfiguration config = mDialog.getConfig();

    if (config == null) {
        if (mSelected != null && !requireKeyStore(mSelected.getConfig())) {
            connect(mSelected.networkId);
        }
    } else if (config.networkId != -1) {
        if (mSelected != null) {
            mWifiManager.updateNetwork(config);
            saveNetworks();
        }
    } else {
        int networkId = mWifiManager.addNetwork(config);
            if (networkId != -1) {
                mWifiManager.enableNetwork(networkId, false);
                config.networkId = networkId;
                if (mDialog.edit || requireKeyStore(config)) {
                    saveNetworks();
                } else {
                    connect(networkId);
                }
            }
        }
    }
```

连接请求部分

一.Settings 的 connect 函数响应连接，更新网络保存配置，更新设置当前选择的优先级最高，并保存。然后通过 enableNetwork 使得其他网络不可用来进行连接。最后调用 WifiManager 的 reconnect 函数连接当前选择的网络。

二.WifiManager 的 reconnect 函数通过 AIDL 的 Binder 机制，调用 WifiService 的 reconnect 函数

三.然后会调用 WifiStateTracker 的 reconnectCommand 函数，通过 JNI（android_net_wifi_Wifi)的 android_net_wifi_reconnectCommand 函数向 WPA_WPASUPPLICANT 发送 RECONNECT 命令。

四. android_net_wifi_Wifi 通过 doCommand(命令名，响应缓冲，响应缓存大小)调用 wifi.c 中的 wifi_command 函数来发送命令。

五.最后通过 wpa_ctrl 的 wpa_ctrl_request 函数向控制通道发送连接命令。

返回请求部分

六.当连接上之后 WPA_SUPPLICANT 会向控制通道发送连接成功命令。wifi.c 的 wifi_wait_for_event 函数阻塞调用并返回这个命令的字符串（CONNECTED）.

七.而后 WifiMonitor 会被执行来处理这个事件，WifiMonitor 再调用 WifiStateTracker 的 notifyStateChange,WifiStateTracker 则接着会往自身发送 EVENT_DHCP_START 消息来启动 DHCP 去获取 IP 地址,然后广播 NETWORK_STATE_CHANGED_ACTION 消息，最后由 WifiSettings 类来响应，改变状态和界面信息。

关键函数功能介绍

一.connect 函数功能

1.updateNetwork：updateNetwork(config)会将当前选择连接的 AP 配置信息信息传递进去，配置信息有（网络 ID 等）。如果网络 ID 为-1 则重新添加网络配置，然后向 wpa_supplicant 发送 SET_NETWORK 命令（即通过这个网络 ID 设置其他一些相关信息，设置 SSID，密码等）如果网络配置不为－1 则直接执行后面步骤即发送 SET_NETWORK 命令。

2.saveNetwork:告诉 supplicant 保存当前网络配置并更新列表。SaveNetwork 会调用 WifiService 的

saveConfiguration 向 wpa_supplicant 发送 SAVE_CONFIG 命令保存当前网络配置信息，
如果返回 false，则向 wpa_supplicant 重新发送 RECONFIGURE 命令获取配置信息，如果获取信

息成功后，会 Intent 一个 NETWORK_IDS_CHANGED_ACTION 事件 WifiSettings 会注册接受这个 时间并更新列表。

3.enableNetwork 函数，向系统获取接口名并使得该接口有效。由于之前传递的 disableOthers 为 true 则向 wpa_supplicant 发送 SELECT_NETWORK（如果传递的为 false 则发送 ENABLE_NETWORK 命令），

4.reconnect 函数：连接 AP

二.reconnect 函数功能：connect 函数会调用 WifiManager 的 reconnect 然后通过 Binder 机制调用 WifiService 的 reconnect，再由 WifiStateTracke 调用 WifiNative 向 wpa_supplicant 发送 RECONNECT 命令去连接网络，当连接上 wpa_supplicant 之后会向控制通道发送连接成功的命令，

wifi_wait_for_event 函数阻塞等待该事件的发生，并返回这个命令的字符串（CONNECTED）

三.android_net_wifi_Wifi 函数的 doCommand 函数会调用 wifi.c 的 wifi_command 函数将上层的命令向 wpa_supplicant 发送。

四.wifi_wait_for_event 函数以阻塞的方式，等待控制通道传递的事件。当有事件传递过来的时候该函数会通过 wpa_ctrl 的 wpa_ctrl_recv 函数读取该事件，并以字符串形式返回该事件名。

```
int wifi_wait_for_event(char *buf, size_t buflen)
{
                    .....
    result = wpa_ctrl_recv(monitor_conn, buf, &nread);
    if (result < 0) {
        LOGD("wpa_ctrl_recv failed: %s/n", strerror(errno));
        strncpy(buf, WPA_EVENT_TERMINATING " - recv error", buflen-1);
        buf[buflen-1] = '/0';
        return strlen(buf);
    }
    buf[nread] = '/0';


    if (result == 0 && nread == 0) {

        LOGD("Received EOF on supplicant socket/n");
        strncpy(buf, WPA_EVENT_TERMINATING " - signal 0 received", buflen-1);
        buf[buflen-1] = '/0';
        return strlen(buf);
    }

    if (buf[0] == '<') {
        char *match = strchr(buf, '>');
        if (match != NULL) {
            nread -= (match+1-buf);
            memmove(buf, match+1, nread+1);
        }
    }
    return nread;
}
```

五.wpa_ctrl_request，通过 socket 方式向 wpa_supplicant 发送命令，以 select 模式阻塞在

wpa_supplicant 发送和接收。

```
int wpa_ctrl_request(struct wpa_ctrl *ctrl, const char *cmd, size_t cmd_len，char *reply, size_t *reply_len,void(*msg_cb)(char *msg, size_t len))
{
        .....
                res = select(ctrl->s + 1, &rfds, NULL, NULL, &tv);
                if (FD_ISSET(ctrl->s, &rfds)) {
                        res = recv(ctrl->s, reply, *reply_len, 0);
                        if (res < 0)
                                return res;
                        if (res > 0 && reply[0] == '<') {

                                if (msg_cb) {

                                        if ((size_t) res == *reply_len)
                                                res = (*reply_len) - 1;
                                        reply[res] = '/0';
                                        msg_cb(reply, res);
                                }
                                continue;
                        }
                        *reply_len = res;
                        break;
                } else {
                        return -2;
                }
        }
        return 0;
}
```

六.WifiMonitor 维护一个监视线程分发处理底层返回上来的事件

```
 void handleEvent(int event, String remainder) {
          switch (event) {
            case DISCONNECTED:
              handleNetworkStateChange(NetworkInfo.DetailedState.DISCONNECTED, remainder);
              break;
            case CONNECTED:
              handleNetworkStateChange(NetworkInfo.DetailedState.CONNECTED, remainder);
              break;
            case SCAN_RESULTS:
              mWifiStateTracker.notifyScanResultsAvailable();
              break;
            case UNKNOWN:
              break;
          }
        }
```

此时返回的事件是 CONNECTED 因此 handleNetworkStateChange 会被调用，验证一下 BSSID,重新获得 networkId

,然后调用 WifiStateTracke 的 notifyStateChange 通知状态改变了的消息（*EVENT_NETWORK_STATE_CHANGED*）

接着处理这个消息，会移除可用网络通告，然后通过 configureInterface()的动态获取 IP 地址。最后

发送一个 NETWORK_STATE_CHANGED_ACTION Intent，WifiSetings 注册了此 Intent 因此会响应该它。由 updateConnectionState 函数响应。

七.updateConnectionState 获取连接信息，更新列表状态，设置为 Connected,然后设置当前网络为可用状态

```
  private void updateConnectionState(DetailedState state) {
```

```
if (!mWifiManager.isWifiEnabled()) {
    mScanner.pause();
    return;
}

if (state == DetailedState.OBTAINING_IPADDR) {
    mScanner.pause();
} else {
    mScanner.resume();
}

mLastInfo = mWifiManager.getConnectionInfo();
if (state != null) {
    mLastState = state;
}

    for (int i = mAccessPoints.getPreferenceCount() - 1; i >= 0; --i) {
        ((AccessPoint) mAccessPoints.getPreference(i)).update(mLastInfo, mLastState);
    }

    if (mResetNetworks && (state == DetailedState.CONNECTED ||
            state == DetailedState.DISCONNECTED || state == DetailedState.FAILED)) {
        updateAccessPoints();
        enableNetworks();
    }
}
```

流程图对应的源代码路径为：

WifiEnabler,WifiSettings 对应的路径如下：

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

WifiManager,WifiMonitor,WifiStateTracker,WifiNative.对应的源代码路径如下：

froyo/frameworrks/base/wifi/java/android/net/wifi/

WifiService 对应代码的位置

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

android_net_wifi_Wifi 源代码路径如下：

froyo/frameworks/base/core/jni/

wifi_command,wifi_wait_for_envent 源代码路径如下：

/hardware/libhardware_legacy/wifi/wifi.c

wpa_ctrl_源代码路径如下：

/external/wpa_supplicant/wpa_ctrl.c

wpa_supplicant 源代码路径如下：

froyo/external/wpa_supplicant/

六、WIFI 移植

手动加载驱动

驱动加载

modprobe libertas

modprobe libertas_sdio

加载第二行时出错拉

modprobe libertas_sdio

libertas_sdio: Libertas SDIO driver

libertas_sdio: Copyright Pierre Ossman

model=0xb

sd8686_helper.bin sd8686.bin

init: untracked pid 958 exited

过了一会出现：

libertas: can't load helper firmware

libertas: failed to load helper firmware

libertas_sdio: probe of mmc2:0001:1 failed with error -2

但是用 lsmod 又能看到

lsmod

libertas_sdio 8776 0 - Live 0xbf022000

libertas 97416 1 libertas_sdio, Live 0xbf009000

usbserial 30256 0 - Live 0xbf000000

###使用 insmod 试一下

insmod /lib/modules/2.6.24.7/kernel/drivers/net/wireless/libertas/libertas.ko

insmod /lib/modules/2.6.24.7/kernel/drivers/net/wireless/libertas/libertas_sdio.ko

还是老样子

###发现系统中没有提到的文件 fireware 文件

从华恒的 romfs/lib 把 fireware 拷贝到 /nfs/rootfs/lib 下

$$还是不行

###把 fireware 拷贝到 system/etc/下面，终于可以加载了

如下：

modprobe libertas_sdio

libertas_sdio: Libertas SDIO driver

libertas_sdio: Copyright Pierre Ossman

model=0xb

sd8686_helper.bin sd8686.bin

init: untracked pid 714 exited

init: untracked pid 717 exited

libertas: eth1: Marvell WLAN 802.11 adapter

$$$建议

Title：Android INIT not loading firmware

android 员工答：You need to run the insmod in a separate process launched by init.

#########

理解原理

Android uses a modified wpa_supplicant (external/wpa_supplicant) daemon for wifi support which is controlled through a socket by hardware/libhardware_legacy/wifi/wifi.c (WiFiHW) that gets controlled from Android UI through android.net.wifi package from frameworks/base/wifi/java/android/net/wifi/ and it's corresponding jni implementation in frameworks/base/core/jni/android_net_wifi_Wifi.cpp

Higher level network management is done in frameworks/base/core/java/android/net

1.在 build/target/board/generic/BoardConfig.mk 中增加

BOARD_WPA_SUPPLICANT_DRIVER := WEXT

2.打开调试信息--可选

默认是不打开调试的

2.1 modify external/wpa_supplicant/common.c and set wpa_debug_level = MSG_DEBUG

2.2 modify common.h and change #define wpa_printf from if ((level) >= MSG_INFO) to if ((level) >= MSG_DEBUG)

3.创建 system/etc/wifi/wpa_supplicant.conf

有 2 种 socket

一种是 android private socket

ctrl_interface=eth1

update_config=1

ap_scan=1 ###取决于 wifi 驱动，如果不行，则改为 0 试试

另一种是 unix 标准 socket，这里先选用第二种

ctrl_interface=DIR=/data/system/wpa_supplicant GROUP=wifi

update_config=1

ap_scan=1 ###取决于 wifi 驱动，如果不行，则改为 0 试试

4 在 init.rc 中增加以下语句

#@qiu

mkdir /system/etc/wifi 0777 wifi wifi

chmod 0777 /system/etc/wifi

chmod 0777 /system/etc/wifi/wpa_supplicant.conf

chown wifi wifi /system/etc/wifi/wpa_supplicant.conf

mkdir /data/misc/wifi 0777 wifi wifi

mkdir /data/misc/wifi/sockets 0770 wifi wifi

chmod 0777 /data/misc/wifi

chmod 0777 /data/misc/wifi/wpa_supplicant.conf

chown wifi wifi /data/misc/wifi

chown wifi wifi /data/misc/wifi/wpa_supplicant.conf

wpa_supplicant socket (unix socket mode)

mkdir /data/system/wpa_supplicant 0777 wifi wifi

chmod 0777 /data/system/wpa_supplicant

chown wifi wifi /data/system/wpa_supplicant

#qiu@

##如果是 private socket，则是

mkdir /system/etc/wifi 0777 wifi wifi

chmod 0777 /system/etc/wifi

chmod 0777 /system/etc/wifi/wpa_supplicant.conf

chown wifi wifi /system/etc/wifi/wpa_supplicant.conf

#wpa_supplicant control socket for android wifi.c (android private socket)

mkdir /data/misc/wifi 0777 wifi wifi

mkdir /data/misc/wifi/sockets 0770 wifi wifi

chmod 0777 /data/misc/wifi

chmod 0777 /data/misc/wifi/wpa_supplicant.conf

chown wifi wifi /data/misc/wifi

chown wifi wifi /data/misc/wifi/wpa_supplicant.conf

5.在 init.rc 中添加 wpa_supplicant 和 dhcpcd 启动服务

service wpa_supplicant /system/bin/wpa_supplicant -dd -Dwext -ieth1 -c /system/etc/wifi/wpa_supplicant.conf

group system wifi inet

disabled

oneshot

##如果是 pricvate socket 则需要在 group 上面增加一行：

socket wpa_eth1 dgram 777 wifi wifi

service dhcpcd /system/bin/dhcpcd -f /system/etc/dhcpcd/dhcpcd.conf -d eth1

group system dhcp wifi

disabled

oneshot

5.5 增加/system/etc/dhcpcd/dhcpcd.conf 文件内容

interface eth1

option subnet_mask, routers, domain_name_servers

6.把 wifi 驱动编译进内核

首先 init.rc 中增加

setprop wifi.interface "eth1"

setprop wlan.driver.status "ok"

其次在修改 hardware/libhardware_legacy/wifi/wifi.c

新的 out/target/product/generic/system/lib/libhardware_legacy.so---v2

在函数 wifi_load_driver()的体的开头增加

//@qiu

LOGE("Weber@wifi driver loaded !");

return 0;

//qiu@

在函数 wifi_unload_driver()体的开头增加

//@qiu

LOGE("Weber@wifi driver unloaded!");

return 0;

//qiu@

####17th,Jul

把驱动从 M 改为*,得 zImage-vvv39

开始启动：

加载 firmware 的时候出错

libertas: can't load helper firmware

libertas: failed to load helper firmware

libertas_sdio: probe of mmc2:0001:1 failed with error -2

看来是无法加载 fireware

###尝试修改 wifi.c 让它自动加载模块

修改

#ifndef WIFI_DRIVER_MODULE_PATH1

#define WIFI_DRIVER_MODULE_PATH1 "/system/lib/modules/libertas.ko"

#endif

#ifndef WIFI_DRIVER_MODULE_PATH2

#define WIFI_DRIVER_MODULE_PATH2 "/system/lib/modules/libertas_sdio.ko"

#endif

#ifndef WIFI_DRIVER_MODULE_NAME1

#define WIFI_DRIVER_MODULE_NAME1 "libertas"

#endif

#ifndef WIFI_DRIVER_MODULE_NAME2

#define WIFI_DRIVER_MODULE_NAME2 "libertas_sdio"

#endif

并在此文件的后面代码中做一定修改，wifi_load_driver()函数中也做相应修改

编译得到 libhardware_legacy.so---v3

-启动后在设置中点击 wifi

串口显示如下，看来是能加载了

libertas_sdio: Libertas SDIO driver

libertas_sdio: Copyright Pierre Ossman

model=0xb

sd8686_helper.bin sd8686.bin

init: untracked pid 954 exited

init: untracked pid 957 exited

libertas: eth1: Marvell WLAN 802.11 adapter

但是 wifi 图标下面显示 wifi 不可用

查看 logcat 的 main：

7899 E/WifiHW（ 708): Cannot access "/data/misc/wifi/wpa_supplicant.conf": Permission denied

7900 E/WifiHW（ 708): Wi-Fi will not be enabled

7901 W/WifiHW（ 708): Weber@wifi driver unloaded!

7902 E/WifiService(708): Failed to start supplicant daemon.

7903 D/SettingsWifiEnabler(791): Received wifi state changed from Enabling to Unknown

于是增加以下语句到 init.rc

mkdir /data/misc/wifi 0770 wifi wifi

mkdir /data/misc/wifi/sockets 0770 wifi wifi

chmod 0770 /data/misc/wifi

chmod 0660 /data/misc/wifi/wpa_supplicant.conf

chown wifi wifi /data/misc/wifi

chown wifi wifi /data/misc/wifi/wpa_supplicant.conf

并且 cp system/etc/wifi/wpa_supplicant.conf ./data/misc/wifi/

启动，终于能打开 wifi，但还是无法打开网页

查看 logcat

4564 E/wpa_supplicant(917): Failed to read or parse configuration '/system/etc/wifi/wpa_supplicant.conf'.

4565 E/WifiHW（ 708): Unable to open connection to supplicant on "/data/system/wpa_supplicant/sta": No such file or directory

4566 D/WifiService(708): ACTION_BATTERY_CHANGED pluggedType: 1

4899 E/WifiHW（ 708): Supplicant not running, cannot connect

5574 E/WifiHW（ 708): Supplicant not running, cannot connect

5575 V/WifiStateTracker(708): Supplicant died unexpectedly

5576 D/dalvikvm(934): DexOpt: load 3102ms, verify 1519ms, opt 24ms

5577 D/installd(665): DexInv: --- END '/system/app/Browser.apk' (success) ---

5578 D/NetworkStateTracker(708): setDetailed state, old =IDLE and new state=DISCONNECTED

5579 D/ConnectivityService(708): ConnectivityChange for WIFI: DISCONNECTED/DISCONNECTED

##在 init.rc 中把 wpa_supplicant.conf 权限修改为 777，

还是有错误：

5573 E/WifiHW（ 708): Unable to open connection to supplicant on "/data/system/wpa_supplicant/sta": No such file or director

##在 init.rc 中增加以下几条语句

setprop wifi.interface "eth1"

setprop wlan.driver.status "ok"

setprop wlan.interface "eth1"

结果还是不行 logcat 显示

E/WifiHW（ 708):Unable to open connection to supplicant on "/data/system/wpa_supplicant/eth1": No such file or directory

##在 init.rc 中 dhcp 服务启动是为 eth1 的增加 wifi 组权限

又出现新错误

如果开始就 enable wifi 就会出现以下错误：

E/WifiHW（ 709): Unable to open connection to supplicant on "/data/system/wpa_supplicant/eth1": Connection refused

因为这是开机默认启动 wifi，如果不起动，错误则是和上一条相同

###18th,Jul

##尝试在 wpa_supplicant.conf 设置 GROUP=system,还是不行，只好改回去

ls /data/system/wpa_supplicant/eth1 -l

srwxrwx--- 1 1010 1010 0 Jul 18 08:22 /data/system/wpa_supplicant/eth1

ls /data/system/wpa_supplicant/eth1 -l

srwxrwxrwx 1 1010 1010 0 Jul 18 08:22 /data/system/wpa_supplicant/eth1

###18th,Jul 手动调试

wpa_supplicant -dd -Dwext -ieth1 -c /system/etc/wifi/wpa_supplicant.conf&

ioctl[SIOCSIWPMKSA]: Invalid argument《《《《输出的错误

ls /data/system/wpa_supplicant/eth1 -l

srwxrwxrwx 1 1010 1010 0 Jul 18 13:14 /data/system/wpa_supplicant/eth1

#wpa_cli -i eth1 -p /data/system/wpa_supplicant

不知道该如何设置 ap

bash# wpa_cli -ieth1 scan //搜索无线网

bash# wpa_cli -ieth1 scan_results //显示搜索结果

bash# wpa_cli -ieth1 add_network

bash# iwconfig eth1 essid "you_wifi_net"

bash# wpa_cli -ieth1 password 0 "password"

bash# wpa_cli -ieth1 enable_network

###19th，Jul

修改 vi frameworks/base/wifi/java/android/net/wifi/WifiStateTracker.java

把 tiwlan0 改 eth1

暂时未编译

###开始没有选择第二步的 debug 选项，选上，在 external/wpa_supplicant 目录下有 mm 编译

得到

system/bin/wpa_cli---vvv1

system/lib/libwpa_client.so---vvv1

system/bin/wpa_supplicant---vvv1

###试一下 android private socket

修改 vi system/etc/wifi/wpa_supplicant.conf

vi data/misc/wifi/wpa_supplicant.conf

修改 init.rc

在 chmod 那里把 unix socket 的那段注释

在启动 wpa_supplicant 的时候增加

socket wpa_eth1 dgram 660 wifi wifi

结果还是不行

###只好在 external/wpa_supplicant/wpa_ctrl.c 中添加

LOGW，注意得先加入头文件和 LOG_TAG

```
 17 #define LOG_TAG "WifiQiu"
 18 #include "cutils/log.h"
```

mm 编译得到

system/bin/wpa_cli---vvv2

system/lib/libwpa_client.so---vvv2

但是竟然没有 WifiQiu 输出，不知道是不是没有用 make 而是 mm 的缘故

###20th,Jul

1.BoardConfig.mk 增加了 HAVE_CUSTOM_WIFI_DRIVER_2 := true

2.修改 wifi/wifi.c 中使得检查和卸载驱动的两个函数，用 make 编译得到

libhardware_legacy.so---v4

system/bin/wpa_cli---vvv2.1

system/bin/wpa_supplicant-v2

system/lib/libwpa_client.so---vvv2.1

###上面的版本还没有来的及测试，有发现问题

1.突然发现 external/wpa_supplicant/wpa_ctrl.c 中 LOG 不够全面，

一共有三种情况，漏了一个函数没有加，于是增加 LOGW 并使之可以区分

2.根据手动 strace wpa_cli 的提示

access("/data/misc/wifi/wpa_supplicant", F_OK) = -1 ENOENT (No such file or directory)

用 grep 搜索到所在文件为 external/wpa_supplicant/wpa_cli.c

把它修改为：/data/misc/wifi

用 mm 分别在 external/wpa_supplicant 和 hardware/libhardware_legacy 中编译

得到

system/bin/wpa_cli---vvvvv3

system/lib/libwpa_client.so---vvv3

libhardware_legacy.so---v5

启动之后发现是在对 socket 进行连接的时候出错了

connect(ctrl->s, (struct sockaddr *) &ctrl->dest,sizeof(ctrl->dest)) < 0)

而且不能关闭 wifi

只好重新修改 wifi.c 直接在 unload 函数中返回 0，使得 wifi 能被关闭

libhardware_legacy.so---v5.1

###21th,Jul 使用 private 的时候再次手动调试

wpa_supplicant -dd -Dwext -ieth1 -c /system/etc/wifi/wpa_supplicant.conf&

ioctl[SIOCSIWPMKSA]: Invalid argument

mkdir[ctrl_interface]: Permission denied

[1] + Done(255) wpa_supplicant -dd -Dwext -ieth1 -c /system/etc/wifi/wpa_supplicant.conf

查找 mkdir 发现是在 ctrl_iface_unix.c 中

错误是出现在#ifdef ANDROID 之外阿，诡异

###用 donut 中 external/wpa_supplicant 下的 driver_wext.c 替换地掉 eclair 中的

mm 重新编译，得到：

wpa_supplicant---vvv3

$$$还是不行

###22th,Jul

把 donut 的 wpa_supplicant 文件夹拷贝到 eclair 下

用 mm 编译得到

system/bin/wpa_cli---vvvvv4

system/bin/wpa_supplicant---vvv4

system/lib/libwpa_client.so---vvv4

wpa_supplicant---vvv3

终于可以搜索到网络了，泪奔

但是，不能获取到 ip 地址呵呵

logcat 提示：

I/WifiStateTracker(713): DhcpHandler: DHCP request failed: Timed out waiting for DHCP to finish

###手动启动 dhcp 提示

eth1: open `/data/misc/dhcp/dhcpcd-eth1.pid': No such file or directory

在 init.rc 中增加

mkdir /data/misc/dhcp 0777 wifi wifi

chmod 0777 /data/misc/dhcp

chown wifi wifi /data/misc/dhcp

还是不能获得地址

eth1: flock `/data/misc/dhcp/dhcpcd-eth1.pid': Try again

后来把上面的 wifi 改为 dhcp，还是不能获得 IP 地址

$$

把 dhcpcd 杀了，手动启动

dhcpcd -f /system/etc/dhcpcd/dhcpcd.conf -d eth1

eth1: dhcpcd 4.0.1 starting

eth1: hardware address = 00:1a:6b:a2:38:65

eth1: executing `/system/etc/dhcpcd/dhcpcd-run-hooks', reason PREINIT

eth1: /system/etc/dhcpcd/dhcpcd-run-hooks: Permission denied

eth1: waiting for carrier

eth1: host does not support a monotonic clock - timing can skew

eth1: timed out

eth1: executing `/system/etc/dhcpcd/dhcpcd-run-hooks', reason FAIL

eth1: /system/etc/dhcpcd/dhcpcd-run-hooks: Permission denied

$$把权限都改为 777，再次执行

dhcpcd -f /system/etc/dhcpcd/dhcpcd.conf -d eth1

eth1: dhcpcd 4.0.1 starting

eth1: hardware address = 00:1a:6b:a2:38:65

eth1: executing `/system/etc/dhcpcd/dhcpcd-run-hooks', reason PREINIT

eth1: waiting for carrier

eth1: host does not support a monotonic clock - timing can skew

eth1: timed out

eth1: executing `/system/etc/dhcpcd/dhcpcd-run-hooks', reason FAIL

$$还没有关闭 android 机呢，重新打开 wifi，嘿嘿，能分配到地址了

但是...但是，还是不能上网，估计是由于地址的原因吧

把 eth0 禁用之后就可以上网了，呵呵

wifi 移植告一段落

七、Wifi 模块代码总结

1 Wifi Application 代码

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

2 Wifi Framework

frameworks/base/wifi/java/android/net/wifi

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

3 Wifi JNI

frameworks/base/core/jni/android_net_wifi_Wifi.cpp

4 Wifi Hardware

hardware/libhardware_legacy/wifi/wifi.c

5 Wifi tool

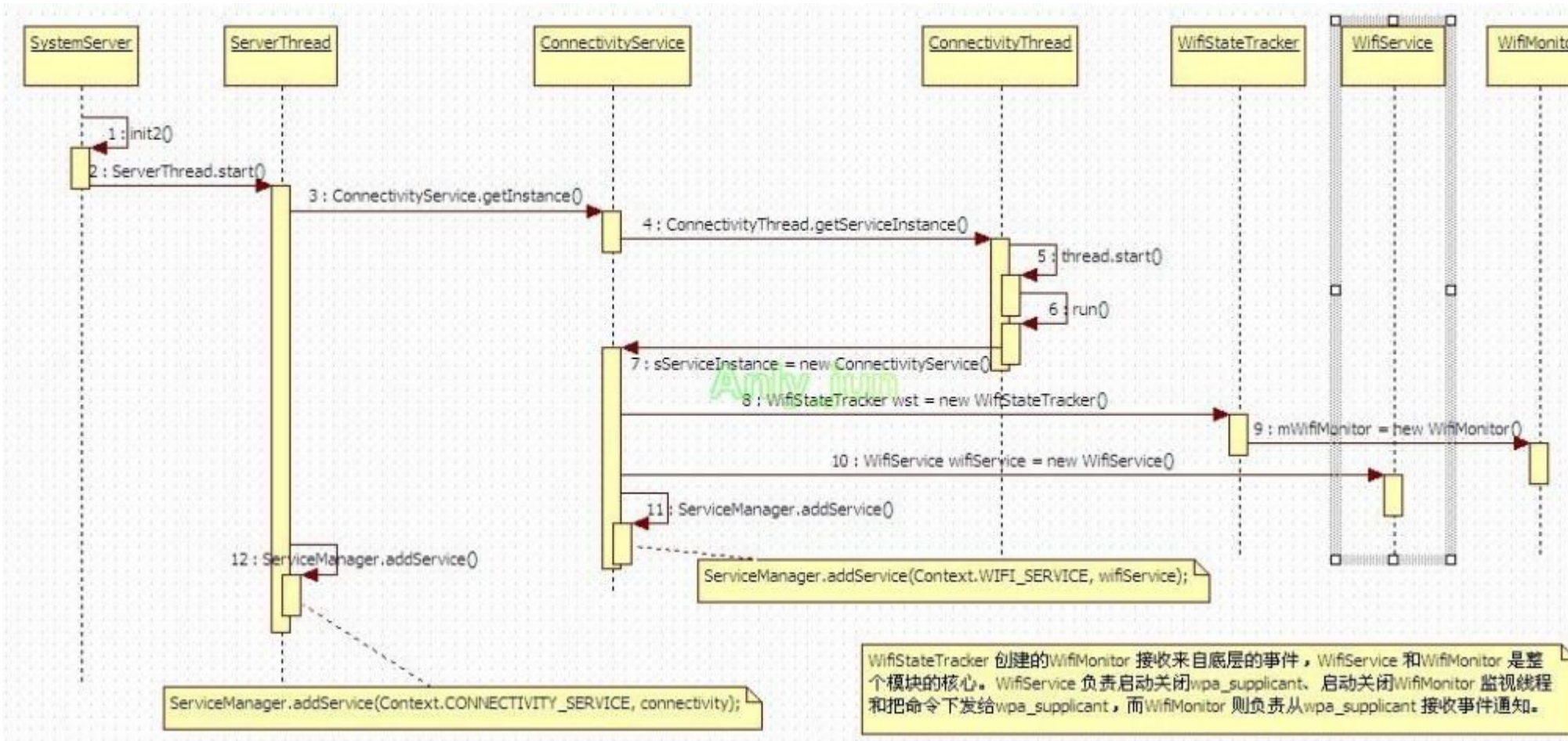
external/wpa_supplicant

6 Wifi kernel

net/wireless drivers/wlan_sd8688 arch/arm/mach-pxa/wlan_pm.c

Wifi 模块的初始化:

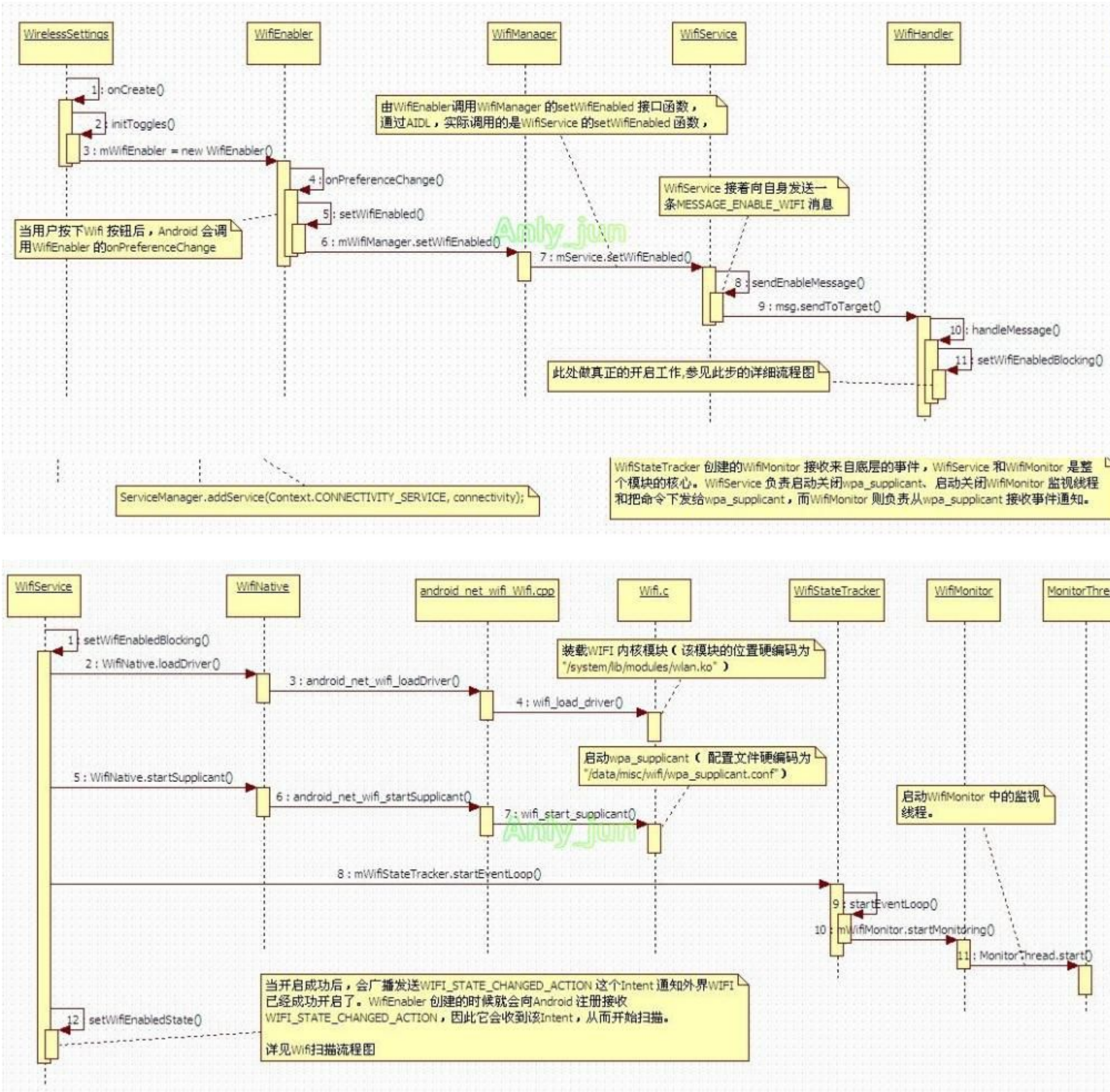
在 SystemServer 启动的时候，会生成一个 ConnectivityService 的实例, ConnectivityService 的构造函数会创建 WifiService，WifiStateTracker 会创建 WifiMonitor 接收来自底层的事件，WifiService 和 WifiMonitor 是整个模块的核心。WifiService 负责启动关闭 wpa_supplicant、启动关闭 WifiMonitor 监视线程和把命令下发给 wpa_supplicant，而 WifiMonitor 则负责从 wpa_supplicant 接收事件通知。



Wifi 模块的启动:

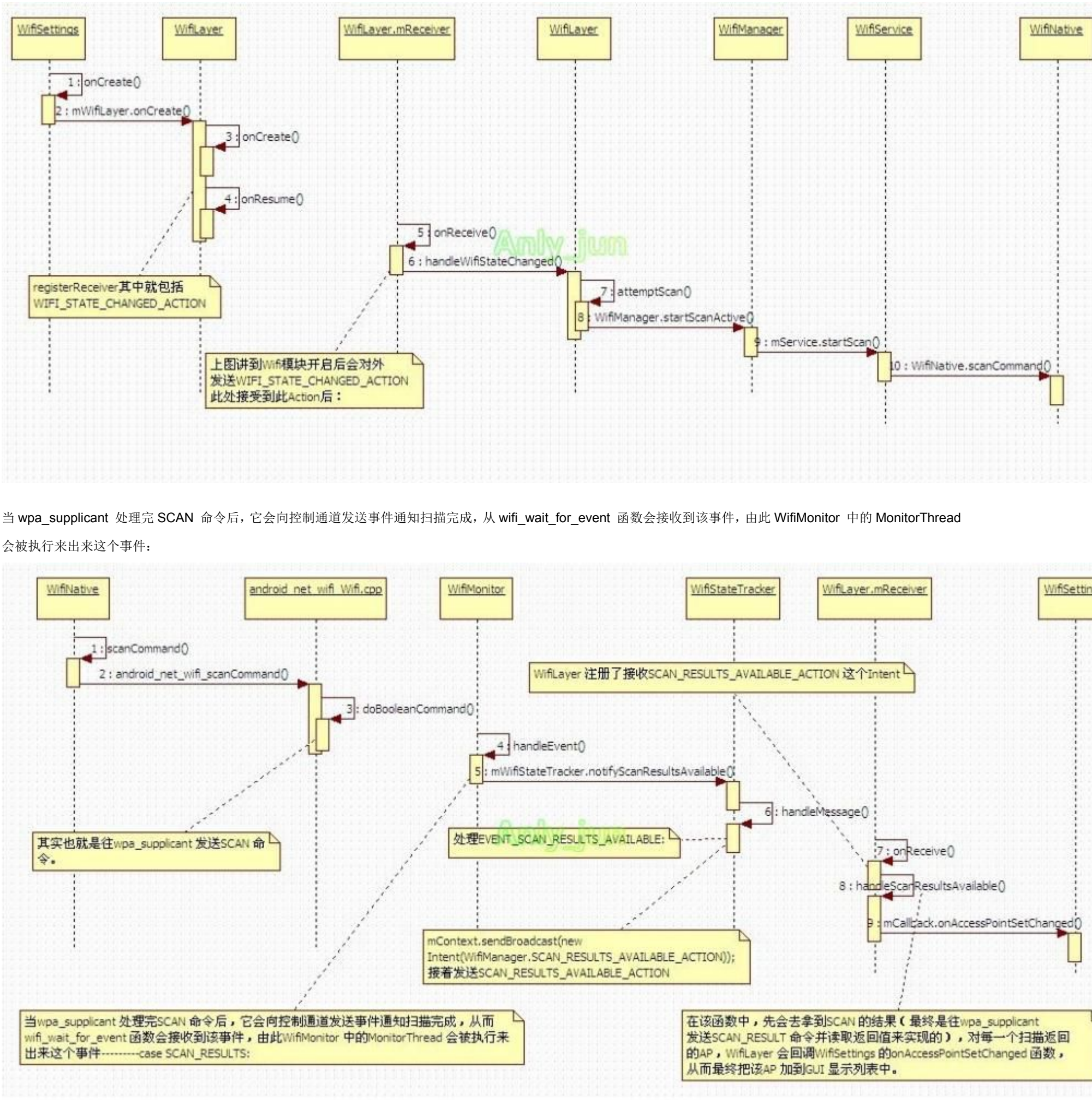
WirelessSettings 在初始化的时候配置了由 WifiEnabler 来处理 Wifi 按钮，

当用户按下 Wifi 按钮后，Android 会调用 WifiEnabler 的 onPreferenceChange，再由 WifiEnabler 调用 WifiManager 的 setWifiEnabled 接口函数，通过 AIDL，实际调用的是 WifiService 的 setWifiEnabled 函数，WifiService 接着向自身发送一条 MESSAGE_ENABLE_WIFI 消息，在处理该消息的代码中做真正的使能工作：首先装载 WIFI 内核模块（该模块的位置硬编码为"/system/lib/modules/wlan.ko" ）， 然后启动 wpa_supplicant （ 配置文件硬编码为"/data/misc/wifi/wpa_supplicant.conf"），再通过 WifiStateTracker 来启动 WifiMonitor 中的监视线程。



查找热点（AP）：

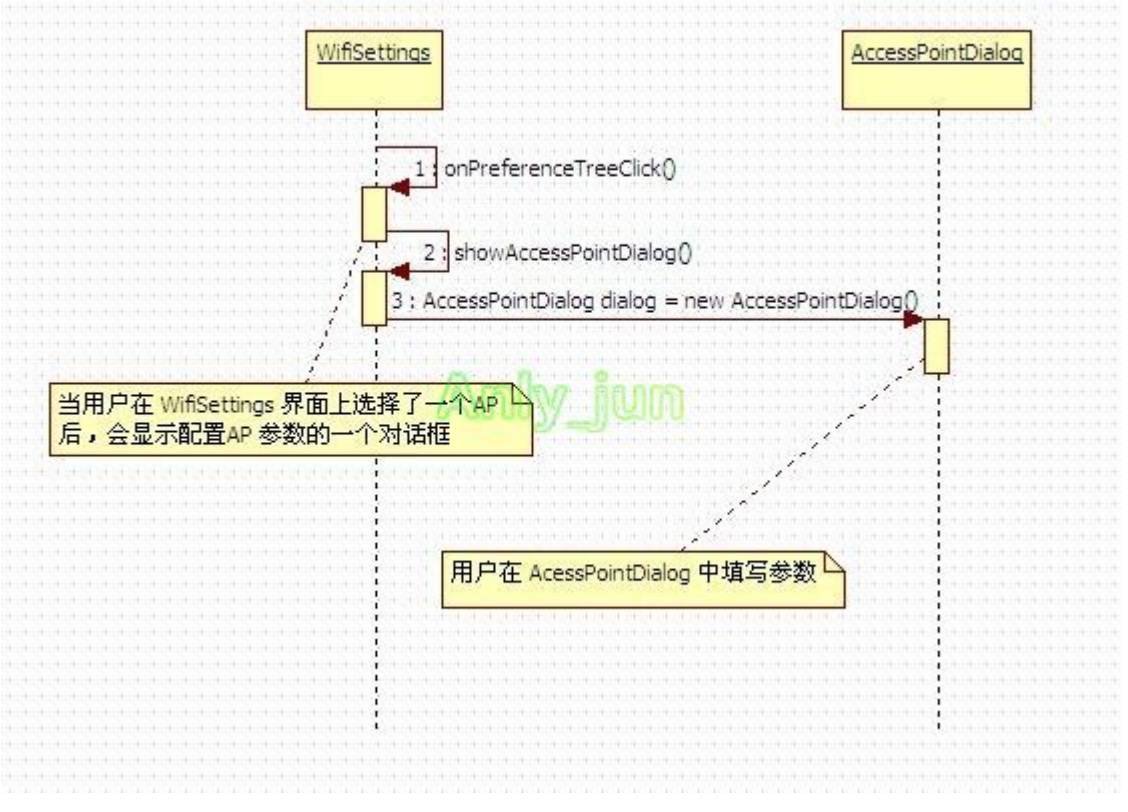
（Wifi 开启）中讲到 Wifi 模块开启后会对外发送 WIFI_STATE_CHANGED_ACTION。WifiLayer 中注册了 Action 的 Receiver。当 WifiLayer 收到此 Action 后开始 scan 的流程，具体如下



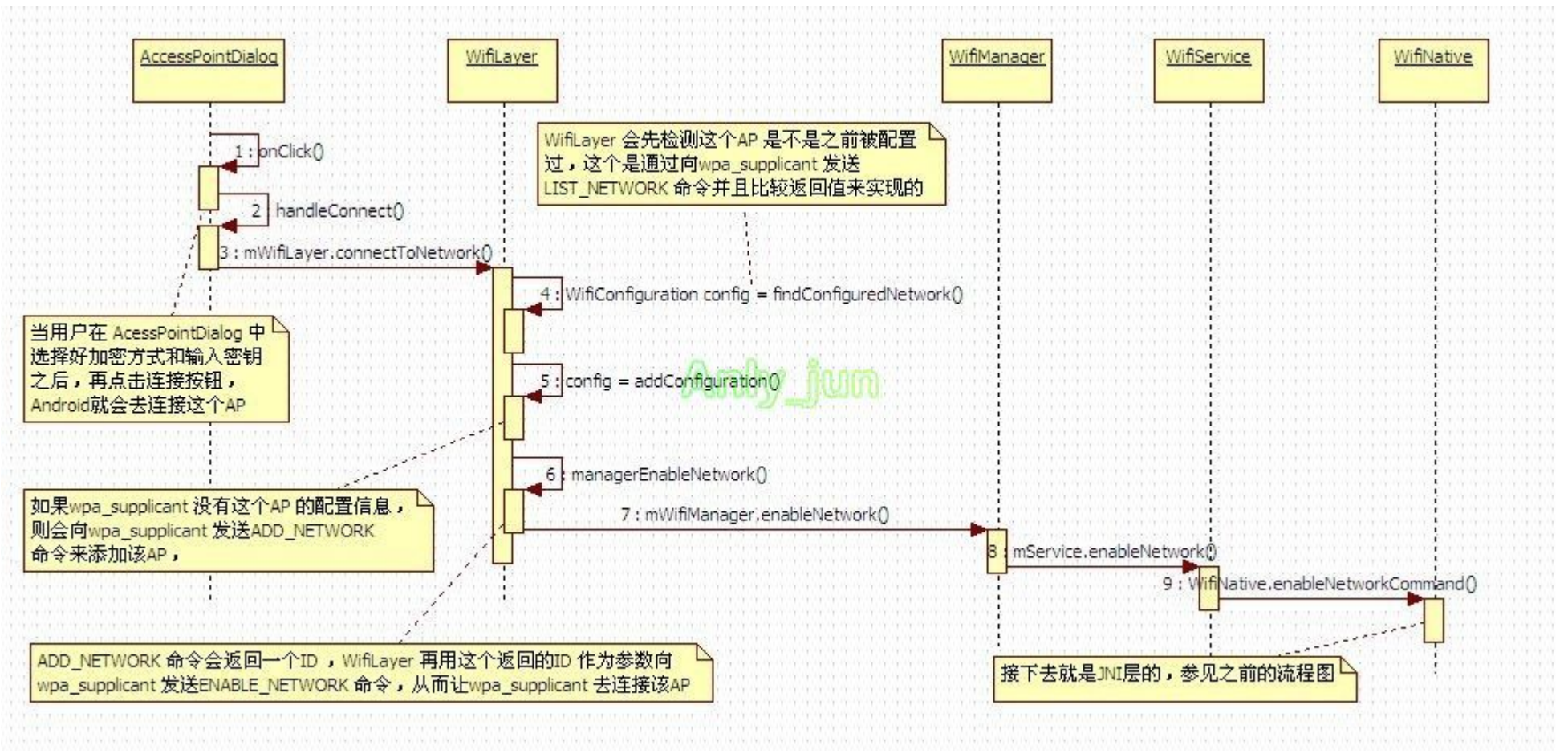
当 wpa_supplicant 处理完 SCAN 命令后，它会向控制通道发送事件通知扫描完成，从 wifi_wait_for_event 函数会接收到该事件，由此 WifiMonitor 中的 MonitorThread 会被执行来出来这个事件：

配置 AP 参数:

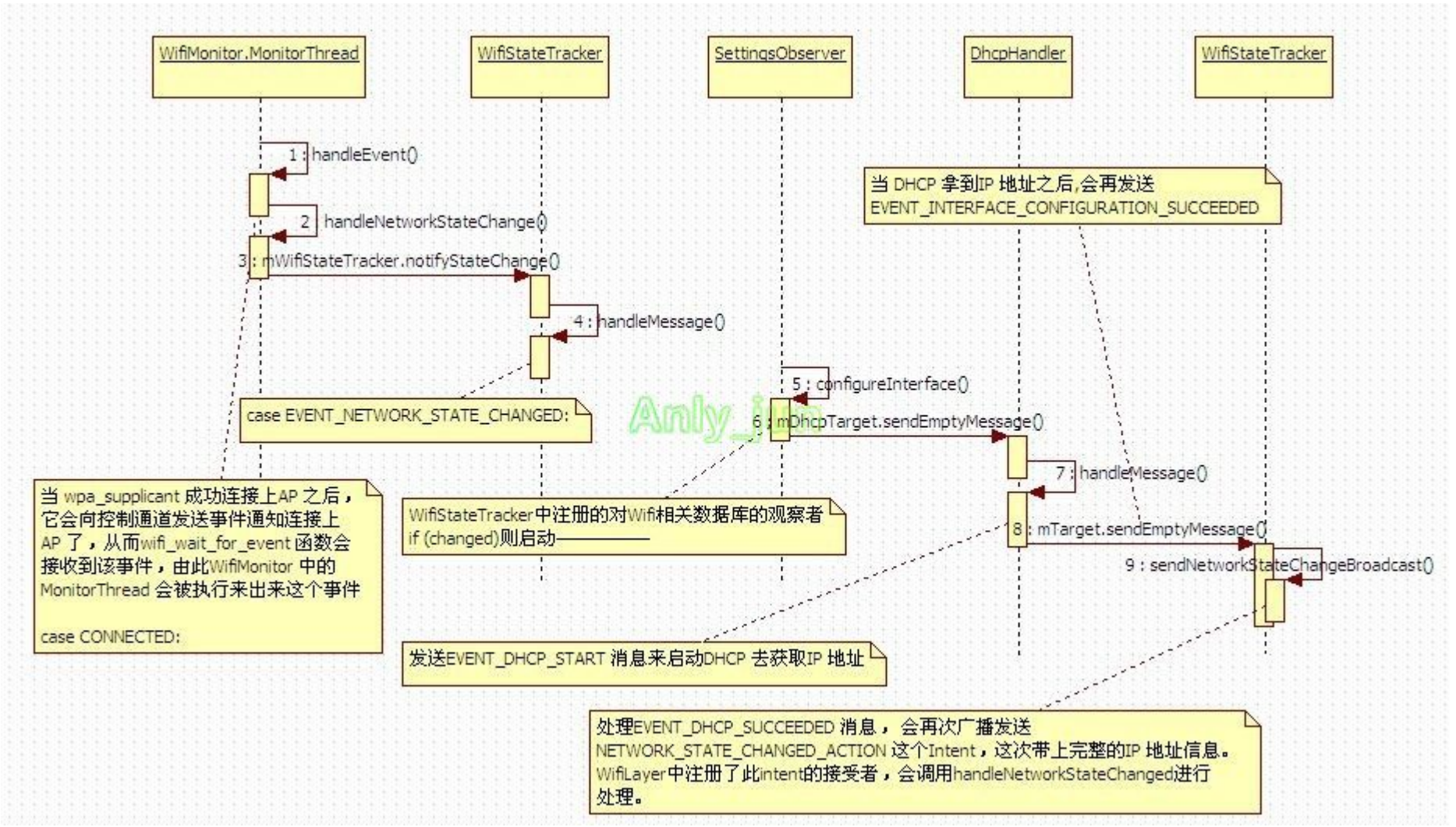
当用户在 WifiSettings 界面上选择了一个 AP 后，会显示配置 AP 参数的一个对话框：

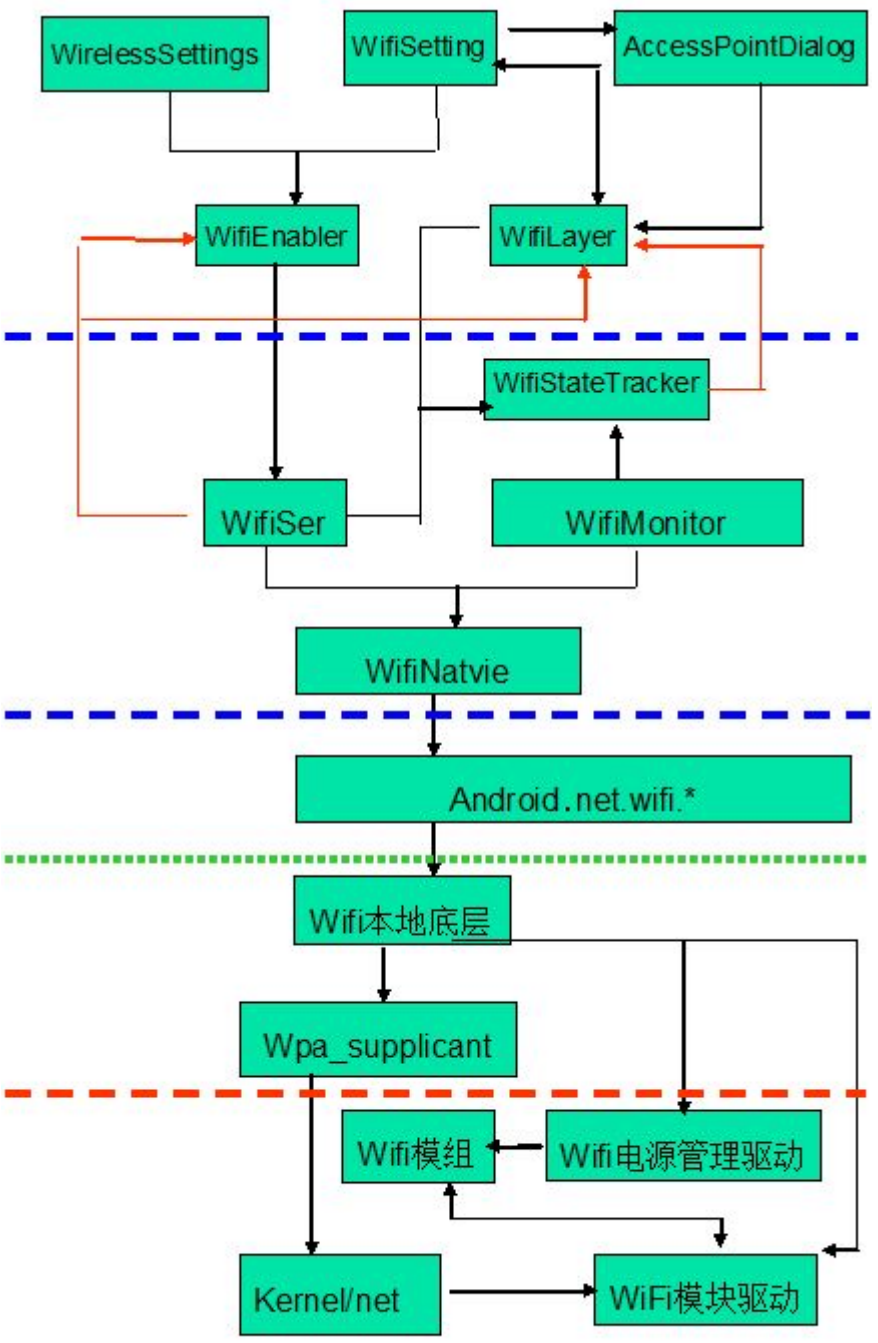

WifiSettings
AccessPointDialog
1 : onPreferenceTreeClick()
2 : showAccessPointDialog()
3 : AccessPointDialog dialog = new AccessPointDialog()
当用户在 WifiSettings 界面上选择了一个AP
后，会显示配置AP 参数的一个对话框
Andy_jun
用户在 AcessPointDialog 中填写参数

Wifi 连接:


AccessPointDialog
WifiLayer
WifiManager
WifiService
WifiNative
1 : onClick()
2 handleConnect()
3 : mWifiLayer.connectToNetwork()
WifiLayer 会先检测这个AP 是不是之前被配置过，这个是通过向wpa_supplicant 发送LIST_NETWORK 命令并且比较返回值来实现的
4 : WifiConfiguration config = findConfiguredNetwork()
当用户在 AcessPointDialog 中选择好加密方式和输入密钥之后，再点击连接按钮，Android就会去连接这个AP
5 : config = addConfiguration()
Andy_jun
6 : managerEnableNetwork()
如果wpa_supplicant 没有这个AP 的配置信息，则会向wpa_supplicant 发送ADD_NETWORK 命令来添加该AP，
7 : mWifiManager.enableNetwork()
8 : mService.enableNetwork()
9 : WifiNative.enableNetworkCommand()
ADD_NETWORK 命令会返回一个ID ，WifiLayer 再用这个返回的ID 作为参数向wpa_supplicant 发送ENABLE_NETWORK 命令，从而让wpa_supplicant 去连接该AP
接下去就是JNI层的，参见之前的流程图

IP 地址的配置:


WifiMonitor.MonitorThread
WifiStateTracker
SettingsObserver
DhcpHandler
WifiStateTracker
1 : handleEvent()
2 : handleNetworkStateChange()
3 : mWifiStateTracker.notifyStateChange()
4 : handleMessage()
当 DHCP 拿到IP 地址之后,会再发送
EVENT_INTERFACE_CONFIGURATION_SUCCEEDED
5 : configureInterface()
6 : mDhcpTarget.sendEmptyMessage()
case EVENT_NETWORK_STATE_CHANGED:
Anly_jun
7 : handleMessage()
8 : mTarget.sendEmptyMessage()
9 : sendNetworkStateChangeBroadcast()
当 wpa_supplicant 成功连接上AP 之后，
它会向控制通道发送事件通知连接上
AP 了，从而wifi_wait_for_event 函数会
接收到该事件，由此WifiMonitor 中的
MonitorThread 会被执行来出来这个事件
case CONNECTED:
WifiStateTracker中注册的对Wifi相关数据库的观察者
if (changed)则启动————————
发送EVENT_DHCP_START 消息来启动DHCP 去获取IP 地址
处理EVENT_DHCP_SUCCEEDED 消息， 会再次广播发送
NETWORK_STATE_CHANGED_ACTION 这个Intent，这次带上完整的IP 地址信息。
WifiLayer中注册了此intent的接受者，会调用handleNetworkStateChanged进行
处理。


WirelessSettings
WifiSetting
AccessPointDialog
WifiEnabler
WifiLayer
WifiStateTracker
WifiSer
WifiMonitor
WifiNatvie
Android.net.wifi.*
Wifi本地底层
Wpa_supplicant
Wifi模组
Wifi电源管理驱动
Kernel/net
WiFi模块驱动

八、wifi 新增功能

Wifi Direct 技术简介

Wi-Fi Direct 标准是指允许无线网络中的设备无需通过无线路由器即可相互连接。与　蓝牙技术类似，这种标准允许无线设备以点对点形式互连，不过在传输速度与传输距离方面　则比蓝牙有大幅提升。

Wi-Fi Direct 技术特点

移动性与便携性：Wi-Fi Direct 设备能够随时随地实现互相连接。由于不需要 Wi-Fi 路由器或接入点，因此 Wi-Fi 设备可以在任何地点实现连接。
即时可用性：用户将得以利用带回家的第一部 Wi-Fi Direct 认证设备建立直接连接。　例如，一部新购买的 Wi-Fi Direct 笔记本可以与用户已有的传统 Wi-Fi 设备创建直接连接。
易用性：Wi-Fi Direct 设备发现（Device Discovery)与服务发现（Service Discovery)功能帮助用户确定可用的设备与服务，然后建立连接。例如，如果用户想要打　印文件，他们可以通过上述服务了解到哪个 Wi-Fi 网络拥有打印机。
简单而安全的连接：Wi-Fi Direct 设备采用 Wi-Fi Protected Setup™简化了在设备　之间创建安全连接的过程。用户可以按下任一设备上的按钮，也可以输入 PIN 码（即设备显　示的 PIN 码），轻松创建安全连接。

Wi-Fi Direct 主要优点：传输速率高，兼容原有设备。
Wi-Fi Direct 主要缺点：耗电量高（较之蓝牙）。

Android P2P 源码学习

①：WifiP2pSettings Wi Fi P2P 设置

方法	解释
mReceiver	处理接收到的各种 Action 方法： WIFI_P2P_PEERS_CHANGED_ACTION：调用 requestPeers； WIFI_P2P_CONNECTION_CHANGED_ACTION：获得 NetworkInfo，判断是否为连接状态； WIFI_P2P_THIS_DEVICE_CHANGED_ACTION：调用 updateDevicePref
onCreate	创建时调用 获得系统服务 WifiP2pManager 和　WifiP2pManager.Channel; mConnectListener/mDisconnectListener 为　wifi P2pDialog 设置　Listener; setHasOptionsMenu(true)设置操作菜单
onResume()	注册 mReceiver; discoverPeers()探索附近设备
onPause()	注销 mReceiver;
onOptionsItemSelected	menu 事件响应： MENU_ID_SEARCH：discoverPeers()探索附近设备； MENU_ID_CREATE_GROUP：createGroup()创建群组； MENU_ID_REMOVE_GROUP：removeGroup()移除群组； MENU_ID_ADVANCED：　未处理；
onPreferenceTreeClick	点击一个设备，根据设备状态弹出相关对话框
updateDevicePref	更新设备属性 首先设置 device 的配置信息，然后调用 onPeersAvailable()方法更新 UI
onPeersAvailable	更新 UI

②

WifiP2pDialog 方法	解释
getConfig()	获得 P2P 配置信息，获得 device 地址和　wps
onCreate	创建时候调用，设置 device 的 address 和 name 等信息。

③

WifiP2pEnabler

方法	理解

mReceiver	处理接收到的各种 Action 方法： WIFI_P2P_STATE_CHANGED_ACTION：调用方法 handleP2pStateChanged()进行处理
handleP2pStateChanged	根据状态设置 checkbox 的属性值
onPreferenceChange	根据 checkbox 的属性变化，开启/关闭 wifi p2p
resume/pause	注册/注销 mReceiver，设置/取消 checkbox 的 Listener

④

WifiP2pPeer

方法	理解
onBindView	设置基本配置信息(名称，地址，rssi 和信号量的图片)，调用 refresh()设置 summary
compareTo	比较某 device 是否和本地的 device 相同
getLevel	获得信号等级

九、Android Wi-Fi Display（Miracast）介绍

2012 年 11 月中旬，Google 发布了 Android 4.2。虽然它和 Android 4.1 同属 Jelly Bean 系列，但却添加了很多新的功能。其中，在显示部分，Android 4.2 在 Project Butter 基础上再接再厉，新增了对 Wi-Fi Display 功能的支持。由此也导致整个显示架构发生了较大的变化。

本文首先介绍 Wi-Fi Display 的背景知识，然后再结合代码对 Android 4.2 中 Wi-Fi Display 的实现进行介绍。

一背景知识介绍

Wi-Fi Display 经常和 Miracast 联系在一起。实际上，Miracast 是 Wi-Fi 联盟（Wi-Fi Alliance）对支持 Wi-Fi Display 功能的设备的认证名称。通过 Miracast 认证的设备将在最大程度内保持对 Wi-Fi Display 功能的支持和兼容。由此可知，Miracast 考察的就是 Wi-Fi Display（本文后续将不再区分 Miracast 和 Wi-Fi Display）。而 Wi-Fi Display 的核心功能就是让设备之间通过 Wi-Fi 无线网络来分享视音频数据。以一个简单的应用场景为例：有了 Wi-Fi Display 后，手机和电视机之间可以直接借助 Wi-Fi，而无需硬连线（如 HDMI）就可将手机中的视频投递到 TV 上去显示[①]。以目前智能设备的发展趋势来看，Wi-Fi Display 极有可能在较短时间内帮助我们真正实现多屏互动。

从技术角度来说，Wi-Fi Display 并非另起炉灶，而是充分利用了现有的 Wi-Fi 技术。图 1 所示为 Wi-Fi Display 中使用的其他 Wi-Fi 技术项。

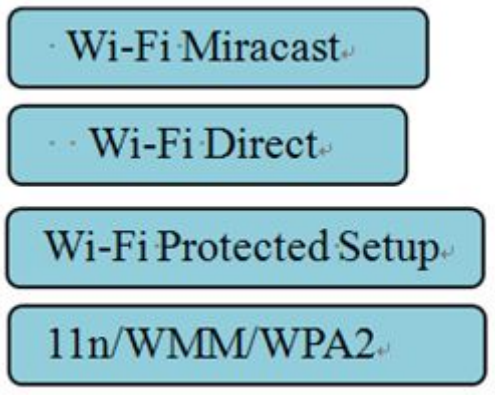


图 1 Miracast 的支撑体系结构

由图 1 可知，Miracast 依赖的 Wi-Fi 技术项[②]有：

- Wi-Fi Direct，也就是 Wi-Fi P2P。它支持在没有 AP（Access Point）的情况下，两个 Wi-Fi 设备直连并通信。
- Wi-Fi Protected Setup：用于帮助用户自动配置 Wi-Fi 网络、添加 Wi-Fi 设备等。
- 11n/WMM/WPA2：其中，11n 就是 802.11n 协议，它将 11a 和 11g 提供的 Wi-Fi 传输速率从 56Mbps 提升到 300 甚至 600Mbps。WMM 是 Wi-Fi Multimedia 的缩写，是一种针对实时视音频数据的 QoS 服务。而 WPA2 意为 Wi-Fi Protected Acess 第二版，主要用来给传输的数据进行加密保护。

上述的 Wi-Fi 技术中，绝大部分功能由硬件厂商实现。而在 Android 中，对 Miracast 来说最重要的是两个基础技术：

- Wi-Fi Direct：该功能由 Android 中的 WifiP2pService 来管理和控制。
- Wi-Fi Multimedia：为了支持 Miracast，Android 4.2 对 MultiMedia 系统也进行了修改。

下边我们对 Miracast 几个重要知识点进行介绍，首先是拓扑结构和视音频格式方面的内容。

Miracast 一个重要功能就是支持 Wi-Fi Direct。但它也考虑了无线网络环境中存在 AP 设备的情况下，设备之间的互联问题。读者可参考如图 2 所示的四种拓扑结构。

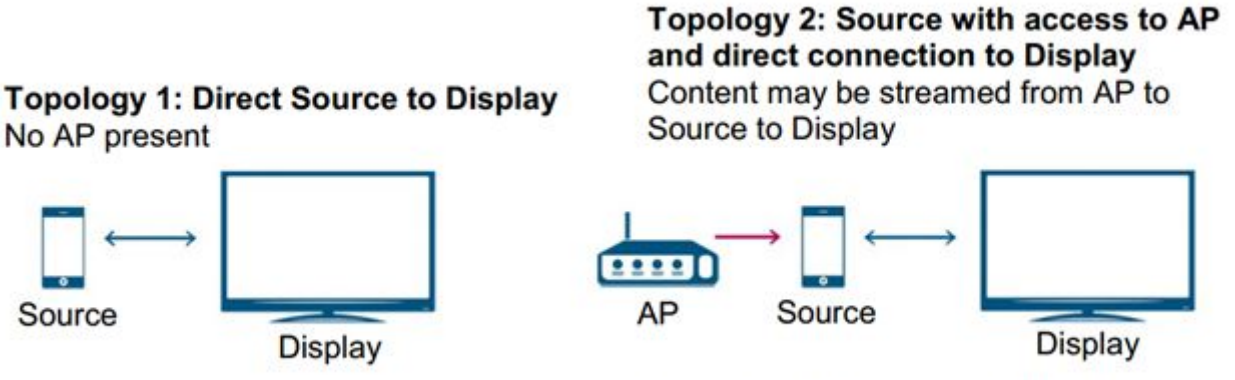


图 2　Miracast 的四种拓扑结构

图 2 所示内容比较简单，此处就不再详述。另外，在 Wi-Fi Display 规范中，还存在着 Source 将 Video 和 Audio 内容分别传送给不同 Render Device 的情况。感兴趣的读者可参考 Wi-Fi Display 技术规范。

另外，Miracast 对所支持的视音频格式也进行了规定，如表 1 所示。

表 1　Miracast 视音频格式支持

分辨率	17 种 CEA 格式，分辨率从 640*480 到 1920*1080，帧率从 24 到 60 29 种 VESA 格式，分辨率从 800*600 到 1920*1200，帧率从 30 到 60 12 种手持设备格式，分辨率从 640*360 到 960*540，帧率从 30 到 60
视频	H.264 高清
音频	必选：LPCM 16bits，48kHz 采样率，双声道 可选： LPCM 16bits，44.1kHz 采样率，双声道 Advanced Audio coding Dolby Advanced Codec 3

最后，我们简单介绍一下 Miracast 的大体工作流程。Miracast 以 session 为单位来管理两个设备之间的交互的工作，主要步骤包括（按顺序）：

- Device Discovery：通过 Wi-Fi P2P 来查找附近的支持 Wi-Fi P2P 的设备。
- Device Selection：当设备 A 发现设备 B 后，A 设备需要提示用户。用户可根据需要选择是否和设备 B 配对。
- Connection Setup：Source 和 Display 设备之间通过 Wi-Fi P2P 建立连接。根据 Wi-Fi Direct 技术规范，这个步骤包括建立一个 Group Owner 和一个 Client。此后，这两个设备将建立一个 TCP 连接，同时一个用于 RTSP 协议的端口将被创建用于后续的 Session 管理和控制工作。
- Capability Negotiation：在正式传输视音频数据前，Source 和 Display 设备需要交换一些 Miracast 参数信息，例如双方所支持的视音频格式等。二者协商成功后，才能继续后面的流程。
- Session Establishment and streaming：上一步工作完成后，Source 和 Display 设备将建立一个 Miracast Session。而后就可以开始传输视音频数据。Source 端的视音频数据将经由 MPEG2TS 编码后通过 RTP 协议传给 Display 设备。Display 设备将解码收到的数据，并最终显示出来。
- User Input back channel setup：这是一个可选步骤。主要用于在传输过程中处理用户发起的一些控制操作。这些控制数据将通过 TCP 在 Source 和 Display 设备之间传递。
- Payload Control：传输过程中，设备可根据无线信号的强弱，甚至设备的电量状况来动态调整传输数据和格式。可调整的内容包括压缩率，视音频格式，分辨率等内容。
- Session teardown：停止整个 Session。

通过对上面背景知识的介绍，读者可以发现：

- Miracast 本质就是一个基于 Wi-Fi 的网络应用。这个应用包括服务端和客户端。
- 服务端和客户端必须支持 RTP/RTSP 等网络协议和相应的编解码技术。

二 Android 4.2 Miracast 功能实现介绍

Miracast 的 Android 实现涉及到系统的多个模块，包括：

- MediaPlayerService 及相关模块：原因很明显，因为 Miracast 本身就牵扯到 RTP/RTSP 及相应的编解码技术。
- SurfaceFlinger 及相关模块：SurfaceFlinger 的作用是将各层 UI 数据混屏并投递到显示设备中去显示。现在，SurfaceFlinger 将支持多个显示设备。而支持 Miracast 的远端设备也做为一个独立的显示设备存在于系统中。

- WindowManagerService 及相关模块：WindowManagerService 用于管理系统中各个 UI 层的位置和属性。由于并非所有的 UI 层都会通过 Miracast 投递到远端设备上。例如手机中的视频可投递到远端设备上去显示，但假如在播放过程中，突然弹出一个密码输入框（可能是某个后台应用程序发起的），则这个密码输入框就不能投递到远端设备上去显示。所以，WindowManagerService 也需要修改以适应 Miracast 的需要。
- DisplayManagerService 及相关模块：DisplayManagerService 服务是 Android 4.2 新增的，用于管理系统中所有的 Display 设备。

由于篇幅原因，本文将重点关注 SurfaceFlinger 和 DisplayManagerService 以及 Miracast 的动态工作流程。

2.1 SurfaceFlinger 对 Miracast 的支持

相比前面的版本，Android 4.2 中 SurfaceFlinger 的最大变化就是增加了一个名为 DisplayDevice 的抽象层。相关结构如图 3 所示：

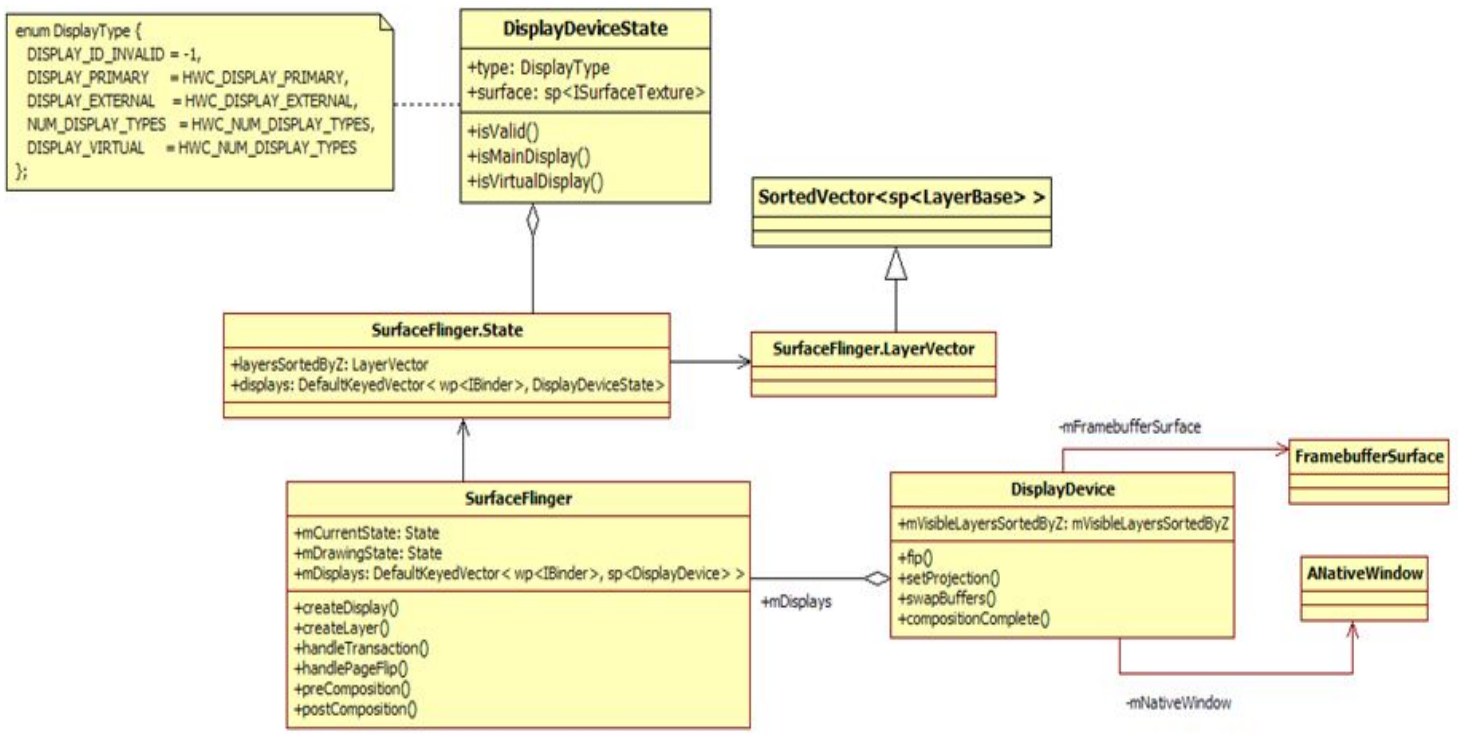


图 3　SurfaceFlinger 家族类图

由图 3 可知：

- Surface 系统定义了一个 DisplayType 的枚举，其中有代表手机屏幕的 DISPLAY_PRIMARY 和代表 HDMI 等外接设备的 DISPLAY_EXTERNAL。比较有意思的是，作为 Wi-Fi Display，它的设备类型是 DISPLAY_VIRTUAL。
- 再来看 SurfaceFlinger 类，其内部有一个名为 mDisplays 的变量，它保存了系统中当前所有的显示设备（DisplayDevice）。另外，SurfaceFlinger 通过 mCurrentState 和 mDrawingState 来控制显示层的状态。其中，mDrawingState 用来控制当前正在绘制的显示层的状态，mCurrentState 表示当前所有显示层的状态。有这两种 State 显示层的原因是不论是 Miracast 还是 HDMI 设备，其在系统中存在的时间是不确定的。例如用户可以随时选择连接一个 Miracast 显示设备。为了不破坏当前正在显示的内容，这个新显示设备的一些信息将保存到 CurrentState 中。等到 SurfaceFlinger 下次混屏前再集中处理。
- mCurrentState 和 mDrawingState 的类型都是 SurfaceFlinger 的内部类 State。由图 3 可知，State 首先通过 layerSortedByZ 变量保存了一个按 Z 轴排序的显示层数组（在 Android 中，显示层的基类是 LayerBase），另外还通过 displays 变量保存了每个显示层对应的 DisplayDeviceState。
- DisplayDeviceState 的作用是保存对应显示层的 DisplayDevice 的属性以及一个 ISurfaceTexure 接口。这个接口最终将传递给 DisplayDevice。
- DisplayDevice 代表显示设备，它有两个重要的变量，一个是 mFrameBufferSurface 和 mNativeWindow。mFrameBufferSurace 是 FrameBufferSurface 类型，当显示设备不属于 VIRTUAL 类型的话，则该变量不为空。对于 Miracast 来说，显示数据是通过网络传递给真正的显示设备的，所有在 Source 端的 SurfaceFlinger 来说，就不存在 FrameBuffer。故当设备为 VIRTUAL 时，其对应的 mFrameBufferSurface 就为空。而 ANativeWindow 是 Android 显示系统的老员工了。该结构体在多媒体的视频 I/O、OpenGL ES 等地方用得较多。而在普通的 UI 绘制中，ISurfaceTexture 接口用得较多。不过早在 Android 2.3，Google 开发人员就通过函数指针将 ANativeWindow 的各项操作和 ISurfaceTexture 接口统一起来。

作为 VIRTUAL 的 Miracast 设备是如何通过 DisplayDevice 这一层抽象来加入到 Surface 系统中来的呢？下面这段代码对理解 DisplayDevice 的抽象作用极为重要。如图 4 所示。

```
[-->SurfaceFlinger::handleTransactionLocked]
......//for循环依次处理每个mCurrentState中的State
if (!state.isVirtualDisplay()) {
      isSecure = true;
      //对于非虚拟设备来说，SurfaceTextureClient构造函数的参数直接来自FramewBufferSurface的
      //getBufferQueue。
      fbs = new FramebufferSurface(*mHwc, state.type);
      stc = new SurfaceTextureClient(
                  static_cast< sp<ISurfaceTexture> >(
                   fbs->getBufferQueue()));
  } else {
     if (state.surface != NULL) //如果是虚拟设备，则直接使用State中的surface变量
           stc = new SurfaceTextureClient(state.surface);
     isSecure = state.isSecure;
  }

   const wp<IBinder>& display(curr.keyAt(i));
   if (stc != NULL) {//创建DisplayDevice，fbs初值为空
         sp<DisplayDevice> hw = new DisplayDevice(this,
             state.type, isSecure, display, stc, fbs,mEGLConfig);
         ......
        mDisplays.add(display, hw);  //将DisplayDevice保存到mDisplays数组中
   }
```

图 4 SurfaceFlinger 代码片段

由图 4 代码可知：

- 对于非 Virtual 设备，DisplayDevice 的 FrameBufferSurface 不为空。而且 SurfaceTextureClient 的构造参数来自于 FrameBufferSurface 的 getBufferQueue 函数。
- 如果是 Virtual 设备，SurfaceTextureClient 直接使用了 State 信息中携带的 surface 变量。

凭着上面这两点不同，我们可以推测出如图 5 所示的 DisplayDevice 的作用

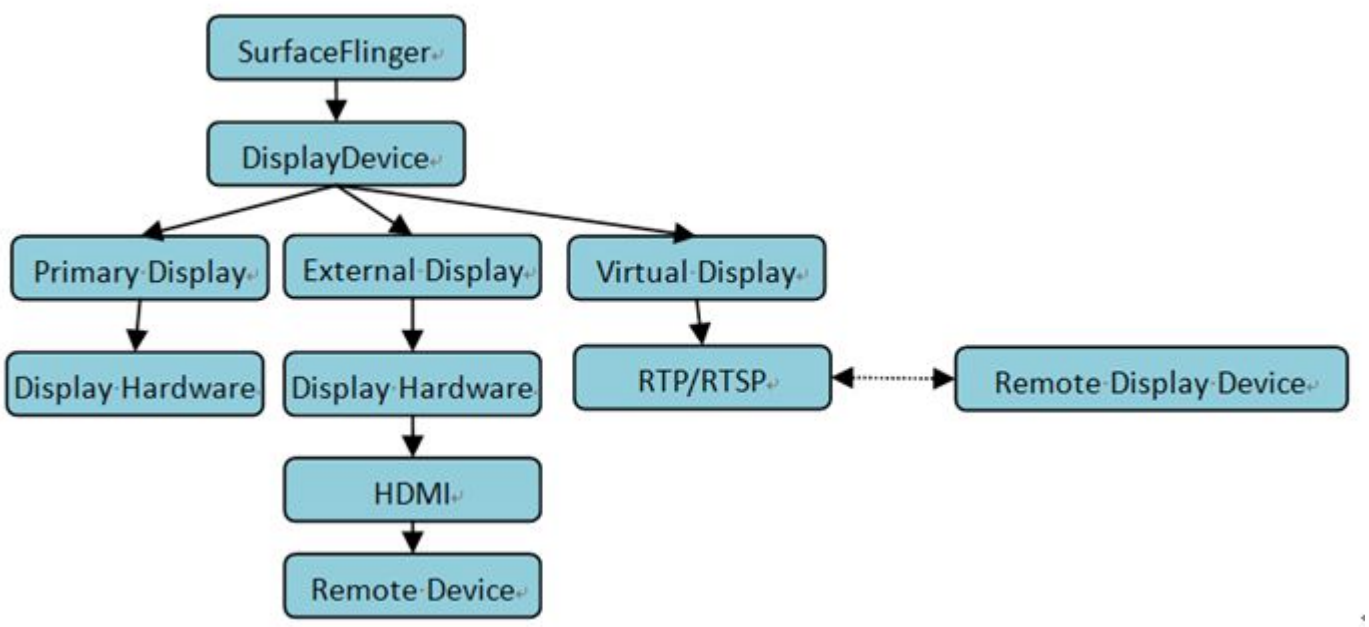


图 5 DisplayDevice 的隔离示意图

最后再来看一下 SurfaceFlinger 中混屏操作的实现，代码如图 6 所示：

```
void SurfaceFlinger::doComposition() {
    ATRACE_CALL();
    const bool repaintEverything = android_atomic_and(0, &mRepaintEverything);
    for (size_t dpy=0 ; dpy<mDisplays.size() ; dpy++) {
        const sp<DisplayDevice>& hw(mDisplays[dpy]);
        if (hw->canDraw()) {
            // transform the dirty region into this screen's coordinate space
            const Region dirtyRegion(hw->getDirtyRegion(repaintEverything));

            // repaint the framebuffer (if needed)
            doDisplayComposition(hw, dirtyRegion);

            hw->dirtyRegion.clear();
            hw->flip(hw->swapRegion);
            hw->swapRegion.clear();
        }
        // inform the h/w that we're done compositing
        hw->compositionComplete();
    }
    postFramebuffer();
} « end doComposition »
```

图 6 SurfaceFilnger 的混屏操作

由图 5 可知，SurfaceFlinger 将遍历系统中所有的 DisplayDevice 来完成各自的混屏工作。

2.2 Framework 对 Miracast 的支持

为了彻底解决多显示设备的问题，Android 4.2 干脆在 Framework 中新增了一个名为 DisplayManagerService 的服务，用来统一管理系统中的显示设备。DisplayManagerService 和系统其它几个服务都有交互。整体结构如图 7 所示。

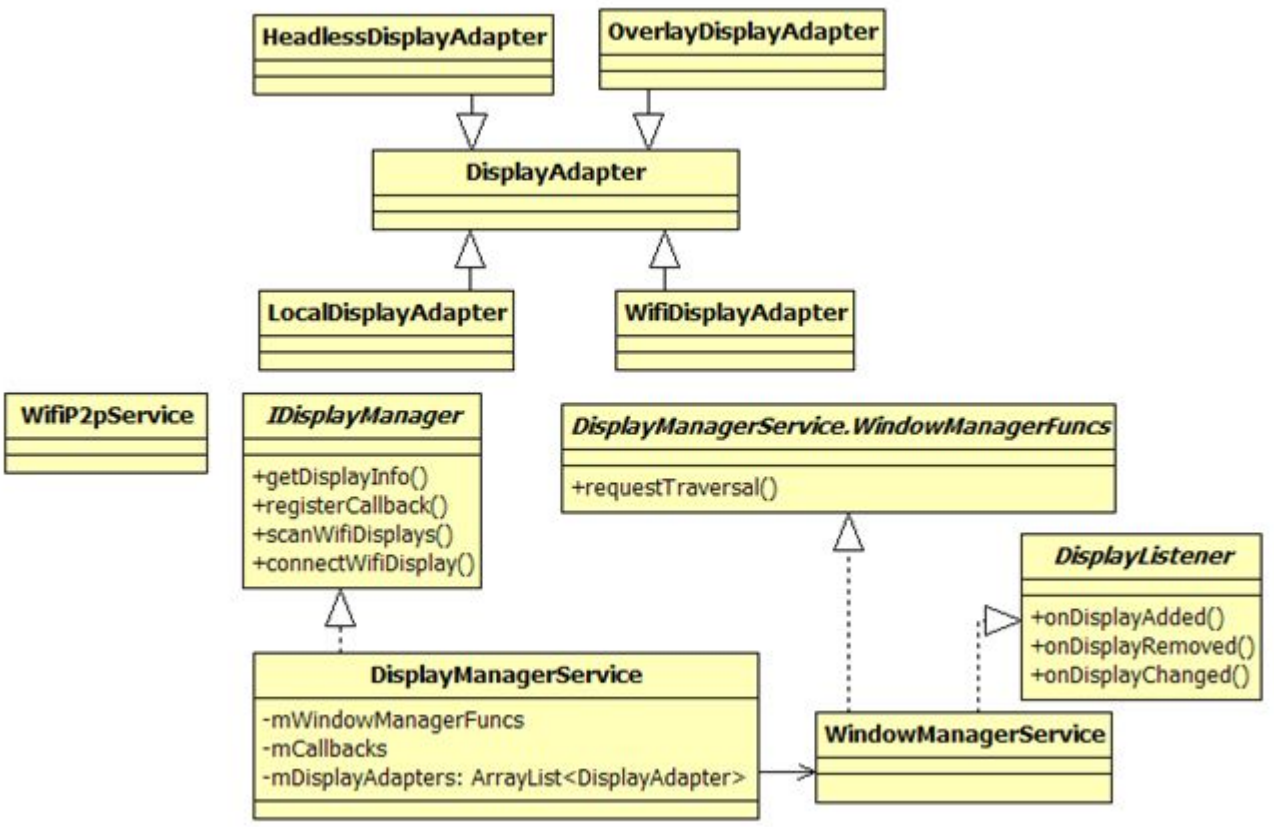


图 7 DisplayManagerService 及相关类图

由图 7 可知：

- DisplayManagerService 主要实现了 IDisplayManager 接口。这个接口的大部分函数都和 Wi-Fi Display 操作相关。
- 另外，DisplayManagerService 和 WindowManagerService 交互紧密。因为 WindowManagerService 管理系统所有 UI 显示，包括属性，Z 轴位置等等。而且，WindowManagerService 是系统内部和 SurfaceFlinger 交互的重要通道。
- DisplayManagerService 通过 mDisplayAdapters 来和 DisplayDevice 交互。每一个 DisplayDevice 都对应有一个 DisplayAdapter。
- 系统定义了四种 DisplayAdapter。HeadlessDisplayAdapter 和 OverlayDisplayAdapter 针对的都是 Fake 设备。其中 OverlayDisplay 用于帮助开发者模拟多屏幕之用。LocalDisplayAdapter 代表主屏幕，而 WifiDisplayAdapter 代表 Wi-Fi Display。

2.3 Android 中 Miracast 动态工作流程介绍

当用户从 Settings 程序中选择开启 Miracast 并找到匹配的 Device 后[③]，系统将通过 WifiDisplayController 的 requestConnect 函数向匹配设备发起连接。代码如图 8 所示：

```
··public·void·requestConnect(String·address)·{¶
·····for·(WifiP2pDevice·device·:·mAvailableWifiDisplayPeers)·{¶
·····if·(device.deviceAddress.equals(address))·{¶
·····connect(device);¶
·····}¶
·····}¶
··}¶
```

图 8 requestConnect 函数实现

图 8 中，最终将调用 connect 函数去连接指定的设备。connect 函数比较中，其中最重要的是 updateConnection 函数，我们抽取其中部分代码来看，如图 9 所示：

```
····mRemoteDisplay·=·RemoteDisplay.listen(iface,·new·RemoteDisplay.Listener()·{¶
····@Override¶
····public·void·onDisplayConnected(Surface·surface,¶
·····int·width,·int·height,·int·flags)·{¶
·····if·(mConnectedDevice·==·oldDevice·&&·!mRemoteDisplayConnected)·{¶
·····Slog.i(TAG,·"Opened·RTSP·connection·with·Wifi·display:·"¶
·····+·mConnectedDevice.deviceName);¶
·····mRemoteDisplayConnected·=·true;¶
·····mHandler.removeCallbacks(mRtspTimeout);¶

·····final·WifiDisplay·display·=·createWifiDisplay(mConnectedDevice);¶
·····advertiseDisplay(display,·surface,·width,·height,·flags);¶
·····}¶
····}¶
```

图 9 updateConnection 函数片段

在图 8 所示的代码中，系统创建了一个 RemoteDisplay，并在这个 Display 上监听（listen）。从注释中可知，该 RemoteDisplay 就是和远端 Device 交互的 RTP/RTSP 通道。而且，一旦有远端 Device 连接上，还会通过 onDisplayConnected 返回一个 Surface 对象。

根据前面对 SurfaceFlinger 的介绍，读者可以猜测出 Miracast 的重头好戏就在 RemoteDisplay 以及它返回的这个 Surface 上了。

确实如此，RemoteDisplay 将调用 MediaPlayerService 的 listenForRemoteDisplay 函数，最终会得到一个 Native 的 RemoteDisplay 对象。相关类图如图 10 所示。

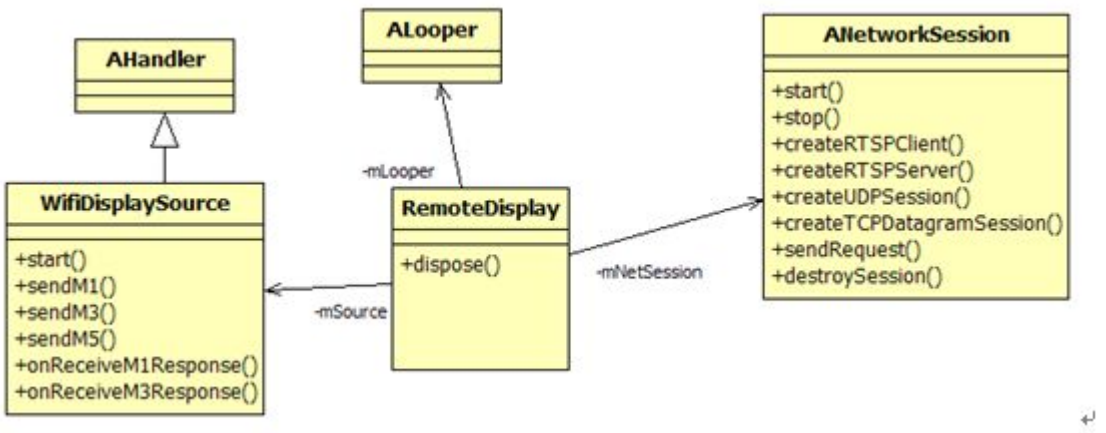


图 10 RemoteDisplay 类图

由图 10 可知，RemoteDisplay 有三个重要成员变量：

- mLooper，指向一个 ALooper 对象。这表明 RemoteDisplay 是一个基于消息派发和处理的系统。
- mNetSession 指向一个 ANetWorkSession 对象。从它的 API 来看，ANetworkSession 提供大部分的网络操作。
- mSource 指向一个 WifiDisplaySource 对象。它从 AHandler 派生，故它就是 mLooper 中消息的处理者。注意，图中的 M1、M3、M5 等都是 Wi-Fi Display 技术规范中指定的消息名。

RemoteDisplay 构造函数中，WifiDisplaySource 的 start 函数将被调用。如此，一个类型为 kWhatStart 的消息被加到消息队列中。该消息最终被 WifiDisplaySource 处理，结果是一个 RTSPServer 被创建。代码如图 11 所示：

```
if (err == OK) {
    if (inet_aton(iface.c_str(), &mInterfaceAddr) != 0) {
        sp<AMessage> notify = new AMessage(kWhatRTSPNotify, id());

        err = mNetSession->createRTSPServer(
                mInterfaceAddr, port, notify, &mSessionID);
    } else {
        err = -EINVAL;
    }
}
```

图 11 kWhatStart 消息的处理结果

以后，客户端发送的数据都将通过类型为 kWhatRTSPNotify 的消息加入到系统中来。而这个消息的处理核心在 onReceiveClientData 函数中，它囊括了设备之间网络交互的所有细节。其核心代码如图 12 所示：

```
status_t err;
if (method == "OPTIONS") {
    err = onOptionsRequest(sessionID, cseq, data);
} else if (method == "SETUP") {
    err = onSetupRequest(sessionID, cseq, data);
} else if (method == "PLAY") {
    err = onPlayRequest(sessionID, cseq, data);
} else if (method == "PAUSE") {
    err = onPauseRequest(sessionID, cseq, data);
} else if (method == "TEARDOWN") {
    err = onTeardownRequest(sessionID, cseq, data);
} else if (method == "GET_PARAMETER") {
    err = onGetParameterRequest(sessionID, cseq, data);
} else if (method == "SET_PARAMETER") {
    err = onSetParameterRequest(sessionID, cseq, data);
} else {
    sendErrorResponse(sessionID, "405 Method Not Allowed", cseq);
```

图 12 onReceiveClientData 核心代码示意

图 12 的内容较多，建议读者根据需要自行研究。

根据前面的背景知识介绍，设备之间的交互将由 Session 来管理。在代码中，Session 的概念由 WifiSource 的内部类 PlaybackSession 来表示。先来看和其相关的类图结构，如图 13 所示：

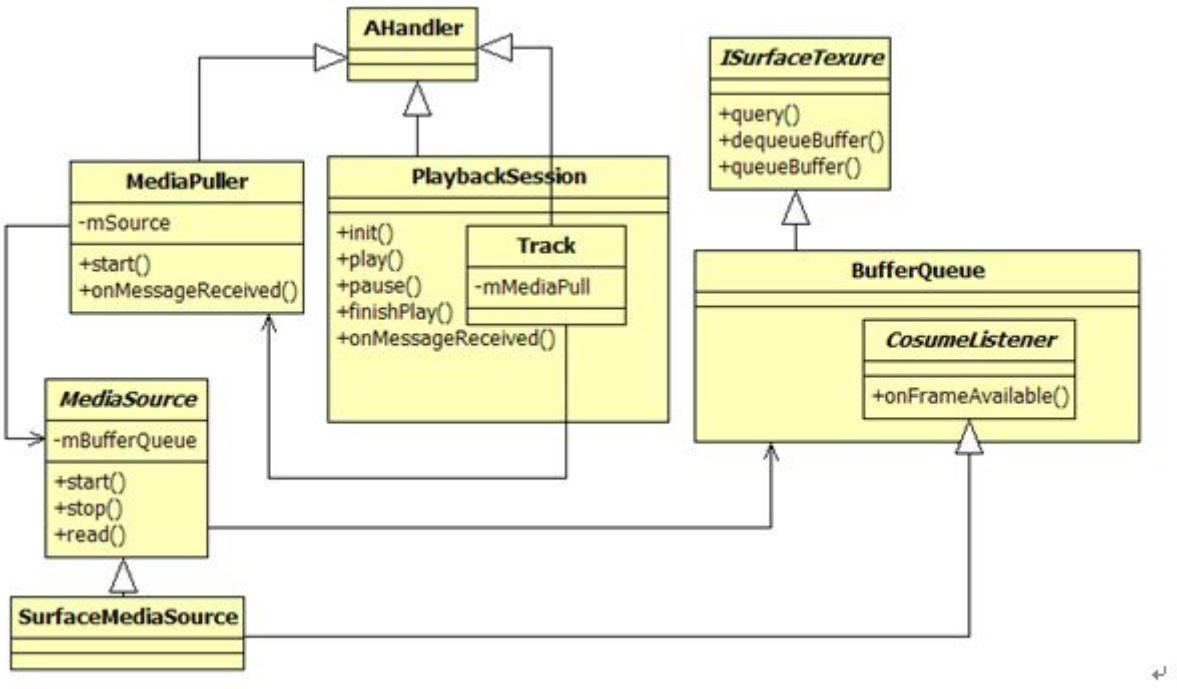


图 13 PlaybackSession 及相关类图

由图 13 可知：

- PlaybackSession 及其内部类 Track 都从 AHandler 派生。故它们的工作也依赖于消息循环和处理。Track 代表视频流或音频流。
- Track 内部通过 mMediaPull 变量指向一个 MediaPull 对象。而 MediaPull 对象则保存了一个 MediaSource 对象。在 PlaybackSession 中，此 MediaSource 的真正类型为 SurfaceMediaSource。它表明该 Media 的源来自 Surface。
- BufferQueue 从 ISurfaceTexure 中派生，根据前面对 SurfaceFlinger 的介绍，它就是 SurfaceFlinger 代码示例中代表虚拟设备的 State 的 surface 变量。

当双方设备准备就绪后，MediaPull 会通过 kWhatPull 消息处理不断调用 MediaSource 的 read 函数。在 SurfaceMediaSource 实现的 read 函数中，来自 SurfaceFlinger 的混屏后的数据经由 BufferQueue 传递到 MediaPull 中。代码如图 14 所示：

```
case kWhatPull:
{
    int32_t generation;
    CHECK(msg->findInt32("generation", &generati

    if (generation != mPullGeneration) {
        break;
    }

    MediaBuffer *mbuf;
    status_t err = mSource->read(&mbuf);
```

```
status_t SurfaceMediaSource::read( MediaBuffer **buffer,
                                   const ReadOptions *options)
{
    while (mStarted) {

        status_t err = mBufferQueue->acquireBuffer(&item);
        if (err == BufferQueue::NO_BUFFER_AVAILABLE) {
            // wait for a buffer to be queued
            mFrameAvailableCondition.wait(mMutex);
        } else if (err == OK) {

            // First time seeing the buffer?  Added it to the SMS slot
            if (item.mGraphicBuffer != NULL) {
                mBufferSlot[item.mBuf] = item.mGraphicBuffer;
```

图 14 MediaPull 和 SurfaceMediaSource 的代码示意

从图 13 可知：

- 左图中，MediaPull 通过 kWhatPull 消息不断调用 MediaSource 的 read 函数。
- 右图中，SurfaceMediaSource 的 read 函数由通过 mBufferQueue 来读取数据。

那么 mBufferQueue 的数据来自什么地方呢？对，正是来自图 4 的 SurfaceFlinger。

当然，PlaybackSession 拿到这些数据后还需要做编码，然后才能发送给远端设备。由于篇幅关系，本文就不再讨论这些问题了。

三总结

本文对 Miracast 的背景知识以及 Android 系统中 Miracast 的实现进行了一番简单介绍。从笔者个人角度来看，有以下几个点值得感兴趣的读者注意：

- 一定要结合 Wi-Fi 的相关协议去理解 Miracast。重点关注的协议包括 Wi-Fi P2p 和 WMM。
- Android Miracast 的实现中，需要重点理解 SurfaceFlinger 和 RemoteDisplay 模块。这部分的实现不仅代码量大，而且类之间，以及线程之间关系复杂。
- 其他需要注意的点就是 DisplayManagerService 及相关模块。这部分内容在 SDK 中有相关 API。应用开发者应关注这些新 API 是否能帮助自己开发出更有新意的应用程序。

另外，Android 的进化速度非常快，尤其在几个重要的功能点上。作者在此也希望国内的手机厂商或那些感兴趣的移动互联网厂商能真正投入力量做一些更有深度和价值的研发工作。

十、wifi 和蓝牙

首先交代开发环境——硬件平台：高通 MSM8225，OS：Android2.3.5，无线模块：brodcom BCM4330。

一、WIFI：

首先保证上好电：在 platform/kernel/arch/arm/mach-msm/board-msm7x30.c 中实现 GPIO 管脚的配置，也要配置好休眠唤醒的脚。再就是在板级__init 里面配置好 WIFI 的上电以及下电时序问题，还有因为我们的无线芯片是基于 SDIO 进行通讯的，所以 SDIO 的 driver 也要配置好。

再就是 WIFI 本身的驱动的开发了，我们是用博通公司提供的代码，在 dhd_custemer_gpio.c 中修改好相应的 Power_en 和 reset 脚的配置，然后配置 wlan linux 相关的驱动。最终编译生成 dhd.ko 文件。

这个 dhd.ko 文件和 nvram.txt 以及 sdio.bin 文件共同加载了 wlan0 的驱动。在中间层有一个 hardware/library_lib/wifi/wifi.c，就是把一些配置文件的加载，DHCP 文件的文件路径等等综合在一起。

在 wifi 移植过程中一定要注意 MAC 地址的有效性和可用性。这样才能连接热点已经被分配到 IP 地址。

二、Bluetooth：

首先还是和 wifi 一样，上电是必须的！在 platform/kernel/arch/arm/mach-msm/board-msm7x30.c 中实现 bluetooth_power_init，注意是在参数初始化的函数__init msm7x30_init 函数中完成。

由于 BT 的协议层，在开发的时候基本不用动什么，所以我们直接找跟芯片相关的文件进行修改。brodcom BT 相关的文件在 system/bluetooth/brcm_patchram_plus/

brcm_patchram_plus.c 中，里面主要完成了如下事件：1、读取 BT 的地址。2、执行 parse 相关命令。3、重启 HCI 服务，4、设置蓝牙通信的波特率。5、设置 hci_pcm

再就是开启 BT 的服务。目录在 system/core/rootdir/A1/etc/init.qcom.bt.sh 中，实现 start_hciattach 和 kill_hciattach 的功能。

网上关于 BT 的驱动很少，所以我在开发过程中把其中的步骤记录下来。供大家相互学习讨论。

一、关于 BT driver 的移植：

1. Enablebluetootch in BoadConfig.mk

BOARD_HAVE_BLUETOOTH := true

2.实现 BT 电源管理 rfkill 驱动。

Kernel/driver/bluetooth/bluetooth-power.c 高通的这个文件基本上不用动。

在 kernel\arch\arm\mach_msm7x27.c: static int bluetooth_power(int on)中

实现：上电：把 bt_reset pin 和 bt_reg_on pin 拉低

mdelay（10）；

把 bt_resetpin 和 bt_reg_on pin 拉高

mdelay（150）

下电：把 bt_reset pin 和 bt_reg_on pin 拉低

3. RebuildAndroid image and reboot

命令行测试：

echo 0 >/sys/class/rfkill/rfkill0/state //BT 下电

echo 1 >/sys/class/rfkill/rfkill0/state //BT 上电

brcm_patchram_plus-d --patchram /etc/firmware/BCM4329B1_002.002.023.0061.0062.hcd/dev/ttyHS0

hciattach -s115200 /dev/ttyHS0 any

没任何错误提示是可以用以下测试

hciconfig hci0up

hcitool scan

4.实现 BT 睡眠唤醒机制

Kernel\drivers\bluetooth\bluesleep.c 一般来说这个文件改动比较少，但可能逻辑上会有些问题。需要小的改动。

在 kernel\arch\arm\mach_xxx/board_xxx.c：bluesleep_resources 中定义 gpio_host_wake（BT 唤醒 host 脚）、gpio_ext_wake（host 唤醒 BT 脚）、host_wake（BT 唤醒 host 的中断号）。

注：各个平台的 board_xxx.c 文件名字不同，请客户确认

5.系统集成

1）在 init.qcom.rc 中确认有下面的内容：

```
service hciattach/system/bin/sh /system/etc/init.qcom.bt.sh
        user bluetooth
        group qcom_oncrpc bluetooth net_bt_admin
        disabled
        oneshot
```

2）修改 init.qcom.bt.sh

确认有：

```
BLUETOOTH_SLEEP_PATH=/proc/bluetooth/sleep/proto
echo 1 >$BLUETOOTH_SLEEP_PATH
```

/system/bin/hciattach-n /dev/ttyHS0 any 3000000 flow & 改为：

```
./brcm_patchram_plus--enable_lpm  - enable_hci --patchram /system/etc/wifi/BCM4329BT.hcd --baudrate3000000 /dev/ttyHS0 &
```

注掉：高通下载 firmware 的命令。

6.重新编译 system。此时 BT 应该能运行了。

二、BT 的休眠唤醒配置

BT 的休眠在 driver/bluetooth/bluesleep.c 中，首先驱动的名字叫“bluesleep”与 arch/arm/mach-msm/board-msm7x30.c 相匹配就执行 platform_driver_probe(&bluesleep_driver, bluesleep_probe)然后调用 static int __init bluesleep_probe(struct platform_device *pdev)，这里会配置两个引脚 HOST_WAKE_BT & BT_WAKE_HOST

```
bsi = kzalloc(sizeof(struct bluesleep_info), GFP_KERNEL);
if (!bsi)
                return -ENOMEM;

res = platform_get_resource_byname(pdev, IORESOURCE_IO,
                                                "gpio_host_wake");
if (!res) {
                BT_ERR("couldn't find host_wake gpio\n");
                ret = -ENODEV;
                goto free_bsi;
}
bsi->host_wake = res->start;
//[SIMT-zhangmin-111230] change the configuration of BT sleep gpio from bt_power to here {
gpio_tlmm_config(GPIO_CFG(bsi->host_wake, 0, GPIO_CFG_INPUT, GPIO_CFG_NO_PULL, GPIO_CFG_2MA),GPIO_CFG_ENABLE);
//[SIMT-zhangmin-111230] }
ret = gpio_request(bsi->host_wake, "bt_host_wake");
if (ret)
```

```
                goto free_bsi;
        ret = gpio_direction_input(bsi->host_wake);

  bsi->host_wake_irq = platform_get_irq_byname(pdev, "host_wake");
```

如上代码所示，主要将 HOST_WAKE_BT 设置为输出脚，BT_WAKE_HOST 设置为输入脚并也设置为中断脚，等待 BT 芯片的唤醒。

然后再 bluesleep_init 函数中建立 BT 目录/proc/bluetooth/sleep/读 btwake 中 HOST_WAKE_BT 的状态。读出的状态值为 0 或 1。

或在 BT 目录/proc/bluetooth/sleep/写 btwake 中 HOST_WAKE_BT 的状态。写出状态值为 0 或 1。

Part 1

问：打开 wifi，连接 wifi 热点，提示连接成功，但 headbar 上不显示 wifi 图标，back 退出 wifi 设置，再进入，提示 wifi 已断开。

答：首先现象复现，当现象复现时进入 adb shell，然后输入 ifconfig 查看是否有 wlan0 端口，然后在 adb shell 中 ping 192.168.1.101(嵌入式设备的 IP)，如果能 Ping 通则说明底层 wifi 设备与 AP 是连接通的。所以把问题转向上层。上层在 frameworks/base/services/java/com/android/server/ConnectivityService.java 文件中查看 private void handleDisconnect(NetworkInfo info)函数的实现，问题可能出现在这里。

Part 2

问：出现新热点，需要重启 WIFI 才可以扫描到。

问：20s 内无法连接到 WIFI 热点。

问：已连接的热点关闭，不会自动连接下一个热点。

答：这些问题都是供应商芯片的 firmware 没有配置好，所以直接找 FAE 换掉。

Part 3

问：调试中如果遇到 dhd_sdio_probe fail 。

问：发现 sdio register timeout 。

答：因为我们用到的 WIFI 数据通道是 SDIO 接口，所以要在 log 信息中查看 sdcard 是否有加载。没加载的话就看看 WIFI 芯片的上电（如 WIFI_REG_ON 这个 PIN 脚）。

Part 4

问：如果在 wifi 的设置里面选中 wifi 选项出现 ERRO（或错误）的提示。

答：首先在 adb shell 中 lsmod 查看.ko 文件是否已经加载。如没加载 cat /proc/kmsg 查看是否是版本匹配的问题。如遇版本匹配则在 kernel/scripts/setlocalversion 中将 echo “+”中去掉。如果顺利加载了驱动，则要看看 MAC 地址是否有，并且是否合理。

Part 5

问：发现在 WIFI 设置选项中有已经选上了 wifi 并且勾应打上了，过一会出现 wifi 的勾自动消失。

答:这种情况在 adb shell 中用 ifconfig 查看 Wlan 的接口用的是否是 wlan0,有可能是 eth0。如果是 eth0 则在 hardware/libhardware_legacy/wifi/wifi.c 中的#define WIFI_DRIVER_MODULE_ARG "firmware_path=/system/etc/firmware/wlan/sdio.bin nvram_path=/data/simcom/nvram.txt iface_name=wlan0" 中查看 iface_name=wlan0 是否已经加上。

Part 6

问：有的路由器不能扫描的到。

答：查看 wlan 的设置的 channel 是否在 1-14 这个频段，因为如果 wlan 设置成 USA 模式则 channel 的范围在 1-11 之间，channel 12,13,14 就不会收索的到。所以要在 nvram.txt 中修改成 ccode=ALL，并且在 gqcom_cfg.ini 中把 APCntryCode=ALL。这样应该就可以扫描到所有的 channel 了。当然如果上层还是没有收索到的话，在 Settings.java 中也要做相应的修改。

常用的命令：wl channels_in_country

wl chanlist

wl channels

wl country

Part 7

问：【预置条件】：wifi 已连接 wifi sleep policy 模式为 When screen turns off

【操作步骤】：wifi 连接网络，静置屏幕熄灭 大约 5 分钟，再次点亮屏幕

【测试结果】：wifi 一直处在 scanning 状态，无法连接网络，关闭再打开恢复正常

【预期结果】：点亮屏幕后，wifi 自动连接成功

答：调试分析记录：

1.在 frameworks/base/services/java/com/android/server/WifiService.java 中修改 LCD 灭屏后关掉 WIFI 的时间长短，由以前的 2 分钟修改为现在的 1 秒钟。

结果是短时间休眠后唤醒可以顺利连上 AP，但经过长时间休眠（如 15 分钟）后唤醒机器还是无法连接 AP。

2.检查 Kernel 中 POWER_ON 使能脚，看是否有其它地方被占用过，结果是屏蔽所有其它用到过该脚的代码还是无用。

3.检查 CP 侧代码，将 CP 侧 POWER_ON 引脚由以前的某种特殊功能设置成普通的 Output 模式，然后 AP 就可以自由的控制其高低电平了。

修改后的结果还是以无效告终。

4.检查 external/wpa_supplicant_6/wpa_supplicant/文件夹下面的代码，检查 WEXT 驱动代码。结果还是失败。

5.最后在机器休眠 15 分钟后，用底层命令行来 scan、连接等动作的操作，结果根本无法扫描到 AP。故把问题集中在驱动底层。经过修改 dhd.ko、sdio.bin 文件现在此 Bug 终于得解。

6.也有问题不出现在驱动 dhd.ko 文件中，可能出现在 mmc 驱动中的 sdio 通道。在函数 mmc_pm_notify 中当出现 PM_POST_SUSPEND 或 PM_POST_HIBERNATION 的情况时就有可能移除掉了 sdio 通道，所以函数 mmc_detect_change(host, 0)要加一个判断，当 sdio 作为 wifi 通道时就不能移除。移除后的后果就是此 Bug。