

# ylyuanlu的专栏

☰ 目录视图

☰ 摘要视图

RSS 订阅

## 个人资料



ylyuanlu

访问：9637次  
积分：335分  
排名：千里之外

原创：19篇    转载：15篇  
译文：1篇    评论：9条

## 文章搜索

## 文章分类

- android 系统开发(15)
- ARM 嵌入式(6)
- C/C++(7)
- Linux 内核与驱动(17)
- 网络编程(1)
- 随笔(0)

## 文章存档

- 2011年09月(2)
- 2011年08月(13)
- 2011年07月(8)
- 2010年11月(12)

## 阅读排行

- Wifi模块分析 (1110)
- android binder机制之-- (... (1074)
- android handlerthrea... (874)
- android binder机制之-- (... (858)
- android binder机制之—— (... (786)

💡 CSDN 博客新增自定义博客专栏模块，方便快捷一站式！  
超级光棍节，你想好怎么过了吗？来聆听大师的演讲吧

📲 提交原创APP，20万等你赢  
参加浏览器性能挑战赛免费去美国！

## 原 Wifi模块分析

分类： android 系统开发 网络编程      2011-07-21 14:25    🔍 1112人阅读    💬 评论(3)    📁 收藏    🚩 举报

### Wifi模块

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：

- (1) Wifi模块相关文件的解析
- (2) Wpa\_suplicant解析
- (3) Wifi的启动流程（有代码供参考分析）

#### 一，Wifi模块相关文件解析

##### 1) wifisettings.java

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

该类数据部分主要定义了下面几个类的变量：

```
{

private final IntentFilter mFilter;

//广播接收器，用来接收消息并做响应的处理工作

privatefinal BroadcastReceiver mReceiver;

//这是一个扫描类，会在用户手动扫描 AP时被调用

privatefinal Scanner mScanner;

private WifiInfo mLastInfo;
```

✕

下载Chrome浏览器

免费领积分

如果你的Chrome版本低于14  
请先卸载，再安装  
去CSDN下载频道

立即下载

下载任意资源  
送30下载积分

CSDN

- [android binder机制之-- \(658\)](#)
- [Camera模块解析之驱动篇 \(471\)](#)
- [Android intent消息通知机制 \(466\)](#)
- [android binder机制之—— \(439\)](#)
- [Linux的i2c驱动详解 \(414\)](#)

评论排行

- [Wifi模块分析 \(3\)](#)
- [android binder机制之—— \(2\)](#)
- [Camera模块解析之驱动篇 \(1\)](#)
- [android handlerthrea... \(1\)](#)
- [Nand Flash探索之旅 \(1\)](#)
- [Linux内核模块设计 \(1\)](#)
- [Camera 图像处理原理分析篇 一 \(0\)](#)
- [Camera 图像处理原理分析篇 二 \(0\)](#)
- [YUV / RGB 格式分析及快速查表算... \(0\)](#)
- [Camera 图像处理原理分析篇 三 \(0\)](#)

推荐文章

- [如何使用HTML5创建在线精美简历](#)
- [为Android应用程序读取/dev下设备而提权](#)
- [敏捷开发产品管理系列之四：新产品研发](#)
- [JXCZT网络管理系统建设方案](#)
- [STL系列之四 heap 堆](#)
- [数据驱动编程之表驱动法](#)

最新评论

- [Nand Flash探索之旅](#)  
conquer320: 写得太好了，条理好清晰，佩服。感觉自己学的东西都是乱七八糟的，不知博主平时是怎么总结这些的？
- [android handlerthread 通知机制](#)  
WSLZNWZD: 请问图在哪里呀？
- [android binder机制之——（我是binder实例）](#)  
vigour\_lu: 那这里remote() 应该是一个BBinder对象,remote()->transact()实际上...
- [android binder机制之——（我是binder实例）](#)  
vigour\_lu: 那这里remote() 应该是一个BBinder对象,remote()->transact()实际上...
- [Wifi模块分析](#)  
weihongcsu: @k1102k27:你好，我以上提的问题能否给点意见，

//服务端代理端，作为WifiService对外的接口类呈现

```
privateWifiManager mWifiManager;
```

//这个类主要实现Wifi的开闭工作

```
privateWifiEnabler mWifiEnabler;
```

//AP

```
private AccessPoint mSelected;
```

```
private WifiDialog mDialog;
```

.....

```
}
```

wifiSettings类的构造函数的主要工作：定义了一个IntentFilter（Intent过滤器）变量，并添加了六个动作，（了解Android的intent机制的同学都知道什么意思，不明白的同学参考Intent机制的资料）接着定义一个广播接收器，并有相应的消息处理函数，下面是该构造函数的定义：

```
public WifiSettings() {
```

```
mFilter = new IntentFilter();
```

//intent机制中的intent消息过滤器，下面添加可以处理的动作

```
mFilter.addAction(WifiManager.WIFI_STATE_CHANGED_ACTION);
```

```
mFilter.addAction(WifiManager.SCAN_RESULTS_AVAILABLE_ACTION);
```

```
mFilter.addAction(WifiManager.NETWORK_IDS_CHANGED_ACTION);
```

```
mFilter.addAction(WifiManager.SUPPLICANT_STATE_CHANGED_ACTION);
```

```
mFilter.addAction(WifiManager.NETWORK_STATE_CHANGED_ACTION);
```

```
mFilter.addAction(WifiManager.RSSI_CHANGED_ACTION);
```

//注册了广播接收器，用来处理接收到的消息事件

```
mReceiver = new BroadcastReceiver() {
```

```
@Override
```

```
public void onReceive(Context context,Intent intent) {
```

```
handleEvent(intent); //事件处理函数
```

感激不尽！

Wifi模块分析

weihongcsu: 你好，想请教一个关于android wifi连接的问题,我的wifi总是死在obtaining Ip...

Camera模块解析之驱动篇

zmyde2010: Goooooooooooooooood

Wifi模块分析

k1102k27: 哥们 你早点写这篇文章就好了我刚做完一个wifi管理的app，摸索了好久。回头看看你的文章，还是有很...

Linux内核模块设计

tianqii:

```
    }

};

mScanner= new Scanner();    //手动扫描类

}
```

在广播接收器中的相应函数onReceive函数中有个handleEvent函数，它就是用来处理广播接收器接受到的intent消息的，它的功能是根据intent消息中的动作类型，来执行相应的操作，每一种动作对应了activity的一项消息处理能力。

在oncreate函数中实例化了mWifiManager和mWifiEnabler两个类，这两个类对wifiSettings来说至关重要，它后面的定义的一系列函数都是通过调用这两个类的相应接口来实现的。

```
.....

mWifiManager = (WifiManager)getSystemService(Context.WIFI_SERVICE);

mWifiEnabler = new WifiEnabler(this,

    (CheckBoxPreference) findPreference("enable_wifi"));

.....
```

WifiSettings中还定义了显示菜单和响应菜单键的函数，即onOptionsItemSelected（）和onOptionsItemSelected（）；还有响应配置对话框中按键的onClick()函数；最后定义了Scanner类，它是一个handler的继承类，实现了消息处理函数，用于处理手动扫描的动作。

2) WifiEnabler.java:

```
packages/apps/Settings/src/com/android/settings/wifi/WifiEnabler.java
```

```
private final Context mContext;

private final CheckBoxPreference mCheckBox;

//两个重要成员

private final WifiManager mWifiManager;

private final IntentFilter mIntentFilter;
```

wifienabler类中定义了四个成员变量很重要，mContext，mCheckBox，mWifiManager和mReceiver，其中mContext用于获取mWifiManager实例，mReceiver用来接收底层发来的消息，mCheckBox用来改变UI的状态。

该类中定义了几个重要的函

数onPreferenceChange，handleWifiStateChanged和handleStateChanged，onPreferenceChange用来处理按下Enbler键，它会调用mWifiManager.setWifiEnabled(enable)，另外两个用来处理接受的消息事件。

在类的构造函数中，主要做了一下工作：初始化了mContext,mCheckBox,mWifimanager，并且初始化了一个mIntentFilter变量，添加了三个动作，在构造函数的上面定义了一个广播接收器，用来接收下层传来的消息，并根据intent动作的类型调用相应的处理函数，这个广播接收器在onResum函数中被注册。

```
public WifiEnabler(Context context, CheckBoxPreferencecheckBox) {

    mContext= context;

    mCheckBox = checkBox;

    mOriginalSummary = checkBox.getSummary();

    checkBox.setPersistent(false);

    mWifiManager = (WifiManager)context.getSystemService(Context.WIFI_SERVICE);

    mIntentFilter= new IntentFilter(WifiManager.WIFI_STATE_CHANGED_ACTION);

    // Theorder matters! We really should not depend on this. :(

    mIntentFilter.addAction(WifiManager.SUPPLICANT_STATE_CHANGED_ACTION);

    mIntentFilter.addAction(WifiManager.NETWORK_STATE_CHANGED_ACTION);

}
```

这里可以总结为：如果上层需要监听或收到下层的消息，那么就要通过定义一个BroadcastReciever，并将它注册，当然在接受到消息后应该有处理消息的函数，然后在onReciever函数中根据消息调用相应的处理函数，这里的消息通知机制是Intent，在BroadcastReciever类的onReciever函数的参数中可以看出。

该类成员函数的也是通过调用mWifimanager的接口来实现的。

3) WifiManager:

```
frameworks/base/wifi/java/android/net/wifi/WifiManager.java
```

两个重要的数据成员:

```
//WifiService
```

```
IWifiManager mService;
```

```
HandlermHandler;
```

IWifiManager mService和HandlermHandler，这个类拥有了一个WifiService实例，就可以通过它进行一系列的调用；WifiManager中定义了的wifi和ap的状态，这些状态会在其他很多类中有使用；然后定义了大量的函数，这些函数几乎都是对WifiService接口函数的封装，直接调用WifiService的函数。

该类的构造函数很简单：

```
public WifiManager(IWifiManager service,Handler handler) {

    mService = service;

    mHandler = handler;

}
```

该类中还定义了一个WifiLock类，这个类用来保证在有应用程序使用Wifi无线电传输数据时，wifiradio可用，即当一个应用程序使用wifi的radio进行无线电数据传输时，就要先获得这个锁，如果该锁已被其他程序占有，就要等到该锁被释放后才能获得，只用当所有持有该锁的程序都释放该锁后，才能关闭radio功能。

4)WifiService：

```
frameworks/base/services/java/com/android/server/WifiService.java
```

```
private final WifiStateTrackermWifiStateTracker;

private Context mContext;

private WifiWatchdogServicemWifiWatchdogService = null;

private final WifiHandler mWifiHandler;
```

这是WifiService中的几个重要的数据成员。

在接下来的构造函数中初始化了mWifiStateTracker，mContext，然后动态生成mWifiThread子线程并启动，在主线程里用mWifiThread调用getLooper()函数获得线程的looper，来初始化创建一个mWifiHandler对象，这个WifiHandler在WifiService类的后面有定义，并重载了Handler类的handlemessage（）函数，这样消息就可以在主线程里被处理了，这是android的handlerthread消息处理机制，可参考相关资料，这里不予详述。在构造函数的最后，注册了两个广播接收器，分别用来ACTION\_AIRPLANE\_MODE\_CHANGED和ACTION\_TETHER\_STATE\_CHANGED这两个动作，这里是android的intent消息通知机制，请参考相关资料，代码如下：

```
mContext = context;

mWifiStateTracker = tracker;

mWifiStateTracker.enableRssiPolling(true);

.....

HandlerThread wifiThread = newHandlerThread("WifiService");

wifiThread.start();

mWifiHandler = newWifiHandler(wifiThread.getLooper());

.....
```

随后定义了一系列的函数，其中有服务器要发送的命令的系列函数，它通过mWifiStateTracker成员类调用自己的发送命令的接口（其实就是对本地接口的一个封装），最后通过适配层发送命令给wpa\_supplicant，而事件处

理只到WifiStateTracker层被处理。

要注意的是，在WifiService中，定义了一些函数来创建消息，并通过mWifiHandler将消息发送到消息队列上，然后在mHandlerThread线程体run()分发\处理消息，在主线程中被mWifiHandler的handlerMessage（）函数处理，最后调用mWifiStateTracker的对应函数来实现的。这里我也不明白为什么WifiService不直接调用mWifiStateTracker对应的函数，还要通过消息处理机制，绕了一圈在调用，当然Google这么做肯定是有它道理的，忘高手指点。

5) WifiStateTracker类：

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

NetworkStateTracker继承了handler类，而WifiStateTracker继承了NetworkStateTracker类，就是说WifiStateTracker间接继承了handler类，属于一个事件处理类。

WifiStateTracker类首先定义了事件日志和事件码（这里包含了所有可能的事件类型），还定义了如下的重要成员数据：

```
//几个重要的数据成员

private WifiMonitor mWifiMonitor; //被启动用来监听supplicant传来的消息

private WifiInfo mWifiInfo;

private WifiManager mWM; //服务代理

private DhcpHandler mDhcpTarget; //IP地址获取线程

private DhcpInfo mDhcpInfo; //Dhcp的相关信息都在这里
```

类的构造函数中，初始化了系列成员变量，包括生成了WifiMonitor的实例，在构造函数中，因为WifiStateTracker是一个handler间接子类，所以他会自动调用handler的无参构造函数，获得looper和Queue消息队列。

然后定义了一些设置supplicant和更新网络信息的辅助函数。

startEventLoop()函数很重要，用来启动WifiMonitor线程，进入消息循环检测。接着定义了系列通知函数，被WifiMonitor调用来向WifiStateTracker传递从wpa\_supplicant接收到的消息，他会调用消息发送函数到消息队列，并被WifiStateTracker的handlermessage（）函数处理。这个handlermessage（）函数就是在随后被定义的，它主要是调用相应的辅助函数完成动作，并可能会将消息封装后，通过intent机制发送出去，被上层的UI活动接收处理。

这里也定义了很多的WifiNative接口函数，这是JNI的本地接口；类DhcpHandler extends Handler{}也是在该类中定义的，它也是一个handler的子类，用来处理DHCP相关的消息EVENT\_DHCP\_START，可以想到它和WifiStateTracker不是共用一个looper。

注意：handleMessage是在该文件中定义的，用来处理经WifiMonitor转换过的消息。

6) WifiMonitor

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

声明了一个重要的成员变量：**mWifiStateTracker**，并在构造函数中由参数提供初始化，还定义了一系列的可能从**wpa\_supplicant**层接收的事件类型及其名字，这些是消息处理机制的基础。

**startMonitoring**（）函数，这是一个线程启动的封装函数，**WifiStateTracker**就是通过这个函数启动的**WifiThread**。

这个重要的类**class MonitorThread extends Thread**{}；它是一个监控进程类，里面有一系列的事件处理函数和一个重要的**Run()**函数，**run**函数主要流程：**connectToSupplicant()**连接精灵进程**wpa\_supplicant**，这里有一个**mWifiStateTracker.notifySupplicantXXX()**的调用，通知上层是否连接成功，然后就是一个轮询过程，其中调用了**WifiNative.waitForEvent()**本地轮询函数接口，并从返回的事件字符串类型中提取事件的名称，最后通过事件的名称调用相应的事件处理函数，并将事件转换成**mWifiStateTracker**能识别的类型上报。

注意：这里的每个事件的捕获（由**wifimonitor**完成），最终都会调用相应的**mWifiStateTracker**的消息通知函数上报消息；轮询和事件处理都是在**Monitor**线程类中实现的。

7)WifiNative

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

里面定义了一个类**WifiNative**：其中声明了许多本地接口，可由**native**的标志看出，这是**Java**代码和本地库之间的联系接口；

8) android\_net\_wifi\_Wifi.java

frameworks/base/core/jni/

里面定义了许多的**JNI**的本地代码实现，每个实现中都会调用**wpa\_supplicant**适配层的接口，通过包含适配层的头文件**wifi.h**获取适配层定义的接口；后面是一个**JNINativeMethod**数组，定义了每个本地接口和本地实现的对对应关系；最后有一个注册函数**register\_XXX\_XX()**，用以把上面的方法类数组注册到系统中。

该类实现了本地接口的相关功能，并通过调用**wpa**的适配层的相关函数和**wpa\_supplicant**通信，所以**JNI**是连接**Java**框架层和**wpa**适配层的桥梁。

9)wpa\_supplicant适配层，wifi.c:目录libhardware/wifi/

里面定义很多字符串变量和适配层的接口实现，是对**wpa\_supplicant**程序通信的接口封装，用来完成上层和**wpa\_supplicant**的通信，头文件在**libhardware/include/hardware**下，这里的函数用来向**JNI**的本地实现提供调用接口。

这里的函数，我把它们分为三类函数：

一类是命令相关的（控制）函数，就是在**JNI**层**android\_XXX\_Command()**函数所调用的：：**Wifi\_Command()**函数，调用流程：**android\_XXX\_command()**=>**docommand()**=>**wifi\_command()**=>**wifi\_send\_command()**=>**wpa\_ctrl\_require()**。

二类是监听函数，即**Wifi\_wait\_for\_event()**函数，调用流

程：`android_net_wifi_Waitforevent()`=>`wifi_wait_for_event()`=>`wpa_ctrl_rcv()`。

三类是剩下的函数。

10) `wpa_supplicant`与上层的接口，`wpa_ctrl.c`：`external/wpa_supplicant`

定义了三类套接字，并分别实现了和`wpa_supplicant`的通信，因此`wpa_supplicant`适配层和`wpa_supplicant`层是通过`socket`通讯的。

要是从`wifi.c`中真的很难看出它和`wpa_supplicant`有什么关系，和它联系密切的是`wpa_ctrl.h`文件，这里面定义了一个类`wpa_ctrl`，这个类中声明了两个`Socket`套接口，一个是本地一个是要连接的套接口，`wpa_ctrl`与`wpa_supplicant`的通信就需要`socket`来帮忙了，而`wpa_supplicant`就是通过调用`wpa_ctrl.h`中定义的函数和`wpa_supplicant`进行通讯的，`wpa_ctrl`类（其实是其中的两个`socket`）就是他们之间的桥梁。

11)`wpa_supplicant`和`driver_wext`驱动接口的联系：

`driver.h`：该文件定义了系列结构，首先是一个`wpa_scan_result`结构，这是一个扫描结果的通用格式，里面包含了扫描的所有信息（如BSSID，SSID，信号质量，噪音水平，支持的最大波特率等信息），每个驱动接口实现负责将从驱动中上传的扫描信息的格式转换到该通用的扫描信息格式；然后是一些宏定义和枚举定义，最后也是最重要的是`wpa_driver_ops`结构，该结构是`wpa driver`的操作函数集合，里面有驱动接口的名称和很多的函数指针。

`drviers.c`：该文件很简单，首先是一些外部变量的引用声明，都是不同驱动操作接口的集合`wpa_driver_XXX_ops`变量；然后就是定义一个驱动操作接口集合的数组，根据宏定义添加对应的驱动操作接口集合的变量。

`drvier_XXX.h`:这是不同驱动接口头文件，主要声明了操作接口

`drvier_XXX.c`:实现操作接口，定义一个操作集合变量，并用上面定义的函数初始化其中的函数指针

注意：下面要搞清楚`wpa_supplicant`守护进程是如何操作，最后调用驱动接口集合中的函数的；要知道`wpa_supplicant`是为不同驱动和操作系统具有更好移植性而被设计的，以便在`wpa_supplicant`层不用实现驱动的具体接口就可以添加新的驱动程序；在`wpa_supplicant`结构中有一个`wpa_drv_ops`类型的`drvier`成员，在`wpa_supplicant`进程中，经常通过`Wpa_supplicant_XXX`函数传递`wpa_supplicant`实例指针`wpa_s`参数给`wpa_drv_XXX`函数来调用它，在`wpa_drv_XX`中会通过`wpa_s->driver->XXX()`的流程来调用通用驱动接口，简单才是生活的真相，可`android`始终让我感到真相还是遥不可及。

12)`WifiWatchdogService`：

首先声明了两个主要变量`mWifiStateTracker`，`mWifiManager`，需要这两个类对象来完成具体的控制工作，在`WifiWatchdogService`的构造函数中，创建了这两个类，并通过`registerForWifiBroadcasts()`注册了`BroadcastReceiver`，`BroadcastReceiver`是用来获取网络变化的，然后启动了一个`WatchdogTread`线程，用来处理从`WifiStateTracker`接收到的消息。

`frameworks/base/services/java/com/android/server/WifiWatchdogService.java`

`WifiWatchdogService(Context context,WifiStateTracker wifiStateTracker) {`



```
mContext = context;

mContentResolver = context.getContentResolver();

mWifiStateTracker =wifiStateTracker;

    mWifiManager = (WifiManager) context.getSystemService(Context.WIFI_SERVICE);

createThread();

// The content observer to listen needs a handler, which createThreadcreates

registerForSettingsChanges();

if (isWatchdogEnabled()) {

    registerForWifiBroadcasts();

}

if (V) {

    myLogV("WifiWatchdogService: Created");

}

}
```

WifiWatchdogHandler继承了handler类，成为一个消息处理类，定义handlemessage()函数，处理消息队列上的消息。

## 二，wpa\_supplicant再解析

### 1)wpa\_ctrl.h:

该文件中并没有定义structwpa\_ctrl结构，因为在其他包含该文件的.c文件中不能直接使用该结构的成员，这里主要声明了几个用于使用socket通信的函数接口，包括：wpa\_ctrl\_open，wpa\_ctrl\_close，wpa\_ctrl\_attach，wpa\_ctrl\_detach，wpa\_ctrl\_cleanup，wpa\_ctrl\_recv,wpa\_ctrl\_request, wpa\_ctrl\_pending, wpa\_ctrl\_get\_fd 等函数。

这些函数的功能从名字上能看出，open函数就是创建一个socket接口，并绑定连接wpa\_supplicant，attach函数用于定义一个控制接口为监听接口，pending函数用于查询有无消息可处理，request和recv分别用来发送和读取消息。

其实，这里就是一个使用socket通信的封装，具体内容可以参考socket编程。

### 2) wpa\_ctrl.c:

这里首先定义了一个wpa\_ctrl结构，里面根据UDP，UNIX和命名管道三种domain类型来定义通信实体：

```
struct wpa_ctrl {

#ifdef CONFIG_CTRL_IFACE_UDP

    int s;

    struct sockaddr_in local;

    struct sockaddr_in dest;

    char *cookie;

#endif /*CONFIG_CTRL_IFACE_UDP */

#ifdef CONFIG_CTRL_IFACE_UNIX

    int s;

    struct sockaddr_un local;

    struct sockaddr_un dest;

#endif /*CONFIG_CTRL_IFACE_UNIX */

#ifdef CONFIG_CTRL_IFACE_NAMED_PIPE

    HANDLE pipe;

#endif /*CONFIG_CTRL_IFACE_NAMED_PIPE */

};
```

然后根据上面三个类型分别实现了wpa\_ctrl.h中声明的接口函数，这里就不做介绍了。

3)wpa\_supplicant.h:

首先，定义了一个枚举wpa\_event\_type，罗列了系列wpa的事件类型，然后就是wpa\_event\_data类型，随后是两个函数：wpa\_supplicant\_event和wpa\_supplicant\_rx\_eapol。

wpa\_supplicant.c: 首先定义一个驱动操作数组extern struct wpa\_driver\_ops \*wpa\_supplicant\_drivers[]，然后是系列wpa\_supplicant\_XXX()函数，很多函数里面调用wpa\_drv\_XXX()函数，这些函数是wpa\_supplicant.i.h中实现的函数。几乎每个函数都需要一个wpa\_supplicant结构，对其进行所有的控制和通信操作。

4)wpa\_supplicant.i.h:

开头定义了几个结构，如：wpa\_blacklist，wpa\_interface，wpa\_params，wpa\_global，wpa\_client\_mlme和wpa\_supplicant等结构，其中wpa\_supplicant最为重要，wpa\_supplicant结构里有一个重要的driver成员，它是wpa\_driver\_ops类型，可以被用来调用抽象层的接口。

接下来是系列函数声明，这些函数声明在wpa\_supplicant.c中实现，然后就是wpa\_drv\_XXX函数，这些函数就是在wpa\_supplicant.c中被wpa\_supplicant\_xxx函数调用的，而这些wpa\_drv\_xxx函数也都有一

个wpa\_supplicant结构的变量指针，用来调用封装的抽象接口。

这里要注意的是：在wpa\_suppliant.c文件中定义的很多函数是在该头文件中声明的，而不是在wpa\_supplicant.h中声明的。

5)driver.h:

该文件中定义了一个重要的数据结构：wpa\_scan\_result，这是一个从驱动发来的数据被封装成的通用的扫描结果数据结构，每个驱动结构的实现都要遵循的扫描结果格式，比如driver\_wext.c中的实现。后面还有定义了很多的数据结构，这里不具表。

文件中最最重要的一个数据结构是：wpa\_driver\_ops，这是所有驱动接口层必须为之实现的API，是所有驱动类型的一个接口封装包，wpa\_supplicant就是通过该接口来和驱动交互的。

6)driver\_wext.h:

该文件很简单，就是声明了该驱动的一些对应驱动API接口的函数。

driver\_wext.c:

此文件就是实现了上面的一些函数，最后初始化了一个wpa\_drv\_ops变量。

三，Wifi模块解析

1) 框架分析

图示1：Wifi框架

首先，用户程序使用WifiManager类来管理Wifi模块，它能够获得Wifi模块的状态，配置和控制Wifi模块，而所有这些操作都要依赖Wifiservice类来实现。

WifiService和WifiMonitor类是Wifi框架的核心，如图所示。下面先来看看WifiService是什么时候，怎么被创建和初始化的。

在systemServer启动之后，它会创建一个ConnectivityServer对象，这个对象的构造函数会创建一个WifiService的实例，代码如下所示：

```
framework/base/services/java/com/android/server/ConnectivityService.java

{
.....

case ConnectivityManager.TYPE_WIFI:

        if (DBG) Slog.v(TAG, "Starting Wifi Service.");
```

```
        WifiStateTracker wst = new WifiStateTracker(context, mHandler);           //创建WifiStateTracker实例

        WifiService wifiService = newWifiService(context, wst); //创建WifiService实例

        ServiceManager.addService(Context.WIFI_SERVICE, wifiService);           //向服务管理系统添加Wifi服务

        wifiService.startWifi(); //启动Wifi

        mNetTrackers[ConnectivityManager.TYPE_WIFI] = wst;

        wst.startMonitoring(); //启动WifiMonitor中的WifiThread线程

        .....

    }
```

WifiService的主要工作：WifiMonitor和Wpa\_supplicant的启动和关闭，向Wpa\_supplicant发送命令。

WifiMonitor的主要工作：阻塞监听并接收来自Wpa\_supplicant的消息，然后发送给WifiStateTracker。

上面两个线程通过AF\_UNIX套接字和Wpa\_supplicant通信，在通信过程中有两种连接方式：控制连接和监听连接。它们创建代码如下：

```
ctrl_conn =wpa_ctrl_open(ifname);

... ..

monitor_conn = wpa_ctrl_open(ifname);
```

2) Wifi启动流程

（1）使能Wifi

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

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。
```

```
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sendMessage（）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.sendMessage()
```

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：

```
.....

if (enable) {

    if (!mWifiStateTracker.loadDriver()) {

        Slog.e(TAG, "Failed toload Wi-Fi driver.");

        setWifiEnabledState(WIFI_STATE_UNKNOWN, uid);

        return false;

    }

    if (!mWifiStateTracker.startSupplicant()) {

        mWifiStateTracker.unloadDriver();

    }

}
```

```
        Slog.e(TAG, "Failed to start supplicant daemon.");

        setWifiEnabledState(WIFI_STATE_UNKNOWN, uid);

        return false;
    }

    registerForBroadcasts();

    mWifiStateTracker.startEventLoop();

    .....

```

至此，Wifi使能结束，自动进入扫描阶段。

(2) 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。

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

```
private void handleDriverEvent(String state) {

    if (state == null) {

        return;

    }

    if (state.equals("STOPPED")) {

        mWifiStateTracker.notifyDriverStopped();

    } else if (state.equals("STARTED")) {

        mWifiStateTracker.notifyDriverStarted();

    } else if (state.equals("HANGED")) {

        mWifiStateTracker.notifyDriverHung();

    }

}

```

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

```
case EVENT_DRIVER_STATE_CHANGED:

    switch(msg.arg1) {

    case DRIVER_STARTED:

        /**

        *Set the number of allowed radio channels according

        *to the system setting, since it gets reset by the

        *driver upon changing to the STARTED state.

        */

        setNumAllowedChannels();

        synchronized (this) {

            if (mRunState == RUN_STATE_STARTING) {

                mRunState = RUN_STATE_RUNNING;

                if (!mIsScanOnly) {

                    reconnectCommand();

                } else {

                    // In some situations, supplicant needs to be kickstarted to

                    // start the background 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。

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

```
public boolean startScan(boolean forceActive) {

    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()。

```
packages/apps/Settings/src/com/android/settings/wifiwifiwifiwifiwifiwifiwifi
```

```
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 boolean onOptionsItemSelected(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);

}

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suppllicant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：

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

```
public void handleMessage(Message msg) {

    Intent intent;

    .....

    case EVENT_SCAN_RESULTS_AVAILABLE:

        if(ActivityManagerNative.isSystemReady()) {

            mContext.sendBroadcast(new
Intent(WifiManager.SCAN_RESULTS_AVAILABLE_ACTION));

        }

        sendScanResultsAvailable();

        /**

         * On receiving the first scanresults after connecting to

         * the supplicant, switch scanmode over to passive.

         */

        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。

(3)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。

```
packages/apps/settings/src/com/android/settings/wifi/WifiSettings.java
```

```
public boolean onPreferenceTreeClick(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/WifiSettings.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

```
public int updateNetwork(WifiConfiguration config) {  
  
    if (config == null || config.networkId < 0) {  
  
        return -1;  
  
    }  
  
    return addOrUpdateNetwork(config);  
  
}  
  
private int addOrUpdateNetwork(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。

#### (4) 获取IP地址

当连接到supplicant后，WifiMonitor就会通知WifiStateTracker。

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

```
Public void Run(){  
  
    if (connectToSupplicant()) {  
  
        // Send a message indicating that 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handlemessage()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);

            .....

            .....

    }

}
```

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

```
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 voidhandleNetworkStateChange(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();

}

caseEVENT_NETWORK_STATE_CHANGED:

.....

configureInterface();

.....

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.sendMessage(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

}

}

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。

```

```

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;

                Log.i(TAG,"DhcpHandler: DHCP request failed: " +

                    NetworkUtils.getDhcpError());

            }

            .....

        }
    }
}

```

这里调用了 `NetworkUtils.runDhcp ()` 函数，`NetworkUtils` 类是一个网络服务的辅助类，它主要定义了一些本地接口，这些接口会通过他们的JNI层 `android_net_NetUtils.cpp` 文件和 `DHCP client` 通信，并获取IP地址。

至此，IP地址获取完毕，Wifi启动流程结束。

上一篇：[android handlerthread 通知机制](#)

分享到： 

下一篇：[android binder机制之--（我是binder）](#)

顶


1

踩

0



查看评论

2楼 [weihongcsu](#) 2011-08-29 09:59 发表 



你好，想请教一个关于android wifi连接的问题,我的wifi总是死在obtaining Ip 那里，以下是logcat。我尝试着根据你的博文“wifi模块”去理清思路，但由于我对此部分不熟悉，进展非常慢，希望能指点迷津，谢谢了！

##通过与成功连接wifi的日志对比，之前的dhcp日志可以确定是对的##

D/dhccpd ( 1905): forking to background

I/logwrapper( 1897): dhccpd terminated by exit(0)

D/dhccpd ( 2128): sending ARP announce (1 of 2), next in 2.00 seconds

D/dhccpd ( 2128): sending ARP announce (2 of 2)


D/dhccpd ( 2128): renew in 43196 seconds

...

I/WifiStateTracker( 199): DhcpHandler: DHCP request failed: Timed out waiting for DHCP to finish ##我用的是froyo,此错在system/core/libnetutils/dhcp\_utils.c的dhcp\_do\_request()中

V/WifiMonitor( 199): Event [CTRL-EVENT-STATE-CHANGE id=0 state=8]

D/dhccpd ( 2128): carrier lost


1楼 [k1102k27](#) 2011-07-22 14:09 发表 



哥们 你早点写这篇文章就好了

我刚做完一个wifi管理的app，摸索了好久。

回头看看你的文章，还是有很大收获

Re: [weihongcsu](#) 2011-08-29 10:00 发表 









回复k1102k27：你好，我以上提的问题能否给点意见，感激不尽！






您还没有登录,请[\[登录\]](#)或[\[注册\]](#)

\* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

专区推荐内容

-  [PHP数组可以用中文索引吗？](#)
-  [欢迎参观GDC2011英特尔展台](#)
-  [WiGig下一代WIFI的集大成者？](#)
-  [结束，“心”的开始](#)
-  [你能WiDi吗？](#)
-  [转型并非放弃 诺基亚年底宣布MeeGo1.1](#)

热门招聘职位

-  [【用友软件】急聘JAVA/.NET开发、测试及需求分析人员](#)
-  [【视博云计算】高薪诚聘系统架构师C++/C#工程师](#)
-  [【杭州贯通】急聘系统分析及.NET/JavaScript等各类网站开发人员](#)
-  [找工作的童鞋看过来，当当网诚聘高级开发工程师](#)
-  [【北塔软件】减招JAVA软件开发/软件测试/FLEX开](#)

 Email:webmaster@csdn.net

Copyright © 1999-2011, CSDN.NET, All Rights Reserved

