# TODO

# Wificond

具体分析：https://blog.csdn.net/sinat\_20059415/article/details/80945447

system/connectivity/wificond/

基于用户需求、功耗优化等，Softap功能常常会在原生Android基础上做一些定制，比如：STA的接入/断开通知；获取当前接入的STA个数和列表；softap的休眠机制等。这些定制都离不开一个基础，就是Android softap中STA接入断开的消息传递机制。掌握了消息传递机制，这些定制只是在该机制中加入一些边边角角，so easy～

前言：之前在[（五十） Android O WiFi的扫描流程梳理](https://blog.csdn.net/sinat_20059415/article/details/80784129#t12) 中梳理到wificond，就是找不到wificond对应的具体实现类

mWificond = mWifiInjector.makeWificond();找不到对应的java实现

IBinder binder = ServiceManager.getService(WIFICOND\_SERVICE\_NAME)

bullhead:/ $ service list | grep wificond

128 wificond: []

这个与android7不同。

服务端是由cpp实现的，这种aidl实现方式被Google成为aidl-cpp

clientInterface = mWificond.createClientInterface()

IWifiScannerImpl也是类似实现

mWificondScanner = mClientInterface.getWifiScannerImpl();

**aidl存放路径**

./system/connectivity/wificond/aidl/android/net/wifi/IWificond.aidl

编译之后

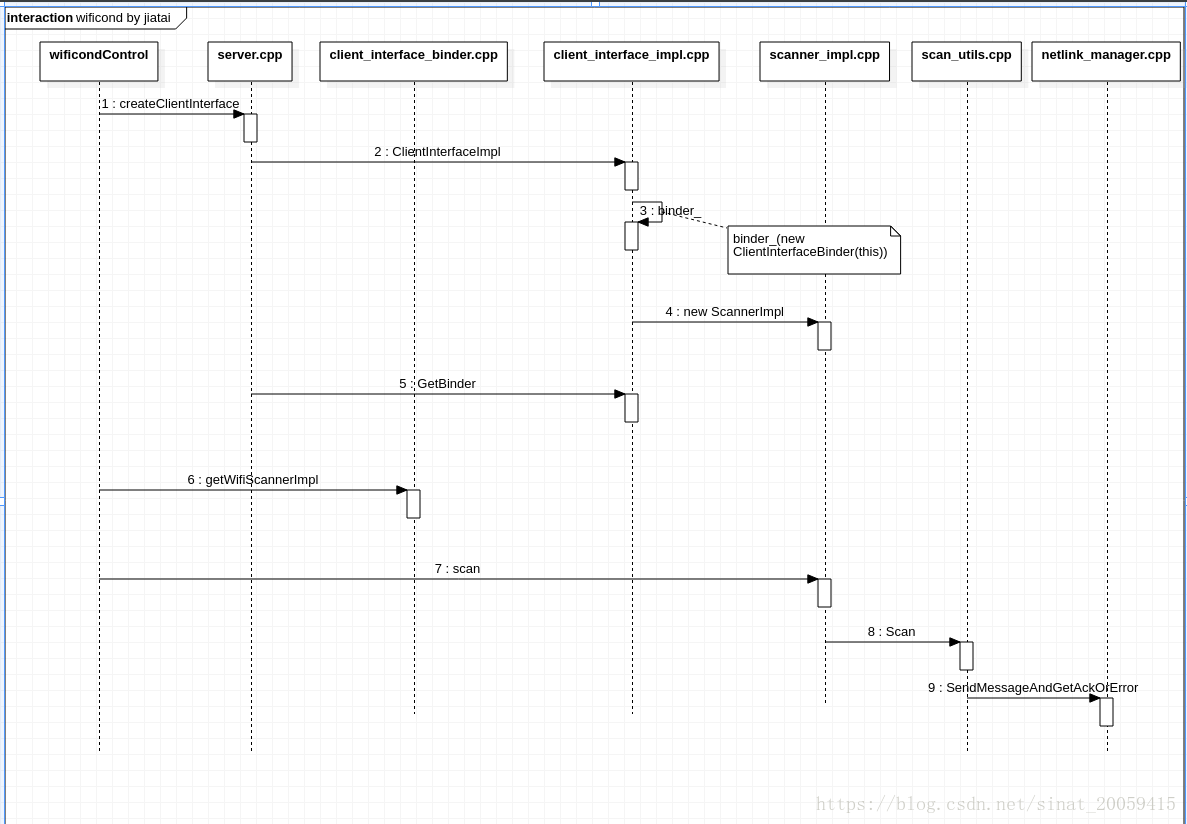
aidl对应的.h文件存放路径：

aidl对应的.cpp文件存放路径：

/out/target/product/xxxxxx/obj/STATIC\_LIBRARIES/libwificond\_ipc\_intermediates/aidl-generated/src/aidl/android/net/wifi/IWificond.cpp

可以通过“make wificond”编译出如下的执行文件，在system/bin下

## 框架



## main

system/connectivity/wificond/main.cpp

**int** main(**int** argc, **char**\*\* argv) {  
 android::base::InitLogging(argv, android::base::LogdLogger(android::base::SYSTEM));  
 LOG(INFO) << **"wificond is starting up..."**;  
  
 unique\_ptr<android::wificond::LooperBackedEventLoop> event\_dispatcher(  
 **new** android::wificond::LooperBackedEventLoop());  
 ScopedSignalHandler scoped\_signal\_handler(event\_dispatcher.get());  
  
 **int** binder\_fd = SetupBinderOrCrash();  
 CHECK(event\_dispatcher->WatchFileDescriptor(  
 binder\_fd,  
 android::wificond::EventLoop::kModeInput,  
 &OnBinderReadReady)) << **"Failed to watch binder FD"**;  
  
 android::wificond::NetlinkManager netlink\_manager(event\_dispatcher.get());  
 CHECK(netlink\_manager.Start()) << **"Failed to start netlink manager"**;  
 android::wificond::NetlinkUtils netlink\_utils(&netlink\_manager);  
 android::wificond::ScanUtils scan\_utils(&netlink\_manager);  
  
 unique\_ptr<android::wificond::Server> server(**new** android::wificond::Server(  
 unique\_ptr<InterfaceTool>(**new** InterfaceTool),  
 unique\_ptr<SupplicantManager>(**new** SupplicantManager()),  
 unique\_ptr<HostapdManager>(**new** HostapdManager()),  
 &netlink\_utils,  
 &scan\_utils));  
 server->CleanUpSystemState();  
 RegisterServiceOrCrash(server.get());  
  
 event\_dispatcher->Poll();  
 LOG(INFO) << **"wificond is about to exit"**;  
 **return** 0;  
}

### RegisterServiceOrCrash

**void** RegisterServiceOrCrash(**const** android::sp<android::IBinder>& service) {  
 android::sp<android::IServiceManager> sm = android::defaultServiceManager();  
 CHECK\_EQ(sm != NULL, **true**) << **"Could not obtain IServiceManager"**;  
  
 CHECK\_EQ(sm->addService(android::String16(kServiceName), service),  
 android::NO\_ERROR);  
}

system/connectivity/wificond/ipc\_constants.cpp

**namespace** android {  
**namespace** wificond {  
**namespace** ipc\_constants {  
  
**const char** kServiceName[] = **"wificond"**;  
  
} *// namespace ipc\_constants*} *// namespace wificond*} *// namespace android*

**using** android::wificond::ipc\_constants::kServiceName;

## Server

### Server（）

Server::Server(unique\_ptr<InterfaceTool> if\_tool,  
 unique\_ptr<SupplicantManager> supplicant\_manager,  
 unique\_ptr<HostapdManager> hostapd\_manager,  
 NetlinkUtils\* netlink\_utils,  
 ScanUtils\* scan\_utils)  
 : if\_tool\_(std::move(if\_tool)),  
 supplicant\_manager\_(std::move(supplicant\_manager)),  
 hostapd\_manager\_(std::move(hostapd\_manager)),  
 netlink\_utils\_(netlink\_utils),  
 scan\_utils\_(scan\_utils) {  
}

## NetlinkManager

### SendMessageAndGetSingleResponse

**bool** NetlinkManager::SendMessageAndGetSingleResponse(  
 **const** NL80211Packet& packet,  
 unique\_ptr<**const** NL80211Packet>\* response) {  
 unique\_ptr<**const** NL80211Packet> response\_or\_error;  
 **if** (!SendMessageAndGetSingleResponseOrError(packet, &response\_or\_error)) {  
 **return false**;  
 }  
 **if** (response\_or\_error->GetMessageType() == NLMSG\_ERROR) {  
 *// We use ERROR because we are not expecting to receive a ACK here.  
 // In that case the caller should use |SendMessageAndGetAckOrError|.* LOG(ERROR) << **"Received error message: "** << strerror(response\_or\_error->GetErrorCode());  
 **return false**;  
 }  
 \*response = std::move(response\_or\_error);  
 **return true**;  
}

### SendMessageAndGetSingleResponseOrError

**bool** NetlinkManager::SendMessageAndGetSingleResponseOrError(  
 **const** NL80211Packet& packet,  
 unique\_ptr<**const** NL80211Packet>\* response) {  
 vector<unique\_ptr<**const** NL80211Packet>> response\_vec;  
 **if** (!SendMessageAndGetResponses(packet, &response\_vec)) {  
 **return false**;  
 }  
 **if** (response\_vec.size() != 1) {  
 LOG(ERROR) << **"Unexpected response size: "** << response\_vec.size();  
 **return false**;  
 }  
  
 \*response = std::move(response\_vec[0]);  
 **return true**;  
}

### SendMessageAndGetResponses

**bool** NetlinkManager::SendMessageAndGetResponses(  
 **const** NL80211Packet& packet,  
 vector<unique\_ptr<**const** NL80211Packet>>\* response) {  
 **if** (!SendMessageInternal(packet, sync\_netlink\_fd\_.get())) {  
 **return false**;  
 }  
 *// Polling netlink socket, waiting for GetFamily reply.* **struct** pollfd netlink\_output;  
 memset(&netlink\_output, 0, **sizeof**(netlink\_output));  
 netlink\_output.fd = sync\_netlink\_fd\_.get();  
 netlink\_output.events = POLLIN;  
  
 uint32\_t sequence = packet.GetMessageSequence();  
  
 **int** time\_remaining = kMaximumNetlinkMessageWaitMilliSeconds;  
 *// Multipart messages may come with seperated datagrams, ending with a  
 // NLMSG\_DONE message.  
 // ReceivePacketAndRunHandler() will remove the handler after receiving a  
 // NLMSG\_DONE message.* message\_handlers\_[sequence] = std::bind(AppendPacket, response, \_1);  
  
 **while** (time\_remaining > 0 &&  
 message\_handlers\_.find(sequence) != message\_handlers\_.end()) {  
 nsecs\_t interval = systemTime(SYSTEM\_TIME\_MONOTONIC);  
 **int** poll\_return = poll(&netlink\_output,  
 1,  
 time\_remaining);  
  
 **if** (poll\_return == 0) {  
 LOG(ERROR) << **"Failed to poll netlink fd: time out "**;  
 message\_handlers\_.erase(sequence);  
 **return false**;  
 } **else if** (poll\_return == -1) {  
 LOG(ERROR) << **"Failed to poll netlink fd: "** << strerror(errno);  
 message\_handlers\_.erase(sequence);  
 **return false**;  
 }  
 ReceivePacketAndRunHandler(sync\_netlink\_fd\_.get());  
 interval = systemTime(SYSTEM\_TIME\_MONOTONIC) - interval;  
 time\_remaining -= **static\_cast**<**int**>(ns2ms(interval));  
 }  
 **if** (time\_remaining <= 0) {  
 LOG(ERROR) << **"Timeout waiting for netlink reply messages"**;  
 message\_handlers\_.erase(sequence);  
 **return false**;  
 }  
 **return true**;  
}

### ReceivePacketAndRunHandler

**void** NetlinkManager::ReceivePacketAndRunHandler(**int** fd)

uint32\_t sequence\_number = packet->GetMessageSequence();  
  
*// Handle multicasts.***if** (sequence\_number == kBroadcastSequenceNumber) {  
 BroadcastHandler(std::move(packet));  
 **continue**;  
}

### BroadcastHandler

Df

**void** NetlinkManager::BroadcastHandler(unique\_ptr<**const** NL80211Packet> packet) {

uint32\_t command = packet->GetCommand();

…

*// Station eventsFor AP mode.* **if** (command == NL80211\_CMD\_NEW\_STATION ||  
 command == NL80211\_CMD\_DEL\_STATION) {  
 uint32\_t if\_index;  
 **if** (!packet->GetAttributeValue(NL80211\_ATTR\_IFINDEX, &if\_index)) {  
 LOG(WARNING) << **"Failed to get interface index from station event"**;  
 **return**;  
 }  
 **const auto** handler = on\_station\_event\_handler\_.find(if\_index);  
 **if** (handler != on\_station\_event\_handler\_.end()) {  
 vector<uint8\_t> mac\_address;  
 **if** (!packet->GetAttributeValue(NL80211\_ATTR\_MAC, &mac\_address)) {  
 LOG(WARNING) << **"Failed to get mac address from station event"**;  
 **return**;  
 }  
 **if** (command == NL80211\_CMD\_NEW\_STATION) {  
 handler->second(NEW\_STATION, mac\_address);  
 } **else** {  
 handler->second(DEL\_STATION, mac\_address);  
 }  
 }  
 **return**;  
 }  
}

解析每一个80211的wifi包，NL80211\_CMD\_NEW\_STATION表示新的设备接入进来，从而通知Ap基站变化

## ApInterfaceImpl

system/connectivity/wificond/ap\_interface\_impl.cpp

pInterfaceImpl::ApInterfaceImpl(**const** string& interface\_name,  
 uint32\_t interface\_index,  
 NetlinkUtils\* netlink\_utils,  
 InterfaceTool\* if\_tool,  
 HostapdManager\* hostapd\_manager)  
 : interface\_name\_(interface\_name),  
 interface\_index\_(interface\_index),  
 netlink\_utils\_(netlink\_utils),  
 if\_tool\_(if\_tool),  
 hostapd\_manager\_(hostapd\_manager),  
 binder\_(**new** ApInterfaceBinder(**this**)),  
 number\_of\_associated\_stations\_(0) {  
 *// This log keeps compiler happy.* LOG(DEBUG) << **"Created ap interface "** << interface\_name\_  
 << **" with index "** << interface\_index\_;  
  
 netlink\_utils\_->SubscribeStationEvent(  
 interface\_index\_,  
 std::bind(&ApInterfaceImpl::OnStationEvent,  
 **this**,  
 \_1, \_2));

OnStationEvent是重要的网络监听方法

### StartHostapd时序

**bool** ApInterfaceImpl::StartHostapd() {  
 **return** hostapd\_manager\_->StartHostapd();  
}

### 注册设备监听

在构造函数中注册了

**void** NetlinkUtils::SubscribeStationEvent(uint32\_t interface\_index,  
 OnStationEventHandler handler) {  
 netlink\_manager\_->SubscribeStationEvent(interface\_index, handler);  
}

最终注册到NetlinkManager的map表里面

**void** NetlinkManager::SubscribeStationEvent(  
 uint32\_t interface\_index,  
 OnStationEventHandler handler) {  
 on\_station\_event\_handler\_[interface\_index] = handler;  
}

### OnStationEvent（监听设备关联通知）

**void** ApInterfaceImpl::OnStationEvent(StationEvent event,  
 **const** vector<uint8\_t>& mac\_address) {  
 **if** (event == NEW\_STATION) {  
 LOG(INFO) << **"New station "** << LoggingUtils::GetMacString(mac\_address)  
 << **" associated with hotspot"**;  
 number\_of\_associated\_stations\_++;  
 } **else if** (event == DEL\_STATION) {  
 LOG(INFO) << **"Station "** << LoggingUtils::GetMacString(mac\_address)  
 << **" disassociated from hotspot"**;  
 **if** (number\_of\_associated\_stations\_ <= 0) {  
 LOG(ERROR) << **"Received DEL\_STATION event when station counter is: "** << number\_of\_associated\_stations\_;  
 } **else** {  
 number\_of\_associated\_stations\_--;  
 }  
 }  
}

如果设备接入进来，可以查看这个

01-18 17:05:40.066 567-567/? I/wificond: New station ec:d0:9f:b0:df:97 associated with hotspot

### GetNumberOfAssociatedStations

获取关联设备数量，可以通过这个啊

**int** ApInterfaceImpl::GetNumberOfAssociatedStations() **const** {  
 **return** number\_of\_associated\_stations\_;  
}

**void** NetlinkUtils::SubscribeStationEvent(uint32\_t interface\_index,  
 OnStationEventHandler handler) {  
 netlink\_manager\_->SubscribeStationEvent(interface\_index, handler);  
}

## AIDL

### IApInterface.aidl

system/connectivity/wificond/ap\_interface\_binder.cpp

system/connectivity/wificond/aidl/android/net/wifi/IApInterface.aidl

实现了cpp层和java层的通信

**interface** IApInterface {  
  
 const **int** ENCRYPTION\_TYPE\_NONE = 0;  
 const **int** ENCRYPTION\_TYPE\_WPA = 1;  
 const **int** ENCRYPTION\_TYPE\_WPA2 = 2;  
  
 *// Start up an instance of hostapd associated with this interface.  
 // @return true on success.* **boolean** startHostapd();  
  
 *// Stop a previously started instance of hostapd.  
 // @return true on success.* **boolean** stopHostapd();  
  
 *// Write out a configuration file for hostapd. This will be used on the next  
 // successful call to StartHostapd(). Returns true on success.  
 //  
 // @param ssid string of <=32 bytes to use as the SSID for this AP.  
 // @param isHidden True iff the AP should not broadcast its SSID.  
 // @param channel WiFi channel to expose the AP on.  
 // @param encryptionType one of ENCRYPTION\_TYPE\* above.  
 // @param passphrase string of bytes to use as the passphrase for this AP.  
 // Ignored if encryptionType is None.  
 // @return true on success.* **boolean** writeHostapdConfig(**in byte**[] ssid, **boolean** isHidden, **int** channel,  
 **int** encryptionType, **in byte**[] passphrase);  
  
 *// Retrieve the name of the network interface corresponding to this  
 // IApInterface instance (e.g. "wlan0")* @utf8InCpp  
 String getInterfaceName();  
  
 *// @return Returns the number of associated devices to this hotspot.  
 // Returns -1 on failure.* **int** getNumberOfAssociatedStations();  
  
}

#### 对应的实现

binder::Status ApInterfaceBinder::getNumberOfAssociatedStations(  
 **int**\* out\_num\_of\_stations) {  
 **if** (!impl\_) {  
 LOG(WARNING) << **"Cannot get number of associated stations "** << **"from dead ApInterface"**;  
 \*out\_num\_of\_stations = -1;  
 **return** binder::Status::ok();  
 }  
 \*out\_num\_of\_stations = impl\_->GetNumberOfAssociatedStations();  
 **return** binder::Status::ok();  
}

### IInterfaceEventCallback.aidl

system/connectivity/wificond/aidl/android/net/wifi/IInterfaceEventCallback.aidl

*// A callback for receiving events related to this chip.***interface** IInterfaceEventCallback {  
  
 *// Signals that the provided interface is ready for future commands.* **oneway void** OnClientInterfaceReady(IClientInterface network\_interface);  
 **oneway void** OnApInterfaceReady(IApInterface network\_interface);  
  
 *// Signals that an interface was torn down.  
 // No future callbacks will be delivered via this callback, and the callback  
 // is automatically unregistered.* **oneway void** OnClientTorndownEvent(IClientInterface network\_interface);  
 **oneway void** OnApTorndownEvent(IApInterface network\_interface);  
}

# REF

Android O WiFi的扫描流程梳理续——梳理java与c++之间的aidl-cpp通信

https://blog.csdn.net/sinat\_20059415/article/details/80945447