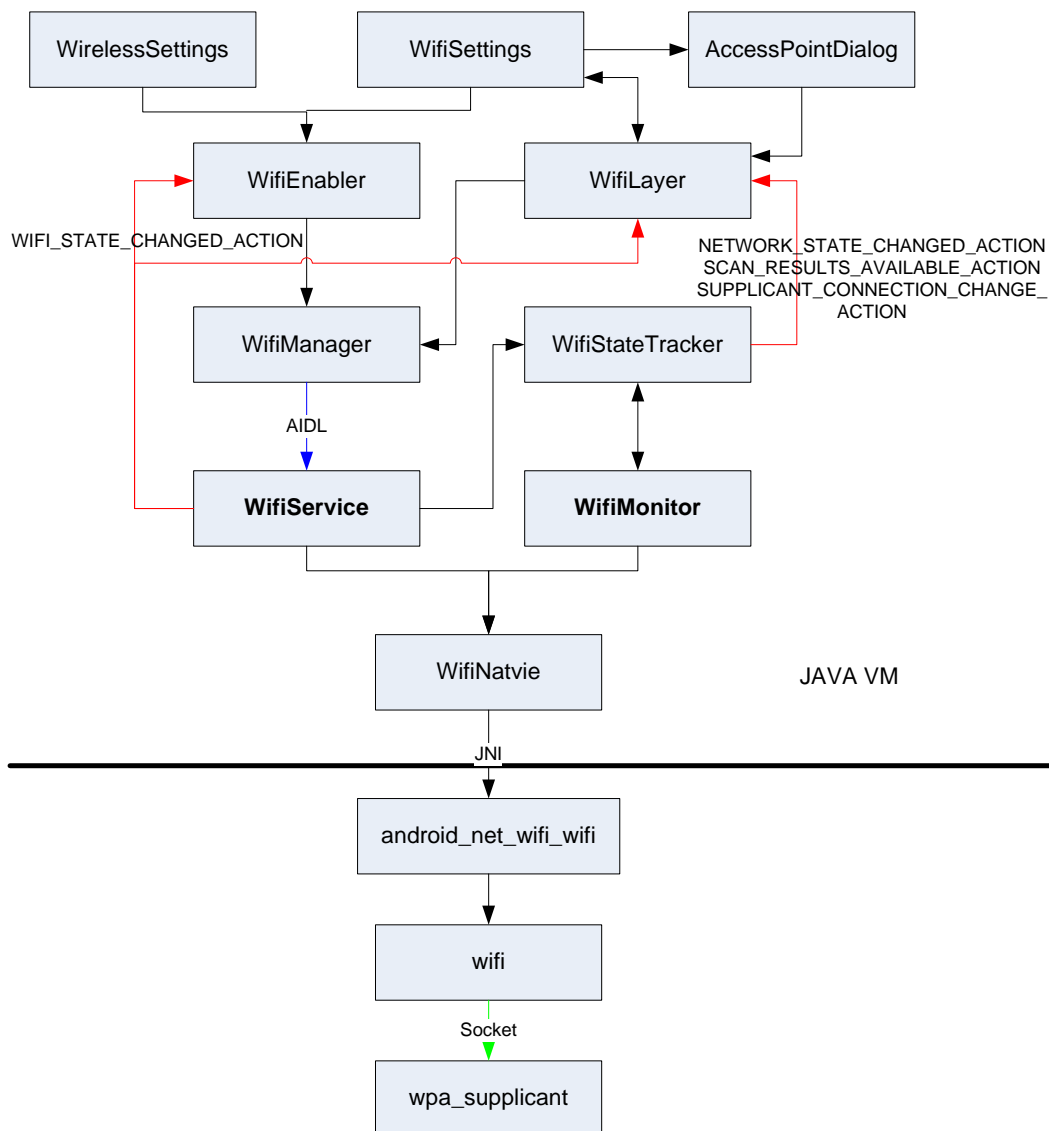


## WIFI 模块



## 初始化

在 SystemServer 启动的时候，会生成一个 ConnectivityService 的实例，

```
try {
    Log.i(TAG, "Starting Connectivity Service.");
    ServiceManager.addService(Context.CONNECTIVITY_SERVICE, new
ConnectivityService(context));
} catch (Throwable e) {
    Log.e(TAG, "Failure starting Connectivity Service", e);
}
```

ConnectivityService 的构造函数会创建 WifiService，

```

if (DBG) Log.v(TAG, "Starting Wifi Service.");
mWifiStateTracker = new WifiStateTracker(context, handler);
WifiService wifiService = new WifiService(context, mWifiStateTracker);
ServiceManager.addService(Context.WIFI_SERVICE, 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。

## 连接 AP

### 1. 使能 WIFI

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

```

private void initToggles() {
    mWifiEnabler = new WifiEnabler(
        this,
        (WifiManager) getSystemService(WIFI_SERVICE),
        (CheckBoxPreference) findPreference(KEY_TOGGLE_WIFI));
}

```

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。

```

private boolean setWifiEnabledBlocking(boolean enable) {
    final int eventualWifiState = enable ? WIFI_STATE_ENABLED :
WIFI_STATE_DISABLED;

    updateWifiState(enable ? WIFI_STATE_ENABLING : WIFI_STATE_DISABLING);

    if (enable) {
        if (!WifiNative.loadDriver()) {
            Log.e(TAG, "Failed to load Wi-Fi driver.");
            updateWifiState(WIFI_STATE_UNKNOWN);
            return false;
        }
        if (!WifiNative.startSupplicant()) {
            WifiNative.unloadDriver();
            Log.e(TAG, "Failed to start supplicant daemon.");
            updateWifiState(WIFI_STATE_UNKNOWN);
            return false;
        }
        mWifiStateTracker.startEventLoop();
    }
}

```

```

        // Success!

        persistWifiEnabled(enable);
        updateWifiState(eventualWifiState);

        return true;
    }

```

当使能成功后，会广播发送 `WIFI_STATE_CHANGED_ACTION` 这个 `Intent` 通知外界 `WIFI` 已经成功使能了。`WifiEnabler` 创建的时候就会向 `Android` 注册接收 `WIFI_STATE_CHANGED_ACTION`，因此它会收到该 `Intent`，从而开始扫描。

```

private void handleWifiStateChanged(int wifiState) {

    if (wifiState == WIFI_STATE_ENABLED) {
        loadConfiguredAccessPoints();
        attemptScan();
    }
}

```

## 2. 查找 AP

扫描的入口函数是 `WifiService` 的 `startScan`，它其实也就是往 `wpa_supplicant` 发送 `SCAN` 命令。

```

static jboolean android_net_wifi_scanCommand(JNIEnv* env, jobject clazz)
{
    jboolean result;
    // Ignore any error from setting the scan mode.
    // The scan will still work.
    (void)doBooleanCommand("DRIVER SCAN-ACTIVE", "OK");
    result = doBooleanCommand("SCAN", "OK");
    (void)doBooleanCommand("DRIVER SCAN-PASSIVE", "OK");
    return result;
}

```

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，

```

void handleEvent(int event, String remainder) {
    case SCAN_RESULTS:
        mWifiStateTracker.notifyScanResultsAvailable();
        break;
}

```

`WifiStateTracker` 则接着广播发送 `SCAN_RESULTS_AVAILABLE_ACTION` 这个 `Intent`

```

case EVENT_SCAN_RESULTS_AVAILABLE:
    mContext.sendBroadcast(new
Intent(WifiManager.SCAN_RESULTS_AVAILABLE_ACTION));

```

`WifiLayer` 注册了接收 `SCAN_RESULTS_AVAILABLE_ACTION` 这个 `Intent`，所以它的相关处理函数 `handleScanResultsAvailable` 会被调用，在该函数中，先去拿到 `SCAN` 的结果（最

终是往 wpa\_supplicant 发送 SCAN\_RESULT 命令并读取返回值来实现的),

```
List<ScanResult> list = mWifiManager.getScanResults();
```

对每一个扫描返回的 AP, WifiLayer 会调用 WifiSettings 的 onAccessPointSetChanged 函数, 从而最终把该 AP 加到 GUI 显示列表中。

```
public void onAccessPointSetChanged(AccessPointState ap, boolean added) {

    AccessPointPreference pref = mAps.get(ap);

    if (added) {

        if (pref == null) {
            pref = new AccessPointPreference(this, ap);
            mAps.put(ap, pref);
        } else {
            pref.setEnabled(true);
        }

        mApCategory.addPreference(pref);

    }

}
```

### 3. 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,

```
public boolean onPreferenceTreeClick(PreferenceScreen preferenceScreen, Preference
preference) {
    if (preference instanceof AccessPointPreference) {
        AccessPointState state = ((AccessPointPreference)
preference).getAccessPointState();
        showAccessPointDialog(state, AccessPointDialog.MODE_INFO);
    }
}
```

### 4. 连接

当用户在 AccessPointDialog 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。

```
private void handleConnect() {
    String password = getEnteredPassword();
    if (!TextUtils.isEmpty(password)) {
        mState.setPassword(password);
    }

    mWifiLayer.connectToNetwork(mState);
}
```

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,

```
// Need WifiConfiguration for the AP
WifiConfiguration config = findConfiguredNetwork(state);
```

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,

```
if (config == null) {
    // Connecting for the first time, need to create it
    config = addConfiguration(state,
ADD_CONFIGURATION_ENABLE|ADD_CONFIGURATION_SAVE);
}
```

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。

```
// Make sure that network is enabled, and disable others
mReenableApsOnNetworkStateChange = true;
if (!mWifiManager.enableNetwork(state.networkId, true)) {
    Log.e(TAG, "Could not enable network ID " + state.networkId);
    error(R.string.error_connecting);
    return false;
}
```

## 5. 配置 IP 地址

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,

```
void handleEvent(int event, String remainder) {
    case CONNECTED:
        handleNetworkStateChange(NetworkInfo.DetailedState.CONNECTED,
remainder);
        break;
```

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,

```
private void handleConnectedState() {
    setPollTimer();
    mLastSignalLevel = -1;
    if (!mHaveIPAddress && !mObtainingIPAddress) {
        mObtainingIPAddress = true;
        mDhcpTarget.obtainMessage(EVENT_DHCP_START).sendToTarget();
    }
}
```

然后再广播发送 NETWORK\_STATE\_CHANGED\_ACTION 这个 Intent

```
case EVENT_NETWORK_STATE_CHANGED:
    if (result.state != DetailedState.DISCONNECTED || !mDisconnectPending) {
        intent = new
Intent(WifiManager.NETWORK_STATE_CHANGED_ACTION);
        intent.putExtra(WifiManager.EXTRA_NETWORK_INFO,
mNetworkInfo);
```

```

        if (result.BSSID != null)
            intent.putExtra(WifiManager.EXTRA_BSSID, result.BSSID);
        mContext.sendStickyBroadcast(intent);
    }
    break;

```

WifiLayer 注册了接收 NETWORK\_STATE\_CHANGED\_ACTION 这个 Intent，所以它的相关处理函数 handleNetworkStateChanged 会被调用，当 DHCP 拿到 IP 地址之后，会再发送 EVENT\_DHCP\_SUCCEEDED 消息，

```

private class DhcpHandler extends Handler {
    public void handleMessage(Message msg) {
        switch (msg.what) {
            case EVENT_DHCP_START:
                if (NetworkUtils.runDhcp(mInterfaceName, mDhcpInfo)) {
                    event = EVENT_DHCP_SUCCEEDED;
                }
            }
        }
    }
}

```

WifiLayer 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。

```

        case EVENT_DHCP_SUCCEEDED:
            mWifiInfo.setIpAddress(mDhcpInfo.ipAddress);
            setDetailedState(DetailedState.CONNECTED);
            intent = new
Intent(WifiManager.NETWORK_STATE_CHANGED_ACTION);
            intent.putExtra(WifiManager.EXTRA_NETWORK_INFO, mNetworkInfo);
            mContext.sendStickyBroadcast(intent);
            break;

```

至此为止，整个连接过程完成。

**问题：**

目前的实现不支持 Ad-hoc 方式。