



经典畅销书“深入理解Android”系列新作，资深Android系统专家邓凡平撰写，  
全志、高通等公司资深专家担任技术审核并强烈推荐  
从专业知识角度和Android系统代码实现角度对Netd、Wi-Fi、NFC和GPS模块代  
码进行深入剖析，深刻揭示其实现原理和工作机制

移动开发



邓凡平◎著

Understanding Android Internals  
Wi-Fi, NFC and GPS

# 深入理解Android Wi-Fi、NFC和GPS卷

移动开发

深入理解Android: Wi-Fi、NFC和GPS卷

邓凡平 著

ISBN: 978-7-111-45683-4

本书纸版由机械工业出版社于2014年出版，电子版由华章分社（北京华章图文信息有限公司）全球范围内制作与发行。

版权所有，侵权必究

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.hzmedia.com.cn

新浪微博 @研发书局

腾讯微博 @yanfabook

# 目 录

前言

本书主要内容及特色

读者对象

如何阅读本书

勘误和支持

致谢

第1章 准备工作

1. 1 Android系统架构

1. 2 工具使用

1. 2. 1 Source Insight的使用

1. 2. 2 Eclipse的使用

1. 2. 3 BusyBox的使用

1. 3 本书资源下载说明

第2章 深入理解Netd

2. 1 概述

2. 2 Netd工作流程

2. 2. 1 main函数分析

2. 2. 2 NetlinkManager分析

2. 2. 3 CommandListener分析

2. 2. 4 DnsProxyListener分析

2. 2. 5 MDnsSdListener分析

2. 3 CommandListener中的命令

2. 3. 1 iptables、tc和ip命令

2. 3. 2 CommandListener构造函数和测试工具ndc

2. 3. 3 InterfaceCmd命令

2. 3. 4 IpFwd和FirewallCmd命令

2. 3. 5 ListTtysCmd和PppdCmd命令

2. 3. 6 BandwidthControlCmd和IdletimerControlCmd命令

2. 3. 7 NatCmd命令

2. 3. 8 TetherCmd和SoftapCmd命令

2. 3. 9 ResolverCmd命令

2. 4 NetworkManagementService介绍

2. 4. 1 create函数详解

2. 4. 2 systemReady函数详解

2. 5 本章总结和参考资料说明

2. 5. 1 本章总结

## 2.5.2 参考资料说明

### 第3章 Wi-Fi基础知识

#### 3.1 概述

#### 3.2 无线电频谱和802.11协议的发展历程

##### 3.2.1 无线电频谱知识[2]

##### 3.2.2 IEEE 802.11发展历程

##### 3.3 802.11无线网络技术

###### 3.3.1 OSI基本参考模型及相关基本概念

###### 3.3.2 802.11知识点导读

###### 3.3.3 802.11组件

###### 3.3.4 802.11 Service介绍[13]

###### 3.3.5 802.11 MAC服务和帧

###### 3.3.6 802.11 MAC管理实体[22][23]

###### 3.3.7 无线网络安全技术知识点

#### 3.4 Linux Wi-Fi编程API介绍

##### 3.4.1 Linux Wireless Extensions介绍

##### 3.4.2 nl80211介绍

#### 3.5 本章总结和参考资料说明

##### 3.5.1 本章总结

##### 3.5.2 参考资料说明

### 第4章 深入理解wpa\_supplicant

#### 4.1 概述

#### 4.2 初识wpa\_supplicant

##### 4.2.1 wpa\_supplicant架构

##### 4.2.2 wpa\_supplicant编译配置

##### 4.2.3 wpa\_supplicant命令和控制API

##### 4.2.4 git的使用

#### 4.3 wpa\_supplicant初始化流程

##### 4.3.1 main函数分析

##### 4.3.2 wpa\_supplicant init函数分析

##### 4.3.3 wpa\_supplicant add iface函数分析

##### 4.3.4 wpa\_supplicant init iface函数分析

#### 4.4 EAP和EAPOL模块

##### 4.4.1 EAP模块分析[20]

##### 4.4.2 EAPOL模块分析[21]

#### 4.5 wpa\_supplicant连接无线网络分析

##### 4.5.1 ADD NETWORK命令处理

4.5.2 SET\_NETWORK命令处理

4.5.3 ENABLE\_NETWORK命令处理

4.6 本章总结和参考资料说明

4.6.1 本章总结

4.6.2 参考资料说明

第5章 深入理解WifiService

5.1 概述

5.2 WifiService的创建及初始化

5.2.1 HSM和AsyncChannel介绍

5.2.2 WifiService构造函数分析

5.2.3 WifiStateMachine介绍

5.3 加入无线网络分析

5.3.1 Settings操作Wi-Fi分析

5.3.2 WifiService操作Wi-Fi分析

5.4 WifiWatchdogStateMachine介绍

5.5 Captive Portal Check介绍

5.6 本章总结和参考资料说明

5.6.1 本章总结

5.6.2 参考资料说明

第6章 深入理解Wi-Fi Simple Configuration

6.1 概述

6.2 WSC基础知识

6.2.1 WSC应用场景[1]

6.2.2 WSC核心组件及接口[2]

6.3 Registration Protocol详解[3]

6.3.1 WSC IE和Attribute介绍[4]

6.3.2 802.11管理帧WSC IE设置[4]

6.3.3 EAP-WSC介绍[4][5]

6.4 WSC代码分析

6.4.1 Settings中的WSC处理

6.4.2 WifiStateMachine的处理

6.4.3 wpa\_supplicant中的WSC处理

6.4.4 EAP-WSC处理流程分析

6.5 本章总结和参考资料说明

6.5.1 本章总结

6.5.2 参考资料说明

第7章 深入理解Wi-Fi P2P

- [7.1 概述](#)
- [7.2 P2P基础知识](#)
- [7.2.1 P2P架构\[1\]](#)
- [7.2.2 P2P Discovery技术\[4\]](#)
- [7.2.3 P2P工作流程](#)
- [7.3 WifiP2pSettings和WifiP2pService介绍](#)
- [7.3.1 WifiP2pSettings工作流程](#)
- [7.3.2 WifiP2pService工作流程](#)
- [7.4 wpa\\_supplicant中的P2P](#)
- [7.4.1 P2P模块初始化](#)
- [7.4.2 P2P Device Discovery流程分析](#)
- [7.4.3 Provision Discovery流程分析](#)
- [7.4.4 GO Negotiation流程分析](#)
- [7.5 本章总结和参考资料说明](#)
- [7.5.1 本章总结](#)
- [7.5.2 参考资料说明](#)
- [第8章 深入理解NFC](#)
- [8.1 概述](#)
- [8.2 NFC基础知识](#)
- [8.2.1 NFC概述](#)
- [8.2.2 NFC R/W运行模式](#)
- [8.2.3 NFC P2P运行模式\[12\]](#)
- [8.2.4 NFC CE运行模式\[15\]\[16\]](#)
- [8.2.5 NCI原理\[20\]](#)
- [8.2.6 NFC相关规范](#)
- [8.3 Android中的NFC](#)
- [8.3.1 NFC应用示例](#)
- [8.3.2 NFC系统模块](#)
- [8.4 NFC HAL层讨论](#)
- [8.5 本章总结和参考资料说明](#)
- [8.5.1 本章总结](#)
- [8.5.2 参考资料说明](#)
- [第9章 深入理解GPS](#)
- [9.1 概述](#)
- [9.2 GPS基础知识](#)
- [9.2.1 卫星导航基本原理](#)
- [9.2.2 GPS系统组成及原理](#)

- 9.2.3 OMA-SUPL协议[27][28].
- 9.3 Android中的位置管理
- 9.3.1 LocationManager架构
- 9.3.2 LocationManager应用示例
- 9.3.3 LocationManager系统模块
- 9.4 本章总结和参考资料说明
- 9.4.1 本章总结
- 9.4.2 参考资料说明
- 附录

# 前言

## 本书主要内容及特色

本书所讲解的Wi-Fi、NFC以及GPS模块的背后都涉及非常多的专业知识，例如与Wi-Fi相关的802.11协议、Wi-Fi Alliance（Wi-Fi联盟）定义的Wi-Fi Simple Configuration和Wi-Fi P2P协议、NFC Forum定义的一整套与NFC相关的协议、与GPS相关的卫星导航原理、AGPS和OMA-SUPL协议等。显然，如果不了解这些专业知识，就不可能真正掌握它们在Android平台中的代码实现。

考虑到这些专业知识的重要性，本书在讲解Android平台中Wi-Fi、NFC和GPS模块的实现之前，先重点介绍与代码相关的专业知识。当然，这些专业知识内容如此丰富，在一本书中无法全部涵盖。为了方便读者进一步深入学习，本书每章的最后都会列举笔者在撰写各章时所阅读的参考资料。

以下是本书的内容概述。

- 第1章介绍本书的内容组成、使用的工具以及参考源码的下载方法。
- 第2章介绍Netd以及相关的背景知识。
- 第3章介绍Wi-Fi基础知识。Wi-Fi是本章的重点，而且也是当下最热门的技术。
- 第4章介绍wpa\_supplicant，它是Wi-Fi领域中最核心的软件实现。
- 第5章介绍WifiService，它是Android平台中特有的Wi-Fi服务模块。
- 第6章和第7章介绍Wi-Fi Alliance推出的两项重要技术——Wi-Fi Simple Configuration和Wi-Fi P2P，以及它们在Android平台中的代码实现。

- 第8章介绍NFC背景知识以及NFC在Android平台中的代码实现。NFC也是历史比较悠久的技术，希望它能随着Android的普及而走向大众。
- 第9章介绍GPS原理及Android平台中的位置管理服务架构。
- 附录为笔者和审稿专家之一的吴劲良先生关于本书定位、学习方法等方面的讨论。相信这些讨论内容能引起读者的共鸣。

本书通过理论和代码相结合的方式进行讲解，旨在引领读者一步步了解Wi-Fi、NFC和GPS模块的工作原理。总之，笔者希望读者在阅读完本书后能有以下收获。

- 初步掌握Wi-Fi、NFC和GPS的专业知识。
- 根据其实现代码，进一步加深对这些专业知识的理解。

## 读者对象

适合阅读本书的读者包括：

- Android系统开发工程师
- Wi-Fi、NFC或GPS的BSP开发工程师

BSP开发工程师更需要对Android平台中这些模块的工作原理及背景知识有深入的理解。虽然本书没有介绍这些模块在Linux Kernel层的实现，但了解它们在用户空间的工作流程也将极大帮助BSP开发工程师拓展自己的知识面。

- 对Wi-Fi、NFC和GPS感兴趣的在校高年级本科生、研究生和其他读者在掌握理论的基础上，如何在实际代码中来实现或使用它们也许是众多学子最想知道的。希望这本理论与代码实现深度结合的书籍会助您

一臂之力。

## 如何阅读本书

本书是一本专业知识和代码实现相结合的书籍，所以读者在阅读时应注意以下事项。

- 首先阅读专业知识。如果对这些内容比较了解，可以直接跳转到代码实现。
- 然后是Android平台中相关模块的代码实现。这些代码实现往往基于一定的专业知识，所以在阅读代码时务必和前述的专业知识相结合。
- 每章最后都列出了笔者在撰写各章时所参考的资料。资料较多，读者可根据这些内容开展进一步的研究工作。
- 每章开头都把本章涉及的源码路径全部列出，而在具体分析源码时，只列出该源码的文件名及所分析的函数或相关数据结构名。例如：

[-->AndroidRuntime.cpp: : 函数或数据结构名]

//源码分析和一些注释

最后，本书在描述类之间的关系及函数调用流程上，使用了UML的静态类图及序列图。UML是一个强大的工具，但它的建模规范过于烦琐，为更简单清晰地描述事情的本质，本书并未完全遵循UML的建模规范。如图1所示，外部类内部的方框用于表示内部类。另外，“外部类A. 内部类B”也用于表示内部类。接口和普通类用同一种框图表示。



## 图1 类图

图2所示为本书描述数据结构及成员时使用的UML图例。



## 图2 数据结构图

特别注意 本书所使用的UML图都比较简单，读者不必花费大量时间专门学习UML。另外，出于方便考虑，本书所绘制的UML图没有严格遵守UML规范，这一点敬请读者谅解。

本书涉及的Android源码及一些开发工具的下载地址为  
<http://115.com/lb/51bdugrdt4r>。关于它们的使用详情，请读者阅读1.3节。

## 勘误和支持

由于作者的水平有限，加之编写时间仓促，书中难免会出现一些错误或不准确的地方，恳请读者不吝批评指正。若有问题，可通过邮箱或在博客上留言与笔者共同商讨。笔者的联系方式如下。

- 邮箱：fanping.deng@gmail.com
- 博客：[blog.csdn.net/innost](http://blog.csdn.net/innost)和  
<http://my.oschina.net/innost/blog>

## 致谢

首先要感谢杨福川编辑的大力支持。另外，要感谢本书审稿编辑白宇严谨负责的工作。

特别感谢Tieto公司。Tieto开放的企业文化、Android团队高效的工作效率、团队成员之间默契的工作配合程度，以及领导无私而有力的支持着实让我感到幸运和自豪。在Tieto就职的一年中，笔者所在的Android团队不仅成功赢得了客户的信任，更是得到了Tieto公司总部和其他国家分公司同事们的一致认可。同时，团队成员还积极分享，并在《程序员》杂志上发表了六篇高质量的文章。

在此，笔者借助本书对Tieto的领导和同事表示衷心的感谢。他们是中國北京分公司的Leo、hongbin、James、yantao、meiyang、dujiang、changgeng、caimin、wenjing、huaizhi、huirong、xinzhi、huimin、yuzheng、Liuxuan、Emily、Diego、jinghua、Jenny等，中國成都分公司的tianxiang、chengguo等，波兰分公司的Marcin、Marciej、Filip Matusiak等、捷克分公司的Vaclav、Bronislav、Petrous Jan等、芬兰分公司的Mikel Echegoyen。

当然，本书能得以快速出版，还需要感谢两位功力深厚并热心参与技术审稿的专家。他们是全志（Allwinner）公司Wireless Team负责人吴劲良，以及高通（Qualcomm）中国资深研发经理杨洋。二位专家在各自领域所表现出来的专业素养和技术水平，时刻提醒笔者应牢记“路漫漫其修远兮，吾将上下而求索”。另外，高通中国资深研发经理毛晓冬也对本书成功编写提供了帮助，在此一并表示感谢。

最后，感谢我的家人，尤其是我的妻子。希望明年上天能恩赐一个健康可爱的宝宝，这样，我将拥有更加无穷的动力编写更多书籍来回馈花费宝贵时间和精力关注本书的读者，以及所有在人生和职业道路上曾给予我指导的诸位师长。

# 第1章 准备工作

本章主要内容

- 本书的内容组成；
- 工具使用；
- 本书资源下载说明。

## 1.1 Android系统架构

Android是Google公司推出的一款手机开发平台。该平台本身是基于Linux内核的，图1-1展示了这个系统的架构。

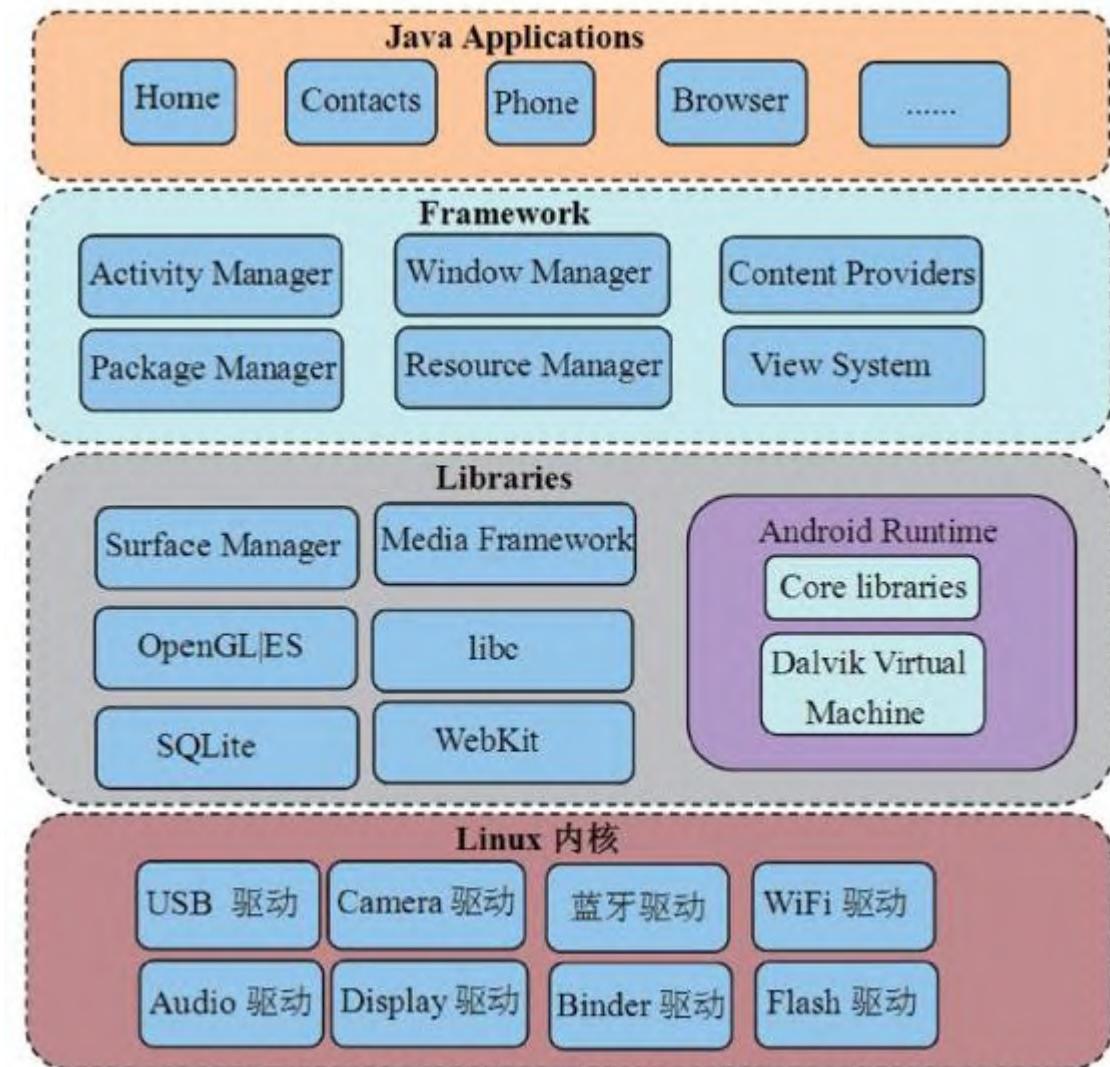


图1-1 Android系统架构

从图1-1可知，Android系统大体可分为四层，从下往上依次如下。

- Linux内核层，目前Android 4.4（代号为KitKat）基于Linux内核3.4版本。

- Libraries层，这一层提供动态库（也叫共享库）、Android运行时库、Dalvik虚拟机<sup>①</sup>等。从编程语言方面来说，这一层大部分都是用C或C++写的，所以也可以简单地把它看成是Native层。
  - Libraries层之上是Framework层，这一层大部分用Java语言编写。它是Android平台上Java世界的基石。
  - Framework层之上是Applications层，和用户直接交互的就是这些应用程序，它们都是用Java开发的。
- ①** 4.4版本新增了ART虚拟机运行时，相信它的出现能提升应用程序的运行速度。

## 1.2 工具使用

本节介绍Android开发和源码研究过程中的三件利器。

### 1.2.1 Source Insight的使用

Source Insight是阅读源码的必备工具，是一个Windows下的软件，在Linux平台上可通过wine安装。下面介绍一下如何在Source Insight中导入源码。

使用Source Insight时，需要新建一个源码工程，通过菜单项Project→New Project，可指定源码的目录。

提示 如果把Android所有源代码都加到工程中，将导致Source Insight运行速度非常慢。

实际上，只需要将当前分析的源码目录加到工程即可。例如，新建一个Source Insight工程后，只把源码/framework/base目录加进去了。另外，当一个目录下的源码分析完后，可以通过Project→Add and Remove Project Files选项把无须分析的目录从工程中去掉。上述步骤如图1-2所示。

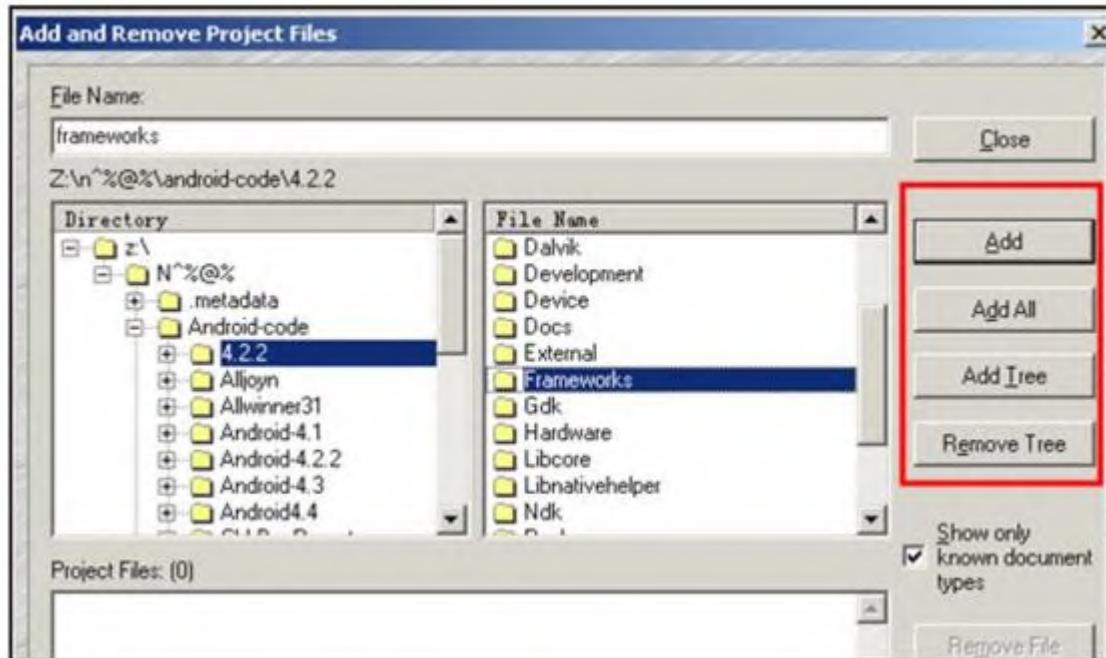


图1-2 添加或删除工程中的目录

从图1-2右边的框可知，Source Insight支持动态添加或删除目录。通过这种方式可极大减少Source Insight的工作负担。

**提示** 一般首先把framework/base下的目录加到工程，以后如有需要，再把其他目录加进来。另外，关于Source Insight其他使用技巧，可参考《深入理解Android：卷 I》第1章。

## 1.2.2 Eclipse的使用

笔者一般使用Source Insight来查看Native代码，而Android推荐的集成开发工具Eclipse既能查看Java代码和Native代码，也能调试系统核心进程。

### 1. 导入Android Framework Java源码

注意，这一步必须编译整个Android源码才可以实施，步骤如下。

1) 将Android源码目录/development/ide/eclipse/.classpath复制到Android源码根目录。

2) 打开Android源码根目录下的.classpath文件。该文件是供Eclipse使用的，其中保存的是源码目录中各个模块的路径。

由于我们只关心Framework相关的模块，因此可以把一些不是Framework的目录从该文件中注释掉。同时，去掉不必要的模块也可加快Android源码导入速度。图1-3所示为该文件的部分内容。

```
-->
<classpathentry kind="src" path="frameworks/basecmds/am/src"/>
<classpathentry kind="src" path="frameworks/basecmds/input/src"/>
<classpathentry kind="src" path="frameworks/basecmds/pm/src"/>
<classpathentry kind="src" path="frameworks/basecmds/svc/src"/>
<classpathentry kind="src" path="frameworks/base/core/java"/>
<classpathentry kind="src" path="frameworks/base/drm/java"/>
<classpathentry kind="src" path="frameworks/base/graphics/java"/>
<classpathentry kind="src" path="frameworks/base/icu4j/java"/>
<classpathentry kind="src" path="frameworks/base/keystore/java"/>
<classpathentry kind="src" path="frameworks/base/location/java"/>
<classpathentry kind="src" path="frameworks/base/location/lib/java"/>
<classpathentry kind="src" path="frameworks/base/media/java"/>
<classpathentry kind="src" path="frameworks/base/media/mca/effect/java"/>
<classpathentry kind="src" path="frameworks/base/media/mca/filterfw/java"/>
<classpathentry kind="src" path="frameworks/base/media/mca/filterpacks/java"/>
<classpathentry kind="src" path="frameworks/base/nfc-extras/java"/>
<classpathentry kind="src" path="frameworks/base/obex"/>
<classpathentry kind="src" path="frameworks/base/opengl/java"/>
<classpathentry kind="src" path="frameworks/base/packages/FusedLocation/src"/>
<classpathentry kind="src" path="frameworks/base/packages/SettingsProvider/src"/>
<classpathentry kind="src" path="frameworks/base/packages/SystemUI/src"/>
<classpathentry kind="src" path="frameworks/base/policy/src"/>
<classpathentry kind="src" path="frameworks/base/sax/java"/>
<classpathentry kind="src" path="frameworks/base/services/java"/>
<classpathentry kind="src" path="frameworks/base/telephony/java"/>
<classpathentry kind="src" path="frameworks/base/test-runner/src"/>
<classpathentry kind="src" path="frameworks/base/voip/java"/>
<classpathentry kind="src" path="frameworks/base/wifi/java"/>
<classpathentry kind="src" path="frameworks/ex/carousel/java"/>
<classpathentry kind="src" path="frameworks/ex/chips/src"/>
<classpathentry kind="src" path="frameworks/ex/common/java"/>
▼<!--
    classpathentry kind="src" path="frameworks/ex/photoviewer/src"/
-->
```

图1-3 .classpath文件内容

然后，单击Eclipse菜单栏New→Java Project，弹出如图1-4所示的对话框。设置Location为Android 4.2源码所在路径。

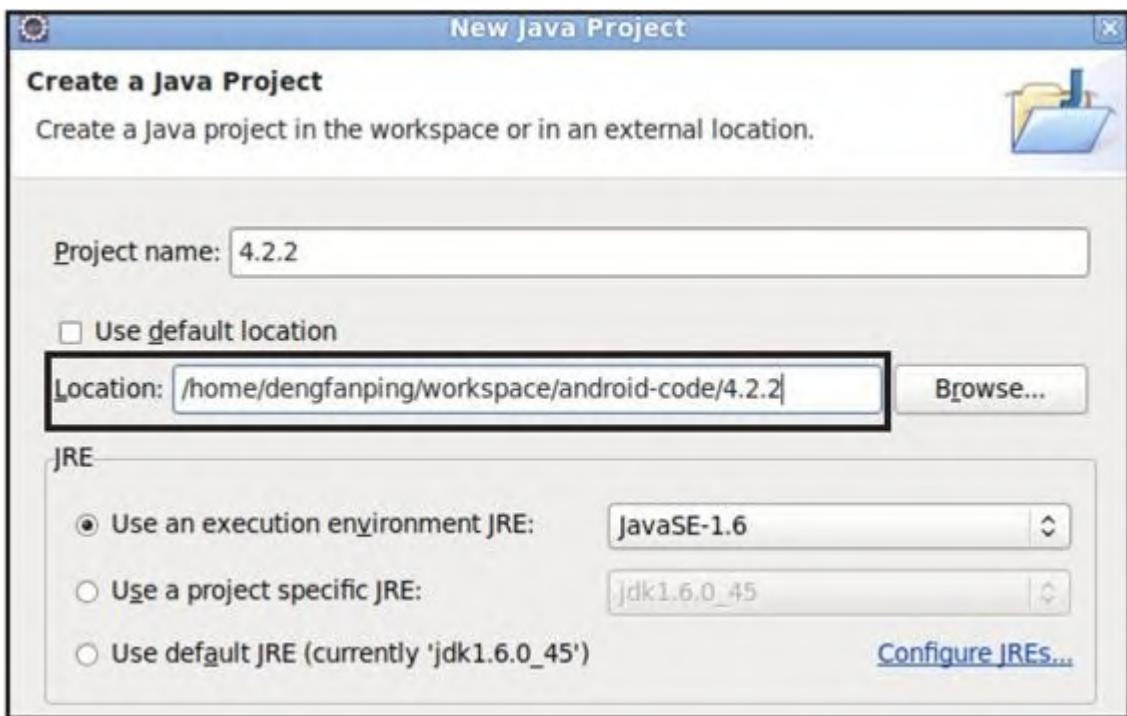


图1-4 导入Android源码

由于Android 4.2源码文件较多，导入过程会持续较长一段时间。

**提示** 导入源码前一定要取消Eclipse的自动编译选项（通过菜单栏Project→Build Project Automatically设置）。另外，源码导入完毕后，千万不要清理（clean）这个工程。清理会删除之前源码编译所生成的文件，导致后续又要重新编译Android系统了。

本书的共享资源中提供了一个已经配置好的. classpath文件，供读者下载并使用。

## 2. 导入Android Native代码

本节介绍如何在Eclipse中导入Android Native代码，其步骤如下所示。

导入Android Framework中的Java文件后，首先切换到C/C++视图（通过单击菜单栏Window→Open Perspective→选择C/C++）。

单击菜单栏New→C/C++，然后选择Convert to a C/C++project，如图1-5所示。

按照图1-5操作之后，之前的Java工程就被转成C++工程。不过读者仍需要完成以下几个步骤。

- 1) 通过Properties→C/C++General→Paths and symbols，打开该工程的路径和符号设置对话。如图1-6所示。
- 2) 在图1-6上边的框中选择Includes页面，然后单击下方框中的Import Settings，选择4.2源码/development/ide/eclipse/android-include-paths.xml以导入路径配置。
- 3) 同上，选择图1-6中的Symbols页面，然后通过下方的Import Settings导入4.2源码/development/ide/eclipse/android-symbols.xml文件。

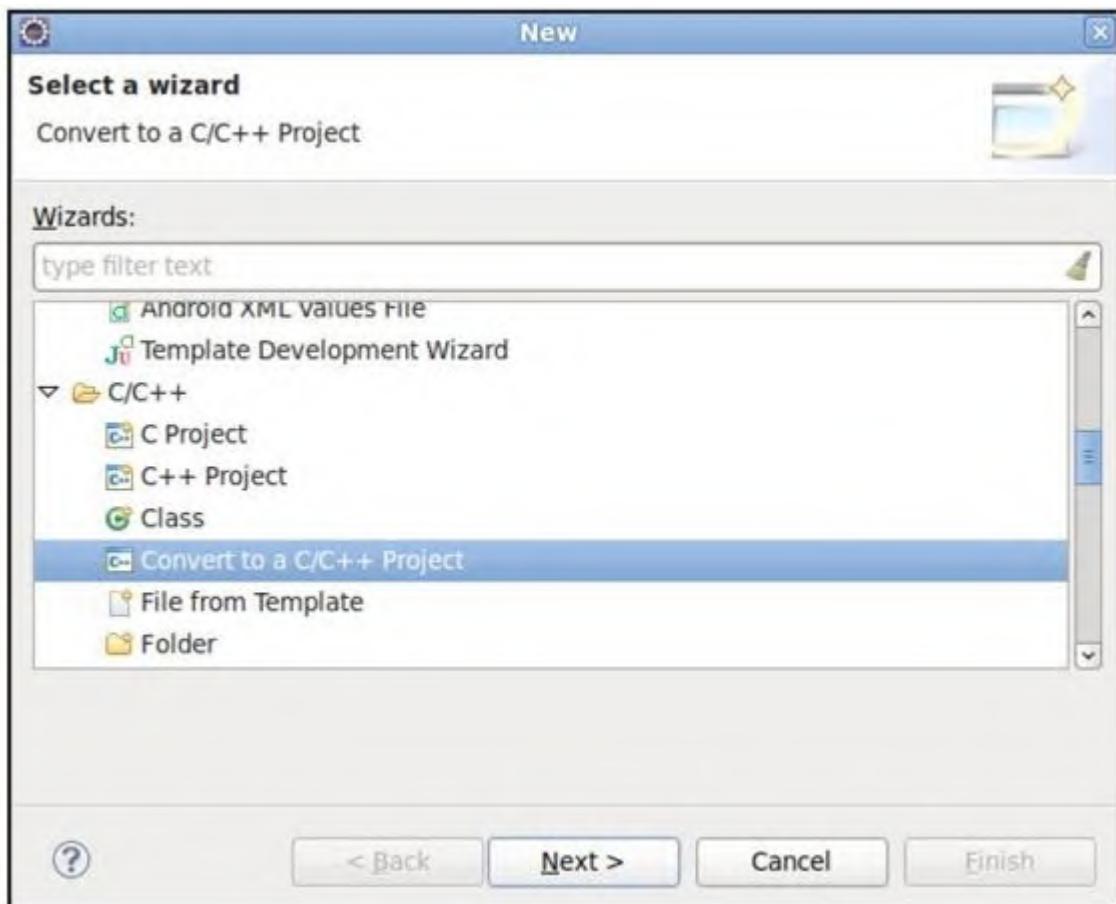


图1-5 转换成C++工程

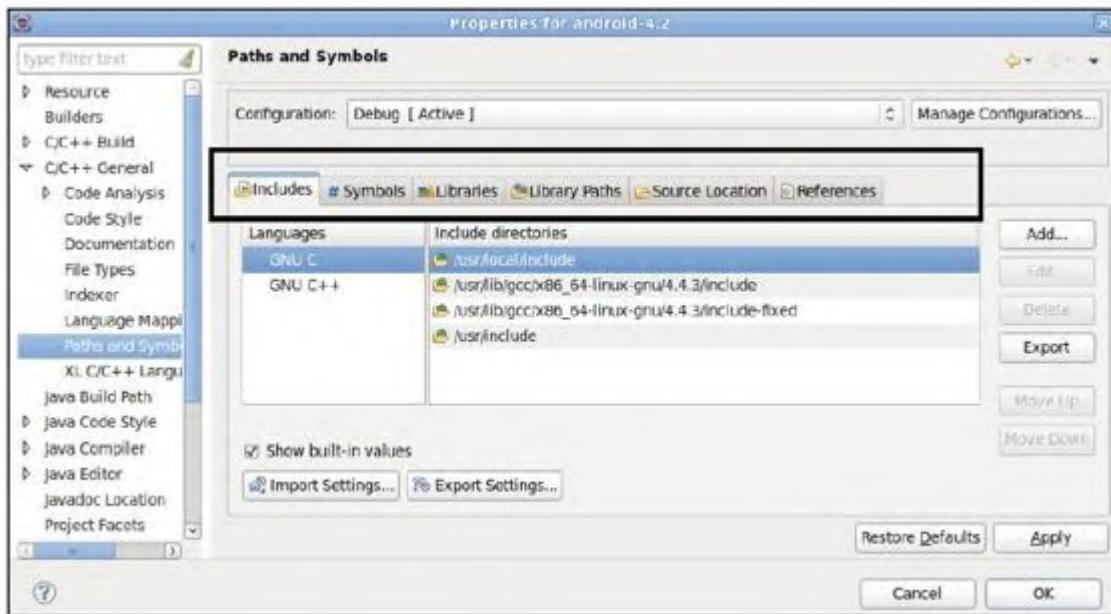


图1-6 路径和符号设置

配置好路径和符号文件后，可进一步通过图1-6中的Source Location页面选择此次需要导入的C++文件，如图1-7所示。

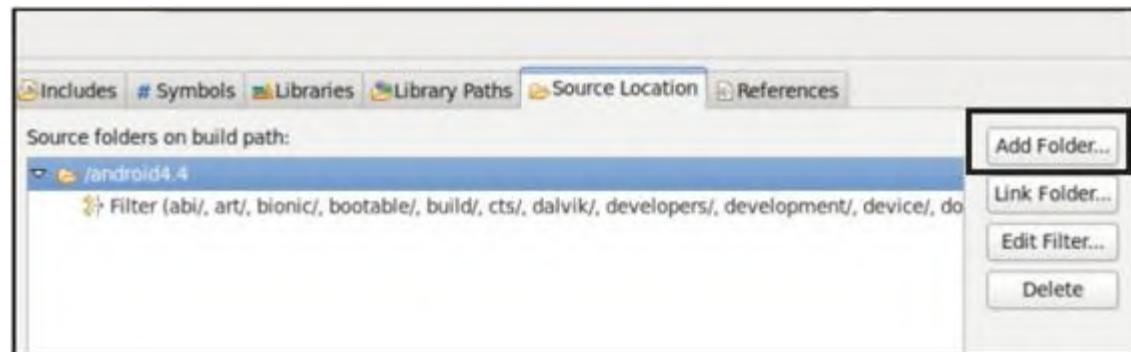


图1-7 过滤C++文件

通过图1-7中Add Folder选项，可以选择哪些目录下的C++文件被过滤掉。笔者目前仅导入了frameworks目录下的Native代码。

当所有文件都导入完毕后，读者通过右击C++工程，然后选择Index→Rebuild来重新生成Native代码的索引。

通过上述方法就能导入Android中的Native代码了。

### 3. 调试SystemServer

调试SystemServer的步骤如下。

- 1) 首先编译Android源码工程。编译过程中会有很多警告，如果有错误，大部分原因是.classpath文件将不需要的模块包含进来，可根据Eclipse的提示做相应处理。笔者配置的几台机器基本都是一次配置就成功了。
- 2) 在Android源码工程上单击右键，依次单击Debug As→Debug Configurations，弹出如图1-8所示的对话框，然后从左边找到Remote Java Application一栏。
- 3) 单击图1-8中黑框所示的新建按钮，按图1-9中的内容设置该对话框。

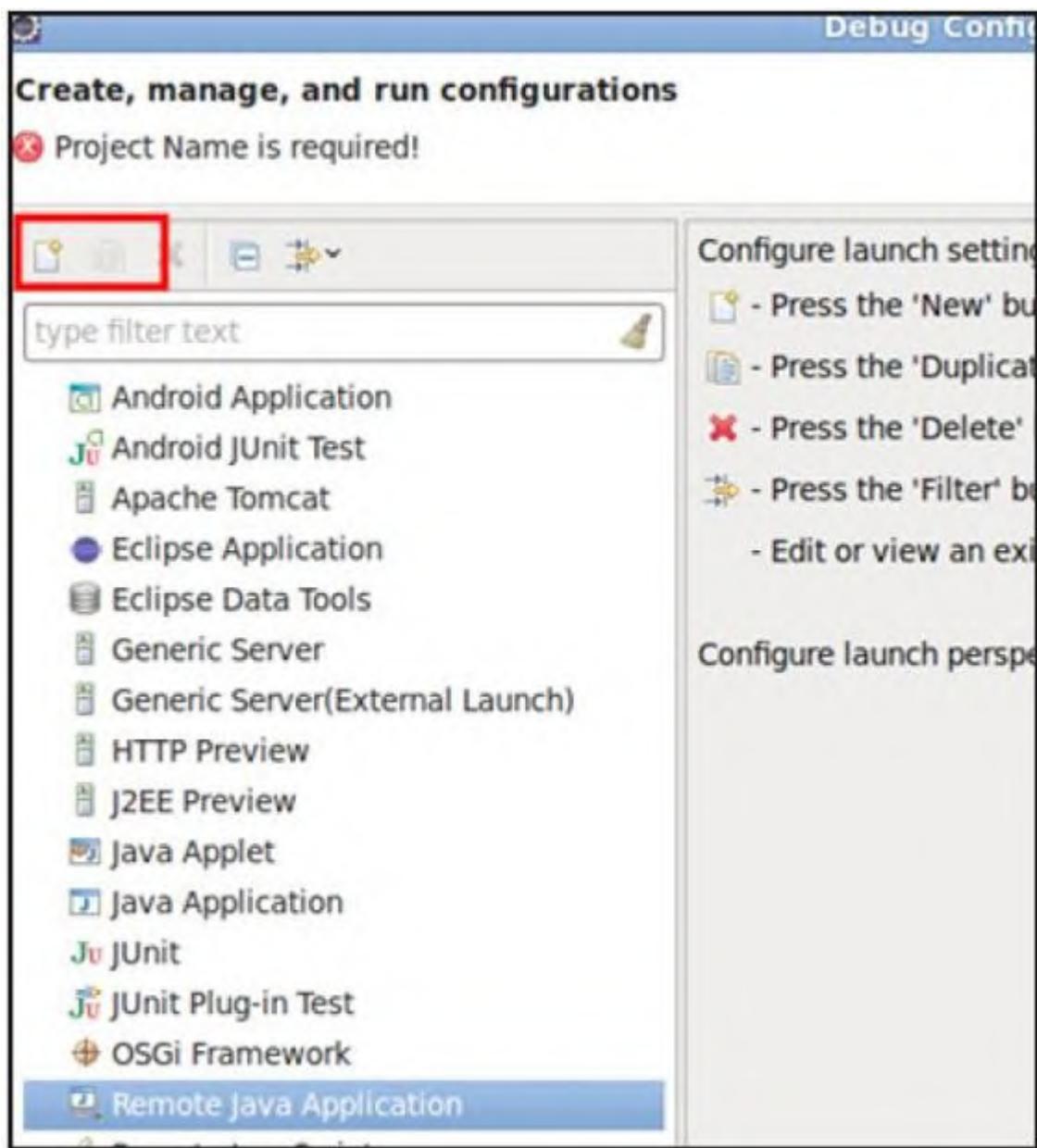


图1-8 Debug配置

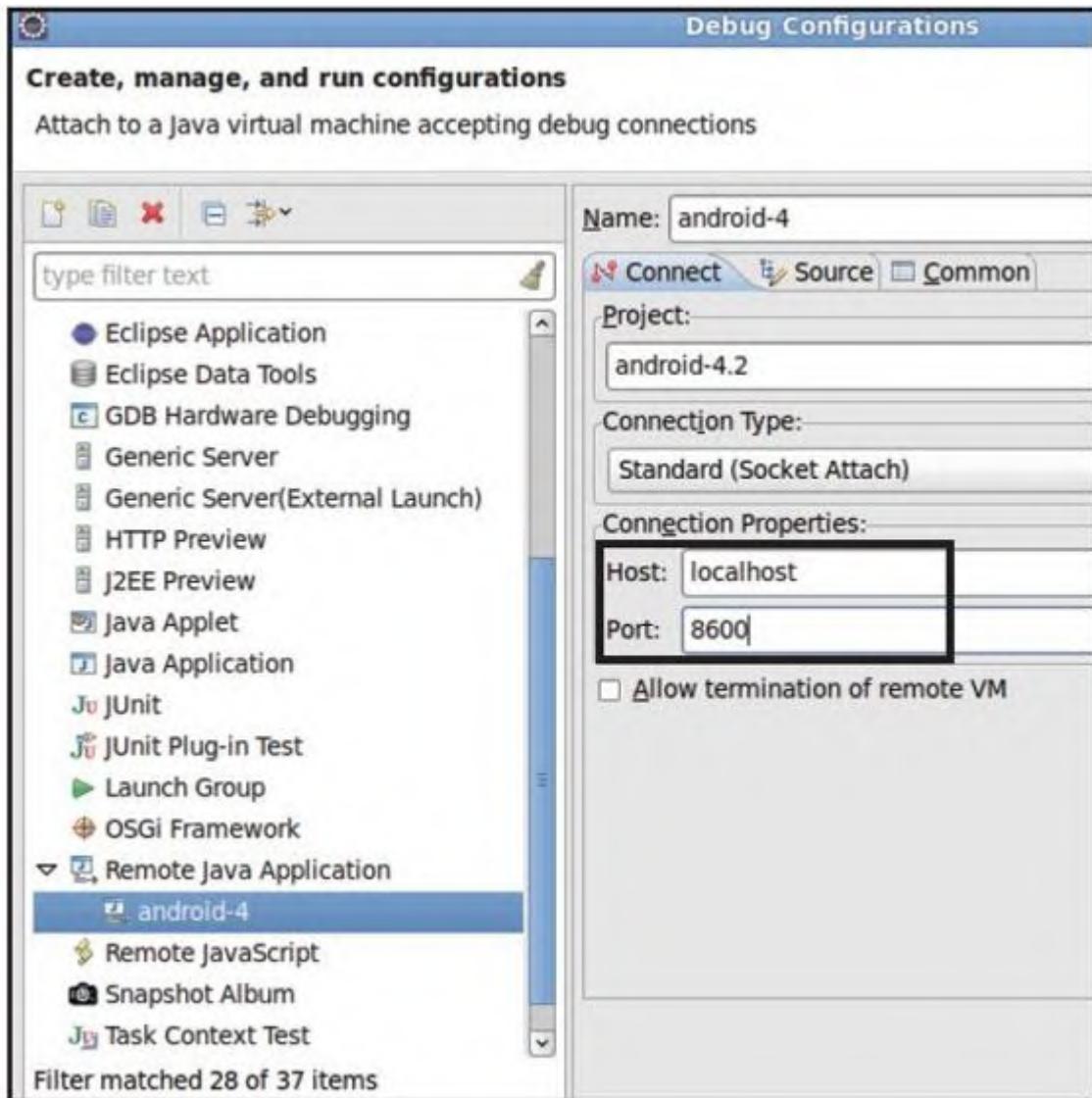


图1-9 Remote Java Application配置

如图1-9所示，需要选择Remote调试端口号为8600，Host类型为localhost。8600是SystemServer进程的调试端口号。Eclipse一旦连接到该端口，即可通过JDWP协议来调试SystemServer。

提示 读者可阅读《深入理解Android：卷II》第1章了解更多使用Eclipse的建议。

### 1.2.3 BusyBox的使用

BusyBox，号称Linux平台上的“瑞士军刀”，它提供了很多常用的工具，例如grep、find等。这些工具在标准Linux上都有，但Android系统却去掉了其中的大多数工具。这导致了我们在调试程序、研究Android系统时步履维艰，所以需要在手机上安装BusyBox。

#### 1. 下载BusyBox

可从网站<http://www.busybox.net/downloads/binaries/1.21.1/>下载已编译好的BusyBox，如图1-10所示。

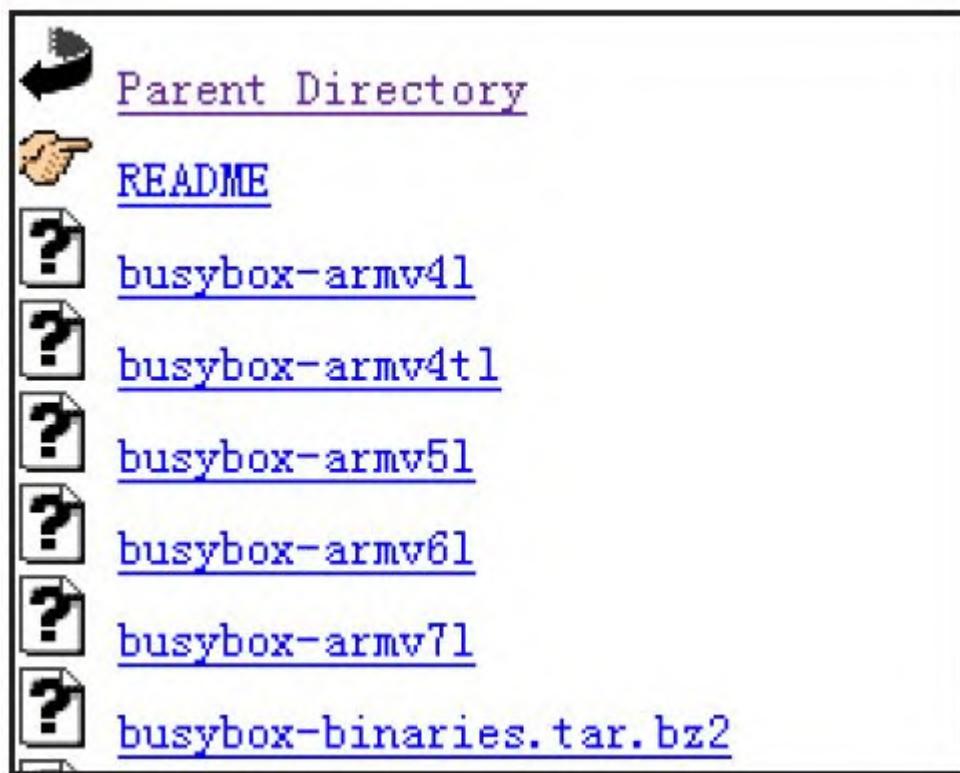


图1-10 BusyBox下载

注意，该网站已经根据不同平台编译好对应的BusyBox，可根据自己手机的情况下载对应的文件。笔者下载了支持Galaxy Note 2的busybox-armv7l。

提升： arm v7表示ARM指令集为v7， 目前ARM Cortex-A8/A9系列的CPU支持该指令集。

## 2. 安装和使用BusyBox

下载完BusyBox后，需使用adb push命令将它安装到手机上。如：

```
adb push busybox /system/xbin          #为了避免冲突，笔者push到了/system/xbin目录下
cd /system/xbin                         #进入对应目录
chmod 755 busybox                      #更改busybox权限为可执行
busybox --install .                     #安装busybox
grep                                #执行busybox提供的grep命令，或者busybox xxx执行xxx命令也行
```

BusyBox安装完，如果执行busybox命令，就会打印如图1-11所示的输出。

```
Currently defined functions:
[. [[, acpid, add-shell, addgroup, adduser, adjtimex, arp, arping, ash,
awk, base64, basename, beep, blkid, blockdev, bootchartd, brctl,
bunzip2, bzcat, bzip2, cal, cat, catv, chat, chattr, chgrp, chmod,
chown, chpasswd, chpst, chroot, chrt, chut, cksum, clear, cmp, comm,
cp, cpio, crond, crontab, cryptpw, cttyhack, cut, date, dc, dd,
deallocut, delgroup, deluser, depmod, devmem, df, dhcprelay, diff,
dirname, dmesg, dnsd, dnsdomainname, dos2unix, du, dumpkmap,
dumpleases, echo, ed, egrep, eject, env, envudir, envuidgid, ether-wake,
expand, expr, fakeidentd, false, fbset, fbsplash, fdflush, fdformat,
fdisk, fgconsole, fgrep, find, findfs, flock, fold, free, freeramdisk,
fsck, fsck.minix, fsync, ftpd, ftppget, ftpput, fuser, getopt, getty,
grep, gunzip, gzip, halt, hd, hdparm, head, hexdump, hostid, hostname,
httpd, hush, hwclock, id, ifconfig, ifdown, ifenslave, ifplugd, ifup,
inetd, init, insmod, install, ionice, iostat, ip, ipaddr, ipcalc,
ipcrm, ipcs, iplink, iproute, iprule, iptunnel, kbd_mode, kill,
killall, killall5, klogd, last, length, less, linux32, linux64,
linuxrc, ln, loadfont, loadkmap, logger, login, logname, logread,
losetup, lpd, lpq, lpr, ls, lsattr, lsmod, lspci, lsusb, lzcat, lzma,
lzop, lzopcat, makedeus, makemime, man, md5sum, mdev, mesg, microcom,
mkdir, mkdosfs, mke2fs, mkfifo, mkfs.ext2, mkfs.minix, mkfs.ufat,
mknod, mkpasswd, mkswap, mktemp, modinfo, modprobe, more, mount,
mountpoint, mpstat, mt, mu, nameif, nbd-client, nc, netstat, nice,
nmeter, nohup, nslookup, ntpd, od, openut, passwd, patch, pgrep, pidof,
ping, ping6, pipe_progress, pivot_root, pkill, pmmap, popmaildir,
poweroff, powertop, printenv, printf, ps, pscan, pwd, raidautorun,
```

图1-11 BusyBox提供的工具

从上图中可看出，BusyBox提供了不少的工具，这样，我们在研究Android系统时就如虎添翼了。

**提示** 本书共享资源中提供了busybox-armv7l的下载。

## 1.3 本书资源下载说明

为了减轻国内读者无法从Android官网下载源码的烦恼，笔者在115网盘上分享了本书所使用的Android源码及其他一些资源，如图1-12所示。

文件名称 / 文件夹名称	类型
4.2.2.tar.gz	文件 (2.5GB)
classpath	文件 (10.1KB)
busybox-armv7l	文件 (1.1MB)
com.cb.eclipse.folding_1.0.6.jar	文件 (101.0KB)
p2p_cap.pcapng	文件 (1.2MB)
wps_pbc.pcapng	文件 (1.3MB)
wpa_supplicant_analysis.pcapng	文件 (15.7MB)
wps_pin.pcapng	文件 (294.3KB)

图1-12 本书资源

图1-12中所示：

- 4.2.2.tar.gz为4.2.2源码压缩包。请读者特别注意，本书对wpa\_supplicant的分析使用的是Android 4.1源码中的wpa\_supplicant，故笔者在external目录中将4.1版本中的wpa\_supplicant代码复制到wpa\_supplicant\_8\_4.1中。
- classpath为Android Java配置文件，使用时请将它改名为".classpath"。
- busybox-armv7l是BusyBox。

- com.cb.eclipse.folding\_1.0.6.jar是一个Eclipse的插件，名叫Coffee Byte Java，它可以折叠代码段以方便阅读①。

最后四个文件为笔者研究Wi-Fi时，利用Wi-Fi数据截获工具AirPcap保存的相关协议数据包，其中p2p\_cap为测试P2P时保存的数据包，wps\_pbc和wps\_pin为测试WSC PBC和PIN时保存的数据包，wpa\_supplicant\_analysis为测试STA加入AP时所保存的数据包。没有AirPcap工具的读者可通过Wireshark工具直接导入这些数据以更直观的方式来分析Wi-Fi。该资源的下载地址是<http://115.com/lb/51bdugrdt4r>。另外，请读者务必关注笔者的博客blog.csdn.net/innost以获取更新信息。

① 详细用法请参考《深入理解Android：卷II》第1章。

# 读累了记得休息一会儿哦~

公众号：古德猫宁李

- 电子书搜索下载
- 书单分享
- 书友学习交流

网站：[沉金书屋 https://www.chenjin5.com](https://www.chenjin5.com)

- 电子书搜索下载
- 电子书打包资源分享
- 学习资源分享

# 第2章 深入理解Netd

本章所涉及的源代码文件名及位置

- main.cpp system/netd/main.cpp
- NetlinkManager.cpp system/netd/NetlinkManager.cpp
- NetlinkHandler.cpp system/netd/NetlinkHandler.cpp
- CommandListener.cpp system/netd/CommandListener.cpp
- DnsProxyListener.cpp system/netd/DnsProxyListener.cpp
- MDnsSdListener.cpp system/netd/MDnsSdListener.cpp
- getaddrinfo.c bionic/libc/netbsd/net/getaddrinfo.c
- dns\_sd.h external/mdiressponder/mDNSShared/dns\_sd.h
- ifc\_utils.c system/core/libnetutils/ifc\_utils.c
- ndc.c system/netd/ndc.c
- SecondaryTableController.cpp  
system/netd/SecondaryTableController.cpp
- InterfaceController.cpp  
system/netd/InterfaceController.cpp
- FirewallController.cpp  
system/netd/FirewallController.cpp
- logwrap.c system/netd/logwrap.c
- TetherController.cpp system/netd/TetherController.cpp

- SoftapController.cpp system/netd/SoftapController.cpp

- SystemServer.java

framework/base/services/java/com/android/server/SystemServer.java

- NetworkManagementService.java

framework/base/services/java/com/android/server/NetworkManagementService.java

## 2.1 概述

Netd是Android系统中专门负责网络管理和控制的后台daemon程序，其功能主要分三部分。

- 设置防火墙（Firewall）、网络地址转换（NAT）、带宽控制、无线网卡软接入点（Soft Access Point）控制，网络设备绑定（Tether）等。
- Android系统中DNS信息的缓存和管理。
- 网络服务搜索（Net Service Discovery，NSD）功能，包括服务注册（Service Registration）、服务搜索（Service Browse）和服务名解析（Service Resolve）等。

Netd的工作流程和Vold类似，其工作可分成两部分。

- Netd接收并处理来自Framework层中NetworkManagementService或NsdService的命令。这些命令最终由Netd中对应的Command对象去处理。
- Netd接收并解析来自Kernel的UEvent消息，然后再转发给Framework层中对应Service去处理。

由上述内容可知，Netd位于Framework层和Kernel层之间，它是Android系统中网络相关消息和命令转发及处理的中枢模块。Netd的代码量不大，难度较低，但其所涉及的相关背景知识却比较多。本章对Netd的分析将从以下几个方面入手。

- 首先介绍Netd的大体工作流程以及DNS、MDns相关的背景知识。关于Netd的工作流程分析，读者也可参考其他资料[①](#)。
- 本章集中介绍Netd中涉及的Android系统中网络管理和控制的相关工具。它们是iptables、tc和ip。

- 然后介绍Netd中CommandListener的命令处理。这些命令的正常工作依赖于前面介绍的iptables等工具。

- 最后，介绍Java Framework中的NetworkManagementService服务。

**提示** NsdService比较简单，感兴趣的读者不妨阅读作者的一篇博文“Android Says Bonjour”中的2.2节“NsdService介绍”。地址位于<http://blog.csdn.net/innost/article/details/8629139>。

① 可参考《深入理解Android：卷I》第9章关于Vold的分析。

## 2.2 Netd工作流程

Netd进程由init进程根据init.rc的对应配置项①而启动，其配置项如图2-1所示。

由图2-1可知，Netd启动时将创建三个TCP监听socket，其名称分别为netd、dnsproxyd和mdns。

```
service.netd./system/bin/netd
....class.main
....socket.netd.stream.0660.root.system
....socket.dnsproxyd.stream.0660.root.inet
....socket.mdns.stream.0660.root.system
```

图2-1 Netd启动配置参数

根据本章后续分析，读者将会看到以下内容。

- Framework层中的NetworkManagementService和Nsd-Service将分别和netd及mdns监听socket建立链接并交互。
- 每一个调用和域名解析相关的socket API（如getaddrinfo或gethostbyname等）的进程都会借由dnsproxyd监听socket与netd建立链接。

下面开始分析Netd进程。

① 关于init工作原理以及init.rc的分析方法，可参考《深入理解Android：卷I》第3章关于init进程的分析。

## 2.2.1 main函数分析

Netd进程的入口函数是其main函数，代码如下所示。

[-->main.cpp]

```
int main() {  
  
    CommandListener *cl;  
    NetlinkManager *nm;  
    DnsProxyListener *dpl;  
    MDnsSdListener *mdns1;  
  
    ALOGI("Netd 1.0 starting");  
  
    // 为Netd进程屏蔽SIGPIPE信号  
    blockSigpipe();  
  
    // ①创建NetlinkManager  
    nm = NetlinkManager::Instance();  
    // ②创建CommandListener，它将创建名为"netd"的监听socket  
    cl = new CommandListener();  
    // 设置NetlinkManager的消息发送者（Broadcaster）为  
    // CommandListener。  
    nm->setBroadcaster((SocketListener *) cl);  
    // 启动NetlinkManager  
    nm->start();  
    ....  
    // 注意下面这行代码，它为本Netd设置环境变量ANDROID_DNS_MODE  
    // 为"local"，其作用将在2.2.4节介绍  
    setenv("ANDROID_DNS_MODE", "local", 1);  
    // ③创建DnsProxyListener，它将创建名为"dnsproxyd"的监听socket  
    dpl = new DnsProxyListener();  
    dpl->startListener();  
  
    // ④创建MDnsSdListener并启动监听，它将创建名为"mdns"的监听socket  
    mdns1 = new MDnsSdListener();  
    mdns1->startListener();  
  
    cl->startListener();  
  
    while(1) {  
        sleep(1000);  
    }  
}
```

```
    }
    exit(0);
}
```

Netd的main函数非常简单，主要是创建几个重要成员并启动相应的工作，这四个重要成员分别如下。

- **NetlinkManager**: 接收并处理来自Kernel的UEvent消息。这些消息经NetlinkManager解析后将借助它的Broadcaster（也就是代码中为NetlinkManager设置的CommandListener）发送给Framework层的NetworkManagementService。
- **CommandListener、DnsProxyListener、MDnsSdListener**: 分别创建名为"netd"、"dnsproxyd"、"mdns"的监听socket，并处理来客户端的命令。

下面将分别讨论这四位成员的作用。

## 2.2.2 NetlinkManager分析

NetlinkManager（以后简称NM）主要负责接收并解析来自Kernel的UEvent消息。其核心代码在start函数中，如下所示。

[-->NetlinkManager.cpp: : start]

```
int NetlinkManager::start() {
    // 创建接收NETLINK_KOBJECT_UEVENT消息的socket，其值保存在
    mUeventSock中
    // 其中，NETLINK_FORMAT_ASCII代表UEvent消息的内容为ASCII字符串
    mUeventHandler = setupSocket(&mUeventSock,
        NETLINK_KOBJECT_UEVENT,
        0xffffffff, NetlinkListener::NETLINK_FORMAT_ASCII);
    // 创建接收RTMGPR_LINK消息的socket，其值保存在mRouteSock中
    // 其中，NETLINK_FORMAT_BINARY代表UEvent消息的类型为结构体，故需要
    // 进行二进制解析
    mRouteHandler = setupSocket(&mRouteSock, NETLINK_ROUTE,
        RTMGRP_LINK,
        NetlinkListener::NETLINK_FORMAT_BINARY);
    // 创建接收NETLINK_NFLOG消息的socket，其值保存在mQuotaSock中
    mQuotaHandler = setupSocket(&mQuotaSock, NETLINK_NFLOG,
        NFLOG_QUOTA_GROUP,
        NetlinkListener::NETLINK_FORMAT_BINARY);
    return 0;
}
```

NM的start函数主要是向Kernel注册三个用于接收UEvent事件的socket，这三个UEvent<sup>[1][2]</sup> 分别对应于以下内容。

- NETLINK\_KOBJECT\_UEVENT：代表kobject事件，由于这些事件包含的信息由ASCII字符串表达，故上述代码中使用了NETLINK\_FORMAT\_ASCII。它表示将采用字符串解析的方法去解析接收到的UEvent消息。kobject一般用来通知内核中某个模块的加载或卸载。对于NM来说，其关注的是/sys/class/net下相应模块的加载或卸载消息。
- NETLINK\_ROUTE：代表Kernel中routing或link改变时对应的消息。NETLINK\_ROUTE包含很多子项，上述代码中使用了RTMGRP\_LINK项。二者结合起来使用，表示NM希望收到网络链路断开或接通时对应的

UEvent消息（笔者在Ubuntu PC上测试过，当网卡上拔掉或插入网线时，会触发这些UEvent消息的发送）。由于对应UEvent消息内部封装了nlmsghdr等相关结构体，故上述代码使用了NETLINK\_FORMAT\_BINARY来指示解析UEvent消息时将使用二进制的解析方法。

- NETLINK\_NFLOG：和2.3.6节介绍的带宽控制有关。Netd中的带宽控制可以设置一个预警值，当网络数据超过一定字节数就会触发Kernel发送一个警告。该功能属于iptables的扩展项，但由于iptables的文档更新速度较慢（这也是很多开源项目的一大弊端），笔者一直未能找到相关的正式说明。值得指出的是，上述代码中有关NETLINK\_NFLOG相关socket的设置并非所有Kernel版本都支持。同时，NFLOG\_QUOTA\_GROUP的值是直接定义在NetlinkManager.cpp中的，而非和其他类似系统定义一样定义在系统头文件中，这也表明NFLOG\_QUOTA\_GROUP的功能比较新。

**提示** 读者可通过在Linux终端中执行man PF\_LINK得到有关NETLINK的详细说明。

上述start函数将调用setupSocket创建用于接收UEvent消息的socket以及一个解析对象NetlinkHandler。setupSocket代码本身比较简单，此处就不展开分析。

下面来看NM及其家族成员，它们之间的关系如图2-2所示。

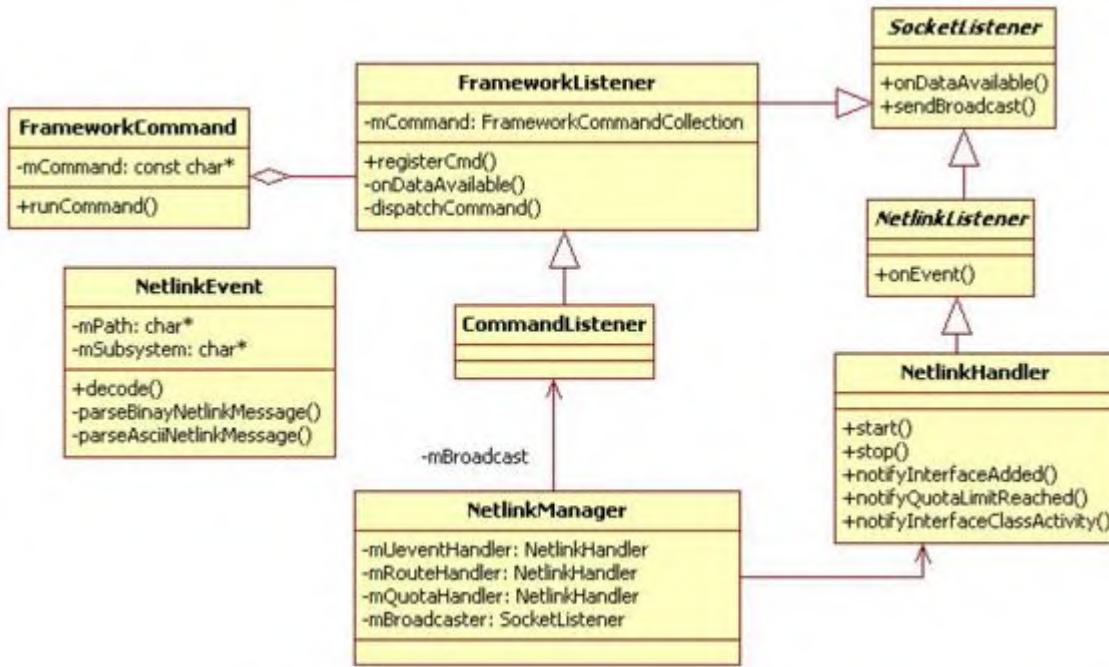


图2-2 NM家族成员

由图2-2可知：

- NetlinkHandler和CommandListener均间接从SocketListener派生。其中，NetlinkHandler收到的socket消息将通过onEvent回调处理。
- 结合前文所述，NetlinkManager分别注册了三个用于接收UEvent的socket，其对应的NetlinkHandler分别是mUeventHandler、mRouteHandler和mQuotaHandler。
- NetlinkHandler接收到的UEvent消息会转换成一个NetlinkEvent对象。NetlinkEvent对象封装了对UEvent消息的解析方法。对于NETLINK\_FORMAT\_ASCII类型，其parseAsciiNetlinkMessage函数会被调用，而对于NETLINK\_FORMAT\_BINARY类型，其parseBinaryNetlinkMessage函数会被调用。
- NM处理流程的输入为一个解析后的NetlinkEvent对象。NM完成相应工作后，其处理结果将经由mBroadcaster对象传递给Framework层的接收者，也就是NetworkManagementService。

- CommandListener从FrameworkListener派生，而FrameworkListener内部有一个数组mCommands，用来存储注册到FrameworkListener中的命令处理对象。

下面简单了解NetlinkHandler的onEvent函数，由于其内部已针对不同属性的NetlinkEvent进行了分类处理，故浏览这段代码能对前文所述不同UEvent消息的作用加深理解。

[-->NetlinkHandler.cpp: : onEvent]

```

void NetlinkHandler::onEvent(NetlinkEvent *evt) {
    const char *subsys = evt->getSubsystem();
    .....
    // 处理对应NETLINK_KOBJECT_UEVENT和NETLINK_ROUTE的信息
    if (!strcmp(subsys, "net")) {
        int action = evt->getAction();
        const char *iface = evt->findParam("INTERFACE");
        // 查找消息中携带的网络设备名
        if (action == evt->NlActionAdd) {
            notifyInterfaceAdded(iface); // 添加NIC (Network
Interface Card) 的消息
        } else if (action == evt->NlActionRemove) {
            notifyInterfaceRemoved(iface); // NIC
被移除的消息
        } else if (action == evt->NlActionChange) {
            evt->dump();
            notifyInterfaceChanged("nana", true); // NIC
变化消息
        } else if (action == evt->NlActionLinkUp) { // 下面两个消息来自NETLINK_ROUTE
            notifyInterfaceLinkChanged(iface, true); // 链路
启用 (类似插网线)
        } else if (action == evt->NlActionLinkDown) {
            notifyInterfaceLinkChanged(iface, false); // 链路
断开 (类似拔网线)
        }
    } else if (!strcmp(subsys, "qlog")) { // 对应
NETLINK_NFLOG
        const char *alertName = evt->findParam("ALERT_NAME");
        const char *iface = evt->findParam("INTERFACE");
        notifyQuotaLimitReached(alertName, iface); // 当数据量超过
预警值，则会收到该通知
    } else if (!strcmp(subsys, "xt_idletimer")) {

```

```
// 这和后文的idletimer有关，用于跟踪某个NIC的工作状态，  
即"idle"或"active"  
    // 检测时间按秒计算  
    int action = evt->getAction();  
    const char *label = evt->findParam("LABEL");  
    const char *state = evt->findParam("STATE");  
    if (label == NULL) {  
        label = evt->findParam("INTERFACE");  
    }  
    if (state)  
        notifyInterfaceClassActivity(label,  
!strcmp("active", state));  
    }  
    ....  
}
```

由上边代码可知，NETLINK\_KOBJECT\_UEVENT和NETLINK\_ROUTE主要反映网络设备的事件和状态，包括NIC的添加、删除和修改，以及链路的连接状态等。NETLINK\_NFLOG用于反映设置的log是否超过配额。另外，上边代码中还处理了xt\_idletimer的uevent消息，它和后文介绍的IdleTimerCmd有关，主要用来监视网络设备的收发工作状态。当对应设备工作或空闲时间超过设置的监控时间后，Kernel将会发送携带其状态（idle或active）的UEvent消息。

图2-3所示为NetlinkHandler的工作流程。

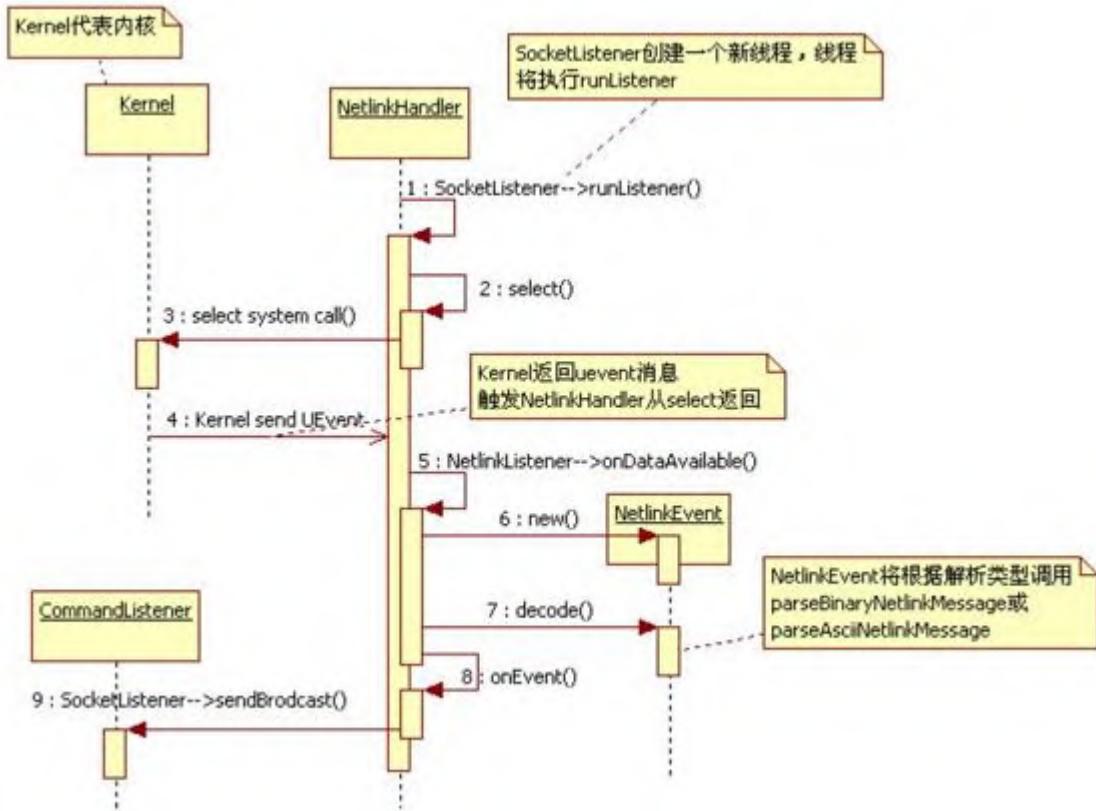


图2-3 NM工作流程

由图2-3可知，NM创建NetlinkHandler后，工作便转交给NetlinkHandler来完成，而每个NetlinkHandler对象均会单独创建一个线程用于接收socket消息。当Kernel发送UEvent消息后，NetlinkHandler便从select调用中返回，然后调用其onDataAvailable函数，该函数内部会创建一个NetlinkEvent对象。NetlinkEvent对象根据socket创建时指定的解析类型去解析来自Kernel的UEvent消息。最终NetlinkHandler的onEvent将被调用，不同的UEvent消息将在此函数中进行分类处理。NetlinkHandler最终将处理结果经由NM内部变量mBroadcaster转发给NetworkManagementService。

提醒 请读者结合上文所述流程自行研读相关代码。

### 2.2.3 CommandListener分析

Netd中第二个重要成员是CommandListener（以后简称CL），其主要作用是接收来自Framework层NetworkManageService的命令。从角色来看，CL仅是一个Listener。它在收到命令后，只是将它们转交给对应的命令处理对象去处理。CL内部定义了许多命令，而这些命令都有较深的背景知识。本节以分析CL的工作流程为主，而相关的命令处理则放到后文分析。

图2-4所示为CL中的Command对象及对应的Controller对象。

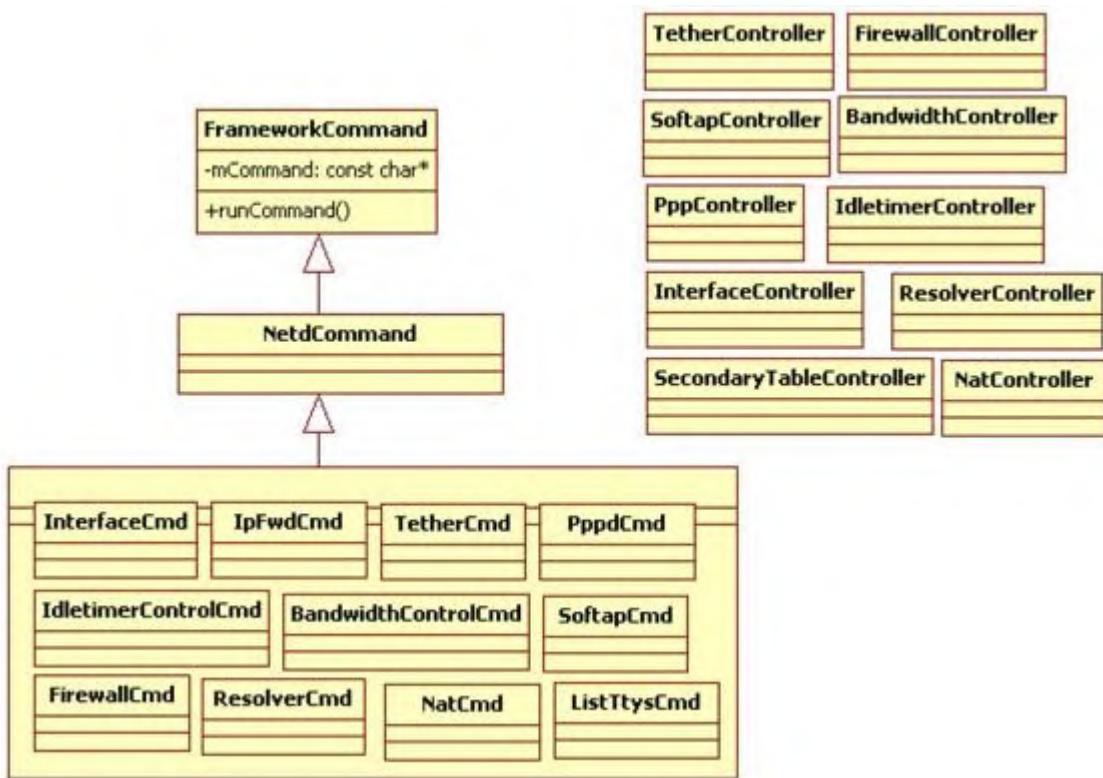


图2-4 CL中的命令及控制类

由图2-4可知，CL定义了11个和网络相关的Command类。这些类均从NetdCommand派生（注意，为保持绘图简洁，这11个Command的派生关系由1个派生箭头表达）。CL还定义了10个控制类，这些控制类将和命令类共同完成相应的命令处理工作。

结合图2-2中对NM家族成员的介绍，CL创建时，需要注册自己支持的命令类。这部分代码在其构造函数中实现，代码如下所示。

[-->CommandListener: : CommandListener构造函数]

```
CommandListener::CommandListener() :  
    FrameworkListener("netd", true) {  
    registerCmd(new InterfaceCmd()); // 注册11个命令类对象  
    registerCmd(new IpFwdCmd());  
    registerCmd(new TetherCmd());  
    registerCmd(new NatCmd());  
    registerCmd(new ListTtysCmd());  
    registerCmd(new PppdCmd());  
    registerCmd(new SoftapCmd());  
    registerCmd(new BandwidthControlCmd());  
    registerCmd(new IdletimerControlCmd());  
    registerCmd(new ResolverCmd());  
    registerCmd(new FirewallCmd());  
    // 创建对应的控制类对象  
    if (!sSecondaryTableCtrl)  
        sSecondaryTableCtrl = new SecondaryTableController();  
    if (!sTetherCtrl)  
        sTetherCtrl = new TetherController();  
    if (!sNatCtrl)  
        sNatCtrl = new NatController(sSecondaryTableCtrl);  
    if (!sPppCtrl)  
        sPppCtrl = new PppController();  
    if (!sSoftapCtrl)  
        sSoftapCtrl = new SoftapController();  
    if (!sBandwidthCtrl)  
        sBandwidthCtrl = new BandwidthController();  
    if (!sIdletimerCtrl)  
        sIdletimerCtrl = new IdletimerController();  
    if (!sResolverCtrl)  
        sResolverCtrl = new ResolverController();  
    if (!sFirewallCtrl)  
        sFirewallCtrl = new FirewallController();  
    if (!sInterfaceCtrl)  
        sInterfaceCtrl = new InterfaceController();  
    // 其他重要工作，后文再分析  
}
```

由于CL的间接基类也是SocketListener，所以其工作流程和NetlinkHandler类似。图2-5给出了CL的工作流程。

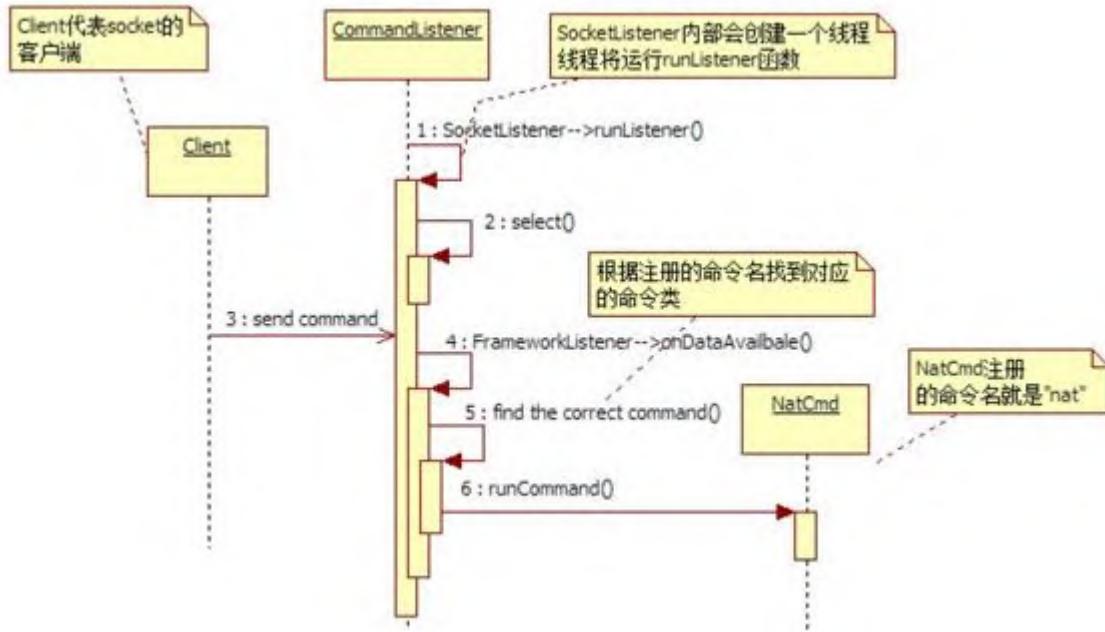


图2-5 CL的工作流程

图2-5中，假设Client端发送的命令名是“nat”，当CL收到这个命令后，首先会从其构造函数中注册的那些命令对象中找到对应该名字（即nat）的命令对象，其结果就是图中的NatCmd对象。而该命令最终的处理工作将由此NatCmd对象的runCommand函数完成。

## 2.2.4 DnsProxyListener分析

DnsProxyListener和Android系统中的DNS管理有关。什么是DNS呢？Android系统中DNS又有什么特点呢？来看下文。

### 1. Android DNS简介<sup>[3]</sup>

DNS (Domain Name System, 域名系统) 主要作用是在域名和IP地址之间建立一种映射。简单来说，DNS的功能类似于电话簿，它可将人名映射到相应的电话号码。在DNS中，人名就是域名，电话号码就是IP地址。域名系统的管理由DNS服务器来完成。全球范围内的DNS服务器共同构成了一个分布式的域名-IP数据库。

对使用域名来发起网络操作的网络程序来说，其域名解析工作主要分两步。

- 1) 将域名转换成IP。由于域名和IP的转换关系存储在DNS服务器上，所以该网络程序要向DNS服务器发起请求，以获取域名对应的IP地址。
- 2) DNS服务器根据DNS解析规则解析并得到该域名对应的IP地址，然后返回给客户端。在DNS中，每一个域名和IP的对应关系称为一条记录。客户端一般会缓存这条记录以备后续之用。

提醒 DNS解析规则比较复杂，感兴趣的读者可研究DNS的相关协议。

对软件开发者来说，常用的域名解析socket API有两个。

- `getaddrinfo` : 根据指定的host名或service名得到对应的IP地址（由结构体addrinfo表达）。
- `getnameinfo` : 根据指定的IP地址（由结构体sockaddr表达）得到对应的host或service的名称。

Android中，这两个函数均由Bionic C实现。其代码实现基于NetBSD的解析库（resolver library），并经过一些修改。这些修改如下。

- 没有实现name-server-switch功能。这是为了保持Bionic C库的轻便性而做的裁剪。
- DNS服务器的配置文件由/etc/resolv.conf变成/system/etc/resolv.conf①。在Android系统中，/etc目录实际上为/system/etc目录的链接。resolv.conf存储的是DNS服务器的IP地址。
- 系统属性中保存了一些DNS服务器的地址，它们通过诸如"net.dns1"或"net.dns2"之类的属性来表达。这些属性由dhcpd进程或其他系统模块负责维护。
- 每个进程还可以设置进程特定的DNS服务器地址，它们通过诸如"net.dns1.<pid>"或"net.dns2.<pid>"的系统属性来表达。
- 不同的网络设备也有对应的DNS服务器地址，例如通过wlan接口发起的网络操作，其对应的DNS服务器由系统属性"net.wlan.dns1"表示。

图2-6所示为三星Galaxy Note 2中有关dns的信息。由图可知，系统中有些进程有自己特定的DNS服务器。不同网络设备也设置了对应的DNS服务器地址。

```
[net.dns1]: [219.239.26.42]
[net.dns2.1867]: []
[net.dns2.29454]: []
[net.dns2.30890]: []
[net.dns2.31631]: []
[net.dns2.31813]: []
[net.dns2.32230]: []
[net.dns2.32520]: []
[net.dns2]: [124.207.160.106]
[net.dnschange]: [29]
[net.rmnet0.dns1]: []
[net.rmnet0.dns2]: []
[net.rmnet1.dns1]: []
[net.rmnet1.dns2]: []
[net.rmnet2.dns1]: []
[net.rmnet2.dns2]: []
[net.wlan0.dns1]: [50.0.0.0]
[net.wlan0.dns2]: [49.0.0.0]
```

图2-6 net.dns设置

## 2. getaddrinfo函数分析

本节介绍Android中getaddrinfo的实现，我们将只关注Android对其做的改动。

[-->getaddrinfo.c: : getaddrinfo]

```
int getaddrinfo(const char *hostname, const char *servname,
    const struct addrinfo *hints, struct addrinfo **res)
{
    .....// getaddrinfo的正常处理
    // Android平台的特殊定制
    if (android_getaddrinfo_proxy(hostname, servname, hints,
        res) == 0) {
```

```

        return 0;
    }
    .....// 如果上述函数处理失败，则继续getaddrinfo的正常处理
    return error
}

```

由上述代码可知，Android平台中的getaddrinfo会调用其定制的 android\_getaddrinfo\_proxy 函数完成一些特殊操作，该函数的实现如下所示。

```

[-->getaddrinfo.c: : android_getaddrinfo_proxy]

static int android_getaddrinfo_proxy(const char *hostname,
const char *servname,
          const struct addrinfo *hints, struct addrinfo
**res)
{
    .....
    // 取ANDROID_DNS_MODE环境变量。只有Netd进程设置了它
    const char* cache_mode = getenv("ANDROID_DNS_MODE");
    .....
    // 由于Netd进程设置了此环境变量，故Netd进程调用getaddrinfo将不会采用这套定制的方法
    if (cache_mode != NULL && strcmp(cache_mode, "local") == 0)
    {
        return -1;
    }
    // 获取本进程对应的DNS地址
    snprintf(propname, sizeof(propname), "net.dns1.%d",
getpid());
    if (__system_property_get(propname, propvalue) > 0) {
        return -1;
    }

    // 建立和Netd中DnsProxyListener的连接，将请求转发给它去执行
    sock = socket(AF_UNIX, SOCK_STREAM, 0);
    if (sock < 0) {
        return -1;
    }
    .....
    strlcpy(proxy_addr.sun_path, "/dev/socket/dnsproxyd",
           sizeof(proxy_addr.sun_path));
    .....// 发送请求，处理回复等
    return -1;
}

```

由上述代码可知：

- 当Netd进程调用getaddrinfo时，由于其设置了ANDROID\_DNS\_MODE环境变量，所以该函数会继续原来的流程。
- 当非Netd进程调用getaddrinfo函数时，首先会开展 android\_getaddrinfo\_proxy中的工作，即判断该进程是否有定制的 DNS服务器，如果没有它将和位于Netd进程中的dnsproxoyd监听socket 建立连接，然后把请求发给DnsProxyListener去执行。

### 3. DnsProxyListener命令

下面介绍DnsProxyListener（以后简称DPL），图2-7所示为其家族成员示意图。

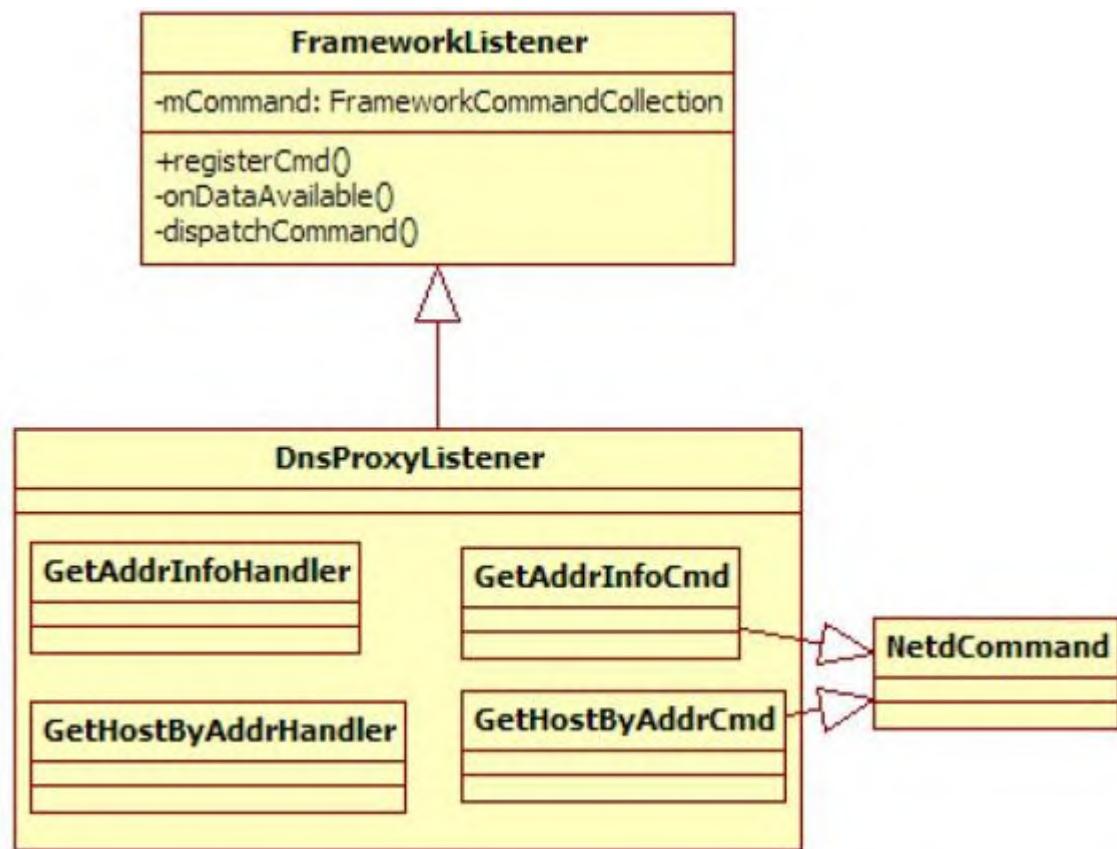


图2-7 DPL家族成员

由图2-7可知，DPL仅定义了两个命令。

- GetAddrInfoCmd : 和Bionic C库的getaddrinfo函数对应。
- GetHostByAddrCmd : 和Bionic C库的gethostbyaddr函数对应。

这个两条命令的处理比较简单，此处不展开详细的代码。为方便读者理解，我们将给出调用序列图，如图2-8所示。

如图2-8所示，GetAddrInfoHandler最终的处理还是交由Bionic C的getaddrinfo函数来完成。根据前文所述，由于Netd进程设置了ANDROID\_DNS\_MODE环境变量，故Netd调用的getaddrinfo将走正常的流程。这个正常流程就是Netd进程将向指定的DNS服务器发起请求以解析域名。

Android系统中，通过这种方式来管理DNS的好处是，所有解析后得到的DNS记录都将缓存在Netd进程中，从而使这些信息成为一个公共的资源，最大程度做到信息共享。

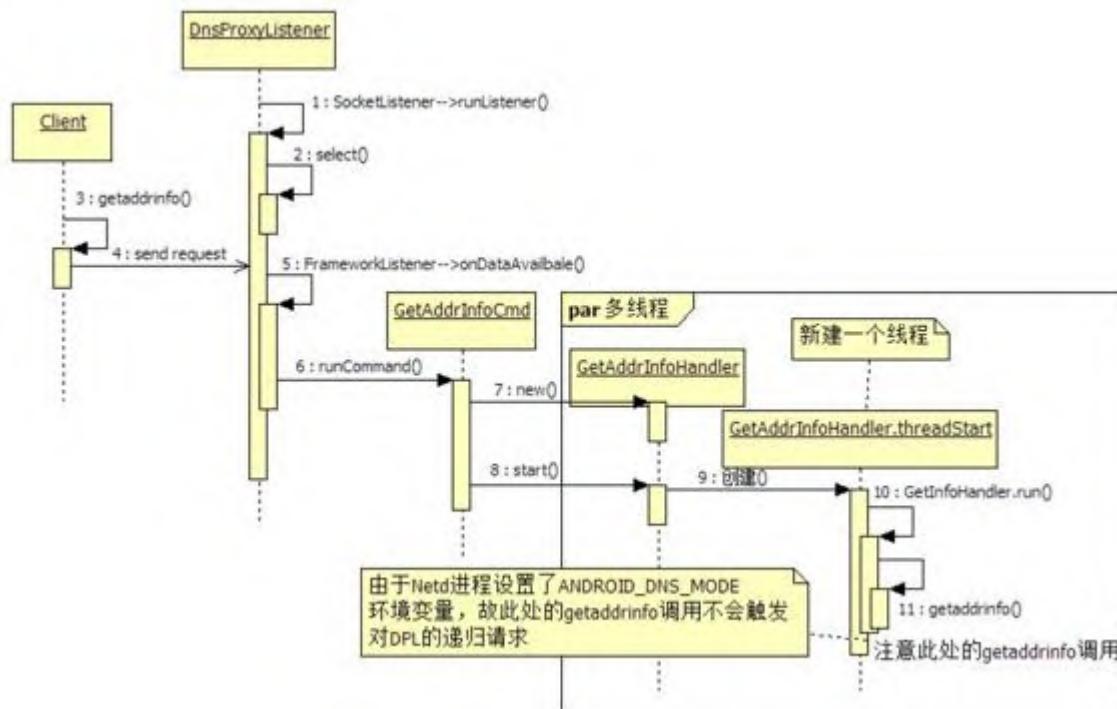


图2-8 GetAddrInfoCmd处理流程

① 此处结论来自bionic/libc/docs/OVERVIEW.txt文件，不过根据同目录下CHANGES.txt的说明，resolv.conf将不再使用。

## 2.2.5 MDnsSdListener分析

MDnsSd是Multicast DNS Service Discovery的简称，它和Apple公司的Bonjour技术有关，故本节将先介绍Apple Bonjour技术。

### 1. Apple Bonjour技术<sup>[4][5][6]</sup>

Bonjour是法语中的Hello之意。它是Apple公司为基于组播域名服务（multicast DNS）的开放性零配置网络标准所起的名字。使用Bonjour的设备在网络中自动组播它们自己的服务信息并监听其他设备的服务信息，设备之间就像在打招呼，这也是该技术命名为Bonjour的原因。Bonjour使得局域网中的系统和服务即使在没有网络管理员的情况下也很容易被找到。

举一个简单的例子，在局域网中，如果要进行打印服务，就必须先知道打印服务器的IP地址。此IP地址一般由IT部门的人负责分配，然后还得全员发邮件以公示此地址。有了Bonjour以后，打印服务器自己会依据零配置网络标准在局域网内部找到一个可用的IP并注册一个打印服务，例如“print service”。当客户端需要打印服务时，会先搜索网络内部的打印服务器。由于不知道打印服务器的IP地址，客户端只能根据诸如“print service”的名字去查找打印机。在Bonjour的帮助下，客户端最终能找到这台注册了“print service”名字的打印机，并获得它的IP地址以及端口号。

从Bonjour角度来看，该技术主要解决了三个问题。

- Addressing：即为主机分配IP。Bonjour的Addressing处理比较简单，即每个主机在网络内部的地址可选范围内找一个IP，然后查看下网络内部是否有其他主机再用。如果该IP没有被分配的话，它将使用此IP。
- Naming：Naming解决的是host和IP地址的对应关系。Bonjour采用的是Multiple DNS技术，即DNS查询消息将通过UDP组播方式发送。一旦网络内部某个机器发现查询的机器名和自己设置的一样，就回复这条请求。此外，Bonjour还拓展了MDNS的用途，即除了能查找host外，

还支持对service的查找。不过，Bonjour的Naming有一个限制，即网络内部不能有重名的host或service。

- Service Discovery : SD基于Naming工作，它使得应用程序能查找到网络内部的服务，并解析该服务对应的IP地址和端口号。应用程序一旦得到服务的IP地址和端口号，就可以直接和该服务建立交互关系。

Bonjour技术在Mac OS以及iTunes、iPhone上都得到了广泛应用。为了进一步推广，Apple通过开源工程mdnsresponder将其发布。在Windows平台上，它将生成一个后台程序mdnsresponder。在Android平台上（或者说支持POSIX的Linux平台）它是一个名为mdnsd的程序。不过，不论是mdnsresponder还是mdnsd，应用开发者要做的仅仅是利用Bonjour的API向它们发起服务注册、服务查询和服务解析等请求并接收来自它们的处理结果。

下面介绍Bonjour API中使用最多的三个函数，它们分别用来服务注册、服务查询和服务解析。理解这三个函数的功能也是理解MDnsSdListener的基础。

使用Bonjour API必须包含如下的头文件和动态库，并连接到：

```
#include <dns_sd.h> // 必须包含此头文件  
libmdnssd.so // 链接到此so
```

Bonjour中，服务注册的API为DNSServiceRegister，原型如下。

```
DNSServiceErrorType DNSSD_API DNSServiceRegister  
(  
    DNSServiceRef *sdRef,  
    DNSServiceFlags flags,  
    uint32_t interfaceIndex,  
    const char *name, /* may be  
    NULL */  
    const char *regtype,  
    const char *domain, /* may be  
    NULL */  
    const char *host, /* may be  
    NULL */  
    uint16_t port, /* In network  
    byte order */  
    uint16_t txtLen,
```

```

    const void           *txtRecord,      /* may be
NULL */          callBack,        /* may be
NULL */          *context       /* may be
void              */
NULL */          );

```

该函数的解释如下。

- sdRef代表一个未初始化的DNSService实体，其类型DNSServiceRef是指针。该参数最终由DNSServiceRegister函数分配内存并初始化。
- flags表示当网络内部有重名服务时的冲突处理。默认是按顺序修改服务名。例如要注册的服务名为“printer”，当检测到重名冲突时，就可改名为“printer (1) ”。
- interfaceIndex表示该服务输出到主机的哪些网络接口上。值-1表示仅对本机支持，也就是该服务的用在loop接口上。
- name表示服务名，如果为空就取机器名。
- regtype表示服务类型，用字符串表达。Bonjour要求格式为“\_服务名.\_传输协议”，例如“\_ftp.\_tcp”。目前传输协议仅支持TCP和UDP。
- domian和host一般都为空。
- port表示该服务的端口。如果为0，Bonjour会自动分配一个。
- txtLen以及txtRecord字符串用来描述该服务。一般都设置为空。
- callBack表示设置回调函数。该服务注册的请求结果都会通过它回调给客户端。
- context表示上下文指针，由应用程序设置。

当客户端需要搜索网络内部特定服务时，需要使用DNSServiceBrowser API，其原型如下。

```
DNSServiceErrorType DNSSD_API DNSServiceBrowse
(
```

```

DNSServiceRef           *sdRef,
DNSServiceFlags         flags,
uint32_t                interfaceIndex,
const char               *regtype,
const char               *domain,          /* may be NULL
*/
DNSServiceBrowseReply    callBack,
void                     *context        /* may be NULL */
);

```

其中，sdref、interfaceIndex、regtype、domain以及context含义与DNSServiceRegister一样。flags在本函数中没有作用。callBack为DNSServiceBrowser处理结果的回调通知接口。

当客户端想获得指定服务的IP和端口号时，需要使用DNSServiceResolve API，其原型如下。

```

DNSServiceErrorType DNSSD_API DNSServiceResolve
(
    DNSServiceRef           *sdRef,
    DNSServiceFlags         flags,
    uint32_t                interfaceIndex,
    const char               *name,
    const char               *regtype,
    const char               *domain,
    DNSServiceResolveReply   callBack,
    void                     *context        /* may be NULL
*/
);

```

其中，name、regtype和domain都从DNSServiceBrowse函数的处理结果中获得。callBack用于通知DNSServiceResolve的处理结果。该回调函数将返回服务的IP地址和端口号。

## 2. MDnsSdListener详解

MDnsSdListener对应的Framework层服务为NsdService（Nsd为Network Service Discovery的缩写），它是Android 4.1新增的一个Framework层Service。该服务的实现比较简单，故本书不详细讨论。感兴趣的读者不妨首先阅读SDK中关于NsdService的相关文档。

**提示** SDK中有一个基于Nsd技术开发的NsdChat例程，读者也可先学习它的实现。相关文档位置为

<http://developer.android.com/training/connect-devices-wirelessly/nsd.html>。

图2-9所示为MDnsSdListener家族成员示意图。

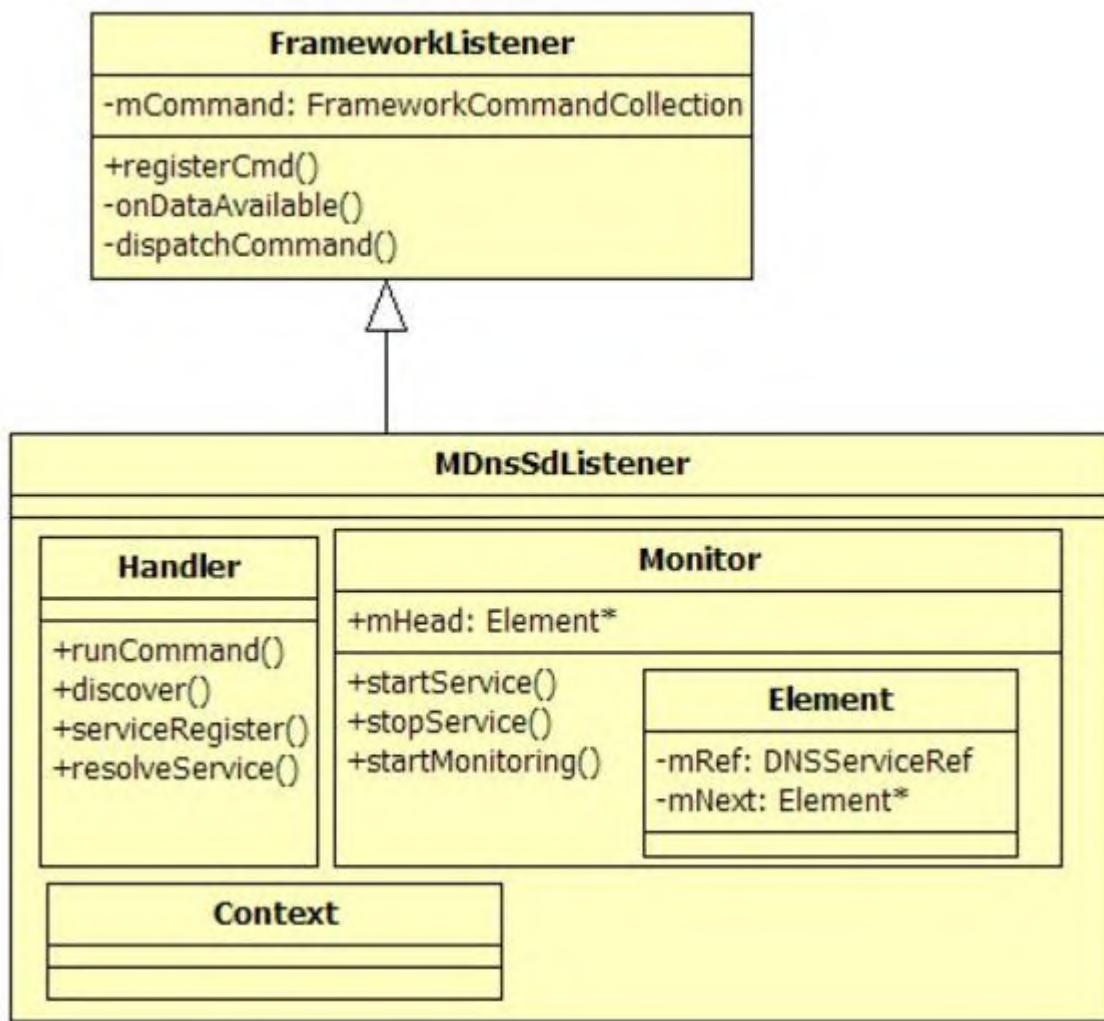


图2-9 MDnsSdListener家族成员

由图2-9可知：

- MDnsSdListener的内部类Monitor用于和mdnsd后台进程通信，它将调用前面提到的Bonjour API。
- Monitor内部针对每个DNSService都会建立一个Element对象，该对象通过Monitor的mHead指针保存在一个list中。

- Handler是MDnsSdListener注册的Command。

下面简单介绍MDnsSdListener的运行过程，主要工作可分成三步。

- 1) Netd创建MDnsSdListener对象，其内部会创建Monitor对象，而Monitor对象将启动一个线程用于和mdnsd通信，并接收来自Handler的请求。
- 2) NsdService启动完毕后将向MDnsSdListener发送“start-service”命令。
- 3) NsdService响应应用程序的请求，向MDnsSdListener发送其他命令，例如“discovery”等。Monitor将最终处理这些请求。

先来看第一步，当MDnsSdListener构造时，会创建一个Monitor对象，代码如下所示。

[-->MDnsSdListener.cpp: : Monitor: Monitor]

```
MDnsSdListener::Monitor::Monitor() {
    mHead = NULL;
    pthread_mutex_init(&mHeadMutex, NULL);
    // 创建两个socket，用于接收MDnsSdListener对象的指令
    socketpair(AF_LOCAL, SOCK_STREAM, 0, mCtrlSocketPair);
    // 创建线程，线程函数是threadStart，其内部会调用run
    pthread_create(&mThread, NULL,
        MDnsSdListener::Monitor::threadStart, this);
}
```

Monitor的threadStart线程将调用其run函数，该函数通过poll方式侦听包括mCtrlSocketPair在内的socket信息。这部分代码属于基本的Linux socket编程，本书不开展深入讨论。

当NsdService发送“start-service”命令后，Handler的runCommand将执行Monitor的startService函数，代码如下所示。

[-->MDnsSdListener.cpp: : Monitor: startService]

```
int MDnsSdListener::Monitor::startService() {
    int result = 0;
    char property_value[PROPERTY_VALUE_MAX];
```

```

    pthread_mutex_lock(&mHeadMutex);
    /*
     MDNS_SERVICE_STATUS是一个字符串，值为"init.svc.mdnsd"，在
     init.rc配置文件中，mdnsd是一个
     service，而"init.svc.mdnsd"将记录mdnsd进程的运行状态。
     */
    property_get(MDNS_SERVICE_STATUS, property_value, "");
    if (strcmp("running", property_value) != 0) {
        // 如果mdnsd的状态不为"running"，则通过设置"ctl.start"命令
        启动mdnsd
        property_set("ctl.start", MDNS_SERVICE_NAME);
        // 如果mdnsd成功启动，则属性值变成"running"
        wait_for_property(MDNS_SERVICE_STATUS, "running", 5);
        result = -1;
    } else {
        result = 0;
    }
    pthread_mutex_unlock(&mHeadMutex);
    return result;
}

```

startService的实现比较有趣，充分利用了init的属性控制以启动mdnsd进程。

当NsdService发送注册服务请求时，Handler的serviceRegister函数将被调用，代码如下所示。

[-->MDnsSdListener.cpp: : Handler: serviceRegister]

```

void MDnsSdListener::Handler::serviceRegister(SocketClient
*cli, int requestId,
      const char *interfaceName, const char *serviceName,
      const char *serviceType,
      const char *domain, const char *host, int port, int
      txtLen, void *txtRecord) {
    Context *context = new Context(requestId, mListener);
    DNSServiceRef *ref = mMonitor-
>allocateServiceRef(requestId, context);
    port = htons(port);
    .....
    DNSServiceFlags nativeFlags = 0;
    int interfaceInt = ifaceNameToI(interfaceName);
    // 调用Bonjour API DNSServiceRegister，并注册回调函数
    MDnsSdListenerRegisterCallback
    DNSServiceErrorType result = DNSServiceRegister(ref,

```

```

interfaceInt,
            nativeFlags, serviceName, serviceType, domain,
host, port,
            txtLen, txtRecord,
&MDnsSdListenerRegisterCallback, context);
if (result != kDNSServiceErr_NoError) {
    .....// 错误处理
}
// 通知Monitor对象进行rescan, 请读者自行研究该函数
mMonitor->startMonitoring(requestId);
cli->sendMsg(ResponseCode::CommandOkay, "serviceRegister
started", false);
return;
}

```

DNSServiceRegister内部将把请求发送给mdnsd去处理，处理的结果通过MDnsSdListener-RegisterCallback返回，该函数代码如下所示。

[-->MDnsSdListener.cpp: : MDnsSdListenerRegisterCallback]

```

void MDnsSdListenerRegisterCallback(DNSServiceRef sdRef,
DNSServiceFlags flags,
        DNSServiceErrorType errorCode, const char *serviceName,
const char *regType,
        const char *domain, void *inContext) {
    MDnsSdListener::Context *context =
        einterpret_cast<MDnsSdListener::Context *>
(inContext);
    char *msg;
    int refNumber = context->mRefNumber;
    if (errorCode != kDNSServiceErr_NoError) {
        .....// 错误处理
    } else {
        char *quotedServiceName =
SocketClient::quoteArg(serviceName);
        asprintf(&msg, "%d %s", refNumber, quotedServiceName);
        free(quotedServiceName);
        // 将处理结果返回给NsdService
        context->mListener-
>sendBroadcast(ResponseCode::ServiceRegistrationSucceeded,
                    msg, false);
    }
    free(msg);
}

```

**提示** Netd的工作流程相关代码相对简单，处理流程也比较固定。

1) NM接收Kernel的UEvent消息，然后转发给Framework层的客户端。

2) CL、DPL以及MDnsSdListener接收来自客户端的请求并处理它们。

对Android中DNS的管理以及Apple Bonjour技术感兴趣的读者不妨阅读章末列出的参考资料以加深理解。

## 2.3 CommandListener中的命令

CL一共定义了11个命令，这些命令充分反映了Netd在Android系统中网络管理和控制方面的职责。本节首先介绍Linux系统中常用的三个网络管理工具，然后再分类介绍CL中的相关命令。

# 读累了记得休息一会儿哦~

公众号：古德猫宁李

- 电子书搜索下载
- 书单分享
- 书友学习交流

网站：[沉金书屋 https://www.chenjin5.com](https://www.chenjin5.com)

- 电子书搜索下载
- 电子书打包资源分享
- 学习资源分享

### 2.3.1 iptables、tc和ip命令

网络管理和控制一直是一项比较复杂和专业的工作，由于Linux系统中原本就有一些强大的网络管理工具，故Netd也毫不犹豫充分利用了它们。目前Netd中最依赖三个网络管控工具，即iptables、tc和ip。

#### 1. iptables命令<sup>[7]</sup><sup>[8]</sup><sup>[9]</sup>

iptables是Linux系统中最重要的网络管控工具。它与Kernel中的netfilter模块配合工作，其主要功能是为netfilter设置一些过滤(filter)或网络地址转换(NAT)的规则。当Kernel收到网络数据包后，将会依据iptables设置的规则进行相应的操作。举个最简单的例子，可以利用iptables设置这样一条防火墙规则：丢弃来自IP地址为192.168.1.108的所有数据包。

##### (1) iptables原理

iptables的语法比较复杂，但工作原理较易理解。清楚iptables的前提是理解它的表(Table)、链(Chain)和规则(Rule)。三者关系如图2-10所示。

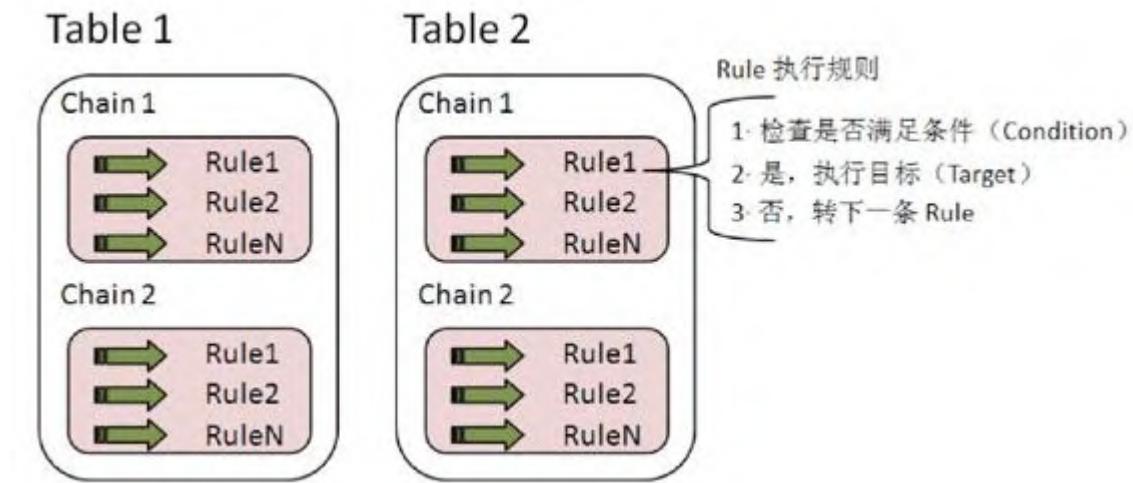


图2-10 iptables三要素关系

由图2-10可知：

- iptables内部（其实是Kernel的netfilter模块）维护着四个Table，分别是filter、nat、mangle和raw，它们对应着不同的功能，稍后将详细介绍它们的作用。
- Table中定义了Chain。一个Table可以支持多个Chain，Chain实际上是Rule的集合，每个Table都有默认的Chain。例如filter表默认的Chain有INPUT、OUTPUT、FORWARD。用户可以自定义Chain，也可修改Chain中的Rule。稍后将介绍不同Table中默认Chain方面的知识。
- Rule就是iptables工作的规则。首先，系统将检查要处理的数据包是否满足Rule设置的条件，如果满足则执行Rule中设置的目标（Target），否则继续执行Chain中的下一条Rule。

由前述内容可知，iptables中的Table和Chain是理解iptables工作的关键。表2-1总结了iptables中默认Table及Chain的相关内容。

表 2-1 iptables 默认 Table 及 Chain

Table 名	Table 说明	默认 Chain	Chain 说明
filter	iptables 默认使用的表，用于数据包的过滤	INPUT	处理目标为本机的数据包
		OUTPUT	处理本机产生的数据包
		FORWARD	处理通过本机转发的数据包
nat	控制网络地址转换（Network Address Translation）的表	PREROUTING	数据包路由前起作用
		POSTROUTING	数据包送出前起作用
		OUTPUT	数据包路由前起作用
mangle	修改包的信息，例如修改数据包的TTL值	PREROUTING	数据包路由前起作用
		OUTPUT	数据包路由前起作用
		FORWARD	数据包转发时起作用
		INPUT	数据包进入时起作用
		POSTROUTING	数据包送出前起作用
raw	根据链接状态进行相关处理，属于iptables的高级用法	PREROUTING	处理任何达到本机网口的数据包
		OUTPUT	处理本机产生的数据包

由表2-1可知，有些Table的默认Chain具有相同的名字，导致我们理解起来有些困难。为此，读者必须结合图2-11所示的iptables数据包处理流程图来理解前述内容。由图可知，不同Table和Chain在此处理流程中起着不同的作用。

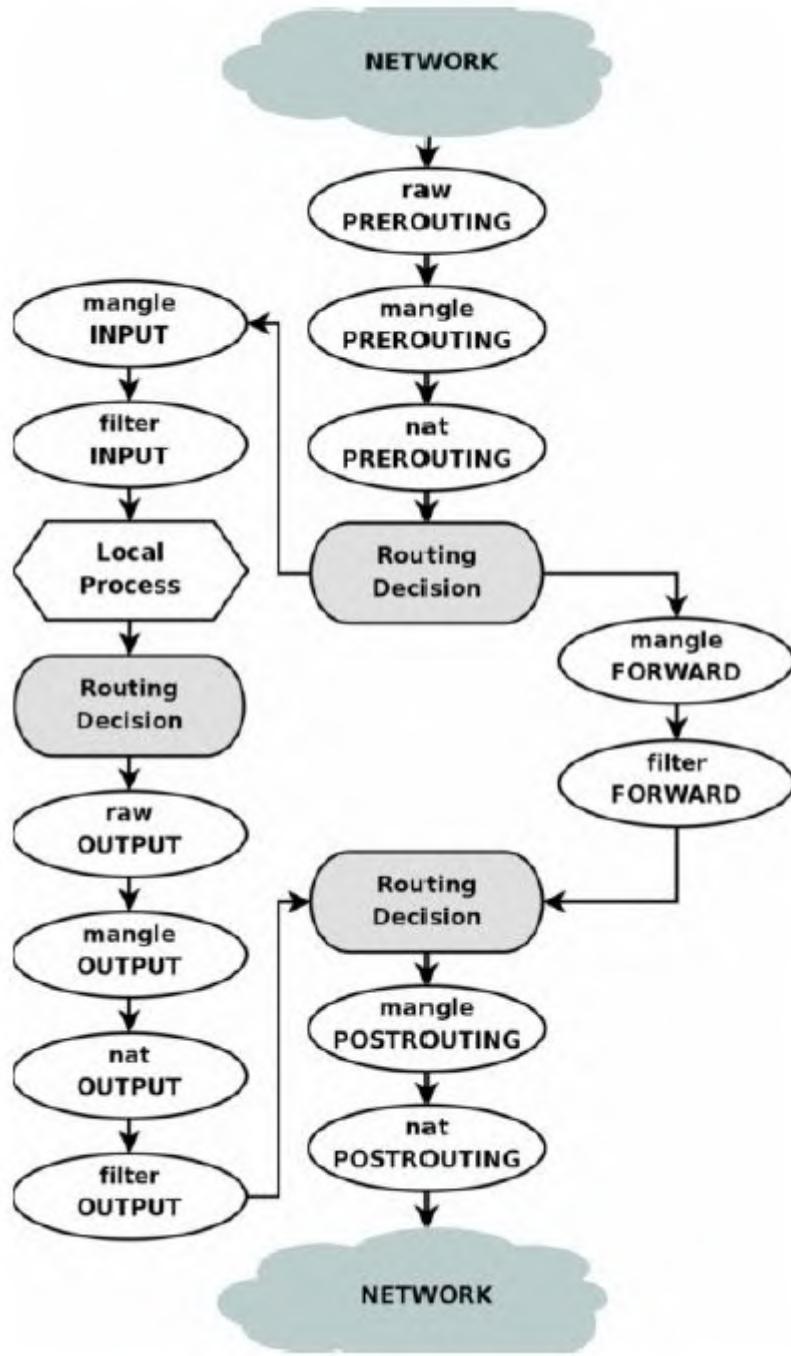


图2-11 iptables数据包处理流程

## (2) iptables Target和常用参数

iptables中的Rule有四个默认定义的Target，如下。

- ACCEPT: 接收数据包。

- **DROP**: 直接丢弃数据包。没有任何信息会反馈给数据源端。
- **RETURN**: 返回到调用Chain，略过后续的Rule处理。
- **QUEUE**: 数据返回到用户空间去处理。

**提示** iptables的扩展Target还支持REJECT。相比DROP而言，REJECT会发送反馈信息给数据源端，如主机不可达之类（`icmp-host-unreachable`）的信息。目前只有INPUT、OUTPUT、FORWARD以及被这三个链调用的自定义链支持REJECT。

iptables有很多参数，此处先介绍一些常用参数。

- t: 指定table。如果不带此参数，则默认为filter表。
- A, --append chain rule-specification: 在指定Chain的末尾添加一条Rule，rule-specification指明该Rule的内容。
- D, --delete chain rule-specification: 删除指定Chain中满足rule-specification的那条Rule。
- I, --insert chain[rule num]rule-specification: 为指定Chain插入一条Rule，位置由rule num指定。如果没有该参数，则默认加到Chain的头部。
- N: 创建一条新Chain。
- L, --list: 显示指定Table的Chain和Rule的信息。

Rule-specification描述该Rule的匹配条件以及目标动作，它也有一些参数来指明这些信息。

- i: 指定接收数据包的网卡名，如eth0、eth1等。
- o: 指定发出数据包的网卡名。
- p: 指定协议，如tcp、udp等。
- s, --source address[/mask]: 指定数据包的源IP地址。
- j, --jump target: 跳转到指定目标，如ACCEPT、DROP等。

以前文提到的设置防火墙为例，其对应的iptables设置参数如下。

```
iptables -t filter -A INPUT -s 192.168.1.108 -j DROP
```

如果仅拦截协议为tcp的数据包，则相应参数如下。

```
iptables -t filter -A INPUT -p tcp -s 192.168.1.108 -j DROP
```

另外，iptables仅支持IPv4，如果需针对IPv6进行相应设置，则要使用ip6tables工具。

**提示** iptables的用法非常灵活，如果没有长期的使用经验，将很难理解它们的真正作用。

## 2. tc命令<sup>[10][11][12][13]</sup>

TC是Traffic Control的缩写。在Linux系统中，流量控制是通过建立数据包队列（Queue），并控制各个队列中数据包的发送方式来实现的。Linux流量控制的基本原理如图2-12所示，该图描述了Linux系统中网络数据的处理流程。

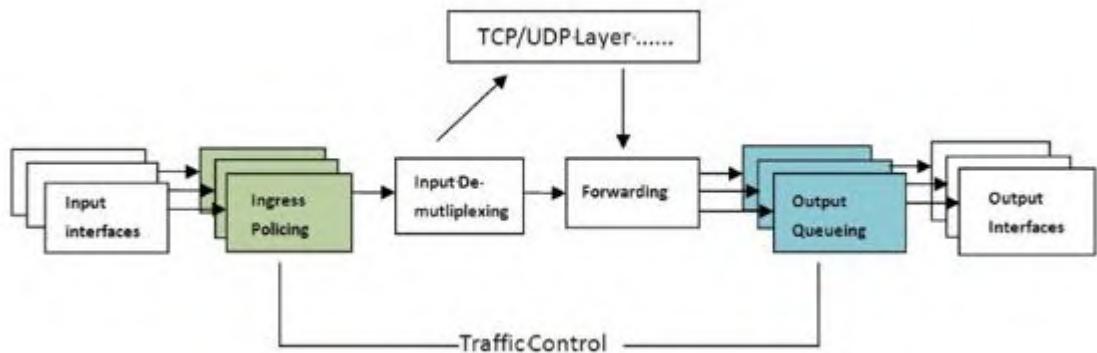


图2-12 网络数据处理流程

由图2-12可知：

- 接收包从输入接口（Input Interface）进来后，将经过输入流量限制（Ingress Policing）以丢弃不符合规定的数据包。而符合规定的数据包则交给输入多路选择器（Input De-Multiplexing）进行判断选择。
- 输入多路选择器的选择结果是，如果数据包的目的是本机，将该包送给上层处理，否则需要将数据包交到转发块（Forwarding Block）去处理。转发块同时也接收来自本机上层（TCP、UDP等）产生的包。
- 转发块通过查看路由表，决定处理包的下一跳目的地。然后，转发块对数据包进行排列整合以便将它们送到对应的输出接口（Output Interface）。

一般而言，我们只能限制本机网卡往外发送的数据包，而不能限制网卡接收的数据包。Linux中的流量控制就是在数据包通过输出接口时，通过改变发送次序等方式来实现控制传输速率的。

提示 也可通过IFB设备在输入接口进行流量控制。相关内容见2.3.3节。

在具体实现中，系统会建立许多队列及对应的队列规则（queuing discipline，简称qdisc）。目前系统包括的qdisc分为两类。

无分类的队列规则（Classless qdisc）：该规则对进入网卡的数据包不加区分，统一对待。使用这种规定的处理能够对数据包重新编排、延迟或丢弃。简而言之，这种类型是针对整个网卡的流量进行调整。常用的qdisc如下。

- fifo (First In First Out，先进先出队列)：最简单的控制。
- SFQ (Stochastic Fairness Queuing，随机公平队列)：对发送会话进行重排，这样每个发送会话都可以公平地发送数据）。
- RED (Random Early Detection，前向随机丢包)：用于模拟流量接近带宽限制时丢包的情况。
- TBF (Token Bucket Filter，令牌桶过滤器)：可较好地使得流量减低到预设值。适合高带宽的环境。这类qdisc使用的流量控制手段主要是排序、限速和丢包。

分类的队列规定（Classfull qdisc）：它对进入网络设备的数据包根据不同的需求以分类的方式区分对待。数据包进入一个分类的队列后，它就需要被送到某一个类中进行分类处理。对数据包进行分类的工具是过滤器（filter）。过滤器会返回一个决定，qdisc根据该决定把数据包送入相应的类进行排队。一个类可以包含多个子类，每个子类可再次使用它们的过滤器进行进一步的分类。当所有分类都处理完后，数据包才进入该类对应的队列排队。

简单言之，如果要利用tc进行流量控制，其主要工作将包含建立队列、建立分类和建立过滤器三个方面，一般的步骤如下。

- 1) 针对网络物理设备（如以太网卡eth0）绑定一个队列QDisc；
- 2) 在该队列上建立分类class；
- 3) 为每一分类建立一个基于路由的过滤器filter；
- 4) 最后与过滤器相配合，建立特定的路由表。

**提示** tc命令所涉及的流量控制方面的知识相当复杂，感兴趣的读者可根据章末列出的参考资料做进一步的深入研究。

### 3. ip命令<sup>[14]</sup>

ip命令是Linux系统中另一个强大的网络管理工具，主要功能如下。

- 可替代ifconfig命令。即通过ip工具可管理系统中的网络接口，包括配置并查看网络接口情况、使能或禁止指定网络接口。
- 可替代route命令。即ip工具支持设置主机路由、网络路由、网关参数等。
- 可替代arp命令。即ip工具支持查看、修改和管理系统的ARP缓存等。

ip命令的语法为：

```
ip [OPTIONS] OBJECT [COMMAND [ARGUMENTS]]
```

例如为网络设备进行配置的ip命令语法如下：

```
ip addr[add|del] IFADDR dev STRING
//接口eth0赋予地址192.168.0.1，掩码是255.255.255.0（24代表掩码中1的个数）
//广播地址是192.168.0.255
ip addr add 192.168.0.1/24 broadcast 192.168.0.255 label eth0
dev eth0
```

**提示** 本节对Linux系统中常用的三个网络管理工具iptables、tc和ip命令进行了一些简单介绍。其中，iptables用于管理数据包过滤、NAT等方面的工作。tc用于流量控制，其背后涉及的知识较为复杂。ip

命令可替代ifconfig、route和arp等命令。ip命令的路由控制示例将在2.3.3节介绍。

## 2.3.2 CommandListener构造函数和测试工具ndc

本节将介绍CL的构造函数以及Netd的测试工具ndc。

### 1. CL构造函数

在前文的2.2.3节中已经介绍了CL构造函数的前半部分，下面接着介绍CL构造函数的后半部分，代码如下所示。

```
[-->CommandListener: : CommandListener构造函数]

CommandListener::CommandListener() :
    FrameworkListener("netd", true) {
    .....// 创建命令和命令控制对象

    // 初始化iptables中的各个Table及相应Chain和Rules
    // createChildChains第一个参数用于指明针对IPv4还是IPv6
    createChildChains(V4V6, "filter", "INPUT", FILTER_INPUT);
    createChildChains(V4V6, "filter", "FORWARD",
FILTER_FORWARD);
    createChildChains(V4V6, "filter", "OUTPUT", FILTER_OUTPUT);
    createChildChains(V4V6, "raw", "PREROUTING",
RAW_PREROUTING);
    createChildChains(V4V6, "mangle", "POSTROUTING",
MANGLE_POSTROUTING);
    createChildChains(V4, "nat", "PREROUTING", NAT_PREROUTING);
    createChildChains(V4, "nat", "POSTROUTING",
NAT_POSTROUTING);

    // 4.2以后，Netd允许OEM厂商可自定义一些规则
    // 这些规则在/system/bin/oem-iptables-init.sh文件中保存
    setupOemIptablesHook();
    // 初始化iptables中的一些chain，以及初始化路由表
    sFirewallCtrl->setupIptablesHooks();
    sNatCtrl->setupIptablesHooks();
    sBandwidthCtrl->setupIptablesHooks();
    sIdletimerCtrl->setupIptablesHooks();
    // 初始时，Netd将禁止带宽控制功能
    sBandwidthCtrl->enableBandwidthControl(false);
```

由上述代码可知，CL构造函数的后半部分工作主要是利用iptables等工具创建较多的Chain和Rule，以及对某些命令控制对象进行初始化。

本节将重点关注iptables执行后的效果。图2-13所示为CL构造后，iptable中filter表内所创建的Chain和Rule。

图2-13中列出的是filter表中部分Chain的截图，其中，target是目标名，prot是protocol之意，opt是选项，source和destination分别表示数据包的源和目标地址。

- bw\_INPUT、bw\_OUTPUT和bw\_FORWARD Chain用于带宽（Bandwidth）控制。
- fw\_INPUT、fw\_OUTPUT和fw\_FORWARD用于防火墙（Firewall）控制。
- natctrl\_FORWARD用于网络地址转换（NAT）控制。
- oem\_fwd、oem\_out用于OEM厂商自定义的控制。

```

Chain INPUT (policy ACCEPT)
target    prot opt source          destination
bw_INPUT  all  --  anywhere       anywhere
fw_INPUT  all  --  anywhere       anywhere

Chain FORWARD (policy ACCEPT)
target    prot opt source          destination
oem_fwd   all  --  anywhere       anywhere
fw_FORWARD all  --  anywhere       anywhere
bw_FORWARD all  --  anywhere       anywhere
natctrl_FORWARD all  --  anywhere       anywhere

Chain OUTPUT (policy ACCEPT)
target    prot opt source          destination
oem_out   all  --  anywhere       anywhere
fw_OUTPUT all  --  anywhere       anywhere
bw_OUTPUT all  --  anywhere       anywhere

Chain bw_FORWARD (1 references)
target    prot opt source          destination

Chain bw_INPUT (1 references)
target    prot opt source          destination
RETURN   all  --  anywhere       anywhere

Chain bw_OUTPUT (1 references)
target    prot opt source          destination

```

图2-13 CL创建后filter表的内容

## 2. ndc测试工具

ndc是Android为Netd提供的一个测试工具。其主要功能有：

- 监视Netd中发生的事情。
- 支持通过命令行发送命令给Netd去执行。

相信读者很轻松就能想到ndc的实现原理，其实它就是连接上位于netd进程中的“netd”监听socket，然后从Netd接收信息或发送命令给Netd。

图2-14为使用ndc monitor选项监控Galaxy Note 2打开Wi-Fi功能时得到的输出。

```
1|shell@android:/ # ndc monitor  
[Connected to Netd]  
600 Iface added p2p0  
600 Iface linkstate p2p0 down  
600 Iface added wlan0  
600 Iface linkstate wlan0 down  
600 Iface linkstate wlan0 up  
600 Iface linkstate wlan0 up  
600 Iface linkstate p2p0 up  
600 Iface linkstate p2p0 up  
600 Iface linkstate wlan0 up
```

图2-14 ndc monitor执行结果

利用ndc来监视Netd的工作状况是一个简单高效的方法。另外，还可利用ndc来测试CommandListener中所支持的各种命令。这对于网络相关模块的HAL层开发者来说无疑是一个很大的帮助。为了方便读者理解，本节下文也将利用ndc来展示命令执行的结果。

在正式介绍CL中的命令对象之前，先介绍这些命令对象处理的通用流程。

如图2-4所示，命令对象的真正控制函数是runCommand，而绝大部分命令的runCommand函数都有类似如下的代码结构（此处以InterfaceCmd

为例)。

```
[-->CommandListener.cpp: : InterfaceCmd: runCommand]

int CommandListener::InterfaceCmd::runCommand(SocketClient
*cli,
                                              int argc, char
**argv) {
    if (argc < 2) { // 先做参数检查
        cli->sendMsg(ResponseCode::CommandSyntaxError, "Missing
argument", false);
        return 0;
    }
    // 然后分别处理自己支持的各种命令选项
    if (!strcmp(argv[1], "list")) {
        .....// 处理"list"选项
    } else if (!strcmp(argv[1], "readrxcounter")) {
        .....// 处理"readrxcounter"选项
    }
    .....// 处理其他选项
}
```

由上述InteraceCmd的处理函数可知， runCommand的处理流程如下。

- 1) 首先参数检查，一般是检查参数个数是否正确。
- 2) 然后根据不同的选项进行对应的处理。

**提示** 接下来分析CL中各个命令，由于其中涉及较多的知识，因此单独增加“背景知识介绍”小节以帮助读者更好地理解它们。

### 2.3.3 InterfaceCmd命令

InterfaceCmd用来管理和控制系统中的网络设备，其支持较多的控制选项。另外，InterfaceCmd除了和控制对象InterfaceController交互外，还会和ThrottleController、SecondaryTableController交互。InterfaceCmd涉及较多的背景知识，本节先集中介绍这部分内容。

#### 1. 背景知识介绍

InterfaceCmd涉及三个重要知识点，分别是IFB设备、Netdevice编程和Linux策略路由管理。

##### (1) IFB设备<sup>[13]</sup>

IFB（Intermediate Functional Block）是IMQ（InterMediate Queuing）的替代者，二者都是Linux为更好地完成流量控制而实现的虚拟设备。相比IMQ而言，IFB的实现代码更少，对SMP（多核）系统支持的也更好。

为什么需要IFB设备呢？2.3.1节介绍tc命令的时候提到，Linux中丰富的流量控制手段和规则都是针对出口流量的，即大多数排队规则（qdisc）都是用于输出方向的。而输入方向主要是入口流量限制，只有一个排队规则，即ingress qdisc。有没有办法让输入流量也能像输出流量那样得到更多的控制呢？

于是，系统新增了一种方式，用于重定向incoming packets。通过ingress qdisc把输入方向的数据包重定向到虚拟设备IFB，而在IFB的输出方向配置多种qdisc，就可以达到对输入方向的流量做队列调度的目的。

图2-15所示为加上IFB后整个流量控制示意图。系统中一共有两个IFB设备，IFB Device 0用于输入流量控制，IFB Device 1用于输出流量控制。

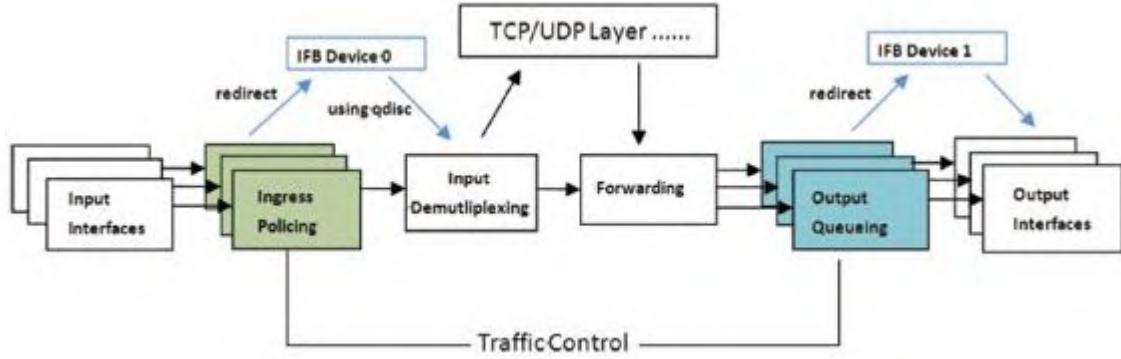


图2-15 增加IFB设备后的流量控制

## (2) Netdevice编程<sup>[15]</sup>

Netdevice是Linux平台中用于直接针对底层网络设备编程的一套接口，其使用方法很简单，就是利用socket句柄和ioctl函数来操作指定的网络设备。常用的数据结构如下所示。

```
#include <net/if.h>
struct ifreq {
    char ifr_name[IFNAMSIZ]; // 指定要操作的NIC名
    union { // 一个联合体，可以设置各种参数
        struct sockaddr ifr_addr; // 网卡地址
        struct sockaddr ifr_dstaddr;
        struct sockaddr ifr_broadaddr; // 组播地址
        struct sockaddr ifr_netmask; // 网络掩码
        struct sockaddr ifr_hwaddr; // MAC地址
        short ifr_flags;
        int ifr_ifindex;
        int ifr_metric;
        int ifr_mtu;
        struct ifmap ifr_map;
        char ifr_slave[IFNAMSIZ];
        char ifr_newname[IFNAMSIZ];
        char *ifr_data;
    };
};
```

Netdevice支持一些特殊的ioctl参数，表2-2展示了几个常见的参数。

表 2-2 Netdevice ioctl 参数说明

参数名	功能描述
SIOCGIFNAME	根据 ifr_index 的值，获得指定网络设备的名字，并保存在 ifr_name 中
SIOCGIFINDEX	获得指定网络设备的索引值
SIOCGIFFLAGS SIOCSIFFLAGS	G 代表 Get，用于获取 flag；S 代表 Set，用于设置 flag。支持的 flags 有如下 <ul style="list-style-type: none"> <li>• IFF_UP：启动网络设备</li> <li>• IFF_PROMISC：网卡是否为混杂模式</li> <li>• IFF_BROADCAST：广播支持</li> <li>• IFF_MULTICAST：组播支持</li> </ul>
SIOCSIFHWADDR SIOCGIFHWADDR	设置 / 获取 MAC 地址
SIOCSIFNAME	设置网络设备名，新名字保存在 ifr_newname 中

关于Netdevice的详细信息，读者可通过man netdevice获得。Android为Netdevice编程提供了一些更为简单的API，它们被封装在libnetutils中，统称为ifc\_utils，下面介绍其中的三个常用API。

使用ifc\_utils之前，需调用ifc\_init进行初始化，其代码如下所示。

```
[-->ifc_utils.c: : ifc_init]

int ifc_init(void)                                     // ifc是interface
control的缩写
{
    int ret;
    if (ifc_ctl_sock == -1) {                           // 创建一个
socket句柄
        ifc_ctl_sock = socket(AF_INET, SOCK_DGRAM, 0);
        .....// 错误处理
    }
    ret = ifc_ctl_sock < 0 ? -1 : 0;
    return ret;
}
```

如果要启动某个NIC（如"ifb0"设备），需调用ifc\_up函数，其代码如下所示。

```
[-->ifc_utils.c: : ifc_up]

int ifc_up(const char *name)
{
    // 要启动"ifb0"设备，name设置为"ifb0"
    int ret = ifc_set_flags(name, IFF_UP, 0);      // 设置IFF_UP参
```

数

```
    return ret;
}
```

ifc\_up函数将通过调用ifc\_set\_flags函数并传递IFF\_UP来启动对应的设备，ifc\_set\_flags的代码如下所示。

[-->ifc\_utils.c: : ifc\_set\_flags]

```
static int ifc_set_flags(const char *name, unsigned set,
unsigned clr)
{
    struct ifreq ifr;
    ifc_init_ifr(name, &ifr);           // 初始化ifr参数，把name复制到
其ifr_name字符数组中

    // 先获取该NIC以前的flag信息
    if(ioctl(ifc_ctl_sock, SIOCGIFFLAGS, &ifr) < 0) return -1;
    // 将新的flag加入到原有的flags中
    ifr.ifr_flags = (ifr.ifr_flags & (~clr)) | set;
    return ioctl(ifc_ctl_sock, SIOCSIFFLAGS, &ifr); // 将组合后的
flag传递给Kernel
}
```

相比直接使用Netdevice来说，Android封装的ifc\_utils更加直观和易用。建议有需要的读者使用ifc\_utils对NIC进行操作。

### (3) Linux策略路由<sup>[16]</sup><sup>[17]</sup>

策略路由 是指系统依据网络管理员定下的一些策略对IP包进行的路由选择。例如网管可设置这样的策略：“所有来自网A的包，选择X路径，其他选择Y路径”，或者“所有TOS (Type Of Service, IP协议头的一部分) 为A的包选择路径F，其他选择路径K”。

从Kernel 2.1开始，Linux采用了策略性路由机制。相比传统路由算法，策略路由主要引入了多路由表及规则的概念。

传统路由算法仅使用一张路由表。但在某些情况下，系统需要使用多个路由表。例如，一个子网通过一个路由器与外界相连。而该路由器与外界有两条线路相连，其中一条的速度较快，另一条的速度较慢。对于子网内的大多数用户来说，由于对速度没有特殊要求，可以让他

们用速度较慢的路由；但是子网内有一些特殊用户对速度的要求较苛刻，他们需要使用速度较快的路由。很明显，仅使用一张路由表是无法实现上述要求的。而如果根据源地址或其他参数，对不同的用户使用不同的路由表，就可以实现这项要求。

传统Linux下配置路由的工具是route，而实现策略性路由配置的工具是iproute2工具包，常用的命令就是ip命令。

Linux最多可以支持255张路由表，其中有4张表是内置的。

- 表255：本地路由表（local table）。本地接口地址，广播地址和NAT地址都放在这个表中。该路由表由系统自动维护，管理员不能直接修改。
- 表254：主路由表（main table）。如果没有指明路由所属的表，所有的路由都默认都放在这个表里，一般来说，传统路由工具命令（如route）所添加的路由都会加到这个表中。一般是普通的路由。
- 表253：默认路由表（default table）。一般来说默认的路由都放在这张表，但是如果特别指明，该表也可以存储所有的网关路由。
- 表0：默认保留。

2.3.1节简单介绍了ip命令，ip命令的一些具体用法如下。

```
// ①查看路由表的内容命令  
ip route list table table_number  
// ②对于路由的操作包括change、del、add、append、replace、monitor  
// 向主路由表（main table）即表254添加一条路由，路由的内容是设置  
192.168.0.4成为网关  
ip route add 0/0 via 192.168.0.4 table main  
  
// 向路由表1添加一条路由，子网192.168.3.0（子网掩码是255.255.255.0）的  
网关是192.168.0.3  
ip route add 192.168.3.0/24 via 192.168.0.3 table 1
```

在多路由表的路由体系里，所有的路由操作（如添加路由等）都需要指明要操作的路由表。如果没有指明路由表，系统默认该操作是针对主路由表（表254）开展的。而在单表体系里，路由操作无须指明路由表。

至此，分别介绍了IFB、Netdevice和Linux路由策略等背景知识。在接下来的分析中，读者将看到它们在InterfaceCmd中的应用。

## 2. InterfaceCmd命令选项

InterfaceCmd支持较多命令选项，笔者将它们粗略地分为6类。

### (1) NIC设备信息管理选项

此类选项包括以下内容。

- list：列举系统当前的网络设备。这是通过枚举/sys/class/net目录下的文件名而来。该目录下的文件（其实是一个设备文件）代表一个具体的网络设备，例如eth0、lo等。/sys目录是Linux设备文件系统(sysfs)的挂载点，用于Linux统一设备管理之用。
- readrxcnter和readtxcounter：这两个选项分别用于读取系统所有网络设备的接收字节数和发送字节数等统计信息。它们都是通过读取/proc/net/dev下的对应文件来实现的。

图2-16所示为Galaxy Note 2对应文件的内容。该文件显示了系统所有网络设备的收发字节数、包数、错误及丢包等各种统计信息。

Interface	Receive								Transmit							
	bytes	packets	errs	drop	fifo	frame	compressed	multicast	bytes	packets	errs	drop	fifo	cols	carrier	compressed
lo:	1555875	4391	0	0	0	0	0	0	1555875	4391	0	0	0	0	0	0
sit0:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ip6tnl0:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
rmnet0:	1592597	1322	0	0	0	0	0	0	84015	1174	0	0	0	0	0	0
rmnet1:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
rmnet2:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
p2p0:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
wlan0:	23847	112	0	19	0	0	0	30	7485	86	0	0	0	0	0	0

图2-16 /proc/net/dev文件内容

```
shell@android:/ # ndc interface list
110 0 lo
110 0 sit0
110 0 ip6tnl0
110 0 rmnet0
110 0 rmnet1
110 0 rmnet2
110 0 p2p0
110 0 wlan0
200 0 Interface list completed
shell@android:/ # ndc interface readrxcounter wlan0
216 0 23847
```

图2-17 list和readrxcounter结果

图2-17展示了在Galaxy Note 2中用ndc执行interface list和readrxcounter选项后的结果。其中最后一行，216为readrxcounter选项对应的返回码，0为错误码，23847为wlan0设备的接收字节数。比较图2-16最后一行wlan0的接收字节数可知，二者完全一致。

## (2) 输入和输出流量控制选项

InterfaceCmd支持对指定网络设备设置流量阈值，其中包括如下。

- getthrottle : throttle是节流之意，用于流量控制。该选项支持读取发送和接收阈值。不过目前代码中这两个值都将返回0。
- setthrottle : 该选项的参数为setthrottle interface rx\_kbps tx\_kbps，用于控制网卡输入和输出流量。

setthrottle的实现比较复杂，核心代码在ThrottleController类的setInterfaceThrottle函数中。这部分代码充分利用了tc命令。以setthrottle wlan0 100 200为例，调用的命令顺序如下。

```
#htb是Classful类qdisc的一种，属于tc命令中较为复杂的一种用法
tc qdisc add dev wlan0 root handle 1: htb default 1 r2q 1000
tc class add dev wlan0 parent 1: classid 1:1 htb rate 200kbit
#利用ifc_utils启动IFB0设备
tc qdisc add dev ifb0 root handle 1: htb default 1 r2q 1000
tc class add dev ifb0 parent 1: classid 1:1 htb rate 100kbit
```

```
tc qdisc add dev wlan0 ingress
tc filter add dev wlan0 parent ffff: protocol ip prio 10 u32
match \
    u32 0 0 flowid 1:1 action mirred egress redirect dev ifb0
```

由上述setthrottle的实现可知，其要完成的功能很简单（就是想对wlan0设备施加输入和输出流量控制），但实现过程却比较复杂，使用了tc命令的很多高级选项。

**提示** 本书非专业的Linux网络管理书籍，故此处仅向读者展示这些命令。对其背后原理感兴趣的读者，不妨阅读本章参考资料中列出的书籍。

### (3) Route控制选项

InterfaceCmd支持路由控制，选项名为“route”。它支持对Route的添加、修改和删除。另外还支持对多路由表的配置。相关代码如下所示。

[-->CommandListener: : InterfaceCmd: runCommand]

```
.....// 略去部分代码
if (!strcmp(argv[1], "route")) {
    int prefix_length = 0;
    .....// 参数检测
    if (!strcmp(argv[2], "add")) {
        if (!strcmp(argv[4], "default")) { // 调用ifc_add_route
            for 指定NIC设备添加路由
                ifc_add_route(argv[3], argv[5], prefix_length,
                argv[7]);
            }.....// 错误处理
        } else if (!strcmp(argv[4], "secondary")) { // 对多路由表进
        行操作
            return sSecondaryTableCtrl->addRoute(cli,
            argv[3], argv[5],
            prefix_length, argv[7]);
        } .....
```

由上述代码可知：

- 当添加的是默认路由时，直接调用ifc\_add\_route函数为指定设备添加一个路由。该功能的实现利用了多种ifc\_utils的ifc\_add\_route函数（也就是基于Netdevice封装的API），对应的IOCTL参数为SIOCADDRT。请读者自行研究该函数的实现。
- 如果是针对多路由表的”secondary”，则使用SecondaryTableController的addRoute函数来添加路由。该函数中，SecondaryTableCtrl首先会计算一个Table索引，然后利用ip命令将路由添加到对应的Table中。

addRoute的核心是调用modifyRoute，其代码如下所示。

```
[-->SecondaryTableController.cpp: : modifyRoute]

int SecondaryTableController::modifyRoute(SocketClient *cli,
    const char *action, char *iface,
    char *dest, int prefix, char *gateway, int tableIndex)
{
    char *cmd;

    if (strcmp(":::", gateway) == 0) {
        asprintf(&cmd, "%s route %s %s/%d dev %s table %d",
            IP_PATH, action, dest, prefix, iface,
            tableIndex+BASE_TABLE_NUMBER);
    } else {
        // 调用ip route命令的参数
        asprintf(&cmd, "%s route %s %s/%d via %s dev %s table
%d",
            IP_PATH, action, dest, prefix, gateway,
            iface, tableIndex+BASE_TABLE_NUMBER);
    }

    if (runAndFree(cli, cmd)) {
        .....// 错误处理
        return -1;
    }
    // SecondaryTableControl对各个Table表的使用都有计数控制。请读者自
    行阅读相关代码
    .....// 略去相关代码
    modifyRuleCount(tableIndex, action);
    cli->sendMsg(ResponseCode::CommandOkay, "Route modified",
false);
    return 0;
}
```

上述代码中，以wlan0为例，对应的ip命令情况如下。

```
ip route add 192.168.0.100/24 via 192.168.0.1 dev wlan0 table  
78#使用路由表78
```

#### (4) Driver控制选项

该选项的名为“driver”，由InterfaceController加载/system/lib/libnetcmd.so对Kernel中的Driver进行控制。而针对Driver操作的API由InterfaceController的interfaceCmd来完成。

[-->InterfaceController.cpp]

```
char if_cmd_lib_file_name[] = "/system/lib/libnetcmdiface.so";  
char set_cmd_func_name[] = "net_iface_send_command";  
char set_cmd_init_func_name[] = "net_iface_send_command_init";  
char set_cmd_fini_func_name[] = "net_iface_send_command_fini";
```

InterfaceController构造时会加载libnetcmdiface.so，然后获取上面三个函数的指针。这三个函数用于直接向驱动发送命令。由于代码中没有任何说明，而且也没有地方用到它，故读者仅简单了解即可。

**提示** 笔者认为libnetcmdiface的目的应该是支持某些网络设备生产厂商（如Broadcom）特有的一些命令选项。由于厂商的这些特殊选项不通用，它们不能向其他命令选项一样在代码中被列举出来。

另外，根据本书审稿专家的反馈，在实际产品中不存在libnetcmdiface.so文件。对博通公司的芯片而言，其私有命令的代码在hardware/broadcom/wlan/bcmhdh/wpa\_supplicant\_8\_lib中，对应的库文件为lib\_driver\_cmd\_bcmhd.so。

#### (5) IFC设备控制选项

IFC设备控制选项主要是利用ifc\_utils API对设备进行管理和控制，例如获取某个设备的配置信息、启动某个设备、配置某个设备的MTU等。主要利用上文介绍的IFC接口对设备进行控制，例如启动、设置MTU等。本节仅给出利用getcfg选项查询wlan0网络设备配置情况的示意图，如图2-18所示。

```
shell@android:/ # ndc interface getcfg wlan0  
213 0 90:18:7c:69:88:e2 192.168.1.101 24 up broadcast running multicast
```

图2-18 getcfg wlan0结果

图2-18显示了Galaxy Note 2打开Wi-Fi后wlan0设备的配置信息，其中：

- 90: 18: 7c: 69: 88: e2为该设备的MAC地址。
- 192. 168. 1. 101和24为IP地址以及子网掩码（255. 255. 255. 0）。
- up、broadcast、running和multicast表示该设备当前已经启动并正在运行，并支持广播和组播。

#### (6) IPv6控制选项<sup>[18]</sup>

InterfaceCmd支持两个IPv6选项。

- ipv6privacyextensions：用于控制网卡使用临时IPv6地址的情况。该功能需要要3.0以上的kernel控制方法就是向/proc/sys/net/ipv6/网卡名/conf/use\_tempaddr写入对应的值。值为0表示不允许IPv6临时地址，值为2表示优先使用临时IP地址，值为1则表示优先使用非临时IP地址。
- ipv6：用于启动或禁止网卡的IPv6支持。往/proc/sys/net/ipv6/网卡名/conf/disable\_ipv6中写入0或1即可。

## 2.3.4 IpFwd和FirewallCmd命令

本节将介绍IpFwd和FirewallCmd两个命令。

### 1. IpFwd命令

IpFwd命令比较简单，主要是控制内核ipforward功能，其支持三个选项。

- status：用于判断ipforward功能是否开启。
- enable和disable：分别用于启动和禁止ipforward功能。

上述功能借助TetherController控制对象来完成，其内部代码非常简单，就是通过读写/proc/sys/net/ipv4/ip\_forward文件来实现对内核ipforward功能的管理。

提示 读者可上网搜索/proc/sys/net/ipv4目录下其他文件的作用。

### 2. FirewallCmd命令

FirewallCmd用于防火墙控制。它支持以下命令选项。

- enable和is\_enabled：用于启动防火墙和判断防火墙是否已经启动。
- set\_interface\_rule：针对单个或多个NIC设备设置防火墙规则。
- set\_egress\_source\_rule和set\_egress\_dest\_rule：基于源IP和目标IP地址设置防火墙。
- set\_uid\_rule：根据uid设置单个进程的防火墙。

下面重点介绍防火墙功能的启动以及如果针对单个进程的设置防火墙规则。

#### (1) 启动防火墙

本节介绍enable和set\_uid\_rule。其中，enable将调用FirewallController的enableFireWall函数，代码如下所示。

```
[-->FirewallController.cpp: : enableFirewall]

int FirewallController::enableFirewall(void) {
    int res = 0;

    // 先清空fw_INPUT/fw_OUTPUT/fw_FORWARD链中的规则
    disableFirewall();

    // 启动防火墙就是设置Filter表中fw_INPUT/fw_OUTPUT/fw_FORWARD的
    // 目标为DROP和REJECT
    res |= execIptables(V4V6, "-A", LOCAL_INPUT, "-j", "DROP",
    NULL);
    res |= execIptables(V4V6, "-A", LOCAL_OUTPUT, "-j",
    "REJECT", NULL);
    res |= execIptables(V4V6, "-A", LOCAL_FORWARD, "-j",
    "REJECT", NULL);

    return res;
}
```

由上述代码可知，当启动防火墙时，filter表中三个用于防火墙控制的Chain的目标都改为DROP或REJECT，这样系统就无法收发网络数据包。图2-19所示为笔者在Ubuntu机器上设置的规则。

Chain fw_FORWARD (1 references)			
target prot opt source destination			
REJECT all -- anywhere anywhere			reject-with icmp-port-unreachable
<hr/>			
Chain fw_INPUT (1 references)			
target prot opt source destination			
DROP all -- anywhere anywhere			
<hr/>			
Chain fw_OUTPUT (1 references)			
target prot opt source destination			
REJECT all -- anywhere anywhere			reject-with icmp-port-unreachable

图2-19 enable防火墙

图2-19所示是利用Ubuntu机器模拟enableFirewall函数后的结果。防火墙启动后，机器就无法连接网络了。

读者可能好奇，为什么enableFirewall的破坏作用如此之大？这是因为防火墙设置一般有两种方法。

- 先允许整个系统都能上网，然后再单独设置防火墙规则以禁止某些模块连接网络。

- 先禁止整个系统上网，然后再单独设置防火墙规则以允许某些模块能连接网络。

由此可知，Android采用了第二种更为严厉的方式来设置防火墙。接下来的工作就需要设置各种防火墙规则以允许某些程序能够上网了。本节讨论如何为单个进程设置防火墙规则。

**提示** 采用Ubuntu来测试enableFirewall，因为模拟器和主机也使用socket通信，一旦启动防火墙，主机就无法再使用adb shell来控制模拟器了。

## (2) 设置进程防火墙

本节介绍的针对单个应用程序设置其防火墙规则，是大部分软件管家禁止某些应用程序上网的通用方法。代码在setUidRule函数中，如下所示。

[-->FirewallController.cpp: : setUidRule]

```
int FirewallController::setUidRule(int uid, FirewallRule rule)
{
    char uidStr[16];
    sprintf(uidStr, "%d", uid);

    const char* op;
    if (rule == ALLOW) {
        op = "-I";
    } else {
        op = "-D";
    }

    int res = 0;
    res |= execIptables(V4V6, op, LOCAL_INPUT, "-m", "owner",
    "--uid-owner", uidStr, "-j", "RETURN", NULL);
    res |= execIptables(V4V6, op, LOCAL_OUTPUT, "-m", "owner",
    "--uid-owner", uidStr, "-j", "RETURN", NULL);
    return res;
}
```

该函数很简单，就是调用正确的iptables命令，它们分别如下。

```
iptables -I FORWARD_INPUT -m owner --uid-owner uid -j RETURN  
iptables -I FORWARD_OUTPUT -m owner --uid-owner uid -j RETURN
```

这里要特别指出的是，目前Android只能根据uid来区分进程。由于Android平台允许不同应用程序共享同一个uid<sup>①</sup>。多个应用程序共享同一个uid的情况下，如果用户禁止了其中一个应用程序（软件管家的防火墙控制允许用户选择是否禁止某个应用程序上网），则连带其他的相关程序都不能上网，虽然用户并没有选择禁止其他应用程序。

提示 笔者测试了360安全卫士，当出现上述关联情况时，它能提醒用户哪些关联程序也会被一并禁止上网。

<sup>①</sup> 参考《深入理解Android：卷II》4.3.1节关于Android系统中UID/GID知识的介绍。

## 2.3.5 ListTtysCmd和PppdCmd命令

ListTTysCmd和PppdCmd比较简单，但也涉及两个重要的背景知识。

### 1. 背景知识介绍

本节介绍的两个背景知识点是TTY和ptmx编程，以及PPP和pppd。

#### (1) TTY和ptmx编程<sup>[19]</sup><sup>[20]</sup>

TTY是Linux系统（更确切地说是UNIX）中终端设备的统称，该词源于TeleTYewriter（电传打字机），是一个通过串行线用打印机键盘通过阅读和发送信息的设备。不过随着计算机技术的发展，这类设备早就被键盘和显示器替代了。

从现在的情况来看，Linux系统中的TTY设备包含许多类型的设备，它们大体可分为以下三种。

- 串行端口终端：这类设备一般命名为/dev/ttySn，n为索引号，例如ttyS0、ttyS1等。它们类似于Windows系统的COM0、COM1等，代表串行端口。当某个应用往ttyS0写入数据时，另一个应用就能从ttyS0读到对应的数据。
- 伪终端（psuedo terminal）：这类设备成对出现，其中一个是master设备，另一个是slave设备。二者的关系类似管道（Pipe），当一个程序往slave设备写数据时，另外一个打开master设备的程序就能读到数据。以前（UNIX 98 scheme之前）主从设备命名方式为/dev/ptyp0和/dev/ttyp0。现在master设备命名为/dev/ptmx，而slave设备则对应/dev/pts/目录下的文件名。
- 控制台终端：这类设备命名从/dev/ttyN（N从1~64）和/dev/console。/dev/tty0代表当前使用的终端，而/dev/console一般指向这个终端。例如初始化时登录系统用的是/dev/tty2，那么/dev/tty0指向/dev/tty2，如果切换到/dev/tty1，则/dev/tty0就指向/dev/tty1了。

通过adb shell登录手机时，其实使用的都是伪终端。图2-20所示为笔者用adb shell登录Galaxy Note 2后用tty命令打印的输出。

```
shell@android:/ $ su  
shell@android:/ # tty  
/dev/pts/1
```

图2-20 tty命令

对程序员来说，Linux提供了针对伪终端的编程接口。伪终端在概念上和管道没有太大区别，但是在实际操作时有一些特殊API需要调用。

主从两端要包含头文件：一般是在两个有亲缘关系的父子进程中使用

```
#include<stdlib.h>
```

主端需要打开/dev/ptmx设备，得到一个文件描述符

从端在使用这个文件描述符前，需要调用grantpt以获取权限，然后调用unlockpt解锁

从端通过ptsname获得从端设备文件名

从端打开由ptsname得到的从端设备

Netd中的logwrap将使用伪终端在父子进程中传递log信息。代码如下所示。

[-->logwrap.c]

```
int logwrap(int argc, const char* argv[])
{
    pid_t pid;

    int parent_ptty;
    int child_ptty;
    char child_devname[64];
    // 父进程打开/dev/ptmx
    parent_ptty = open("/dev/ptmx", O_RDWR);
    .....
    // 父进程解锁/dev/ptmx。ptsname_r将得到从端设备的名字，例如/dev/pts/1
    if (grantpt(parent_ptty) || unlockpt(parent_ptty) ||
```

```

        ptsname_r(parent_ptty, child_devname,
sizeof(child_devname)) {
    // ptsname_r是ptsname的可重入（Reentrant，可用于多线程环境）版本
    .....// 错误处理
}

pid = fork();
if (pid < 0) {
    .....// 错误处理
} else if (pid == 0) {
    // 子进程打开从端设备
    child_ptty = open(child_devname, O_RDWR);
    // 关闭从父进程那继承到的主设备文件
    close(parent_ptty);
    dup2(child_ptty, 1); // 重定向标准输入、输出、错误输出到伪终端
    的从设备
    dup2(child_ptty, 2);
    close(child_ptty);
    child(argc, argv); // 执行child函数
} else {
    int rc = parent(argv[0], parent_ptty); // 父进程执行
parent函数
    close(parent_ptty);
    return rc;
}

return 0;
}

```

为什么Netd要使用这种一般很少用的ptmx伪终端呢？结合代码中的注释，笔者认为原因有以下两个。

- 后文会介绍，Netd将fork很多子进程以执行各种任务，而这些进程的代码一般都直接来自于已有的开源代码。它们没有采用Android的ALOG宏来记录日志。所以，采用这种I/O重定向方法，这些进程的输出将全部传递到Netd进程中来。而Netd进程是可以使用 ALOG宏来打印输出的。上述代码中的parent函数将一直读取/dev/ptmx的内容，然后通过ALOG宏输出。
- 使用伪终端来承担父子进程通信重任（还可以使用别的进程间通信手段例如socketpair、pipe等）的另一个原因是伪终端不会缓存数据。例如一旦子进程往从设备中写入数据（借助I/O重定向，当子进程

往标准输出中写数据时，其实是往从设备中写数据），父进程就能读到它们。

## (2) PPP和pppd<sup>[21][22][23][24]</sup>

PPP（Point-to-Point Protocol，点对点协议）是为在同等单元之间传输数据包这样的简单链路设计的链路层协议。这种链路提供全双工操作，并按照顺序传递数据包。设计目的主要是通过拨号或专线方式建立点对点连接发送数据，使其成为各种主机、网桥和路由器之间简单连接的一种共通的解决方案。

PPP提供了一整套方案来解决链路建立、维护、拆除、上层协议协商、认证等问题，它主要包含以下几个部分。

- 链路控制协议（Link Control Protocol, LCP）：负责创建、维护或终止一次物理连接。
- 网络控制协议（Network Control Protocol, NCP）：包含一簇协议，负责解决物理连接上运行什么网络协议，以及解决上层网络协议发生的问题。
- 认证协议：最常用的包括口令验证协议（Password Authentication Protocol, PAP）和挑战握手验证协议（Challenge-Handshake Authentication Protocol, CHAP）。使用时，客户端会将自己的身份发送给远端的接入服务器。期间将使用认证协议避免第三方窃取数据或冒充远程客户接管与客户端的连接。

pppd（PPP Daemon的缩写）是运行PPP协议的后台进程。它和Kernel中的PPP驱动联动，以完成在直连线路（DSL、拨号网络）上建立IP通信链路的工作。读者可简单认为有了pppd，就可以在直连线上传输IP数据包（类似以前我国大部分家庭用Modem和电话线上网）。

## 2. ListTtysCmd和PppdCmd命令分析

这两个命令非常简单，ListTtysCmd用于枚举系统中所有的tty设备，这是通过列举/sys/class/tty目录中以tty开头的文件名来完成的。

PppdCmd仅有attach和detach两个选项。其中，attach用于启动pppd进程，而detach用于杀死pppd进程。代码在PppController的attachPppd和detachPppd函数中，请读者自行阅读。

PppController启动pppd时传递的启动参数如下所示。

```
pppd-detach  \#启动后转成后台进程
devname\  #使用devname这个设备上链接到另一端的设备
115200\  #波特率设置为115200
iplocal: ipremote\  #设置本地和远端IP
ms-dns d1  \#为运行在windows的程序配置dns地址，第一个是主dns的地址
ms-dnsd2  \#第二个是从dns的地址
lcp-max-configure 99999  #这里设置LCP config request最大个数为
99999
```

配置并启动pppd后，手机就相当于一个Modem了，是不是很神奇呢？

## 2.3.6 BandwidthControlCmd和IdletimerControlCmd命令

本节介绍BandwidthControlCmd（简称bwcc）和IdletimerControlCmd（简称icc）。这两个命令都利用了iptables的扩展模块，所以相应功能基本上完全是靠iptables来实现的。

### 1. BandwidthControlCmd命令

bwcc用于Android系统中的带宽控制。目前4.2系统中的带宽控制可针对设备、某个应用。另外还可以设置预警值，当带宽使用超过该值时会收到相应的通知（见2.2.2节中的NETLINK\_NFQLOG）。

和流量控制类似，带宽控制的实现也是利用iptables。它利用了iptables中扩展模块libxt\_quota2的功能，属于iptables的高级用法。这些内容对于非从事网络管理专业工作的人来说难度相当大。考虑到这个因素，本节将把bwcc当做一个黑盒，仅介绍其提供的各项功能。想深入研究的读者可在此基础上结合参考资料进一步了解。

bwcc提供的选项如下。

- enable和disable：开启或关闭带宽控制。
- removequota、getquota、getiquota、setquota、setquotas、removequotas、removeiquota：删除、查询和添加带宽配额。选项中的’i’针对一个或多个interface。选项中的’s’代表该选项可携带多个interface参数。
- addnaughtyapps和removenaughtyapps：以uid为目标，开启或关闭单个进程的带宽控制。
- setglobalalert、removeglobalalert、setsharedalert、removesharedalert、setinterfacealert、removeinterfacealert：预警值添加/删除有关。可以设置全局（即所有网络接口，例如Wi-Fi、3G等）带宽预警值，或者单个设备（如仅针对wlan0）的带宽预警值。

- `gettetherstats`: 获取绑定 (Tether) 设备的数据统计。后文将介绍 Tether 的相关知识点。

图2-21所示为利用ndc命令为Galaxy Note 2 setglobalalert后的结果。图中为bwcc设置了全局配额是1000字节，当使用测试机下载数据超过1000字节时，将得到如图2-22所示的警告消息。

```
Chain bw_FORWARD (1 references)
target  prot opt source          destination
Chain bw_INPUT (1 references)
target  prot opt source          destination
      all  --  anywhere        anywhere      ! quota globalAlert: 1000 bytes
RETURN   all  --  anywhere        anywhere        owner socket exists
      all  --  anywhere        anywhere
Chain bw_OUTPUT (1 references)
target  prot opt source          destination
      all  --  anywhere        anywhere      ! quota globalAlert: 1000 bytes
RETURN   all  --  anywhere        anywhere        owner socket exists
      all  --  anywhere
```

图2-21 bwcc setglobalalert结果

```
shell@android:/sys # ndc monitor
[Connected to Netd]
600 Iface added p2p0
600 Iface linkstate p2p0 down
600 Iface added wlan0
600 Iface linkstate wlan0 down
600 Iface linkstate wlan0 up
600 Iface linkstate wlan0 up
600 Iface linkstate p2p0 up
600 Iface linkstate p2p0 up
600 Iface linkstate wlan0 up
601 limit alert globalAlert wlan0
```

图2-22 ndc monitor得到的警告消息

图2-22所示最后一行打印了来自Kernel的qlog UEvent消息，以通知在wlan0设备上数据流量已超过配额。

提示 bwcc应该是Netd中难度最大的模块了，其难点是如何利用iptables进行带宽控制。相比其内部实现而言，掌握bwcc的功能对绝大多数Android开发者来说也许更加实用。

## 2. IdletimerControlCmd命令

icc利用了iptables另一个扩展模块libxt\_idletimer，其对应的iptables命令格式如下。

```
iptables -t raw -A idletimer -i nic -j IDLETIMER --timeout --label
label --send_nl_msg 1
```

其中，各个参数的含义如下。

- `timeout`: 超时时间，单位是秒。
- `label`: 用来标示该rule的唯一名字。
- `send_nl_msg`: 如果超时，则通过`/sys/net/xt_idletimer`触发UEvent消息（回顾2.2.2节中关于“`xt_idletimer`”UEvent消息的介绍）。

**提示** 奇怪的是，笔者在模拟器和Galaxy Note 2上利用ndc测试icc命令均不能添加超时规则。而同样的命令放到Ubuntu 12.04上却工作正常（但Ubuntu上的iptable却不支持`--send_nl_msg`选项）。看来这个新颖的功能在目前的系统中支持得还不够好。

## 2.3.7 NatCmd命令

NatCmd和NAT相关，本节首先介绍NAT的背景知识。

### 1. 背景知识介绍<sup>[25]</sup>

在传统TCP/IP通信过程中，所有的路由器仅仅是充当一个中间人的角色，也就是通常所说的存储转发。即路由器不会对转发的数据包进行修改。准确地讲，除了将源MAC地址换成自己的MAC地址以外，路由器不会对转发的数据包做任何修改。而NAT恰恰是出于某种特殊需要而对数据包的源IP地址、目的IP地址、源端口、目的端口进行改写的操作。

什么情况下需要NAT（Network Address Translation，网络地址转换）呢？假设想在公司内网搭建一个WWW服务器以对外发布信息。由于公司内网使用的都是内部IP，故无法向外网发布该服务。这时就可通过NAT来解决这个问题。比如网管可在防火墙的外部网卡上绑定多个合法IP地址，然后通过NAT技术使发给其中某一个IP地址的包转发至这个内部的WWW服务器上，然后再将该内部WWW服务器的响应包伪装成该合法IP发出的包。

NAT分为两种，分别是源NAT（SNAT）和目的NAT（DNAT），顾名思义，SNAT就是改变转发数据包的源地址，DNAT就是改变转发数据包的目的地址。Linux系统上的NAT操作是通过iptables的nat表来完成，该表有三条默认Chain，它们分别如下。

- PREROUTING：可在此定义目的NAT的规则，因为路由器进行路由时只检查数据包的目的IP地址，所以为了使数据包得以正确路由，必须在路由之前就进行目的NAT。
- POSTROUTING：可以在这里定义进行源NAT的规则，系统在决定了数据包的路由以后再执行该链中的规则。
- OUTPUT：定义对本地产生的数据包的目的NAT规则。

此处给出使用iptables进行NAT转换的两个例子。

```
//更改所有来自192.168.1.0/24的数据包的源IP地址为1.2.3.4:  
iptables-t nat-A POSTROUTING-s 192.168.1.0/24-o eth0-j SNAT--to  
1.2.3.4  
//更改所有来自192.168.1.0/24的数据包的目的IP地址为1.2.3.4:  
iptables-t nat-A PREROUTING-s 192.168.1.0/24-i eth1-j DNAT--to  
1.2.3.4
```

## 2. NatCmd命令使用

NatCmd仅支持“enable”和“disable”两个命令选项。不过和上一节介绍的不同，Android中的NAT并不是只利用iptables的nat表来做转换，而是借助ip route命令和iptables的filter表在源和目标网络设备及指定IP地址之间进行地址转换。以“enable”选项为例，其使用方式如下。

```
ndc nat enable lo wlan0 1 202.106.25.35/14
```

其中：

- lo为本地回环设备、代表输入设备。
- wlan0为无线NIC，代表输出设备。
- 1代表后面的地址组合只有一个。
- 202.106.25.35/14指明IP地址和子网掩码。

以上面的命令为例，NatCmd执行它所涉及的iptables命令的调用序列如下所示。

```
iptables-A natctrl_FORWARD-i eth0-o lo-m state--state  
ESTABLISHED, RELATED-j RETURN  
iptables-A natctrl_FORWARD-i lo-o eth0-m state--state INVALID-j  
DROP  
iptables-A natctrl_FORWARD-i lo-o eth0-j RETURN  
iptables-D natctrl_FORWARD-j DROP  
iptables-A natctrl_FORWARD-j DROP  
iptables-t nat-A natctrl_nat_POSTROUTING-o eth0-j MASQUERADE  
#MASQUERADE能自动读取eth0现在的IP地址然后做SNAT  
#这样就避免了每次eth0地址发生改变时都需更新iptables的烦恼
```

注意 上边的命令中并没有利用传递的IP地址，这是因为这些地址和ip命令的多路由策略管理有关，只有使用了多路由策略管理，这些IP地址才能被ip命令用上。由于代码中没有任何说明，所以笔者也很难理解这部分内容。如果有知晓的读者，还请和大家一起分享相关知识。

根据上面的iptables调用命令，读者也能猜测出enableNat的目的是修改来自源设备的数据包，使它看起来是目标设备发出的数据包。

图2-23所示为利用ndc在Galaxy Note 2测试nat命令的结果。

```
shell@android:/ # iptables -S natctrl_FORWARD
-N natctrl FORWARD
FIX ME! implement getprotobyname() bionic/libc/bionic/stubs.c:484
-A natctrl_FORWARD -p tcp -m tcp --tcp-flags SYN,RST SYN -j TCPMSS --clamp-mss-to-pmtu
-A natctrl_FORWARD -i wlan0 -o rmnet0 -m state --state RELATED,ESTABLISHED -j RETURN
-A natctrl_FORWARD -i rmnet0 -o wlan0 -m state --state INVALID -j DROP
-A natctrl_FORWARD -i rmnet0 -o wlan0 -j RETURN
-A natctrl_FORWARD -j DROP
```

图2-23 iptables查看natctrl\_FORWARD规则

图2-23所示为利用ndc nat enable rmnet0 wlan0 0添加nat规则后，再通过iptables-S查看natctrl\_FORWARD链的具体规则时得到的结果。

注意 其中TCPMSS这条规则应该是三星公司自己添加的，不过令人匪夷所思的是该命令竟然打出了“FIX ME！”这样的输出。看来，网络管理的确是一个很有技术含量的活。

## 2.3.8 TetherCmd和SoftapCmd命令

TetherCmd和SoftapCmd命令都和手机中一项名为绑定（Tether）的功能相关。简单来说，绑定功能即把手机当成Modem用。智能手机一般都有多种连接网络的方式，例如使用Wi-Fi或3G。在某些环境下如高铁列车上，差旅人士只要把手机接到笔记本上，然后开启3G和Tether，笔记本就可以利用手机上网了（在智能机普及前，类似的场景中就需要使用3G上网卡）。

另外，如果手机中的无线网络设备支持Soft AP（Soft Access Point，软件实现的接入点）功能，还可以通过Softap命令将手机变成一个AP（可以把它看成是一个无线路由器）。

目前Android 4.2系统支持以下三种方式的绑定。

- Soft AP：利用Wi-Fi无线网络的特性，开启手机Soft AP功能。主机和手机间通过Wi-Fi通信。
- Bluetooth：主机（PC或笔记本电脑）和手机通过蓝牙协议通信。
- USB：主机和手机通过USB协议通信。手机相当于一个USB上网卡。

本节主要介绍TetherCmd中的USB绑定和Softap命令。其余内容我们将留给读者自行研究。

提示 TetherCmd还支持所谓的逆绑定（reverse tethering），即手机借助主机上网。这部分内容请读者自行研究。

### 1. TetherCmd命令

本节仅介绍利用USB实现Tether的功能，其中涉及RNDIS以及DHCP相关的背景知识，我们先来介绍它们。

#### （1）背景知识介绍<sup>[26][27][28][29][30]</sup>

RNDIS（Remote Network Driver Interface Specification）是微软公司的，主要用于Windows平台中USB网络设备的驱动开发。RNDIS的协

议栈如图2-24所示。

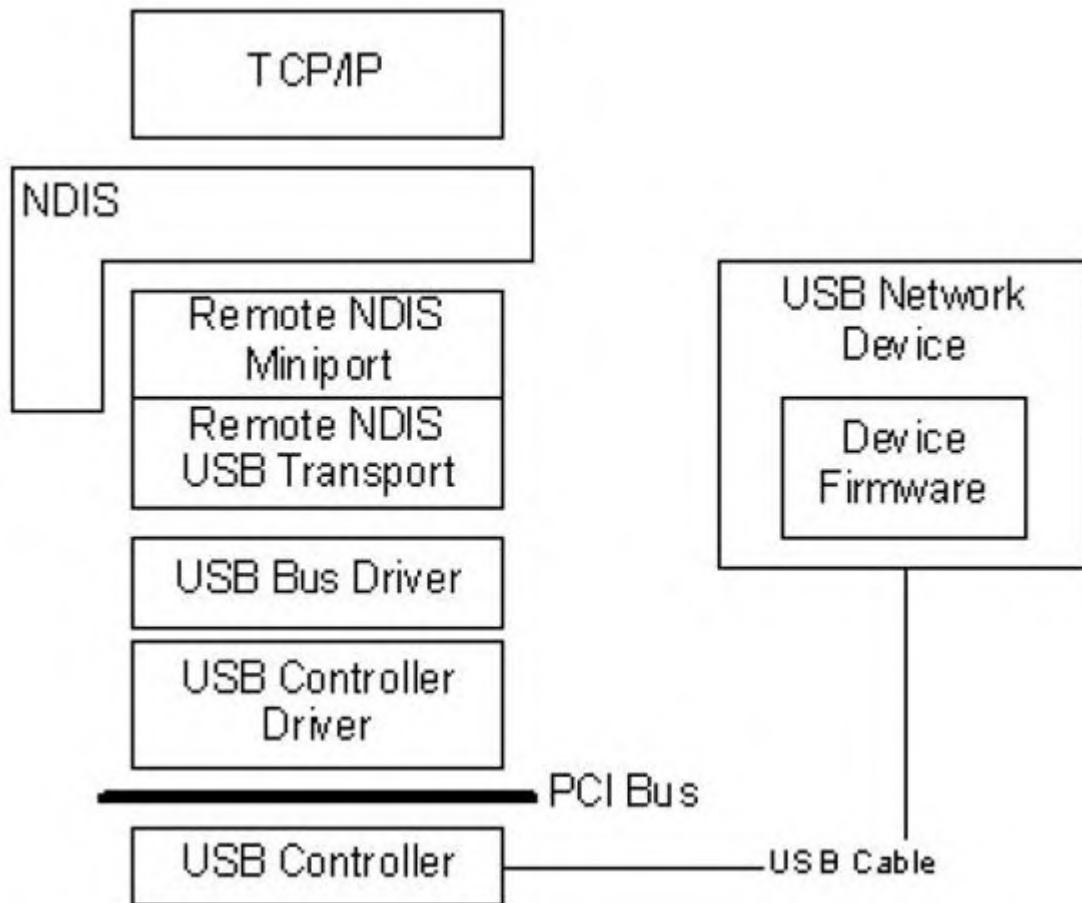


图2-24 RNDIS协议栈

RNDIS的作用是简化Windows平台上USB网络设备驱动开发的流程。此处不讨论相关内容，感兴趣的读者可阅读本章最后列出的参考资料。

RNDIS和Android有什么关系呢？当手机通过USB连接到主机（主机一般运行Windows系统）后，如果要启用USB绑定，必须要把手机的USB设置成RNDIS（绝大部分厂商的手机都是这么实现的）。这样，主机上的OS就能识别到一个新的网卡。然后用户就可以选择使用它来开展网络操作了。

**提示** 本书后续章节将讨论Android平台上USB的相关功能。

假设用户选择使用这个通过USB绑定的网卡，下一步要做的就是给主机分配IP地址了。此处涉及DHCP协议。

DHCP (Dynamic Host Configuration Protocol, 动态主机配置协议) 的前身是BOOTP。BOOTP原本是用于无磁盘主机连接的网络上的，网络主机使用BOOT ROM而不是磁盘启动并连接上网络，BOOTP则可以自动地为那些主机设定TCP/IP环境。DHCP是BOOTP的增强版本，它分为两个部分。

- 服务器端：所有的IP网络设定数据都由DHCP服务器集中管理，并负责处理客户端的DHCP要求。

- 客户端：客户端会使用从服务器分配下来的IP地址等配置信息。

根据上述介绍，相信读者很容易想到，在USB绑定中，主机将是DHCP的客户端，而手机是DHCP的服务器端。那么在Android系统中，DHCP服务器端是谁呢？同Pppd类似，Android也使用了另外一个开源软件DNSmasq来充当DHCP服务器。

DNSmasq是一个用于配置DNS和DHCP的工具，小巧且方便，适用于小型网络，它提供了DNS功能和可选择的DHCP功能。它服务只在本地适用的域名，这些域名是不会在全球的DNS服务器中出现的。DHCP服务器和DNS服务器结合，并且允许DHCP分配的地址能在DNS中正常解析，而这些DHCP分配的地址和相关命令可以配置到每台主机中，也可以配置到一台核心设备中（如路由器）。

DNSmasq适用于拥有NAT的家庭网络、用Modem、ADSL设备连接到互联网等环境。对于需求低资源消耗且配置方便简单的小型网络（最多可支持1000台主机）是一个很好的选择。

## (2) TetherCmd命令使用

Android中启动USB Tether功能将涉及Framework层多个模块，其详细过程留待后续章节介绍。此处读者仅需从TetherCmd角度考虑其中的两个主要步骤。

1) 添加需要Tether的接口。对USB绑定来说，接口名为rndis0。对应的处理函数是TetherController的tetherInterface，代码如下所示。

```
[-->TetherController.cpp: : tetherInterface]
```

```
int TetherController::tetherInterface(const char *interface) {
    mInterfaces->push_back(strdup(interface));
    // 把需要interface名字保存到一个链表即可
    return 0;
}
```

tetherInterface的功能很简单，就是保存需要Tether的设备名。这一步其实没有太多实质性的内容。

2) 通过"start"选项启动Tether。这个步骤将触发TetherController的startTethering函数被调用。该函数的主要功能就是配置dnsmasq的启动参数并启动它。这部分代码比较简单，dnsmasq的启动参数如下所示。

```
dnsmasq\
--keep-in-foreground\#前台运行
--no-resolv\#不解析/etc/resolv.conf，该文件记录域名和dns服务器的一些信息
--no-poll\#不关注/etc/resolv.conf文件的变化
--dhcp-option-force=43, ANDROID_METERED\#强制的dhcp选项。客户端和dnsmasq交互时，首先
#会获取dhcp服务器的一些配置信息。43是DHCP协议中定义的option的一种，代表
vendor specific
#infomation该选项说明vendor specific information就是
ANDROID_METERED
--pid-file\#指定dnsmasq记录自己进程id (pid) 到某个文件。默认
是/var/run/dnsmasq.pid
--dhcp-range=192.168.1.2 192.168.1.100 1h\#该选项开启dnsmasq的
dhcp服务功能。分配的IP地址
#位于192.168.1.2和192.168.1.100之间。1h代表租约时间为1小时。租约时间即
某IP地址可以被DHCP
#客户端使用的时间。如果超过租约时间，dnsmasq必须为该客户端重新分配IP
```

这两步完成后，USB绑定功能中和TetherCmd相关的任务就完成了。从整个绑定过程来看，涉及应用（例如Settings提供的设置功能）、网络模块、USB模块、驱动等，是一个非常复杂的过程。

**提示** 这个过程对软件开发者来说也是一个挑战，只有对USB Tether涉及的各个模块都有相应了解，碰到问题时候才能快速定位和解决它。

## 2. SoftapCmd命令

Softap命令和Wi-Fi有紧密关系。本节先简单介绍Soft AP相关的背景知识，后续章节将对Wi-Fi开展深入讨论。

### (1) 背景知识介绍<sup>[31][32][33]</sup>

Soft AP代表通过软件实现Access Point的功能。那么AP是什么？AP和Soft AP有什么不同？

在Wi-Fi无线技术规范中，AP和Station是其中的两个基本概念。

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

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

以前面提到的高铁列车上的应用场景为例，除了用USB绑定外，还可以打开笔记本和手机的Wi-Fi，并启动手机的Soft AP功能。这样，手机一方面用3G接入互联网，另一方面又利用Soft AP向笔记本提供Wi-Fi接入功能。

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

### (2) SoftapCmd命令使用

和TetherCmd类似，开启Android中手机的Soft AP功能将涉及大量Framework层中的操作，本节仅关注和Netd相关的三个步骤。

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

[-->SoftapController.cpp: : fwReloadSoftap]

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

    .....// 参数检测
    iface = argv[2];
    if (strcmp(argv[3], "AP") == 0) {
        fwpath = (char *)wifi_get_fw_path(WIFI_GET_FW_PATH_AP);
    } else if (strcmp(argv[3], "P2P") == 0) {
        fwpath = (char
*)wifi_get_fw_path(WIFI_GET_FW_PATH_P2P);
    } else {
        fwpath = (char
*)wifi_get_fw_path(WIFI_GET_FW_PATH_STA);
    }
    // 通过往/sys/module/wlan/parameters/fwpath文件中写入固件名
    // 触发驱动去加载对应的固件
    ret = wifi_change_fw_path((const char *)fwpath);
    .....
    return ret;
}
```

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

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

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

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

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

[-->SoftapController.cpp: : setSoftap]

```
int SoftapController::setSoftap(int argc, char *argv[]) {
    char psk_str[2*SHA256_DIGEST_LENGTH+1];
    int ret = 0, i = 0, fd;
    char *ssid, *iface;

    .....// 参数检查

    iface = argv[2];

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

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

    asprintf(&wbuf,
"interface=%s\ndriver=nl80211\nctrl_interface="
"/data/misc/wifi/hostapd\nssid=%s\nchannel=6\nieee80211n=1\n",
        iface, ssid);

    if (argc > 4) { // 判断AP的加密类型
        if (!strcmp(argv[4], "wpa-psk")) {
            generatePsk(ssid, argv[5], psk_str);
        }
    }
}
```

```

        asprintf(&fbuf, "%swpa=1\nwpa_pairwise=TKIP
CCMP\nwpa_psk=%s\n",
                  wbuf, psk_str);
    } else if (!strcmp(argv[4], "wpa2-psk")) {
        generatePsk(ssid, argv[5], psk_str);
        asprintf(&fbuf,
"%swpa=2\nrsn_pairwise=CCMP\nwpa_psk=%s\n",
                  wbuf, psk_str);
    } else if (!strcmp(argv[4], "open")) {
        asprintf(&fbuf, "%s", wbuf);
    }
}
.....  

// HOSTAPD_CONF_FILE指向/data/misc/wifi/hostapd.conf文件
fd = open(HOSTAPD_CONF_FILE, O_CREAT | O_TRUNC | O_WRONLY,
0660);
.....  

if (write(fd, fbuf, strlen(fbuf)) < 0) {
    ALOGE("Cannot write to \'%s\': %s", HOSTAPD_CONF_FILE,
strerror(errno));
    ret = -1;
}
.....// 修改该文件的读写权限等
return ret;
}

```

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

Android4.2/hardware/ti/wlan/mac80211/config目录中有一个hostapd.conf文件，其内容如下所示。

[-->hostapd.conf]

```

driver=n180211                                #指定Wi-Fi驱动的名称
.....#略去部分内容
ssid=AndroidAP                                  #设置接入点名称为AndroidAP
country_code=US
wep_rekey_period=0
eap_server=0
own_ip_addr=127.0.0.1
wpa_group_rekey=0                               #加密方式等设置
wpa_gmk_rekey=0
wpa_ptk_rekey=0

```

```
interface=wlan1          #网络设备接口  
.....#略去部分内容
```

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

3) 最后，SoftapController的startap函数被调用，它将启动hostapd进程。重点关注hostapd启动的参数信息，如下所示。

```
hostapd\  
-e /data/misc/wifi/entropy.bin    \和Wi-Fi协议中的信息加密有关  
/data/misc/wifi/hostapd.conf      \hostapd的配置文件
```

## 2.3.9 ResolverCmd命令

ResolverCmd和Android系统中DNS的实现有关，用于给不同NIC设备配置不同的DNS。其主要支持四个选项。

- `setdefaultif`: 设置DNS查询时默认的NIC。和Android中DNS的实现有关。
- `setifdns`: 设置不同NIC的DNS配置信息。
- `flushdefaultif`、`flushif`: 清空默认或某个NIC的DNS配置信息。

**提示** 感兴趣的读者可结合Android系统中DNS的实现来理解ResolverCmd的功能。

# 读累了记得休息一会儿哦~

公众号：古德猫宁李

- 电子书搜索下载
- 书单分享
- 书友学习交流

网站：[沉金书屋 https://www.chenjin5.com](https://www.chenjin5.com)

- 电子书搜索下载
- 电子书打包资源分享
- 学习资源分享

## 2.4 NetworkManagementService介绍

根据前文所述，NetworkManagementService（以后简称NMService）将通过“netd”socket和Netd交互。NMService代码非常简单，首先来看其创建的代码，如下所示。

[-->SystemServer.java: : ServerThread: run]

```
public void run() {
    .....// 其他Service的创建及相关处理
    try {
        networkManagement =
NetworkManagementService.create(context);

ServiceManager.addService(Context.NETWORKMANAGEMENT_SERVICE,
networkManagement);
    } catch .....
    .....
    final NetworkManagementService networkManagementF =
networkManagement;
    if (networkManagementF != null)
networkManagementF.systemReady();
    .....
}
```

ServerThread是Android Java Framework的中枢，绝大部分重要Service都在该线程中创建，例如ActivityManagerService、WindowManagerService、PackageManagerService以及本书要介绍的WifiService、WifiP2pService等。

ServerThread中和NMService相关的重要知识点仅create和systemReady两个函数。下面将一一介绍。

**提示** 关于ServerThread的详细信息，请读者阅读《深入理解Android：卷II》。

## 2.4.1 create函数详解

create函数的代码如下所示。

```
[-->NetworkManagementService.java: : create]

public static NetworkManagementService create(Context context)
        throws InterruptedException {
    // 创建一个NMService对象
    final NetworkManagementService service = new
NetworkManagementService(context);
    final CountDownLatch connectedSignal =
service.mConnectedSignal;
    service.mThread.start(); // 启动NMService中的一个线程
    // connectedSignal用于等待某个事情的发生。此处是等待mThread完成初始化工作
    connectedSignal.await();
    return service;
}
```

create函数非常简洁，其主要工作就是创建一个NMService对象并启动其中一个线程。create返回前需要确保mThread线程完成初始化工作。下面来看看NMService的构造函数。

```
[-->NetworkManagementService.java]
```

```
private NetworkManagementService(Context context) {
    mContext = context;
    // 对模拟器的处理
    if
("simulator".equals(SystemProperties.get("ro.product.device")))
return;
/*
```

NativeDaemonConnector是Java Framework中一个特别的类，它用于连接指定的socket，并发送和接收

socket数据。

此处，“netd”参数代表目标socket。NetdCallbackReceiver为具体的socket连接及消息处理对象。

1.当Netd连接成功后，NetdCallbackReceiver的onDaemonConnected函数将被调用。

2.当收到来自Netd的数据后，NetdCallbackReceiver的onEvent函数将被调

用。

```
NativeDaemonConnector代码比较简单，感兴趣的读者不妨自行阅读。  
*/  
mConnector = new NativeDaemonConnector(  
    new NetdCallbackReceiver(), "netd", 10,  
    NETD_TAG, 160);  
// 创建一个线程，其Runnable对象就是mConnector  
mThread = new Thread(mConnector, NETD_TAG);  
/*  
把自己添加到Watchdog中的监控队列中。这样，NMService将受到Watchdog的  
监控，一旦NMService  
出现异常，Watchdog将自杀以重启Android Java World。对Watchdog感兴趣的  
读者不妨阅读《深  
入理解Android：卷I》4.5.3节“Watchdog分析”。  
*/  
Watchdog.getInstance().addMonitor(this);  
}
```

对上述代码来说，最重要的是NetdCallbackReceiver，下面来看看。

[-->NetworkManagementService.java: : NetdCallbackReceiver]

```
private class NetdCallbackReceiver implements  
INativeDaemonConnectorCallbacks {  
    public void onDaemonConnected() {  
        if (mConnectedSignal != null) {  
            // 通知NMService构造函数中的connectedSignal.await()返回  
            mConnectedSignal.countDown();  
            mConnectedSignal = null;  
        } else { // mMainHandler和mThread线程绑定  
            mMainHandler.post(new Runnable() {  
                public void run() {  
                    prepareNativeDaemon(); // 下节  
                }  
            });  
        }  
    }  
    // 处理来自Netd的消息  
    public boolean onEvent(int code, String raw, String[]  
cooked) {  
        switch (code) { // 目前NMService只处理下面三种Command对应  
        的消息  
            case NetdResponseCode.InterfaceChange: // 对应  
InterfaceCmd  
                ..... // 略去具体的处理逻辑
```

介绍

```
        case NetdResponseCode.BandwidthControl:// 对应
BandwidthControlCmd
        .....
        case NetdResponseCode.InterfaceClassActivity:// 和
IdletimerCmd有关
        .....
        default: break;
    }
    return false;
}
}
```

create及相关函数都比较简单，此处不详述，下面来看systemReady。

## 2.4.2 systemReady函数详解

systemReady函数详解如下所示。

[-->NetworkManagementService.java: : systemReady]

```
public void systemReady() {  
    prepareNativeDaemon();  
}
```

prepareNativeDaemon用于将系统中一些与带宽控制、防火墙相关的规则发送给Netd去执行，其代码如下所示。

[-->NetworkManagementService.java: : prepareNativeDaemon]

```
private void prepareNativeDaemon() {  
    mBandwidthControlEnabled = false;  
    // 判断kernel是否支持bandwidthcontrol  
    final boolean hasKernelSupport = new  
File("/proc/net/xt_qtaguid/ctrl").exists();  
    if (hasKernelSupport) {  
        try {  
            mConnector.execute("bandwidth", "enable");// 使能  
bandwidth功能  
            mBandwidthControlEnabled = true;  
        } catch .....  
    }  
    // 设置Android系统属性"net.qtaguid_enabled"  
    SystemProperties.set(PROP_QTAGUID_ENABLED,  
mBandwidthControlEnabled ? "1" : "0");  
    // 设置Bandwidth规则  
    synchronized (mQuotaLock) {  
        int size = mActiveQuotas.size();  
        if (size > 0) {  
            // mActiveQuotas保存了每个interface的配额设置  
            final HashMap<String, Long> activeQuotas =  
mActiveQuotas;  
            mActiveQuotas = Maps.newHashMap();  
            for (Map.Entry<String, Long> entry :  
activeQuotas.entrySet())  
                setInterfaceQuota(entry.getKey(),  
entry.getValue());
```

```
        }
        .....// 其他规则
    }
// 设置防火墙规则
setFirewallEnabled(mFirewallEnabled ||
LockdownVpnTracker.isEnabled());
}
```

以setFirewallEnabled函数为例，它和Netd交互的方法如下。

[-->NetworkManagementService.java: : setFirewallEnabled]

```
public void setFirewallEnabled(boolean enabled) {
    enforceSystemUid();
    try {// 发送firewall相关的Command给Netd
        mConnector.execute("firewall", enabled ? "enable" :
"disable");
        mFirewallEnabled = enabled;
    } catch .....
}
```

systemReady函数非常简单，本章就不详述了。

## 2.5 本章总结和参考资料说明

### 2.5.1 本章总结

本章对Netd进行了详细讨论。相信读者读完此章的感受是代码这么容易的模块，竟然涉及如此复杂的背景知识。从代码上看也许它们并不复杂，但是其背后的理论知识却可能大有来头。对于这些内容而言，代码只是外在的表现形式，其核心一定在其背后的那些知识中。所以，读者在阅读专题卷的时候，一定要考察自己是否对背景知识有所掌握。

Netd涉及的内容和网络管理与控制有关，例如DNS、Apple Bonjour、利用iptables等工具实现NAT、防火墙、带宽控制、流量控制、路由控制功能，以及USB绑定Wi-Fi、Soft AP等。请读者在本章的参考资料中找到并继续研究自己感兴趣的内容。

最后，对NetworkManagementService进行了介绍。NMService的内容非常简单。

## 2.5.2 参考资料说明

### 1. Linux PF\_NETLINK相关资料

[1] Linux man PF\_NETLINK

说明：本文档是Linux系统中的帮助文档。从总体上介绍了PF\_NETLINK (AF\_NETLINK) 的作用和相关的数据结构。对高手比较适用。

[2] <http://www.linuxjournal.com/article/8498>

说明：“Manipulating the Networking Environment Using RTNETLINK”，这篇文章以RTNETLINK为主要对象，介绍了如何利用它进行编程以操作网络。此文写得非常详细，建议读者深入阅读，并且自己动手写测试例子。

### 2. DNS、Apple Bonjour相关资料

[3] <http://baike.baidu.com/view/22276.htm>

说明：百度百科中关于DNS的介绍，属于入门级材料，初学者可以先了解相关知识。

[4] <http://en.wikipedia.org/wiki/MDNS>

说明：维基百科中关于Multicast DNS的介绍。入门级材料，包含的信息不是很全，需要跟踪其中的链接才能对MDNS有全面了解。

[5]

[https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/NetServices/Introduction.html#/apple\\_ref/doc/uid/TP40002445-SW1](https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/NetServices/Introduction.html#/apple_ref/doc/uid/TP40002445-SW1)

说明：“Introduction to Bonjour Overview”，苹果开发网站上关于Bonjour基础知识的入口，包含“About Bonjour”、“Bonjour API Architecture”等文档。

[6]

[https://developer.apple.com/library/mac/#documentation/Networking/Conceptual/dns\\_discovery\\_api/Introduction.html#/apple\\_ref/doc/uid/TP30000964](https://developer.apple.com/library/mac/#documentation/Networking/Conceptual/dns_discovery_api/Introduction.html#/apple_ref/doc/uid/TP30000964)

说明：“DNS Service Discovery Programming Guide”，苹果开发网站关于NSD API的说明。

### 3. iptables相关资料

iptables的相关文档非常多，虽然Linux也提供了帮助文档（man iptables），但对新手来说该文档实在不是学习的好资料。

[7] <http://www.thegeekstuff.com/2011/01/iptables-fundamentals/>

说明：“Linux Firewall Tutorial: IPTables Tables, Chains, Rules Fundamentals”，这篇文章首先从原理上介绍了如何理解iptables，然后介绍了相关的例子。笔者认为这是iptables最好的入门资料。

[8] <http://selboo.com/post/721/>

说明：“iptables的相关概念和数据包的流程”，这篇文档介绍了iptables中各个Table及Chain的处理顺序，请读者结合参考资料[7]来理解iptables。

[9] <http://www.frozenthux.net/iptables-tutorial/cn/iptables-tutorial-cn-1.1.19.html>

说明：“Iptables指南1.1.19”，这篇文档介绍的iptables版本比较旧（Android 4.2使用的iptables版本是1.4.11），但对iptables常用参数都有非常详细的介绍。适合已经入门的读者进行深入阅读。

### 4. tc相关资料

tc文献的数量和难度远大于iptables，此处精选几个必读文献。

[10] <http://linux-ip.net/articles/Traffic-Control-HOWTO/intro.html>

说明：“Traffic Control HOWTO”，理解tc的必读文献，覆盖面很广，理论知识讲解到位。难度稍大，需要仔细琢磨才能完全理解。

[11]

<http://wenku.baidu.com/view/f02078db50e2524de5187e45.html>

说明：“TC（Linux下流量控制工具）详细说明及应用实例”，百度文库中的一篇文档，篇幅虽然不长，但也做到了理论和实例结合。建议读者先阅读此文献，然后再深入研究参考资料[10]。

[12]

<http://fanqiang.chinaunix.net/a1/b1/20010811/0705001103.html>

说明：“在Linux中实现流量控制器”，一篇博文，主要对tc命令的用法列举了不少示例，属于tc的实战文章，建议放到最后阅读。

[13]

<http://www.linuxfoundation.org/collaborate/workgroups/networking/ifb>

说明：这是笔者能找到的关于IFB设备最完整的资料，对IFB的使用、常规用法等进行了全方位的介绍。

## 5. ip命令相关资料

[14] <http://blog.chinaunix.net/uid-24921475-id-2547198.html>

说明：“Linux ip命令介绍”。ip命令比较简单，这里仅给出一篇文献。

## 6. NetDevice编程文献

[15] Linux man netdevice

说明：非常详细的NetDevice编程介绍，建议读者认真阅读。

## 7. Linux策略路由相关资料

[16] <http://www.cnblogs.com/iceocean/articles/1594488.html>

说明：“Linux策略路由”，中文文档，知识面覆盖较全，属于入门级资料。

[17]

<http://www.policyrouting.org/PolicyRoutingBook/ONLINE/TOC.html>

说明：《Policy Routing With Linux》，这是一本书籍。个人感觉参考资料[16]是本书的学习总结，属于高级阅读材料，难度较大。

## 8. Linux IPv6控制相关资料

[18] <http://www.ipsidixit.net/2012/08/09/ipv6-temporary-addresses-and-privacy-extensions/>

说明：“IPv6 temporary addresses and privacy extensions”，介绍Linux中IPv6临时地址和privacy extensions方面的知识，知识覆盖面较全，属于入门资料。

## 9. TTY和ptmx编程相关资料

[19] <http://tldp.org/HOWTO/Text-Terminal-HOWTO.html>

说明：“Text-Terminal-HOWTO”，比较旧的资料，覆盖面非常广。读者可阅读自己想了解的章节。

[20] [http://blog.tianya.cn/blogger/post\\_read.asp?BlogID=3616841&PostID=33399981](http://blog.tianya.cn/blogger/post_read.asp?BlogID=3616841&PostID=33399981)

说明：“Linux下tty/pty/pts/ptmx详解”，很好的中文材料，还列出了参考文献。最后，关于ptmx，读者还可通过man ptmx获得如何用它进行编程的指导。

## 10. PPP和pppd相关资料

[21] <http://tldp.org/HOWTO/PPP-HOWTO/>

说明：“Linux PPP HOWTO”，Linux HowTo系列的内容都简单易懂。章节较多，但很多章节仅一两句内容，可做入门参考。

[22] <http://network.51cto.com/art/201009/223784.htm>

说明：“基础解读PPP协议”，中文文档，一页内容，主要介绍PPP框架性的内容。

[23]

<http://wenku.baidu.com/view/0c395f15866fb84ae45c8d4a.html>

说明：“PPP介绍”，百度文库中的一个关于PPP的PPT。内容翔实，不仅介绍了PPP协议的数据包，也从框架上介绍了PPP的工作流程。建议读者首先阅读此文献。

[24] Linux man pppd

说明：介绍Pppd中各个选项的作用。

## 11. NAT相关资料

[25]

<http://oa.jmu.edu.cn/netoa/libq/pubdisc.nsf/66175841be38919248256e35005f4497/7762e8e1056be98f48256e88001ef71d>  
OpenDocument

说明：“用iptables实现NAT”，中文文档，简单易懂。

## 12. Tether、RNDIS、DHCP、DNSmasq相关资料

[26] <http://en.wikipedia.org/wiki/Tethering>

说明：“Tethering”，维基百科中关于Tether的介绍，浅显易懂，属于普及型资料。

[27] <http://msdn.microsoft.com/en-us/library/windows/hardware/gg463293.aspx>

说明：“Remote NDIS (RNDIS) and Windows”，MSDN文档，非常详实（不得不说微软在文档方面的工作真的是一丝不苟）。

[28] [http://baike.baidu.com/view/7992.htm  
subLemmaId=7992&fromenter=%A3%C4%A3%C8%A3%C3%A3%D0](http://baike.baidu.com/view/7992.htm?subLemmaId=7992&fromenter=%A3%C4%A3%C8%A3%C3%A3%D0)

说明：百度百科中关于DHCP的解释，入门资料。

[29] <http://baike.baidu.com/view/6681631.htm>

说明：百度百科中关于DNSmasq的解释。

[30]  
<http://wenku.baidu.com/view/662b536b561252d380eb6ec1.html>

说明：关于DHCP协议中option字段的详细介绍。

### 13. Softap和hostapd相关资料

[31] 《802.11无线网络权威指南（中文第2版）》

说明：读者可先阅读第1、2章中关于Wi-Fi技术的一些基本概念，例如AP和Station。

[32] <http://baike.baidu.com/view/2475889.htm>

说明：百度百科关于Soft AP的入门级介绍。

[33] 关于hostapd，读者可利用man hostapd得到各个选项的用法。  
提示，读者必须先安装hostapd，然后才能查阅其帮助文档。

# 第3章 Wi-Fi基础知识

本章所涉及的源代码文件名及位置

- wireless.h    external/kernel-headers/original/linux/wireless.h
- driver\_wext.c  
external/wpa\_supplicant\_8/src/drivers/driver\_wext.c
- netlink.h    external/kernel-headers/original/linux/netlink.h
- driver\_nl80211.c  
external/wpa\_supplicant\_8/src/drivers/driver\_nl80211.c
- nl80211\_copy.h  
external/wpa\_supplicant8/wpa\_supplicant/src/drivers/nl80211\_copy.h

## 3.1 概述

Wi-Fi (Wireless Fidelity) 是一个无线网络通信技术的品牌，由Wi-Fi联盟 (Wi-Fi Alliance, WFA) 拥有。WFA专门负责Wi-Fi认证与商标授权工作。严格地说，Wi-Fi是一个认证的名称，该认证用于测试无线网络设备是否符合IEEE 802.11系列协议的规范。通过该认证的设备将被授予一个名为Wi-Fi CERTIFIED的商标。不过，随着获得Wi-Fi认证的设备普及，人们也就习以为常得称无线网络为Wi-Fi网络了。

**提示** IEEE 802.11规范和Wi-Fi的关系很难用一两句话说清楚，读者可阅读参考资料[1]进行了解。简单来说，IEEE 802.11是无线网络技术的官方标准，而WFA则参考802.11规范制订了一套Wi-Fi测试方案（Test Plan）。不过，Test Plan和802.11的内容并不完全一致。有些Test项包含了目前802.11还未涉及的内容。另外，Test Plan也未覆盖802.11所有内容。所以参考资料[1]把Wi-Fi定义为802.11规范子集的扩展。

本章从以下两方面向读者介绍Wi-Fi技术。

- IEEE 802.11协议涉及的理论知识，包括无线频谱资源、IEEE 802.11/802.X协议中的相关内容、关键概念（如Access Point）、MAC帧、无线网络安全等。
- 如何在Linux系统中通过Linux Wireless Extension以及n180211 API操作无线网络设备。

**提示** 由于篇幅问题，本书将不讨论WFA制订的一些标准（如Wi-Fi Display、Wi-Fi Simple Configuration等）。不过作为Wi-Fi技术的重要组成部分，笔者将在博客中对它们进行系统介绍①。

① 笔者博客地址为[blog.csdn.net/innost](http://blog.csdn.net/innost)或者[my.oschina.net/innost/blog](http://my.oschina.net/innost/blog)。

## 3.2 无线电频谱和802.11协议的发展历程

本节将介绍无线电频谱和802.11协议发展历程。

### 3.2.1 无线电频谱知识<sup>[2]</sup>

Wi-Fi依靠无线电波来传递数据。绝大多数情况下，这些能收发无线电波的设备往往被强制限制在某个无线频率范围内工作。这是因为无线电频谱（即无线电波的频率，Radio Spectrum，单位为Hz）是一种非常重要的资源。所以目前大部分国家对无线电频谱的使用都有国家级的管制。以下是几个主要国家的管制机构。

- FCC (Federal Communication Commission)，美国联邦通信委员会。
- ERO (European Radio-Communication Office)，欧洲无线电通信局。
- MIIT (Ministry of industry and Information Technology)，中国工信部下属的无线电管理局。
- ITU (International Telecommunication Union)，国际电信联盟。

上述这些机构是如何管理无线电频谱资源的呢？一般而言，无线电频谱资源将按照无线电频率的高低进行划分。有一些频率范围内的频谱资源必须得到这些管制机构的授权才可使用，而有些频率范围的频谱资源无须管制机构的授权就可使用。这些无须授权的频谱大部分集中在ISM (Industrial Scientific Medical)，国际共用频段中。

ISM是位于ISM频段的频谱资源被工业、科学和医学三个主要机构使用。ISM一词最早由FCC定义。不过，各国的ISM频段并不完全一致。例如，美国有三个频段属于ISM，分别是902–908MHz、2400–2483.5MHz和5725–5850MHz。另外，各国都将2.4GHz频段划归于ISM范围，所以Wi-Fi、蓝牙等均可工作在此频段上。

**注意** 虽然无须授权就可以使用这些频段资源，但管制机构对设备的功率却有要求，因为无线电频谱具有易被污染的特点，而较大的功率则会干扰周围其他设备的使用。

# 读累了记得休息一会儿哦~

公众号：古德猫宁李

- 电子书搜索下载
- 书单分享
- 书友学习交流

网站：[沉金书屋 https://www.chenjin5.com](https://www.chenjin5.com)

- 电子书搜索下载
- 电子书打包资源分享
- 学习资源分享

### 3.2.2 IEEE 802.11发展历程

使用过Wi-Fi的读者或多或少都接触过IEEE 802.11，它到底代表什么呢？

IEEE (Institute of Electrical and Electronics Engineers) 是美国电气和电子工程师协会的简称。802是该组织中一个专门负责制定局域网标准的委员会，也称为LMSC (LAN/MAN Standards Committee, 局域网/城域网标准委员会)。该委员会成立于1980年2月，其任务就是制定局域网和城域网标准。

由于工作量较大，该委员会被细分成多个工作组 (Working Group)，每个工作组负责解决某个特定方面问题的标准。工作组也会被赋予一个编号（位于802编号的后面，中间用点号隔开），故802.11代表802项目的第11个工作组<sup>[3]</sup>，专门负责制订无线局域网 (Wireless LAN) 的介质访问控制协议 (Medium Access Control, MAC) 及物理层 (Physical Layer, PHY) 技术规范。

和工作组划分类似，工作组内部还会细分为多个任务组 (Task Group, TG)，任务是修改、更新标准的某个特定方面。TG的编号为英文字母，如a、b、c等。

**提示** TG编号使用大小写字母，其含义不同，小写字母的编号代表该标准不能单独存在。例如802.11b代表它是在802.11上进行的修订工作，其本身不能独立存在。而大写字母的编号代表这是一种体系完备的独立标准，如802.1X是处理安全方面的一种独立标准。

802.11制定了无线网络技术的规范，其发展历经好几个版本。表3-1是IEEE 802.11各版本的简单介绍<sup>[4]</sup>。

表 3-1 802.11 各版本简单介绍

版 本	说 明
802.11	1997 年发布，原始标准（2Mbit/s，工作在 2.4GHz 频段）。由于它在速率和传输距离上都不能满足人们的需要，IEEE 小组又相继推出了 802.11b 和 802.11a 两个新标准
802.11a	1999 年发布，新增物理层补充（54Mbit/s，工作在 5GHz 频段）
802.11b	1999 年发布，新增物理层补充（11Mbit/s，工作在 2.4GHz 频段）。802.11b 是所有无线局域网标准中最著名也是最普及的标准。有时称为 Wi-Fi。不过根据前文的介绍，Wi-Fi 是 WFA 的一个商标
802.11c	它在媒体接入控制 / 链路连接控制（MAC/LLC）层面上进行扩展，旨在制订无线桥接运作标准，但后来将标准追加到即有的 802.1 中，成为 802.1d
802.11d	它和 802.11c 一样在媒体接入控制 / 链路连接控制（MAC/LLC）层面上进行扩展，对应 802.11b 标准，解决 Wi-Fi 在某些不能使用 2.4GHz 频段国家中的使用问题
802.11e	新增对无线网络服务质量（Quality of Service，QoS）的支持。其分布式控制模式可提供稳定合理的服务质量，而集中控制模式可灵活支持多种服务质量策略，让影音传输能及时、定量、保证多媒体的顺畅应用，WFA 将此称为 WMM（Wi-Fi Multi-Media）
802.11f	追加了 IAPP（Inter-Access Point Protocol）协定，确保用户端在不同接入点间的漫游，让用户端能平顺、无形地切换区域。不过，此规范已被废除
802.11g	2003 年发布，它是 IEEE 802.11b 的后继标准，其传送速度为 54Mbit/s。802.11g 是为了更高的传输速率而制定的标准，它采用 2.4GHz 频段，使用 CCK 技术与 802.11b 后向兼容，同时它又通过采用 OFDM 技术支持高达 54Mbit/s 的数据流，所提供的带宽是 802.11a 的 1.5 倍
802.11h	为了与欧洲的 HiperLAN2 相协调的修订标准。由于美国和欧洲在 5GHz 频段上的规划、应用上存在差异，故 802.11h 目的是为了减少对同处于 5GHz 频段的雷达的干扰。802.11h 涉及两种技术，一种是动态频率选择（DFS），另一种技术是传输功率控制（TPC）
802.11i	2004 年发布，新增无线网络安全方面的补充。于 2004 年 7 月完成。其定义了基于 AES 的全新加密协议 CCMP（CTR with CBC-MAC Protocol），以及向前兼容 RC4 的加密协议 TKIP（Temporal Key Integrity Protocol）
802.11j	为适应日本在 5GHz 频段以上的应用不同而定制的标准
802.11k	为无线局域网应该如何进行信道选择、漫游服务和传输功率控制提供了标准
802.11l	由于“11l”容易与安全规范“11i”混淆，并且很像“11l”，因此被放弃编号使用
802.11m	该标准主要对 802.11 家族规范进行维护、修正、改进，以及为其提供解释文件。m 表示 Maintenance
802.11n	2004 年 1 月 IEEE 宣布成立一个新的单位来发展 802.11 标准，其声称支持的数据传输速度可达 540Mbit/s。新增对 MIMO（Multiple-Input Multiple-Output）的支持。MIMO 支持使用多个发射和接收天线来支持更高的数据传输速率和无线网络覆盖范围
802.11p	又称 WAVE（Wireless Access in the Vehicular Environment），由 IEEE 802.11 标准扩充的通信协议，主要用于车载电子无线通信。本质上是 IEEE 802.11 的扩充延伸，符合智能交通系统（Intelligent Transportation Systems，ITS）的相关应用
802.11r	2008 年发布，新增快速基础服务转移（Fast Transition），主要是用来解决客户端在不同无线网络 AP 间切换时的延迟问题
802.11s	制订与实现目前最先进的 MESH 网络，提供自主性组态（self-configuring）、自主性修复（self-healing）等能力。无线 Mesh 网可以把多个无线局域网连在一起从而能覆盖一个大学校园或整个城市。Mesh 本意是指所有节点都相互连接。无线 Mesh 网的核心思想是让网络中的每个节点都可以收发信号。它可以增加无线系统的覆盖范围和带宽容量
802.11t	提供提高无线广播链路特征评估和衡量标准的一致性方法

(续)

版 本	说 明
802.11u	也称“与外部网络互通”(InterWorking with External Networks)，定义了不同种类的无线网络之间的网络安全互连功能，让802.11无线网络能够访问蜂窝网络(Cellular Network)或者WiMax等其他无线网络
802.11v	该标准主要针对无线网络的管理。提供了简化无线网络部署和管理的重要和高效率机制。无线终端设备控制、网络选择、网络优化和统计数据获取与监测都属于802.11v建议的功能
802.11w	其任务是通过保护管理帧(无线网络MAC帧的一种类型，还有数据帧和控制帧。详情见3.3.5节)，以进一步提升无线网络的安全性。因为802.11i所涉及的安全技术只覆盖了数据帧，而随着无线技术的发展，越来越多的敏感信息(如基于位置的标识符以及快速传播的信息)却是通过管理帧来传播的，所以安全保护也需要拓展到管理帧
802.11y	该标准的目标是对在与其他用户共享的美国3.65GHz~3.7GHz频段中802.11无线局域网通信的机制进行标准化

**提示** 可从IEEE官方网站下载802.11-2012标准PDF全文(下载地址为<http://standards.ieee.org/about/get/802/802.11.html>)，长达2793页，包含从802.11a到802.11z各个版本(包括a、b、d、e、g、h、i、j、k、n、p、r、s、u、v、w、y、z)所涉及的技术规范。

### 3.3 802.11无线网络技术

从本节开始，将介绍802.11涉及的无线网络技术。首先介绍OSI基本参考模型。

### 3.3.1 OSI基本参考模型及相关基本概念

#### 1. OSI/RM

ISO (International Organization for Standardization, 国际标准化组织) 和 IEC (International Electrotechnical Commission, 国际电工技术委员会) 于1983年联合发布了ISO/IEC 7498标准。该标准定义了著名的OSI/RM<sup>[5]</sup> (开放系统互联参考模型, Open Systems Interconnection Reference Model)。

在OSI/RM中，计算机网络体系结构被划分成七层，其名称和对应关系如图3-1所示。图中绘制了OSI/RM以及另外一个常用的网络体系TCP/IP的结构。先来看OSI/RM，它将网络划分成七层，由上到下分别如下<sup>[6]</sup>。

- 应用层 (Application Layer) : 应用层能与应用程序界面沟通以达到向用户展示的目的。常见的协议有HTTP、HTTPS、FTP、SMTP等。其数据单位为APDU (Application Protocol Data UNIT)。
- 表示层 (Presentation Layer) : 表示层能为不同客户端提供数据和信息的语法转换，使系统能解读成正确的数据，同时它还能提供压缩解压、加密解密等服务。例如不同格式图像（如GIF、JPEG、TIFF等）的显示就是由位于表示层的协议来支持的。其数据单位为PPDU (Presentation Protocol Data UNIT)。

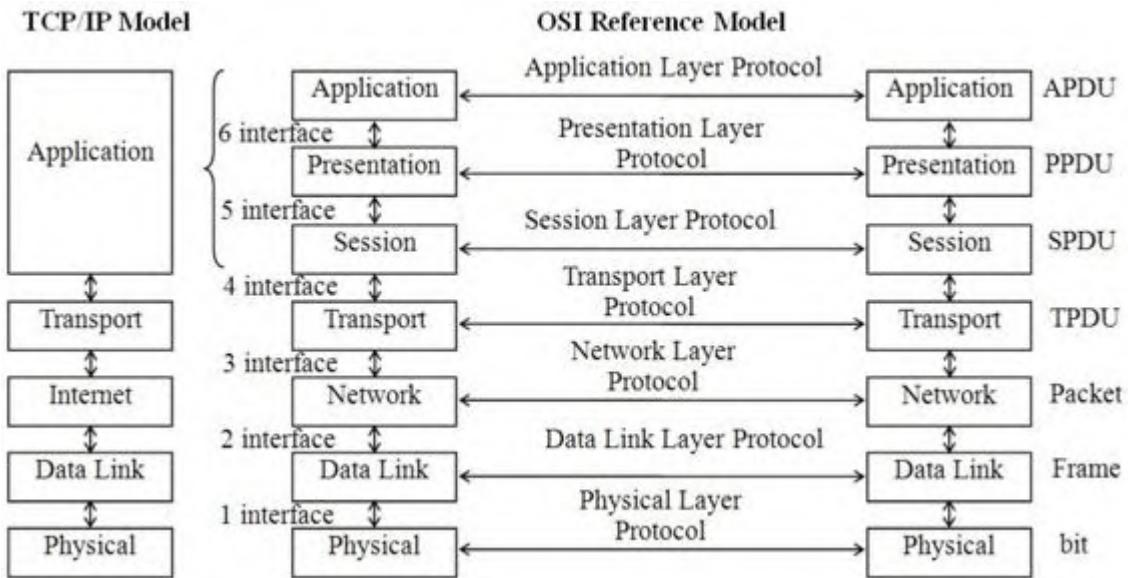


图3-1 OSI RM及TCP/IP结构

- 会话层（Session Layer）：会话层用于为通信双方制定通信方式，创建和注销会话（双方通信）等。常见的协议有ZIP、AppleTalk、SCP等。其数据单位为SPDU（Session Protocol Data UNIT）。
- 传输层（Transport Layer）：传输层用于控制数据流量，同时能进行调试及错误处理，以确保通信顺利。发送端的传输层会为数据分组加上序号，以方便接收端把分组重组为有用的数据或文件。传输层的常见协议有TCP、UDP等。其数据单位为TPDU（Transport Protocol Data Unit）。
- 网络层（Network Layer）：网络层为数据传送的目的地寻址，然后再选择一个传送数据的最佳路线。网络层数据的单位为Packet或Datagram。常见的设备有路由器等。常见协议有IP、IPv6。
- 数据链路层（Data Link Layer）：在物理层提供比特流服务的基础上，建立相邻节点之间的数据链路。通过差错控制提供数据帧（Frame）在信道上无差错的传输。数据链路层在不可靠的物理介质上提供可靠的传输。该层的作用包括物理地址寻址、数据的成帧、流量控制、数据的检错、重发等。数据链路层数据的单位为Frame（帧）。常见的设备有二层交换机、网桥等。

- 物理层（Physical Layer）：物理层定义了通信设备机械、电气、功能和过程等方面的特性，用以建立、维护和拆除物理链路连接。物理层数据的单位为bit。

图3-1中左边所示为另外一个常用的网络体系，即TCP/IP模型。对比图3-1中的两个模型，我们可简单认为TCP/IP Model是OSI/RM的一个简化版本。

提示 关于OSI/RM的详细信息，请读者阅读本章参考资料[5]。

## 2. LLC和MAC子层

虽然ISO/IEC 7498标准所定义的OSI/RM只将网络划分为七层。但实际上每一层还可划分为多个子层（Sub Layer）。所有这些子层中，最为人熟知的就是ISO/IEC 8802<sup>[7]</sup> 规范划分Data Link Layer而得到的LLC（Logic Link Control Sub Layer）和MAC（Medium Access Control Sub Layer）。它们的信息如图3-2所示。

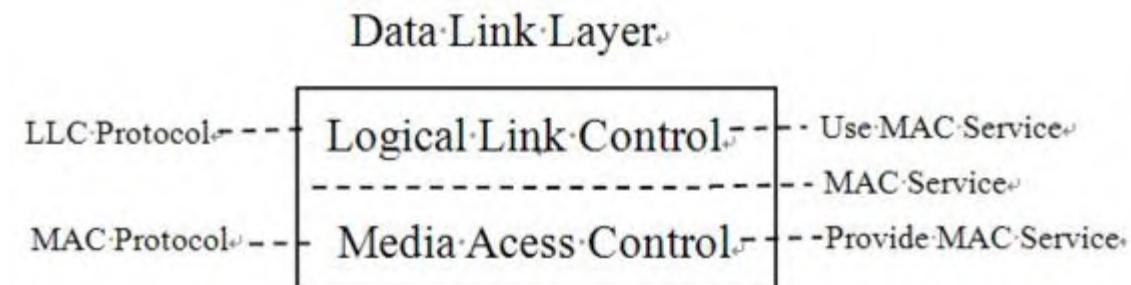


图3-2 MAC和LLC子层

ISO/IEC 8802将Data Link Layer划分成了两个子层。

- 媒介访问控制子层（MAC Sub Layer）：该子层的目的是解决局域网（Local Area Network, LAN）中共用信道的使用产生竞争时，如何分配信道的使用权问题。目前LAN中常用的媒介访问控制方法是CSMA/CD（争用型介质访问控制）。由于无线网络的特殊性，MAC的控制方法略有不同。将在下文介绍相关内容。
- 逻辑链路控制子层（LLC Sub Layer）：该子层实现了两个站点之间帧的交换，实现端到端（源到目的），无差错的帧传输和应答功能

及流量控制功能。

在Data Link层划分的这两个子层中，802.11只涉及MAC层。由于物理介质的不同，无线和有线网络使用的MAC方法有较大差别，主要区别如下<sup>[8]</sup>。

- 有线网络最常使用的方法（此处仅考虑以太网）是CSMA/CD（Carrier Sense Multiple Access/Collision Detect，载波监听多路访问/冲突检测机制）。其主要工作原理是：工作站发送数据前先监听信道是否空闲，若空闲则立即发送数据。并且工作站在发送数据时，边发送边继续监听。若监听到冲突，则立即停止发送数据并等待一段随机时间，然后再重新尝试发送。
- 无线网络主要采用CSMA/CA（Carrier Sense Multiple Access/Collision Avoidance，载波监听多路访问/冲突避免机制）方法。无线网络没有采用冲突检测方法的原因是：如果要支持冲突检测，必须要求无线设备能一边接收数据信号一边传送数据信号，而这种设计对无线网络设备来说性价比太低。另外，冲突检测要求边发送数据包边监听，一旦有冲突则停止发送。很显然，这些因发送冲突而被中断的数据发送将会浪费不少的传输资源。所以，802.11在CSMA/CD基础上进行了一些调整，从而得到了CSMA/CA方法。其主要工作原理见下节内容。

注意 CSMA/CA协议信道利用率低于CSMA/CD协议信道利用率。信道利用率受传输距离和空旷程度的影响，当距离远或者有障碍物影响时会存在隐藏终端问题，降低信道利用率。在802.11b WLAN中，在1Mbps速率时最高信道利用率可达到90%，而在11Mbps时最高信道利用率只有65%。

### 3. CSMA/CA<sup>[8]</sup>

CSMA/CA主要使用两种方法来避免碰撞。

- 设备发送数据前，先监听无线链路状态是否空闲。为了避免发生冲突，当无线链路被其他设备占用时，设备会随机为每一帧选择一段退避（backoff）时间，这样就能减少冲突的发生。

- RTS-CTS握手。设备发送帧前，先发送一个很小的RTS（Request To Send）帧给目标端，等待目标端回应CTS（Clear To Send）帧后才开始传送。此方式可以确保接下来传送数据时，其他设备不会使用信道以避免冲突。由于RTS帧与CTS帧长度很小，使得整体开销也较小。

我们通过图3-3来介绍RTS和CTS的作用。

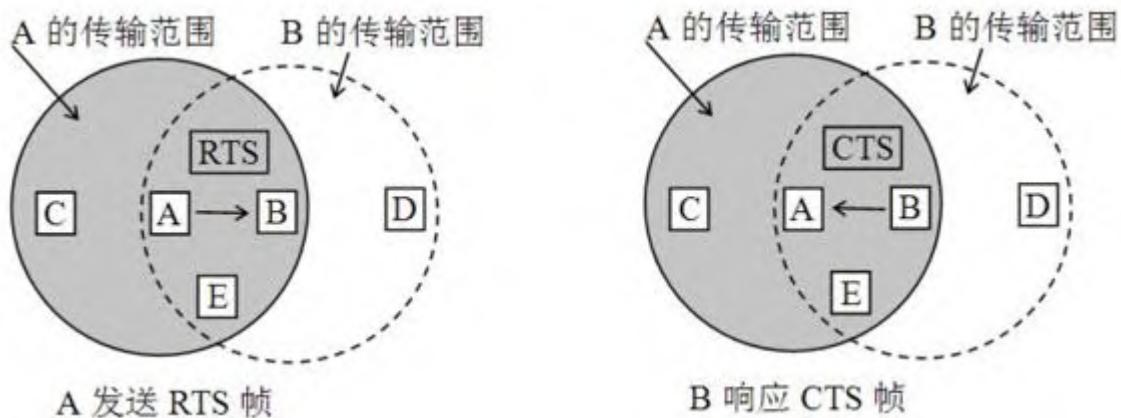


图3-3 RTS/CTS原理

如图3-3所示，以站A和站B之间传输数据为例，站B、站C、站E在站A的无线信号覆盖的范围内，而站D不在其内。站A、站E、站D在站B的无线信号覆盖的范围内，但站C不在其内。

如果站A要向站B发送数据，站A在发送数据帧之前，要先向站B发送一个请求发送帧RTS。在RTS帧中会说明将要发送的数据帧的长度。站B收到RTS帧后就向站A回应一个允许发送帧CTS。在CTS帧中也附上站A欲发送的数据帧的长度（从RTS帧中将此数据复制到CTS帧中）。站A收到CTS帧后就可发送其数据帧了。

怎么保证其他站不会干扰站A和站B之间的数据传输呢？

- 对于站C，站C处于站A的无线传输范围内，但不在站B的无线传输范围内。因此站C能够收听到站A发送的RTS帧，但经过一小段时间后，站C收听不到站B发送的CTS帧。这样，在站A向站B发送数据的同时，站C也可以发送自己的数据而不会干扰站B接收数据（注意，站C收听不到站B的信号表明，站B也收不听到站C的信号）。

- 对于站D，站D收听不到站A发送的RTS帧，但能收听到站B发送的CTS帧。因此，站D在收到站B发送的CTS帧后，应在站B随后接收数据帧的时间内关闭数据发送操作，以避免干扰站B接收自A站发来的数据。
- 对于站E，它能收到RTS帧和CTS帧，因此，站E在站A发送数据帧的整个过程中不能发送数据。

总体而言，使用RTS和CTS帧会使整个网络的效率下降。但由于这两种控制帧都很短（它们的长度分别为20和14字节）。而802.11数据帧则最长可达2346字节，相比之下开销并不算大。相反，若不使用这种控制帧，则一旦发生冲突而导致数据帧重发，则浪费的时间就更大。

另外，802.11提供了三种情况供用户选择以处理。

- 使用RTS和CTS帧。
- 当数据帧的长度超过某一数值时才使用RTS和CTS帧。
- 不使用RTS和CTS帧。

尽管协议经过了精心设计，但冲突仍然会发生。例如，站B和站C同时向站A发送RTS帧。这两个RTS帧发生冲突后，使得站A收不到正确的RTS帧因而站A就不会发送后续的CTS帧。这时，站B和站C像以太网发生冲突那样，各自随机地推迟一段时间后重新发送其RTS帧。

**提示** 根据802.11协议，CSMA/CA具体运作时由协调功能（Coordination Function，CF）来控制。协议规定有四种不同的协调功能，分别是DCF（Distributed CF，分布式协调功能）、基于DCF之上的PCF（Point CF）、HCF（Hybrid CF，混合型协调功能）以及用于Mesh网络的MCF（Mesh CF）。由于无线网络是共享介质，所以协调功能的目的就是用于控制各个无线网络设备使用无线媒介的时机以避免冲突发生。形象点说，这就好比在一个会议室里，所有人都可以发言，但如果多个人同时发言又不知道谁和谁在说话。所以，每个打算发言的人都需要检查当前发言的情况。本章不详细介绍802.11中CF相关的内容。感兴趣的读者可阅读802.11协议第9节“MAC sublayer functional description”。

另外，规范还定义了基于竞争的服务（contention-based service，使用DCF进行数据交换）和基于无竞争的服务（contention-free service，使用PCF进行数据交换），本章也不讨论它们。

#### 4. MAC层Service及其他概念<sup>[5][9]</sup>

ISO/IEC 7498及相关的一些标准文档除了对网络体系进行层次划分外，还定义了层和层之间交互方式及其他一些基本组件。它们可用图3-4来表示。

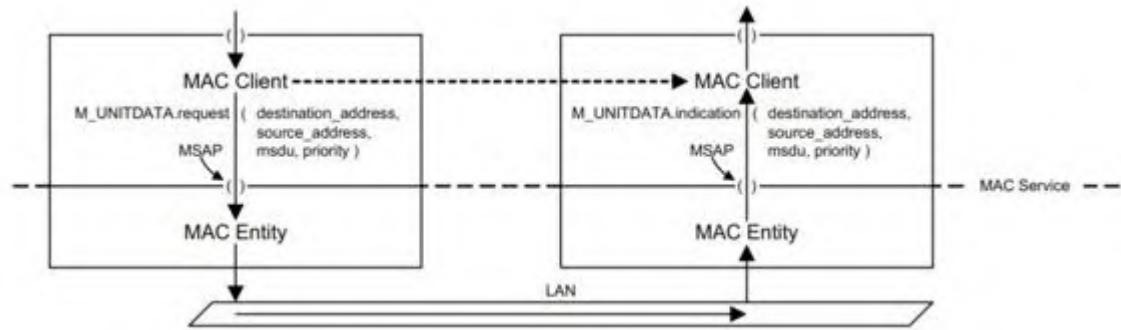


图3-4 MAC Entity、Service和Clients

图3-4绘制了MAC层相关的组件及交互方式。

- 协议规定了每一层所提供的功能，这些功能统一用Service来表示。MAC层为上层提供MAC Service。从Java编程角度来看，就好比定义了一个名为MAC Service的Java Package。
- 每一层内部有数个Entity。Entity代表封装了一组功能的模块。上面提到的Service就是由Entity提供的。根据OSI/RM的层次关系，第N层为第N+1层提供服务。所以，MAC Entity对上一层提供MAC Service。从Java编程角度来看，MAC Entity就好比MAC Service Package中定义的类，不同的Entity代表该Package中实现不同的功能类。一个Package中的Entity可以相互调用以完成某个功能。
- 第N+1层要使用第N层服务时，必须经由Service Access Point来完成。MAC的Service必须借由MSAP（MAC Service Access Point）来访问。

根据上面的介绍，Entity定义了一个类，其中是否定义了相应的功能函数呢？在标准文档中，和功能函数对应的术语是primitives（原语）及它们的parameters（参数）。图3-4列举了MAC Service的两个重要函数，分别是如下。

- M\_UNITDATA.request：Client调用该原语发送数据。该原语的参数是destination\_address（目标地址）、source\_address（源地址）、MSDU（MAC Service Data Unit，即数据）、priority（优先级）。
- M\_UNITDATA.indication：当位于远端机器的MAC层收到上面发送的数据后，将通过indication原语通知上一层以处理该数据。

**提示** 关于MAC层的服务详细定义，请读者阅读3.3.5节MAC服务定义内容。

值得指出的是，规范只是从逻辑上定义了上述内容，它并没有指定具体的实现。关于MAC Service的详细定义，读者可阅读本章参考资料[10]，即ISO/IEC 15802-1。

协议还定义了SDU和PDU两个概念。当上一层调用MAC的request原语时，会把要发送的数据传给MAC层。这个数据被称为MSDU。对MAC层来说，MSDU其实就是MAC要发送的数据，即载荷（Payload）。MAC层处理Payload时还会封装MAC层自己的一些头信息。这些信息和MSDU共同构成了MAC层的Protocol Data Unit（简写为MPDU）。从OSI/RM角度来看，经过N+1层封装后的数据是(N+1)-PDU。该数据传递给N层去处理时，对N层来说就是(N)-SDU。N层封装这个SDU后就变成自己的(N)-PDU了。

## 5. MIB<sup>[11]</sup>

在阅读802.11相关规范时，经常会碰到MIB（Management Information Base，管理信息库）。MIB是一个虚拟的数据库，里边存储了一些设备信息供查询和修改。MIB常见的使用之处是网管利用SNMP协议管理远程主机、路由器等。

MIB内部采用树形结构来管理其数据。内部的每一个管理条目（Entry）通过OID（Object Identifier）来访问。MIB定义了Entry的属性和可取的值。由于属性及其可取值的定义采用了与具体编程语言

无关的方式，所以一个MIB库可以很轻松地通过编译器（compiler）将其转换成对应的编程语言源码文件。

**提示** MIB的定义比较烦琐，可阅读参考资料[11]中的资料列表以学习完整的MIB知识。

从笔者角度来看，对802.11来说，MIB就是定义的一组属性。802.11定义的MIB属性在<http://www.ieee802.org/11/802.11mib.txt>中。下载后将得到一个802.11mib.txt文件。可通过JMIBBrowser<sup>①</sup>工具加载并查看其内容，如图3-5所示。

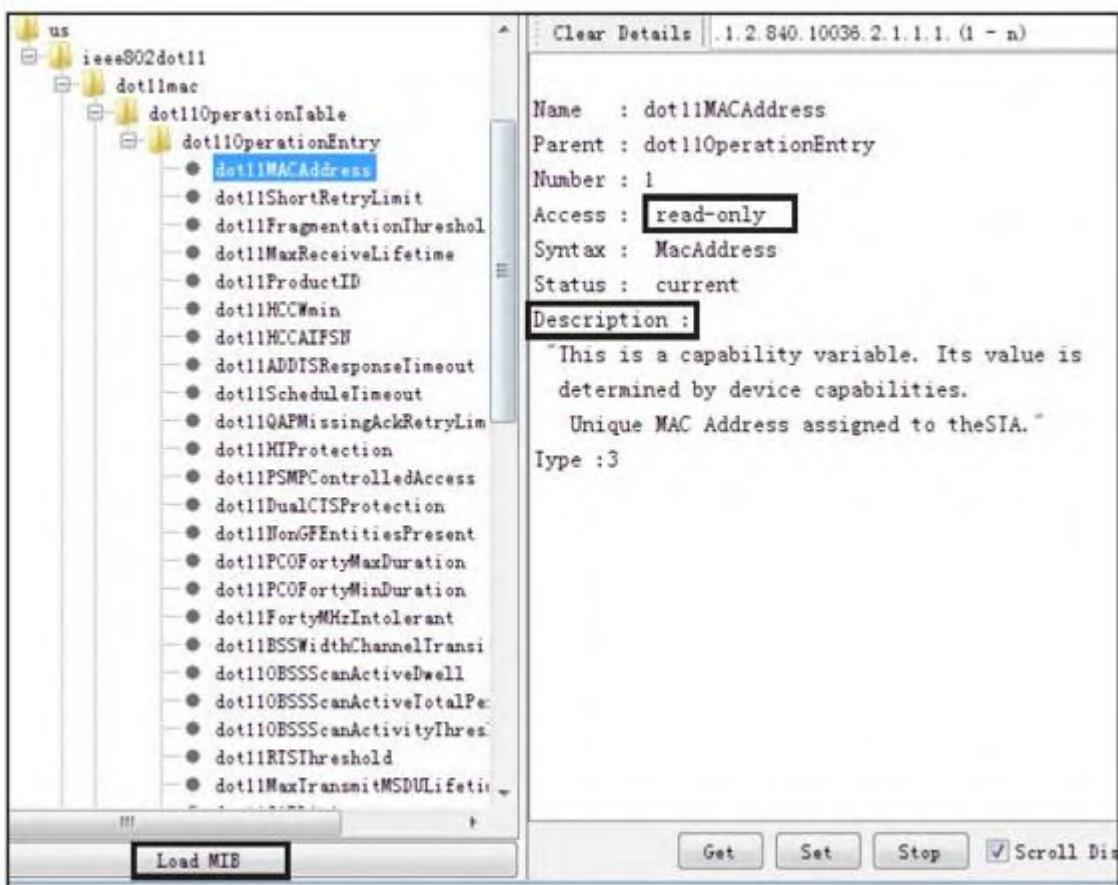


图3-5 802.11 MIB内容

首先点击图3-5所示左下角的“Load MIB”按钮以加载802.11mib.txt文件，然后查看802.11mib定义的一些属性。如图3-5所示，左侧显示当前查看的dot11MACAddress的条目。右侧显示该条目的信息，Access中的“read-only”表示只读，Description表示该条目的意义。

802.11mib定义了一个较全的属性集合，一般而言，设备可能只支持其中一部分属性。图3-6所示为Note 2上wlan0设备的MIB信息截图（由于篇幅问题只包含部分Note 2 wlan0设备的MIB属性）。

以第一条属性dot11RSNAOptionImplemented为例，其在802.11mib.txt的定义如图3-7所示。

相比直接浏览802.11mib.txt文本文件而言，利用JMIBBrowser工具查看属性会更加方便和直观一些。

```
> mib
dot11RSNAOptionImplemented=TRUE
dot11RSNAPreauthenticationImplemented=TRUE
dot11RSNAEnabled=FALSE
dot11RSNAPreauthenticationEnabled=FALSE
dot11RSNAConfigVersion=1
dot11RSNAConfigPairwiseKeysSupported=5
dot11RSNAConfigGroupCipherSize=40
dot11RSNAConfigPMKLifetime=43200
dot11RSNAConfigPMKReauthThreshold=70
dot11RSNAConfigNumberOfPTKSAReplayCounters=1
dot11RSNAConfigSATimeout=60
dot11RSNAAuthenticationSuiteSelected=00-00-00-0
dot11RSNAPairwiseCipherSelected=00-50-f2-1
dot11RSNAGroupCipherSelected=00-50-f2-1
dot11RSNAPMKIDUsed=
dot11RSNAAuthenticationSuiteRequested=00-00-00-0
dot11RSNAPairwiseCipherRequested=00-50-f2-1
dot11RSNAGroupCipherRequested=00-50-f2-1
dot11RSNAConfigNumberOfGTDKSAReplayCounters=0
```

图3-6 Note 2 wlan0设备的MIB属性

```
dot11RSNAOptionImplemented OBJECT-TYPE
    SYNTAX TruthValue
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "This is a capability variable.
         Its value is determined by device capabilities.

        This variable indicates whether the entity is RSNA-capable."
 ::= { dot11StationConfigEntry 26 }
```

图3-7 dot11RSNAOptionImplemented属性内容

提示 以后分析wpa\_supplicant源码时，会碰到802.11mib定义的属性，建议在阅读本节时下载相关文件和工具程序。另外，以上内容涉及的知识点是读者以后阅读802.11协议时必然会碰到的。由于802.11协议引用的参考资料非常多，故本节整理了其中最基础的知识点，请读者务必认真阅读本节内容。

- ① 用Java语言编写的小工具，可利用SNMP协议查看和设置指定设备MIB，下载地址为<http://sourceforge.net/projects/jmibrowser/>。

### 3.3.2 802.11知识点导读

802.11规范全称为《Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications》。从其标题可知，802.11规范定义了无线局域网中MAC层和PHY层的技术标准。

2012年版的802.11协议全文共2793页，包含20小节（clause），23个附录（A~W）。图3-8所示为802.11协议原文目录的一部分。

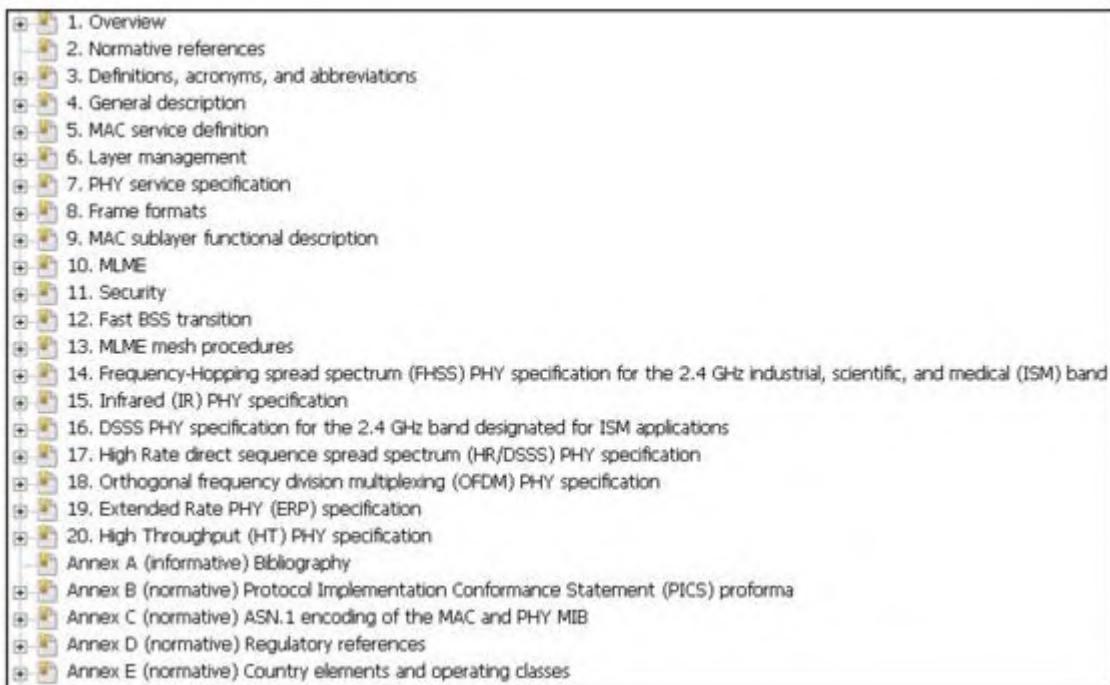


图3-8 802.11协议原文目录

由图3-8可知802.11协议内容非常丰富。读者可尝试阅读该文档，但估计很快会发现这将是一件非常枯燥和令人头疼的事情。主要原因是此规范类似于手册，它非常重视细节的精准，但各技术点前后逻辑上的连贯性较差，使得读者极难将散落在协议中各个角落的技术点整理出一个内容有序，难度由浅入深的核心知识框架来。

基于上述原因，本章后续小节将从以下几个方面介绍802.11规范中的一些核心内容。

- 3.3.3节介绍802.11中的物理组件和网络结构。
- 3.3.4节将在物理组件和网络结构基础上，介绍802.11为无线网络所定义的服务。了解这些服务对我们从整体上理解无线网络的功能有重要意义。
- 3.3.5节介绍802.11 MAC服务和帧方面的内容。这部分知识比较具体，相信读者理解起来没有问题。
- 3.3.6节介绍MAC层管理实体方面的内容。清楚这部分内容有助于读者理解后续有关Linux Wi-Fi编程的知识。
- 3.3.7节介绍802.11安全性方面的知识。

**提示** 篇幅原因，本书不可能囊括规范的所有内容。但相信读者在理解本节内容的基础上，能够轻松开展更加深入的研究。另外，为帮助读者理解规范，本章会在重要知识点之处添加“规范阅读提示”。

由于802.11物理层涉及大量和无线电、信号处理相关的知识。这些知识不仅内容枯燥繁杂，而且对软件工程师来说并无太大意义，故本章不做介绍。愿意深入研究的读者可阅读相关资料。

### 3.3.3 802.11组件

本节介绍802.11规范中的物理组件和相关网络结构。首先来看无线网络中的物理组件。

#### 1. 物理组件<sup>[12]</sup>

802.11无线网络包含四种主要物理组件，如下所示。

- WM (Wireless Medium, 无线媒介)：其本意指能传送无线MAC帧数据的物理层。规范最早定义了射频和红外两种物理层，但目前使用最多的是射频物理层。
- STA (Station, 工作站)：其英文定义是“*A logical entity that is logically addressable instance of aMAC and PHY interface to the WM*”。通俗点说，STA就是指携带无线网络接口卡（即无线网卡）的设备，例如笔记本、智能手机等。另外，无线网卡和有线网卡的MAC地址均分配自同一个地址池以确保其唯一性。
- AP (Access Point, 接入点)：其原文定义是“*An entity that contains one STA and provides access to the distribution services, via the WM for associated STAs*”。由其定义可知，AP本身也是一个STA，只不过它还能为那些已经关联的（associated）STA提供分布式服务（Distribution Service, DS）。什么是DS呢？请读者阅读下文。
- DS (Distribution System, 分布式系统)：其英文定义为“*A system used to interconnect a set of basic service sets (BSSs) and integrated local area networks (LANs) to create an extended service set (ESS)*”。DS的定义涉及BSS、ESS等无线网络架构，其解释见下文。

上述四个物理组件如图3-9所示。其中，最难解释清楚的就是DS。笔者在仔细阅读规范后，感觉其对DS的解释并不直观。此处将列举一个常见的应用场景以帮助读者理解。

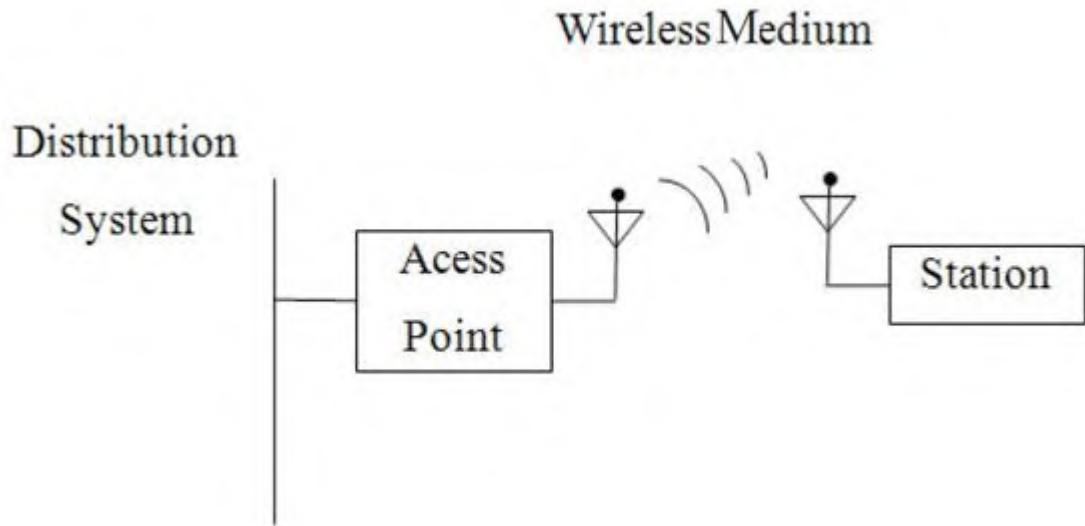


图3-9 802.11四大主要物理组件

一般家用无线路由器一端通过有线接入互联网，另一端通过天线提供无线网络。打开Android手机上的Wi-Fi功能，并成功连接到此无线路由器提供的无线网络（假设其网络名为“TP-LINK\_1F9C5E”，可在路由器中设置）时，我们将得到：

- 路由器一端通过有线接入互联网，故可认为它整合（integrate）了 LAN。
- 不论路由器是否接入有线网络（即本例中的互联网），手机（扮演 STA 的角色）和路由器（扮演 AP 的角色）之间建立了一个小的无线网络。该无线网络的覆盖范围由 AP 即路由器决定。这个小网络就是一个 BSS。另外，定义中提及的 ESS 是对 BSS 的扩展。一个 ESS 可包含一或多个 BSS。在本例中，ESS 对应的 ID 就是“TP-LINK\_1F9C5E”，即我们为路由器设置的网络名。

上述内容中将 BSS 和 LAN 结合到一起以构成一个 ESS 的就是 DS。虽然规范中并未明示 DS 到底是什么，但绝大部分情况下，DS 是指有线网络（通过它可以接入互联网）。后文将介绍 DS 所提供的分布式服务（即 DSS）。现在对读者来说，更重要的概念是其中和无线网络架构相关的 BSS 和 ESS 等。这部分内容将在下节介绍。

规范阅读提示

1) 上文介绍的AP、STA、DS的定义都来自于802.11的3.1节。该节所列的定义是最精确的。以DS为例，此节所定义的DS涉及和有线网络的结合。但规范中其他关于DS的说明均未明示是否一定要和LAN结合。

2) 关于STA，其定义只说明它是一个可“singly addressable”的实体，而没有说明其对应的功能。所以，读者会发现AP也是一个STA。另外还有提供QoS (Quality of Service) 的STA。除此之外，从可移动性的角度来看，还有Mobile STA和Portable STA之分。Portable STA虽然可以移动，但只在固定地点使用（例如AP就是一个典型的Portable STA）。而Mobile STA表示那些只要在Wi-Fi覆盖范围内，都可以使用的STA（例如手机、平板电脑等设备）。

## 2. 无线网络的构建<sup>[12]</sup>

有了上节所述的物理组件，现在就可以搭建由它们构成的无线网络了。802.11规范中，基本服务集（Basic Service Set, BSS）是整个无线网络的基本构建组件（Basic Building Block）。如图3-10所示，BSS有两种类型。

- 独立型BSS（Independent BSS）：这种类型的BSS不需要AP参与。各STA之间可直接交互。这种网络也叫ad-hoc BSS（一般译为自组网络或对等网络）。
- 基础结构型BSS（Infrastructure BSS）：所有STA之间的交互必须经过AP。AP是基础结构型BSS的中控台。这也是家庭或工作中最常见的网络架构。在这种网络中，一个STA必须完成诸如关联、授权等步骤后才能加入某个BSS。注意，一个STA一次只能属于一个BSS。

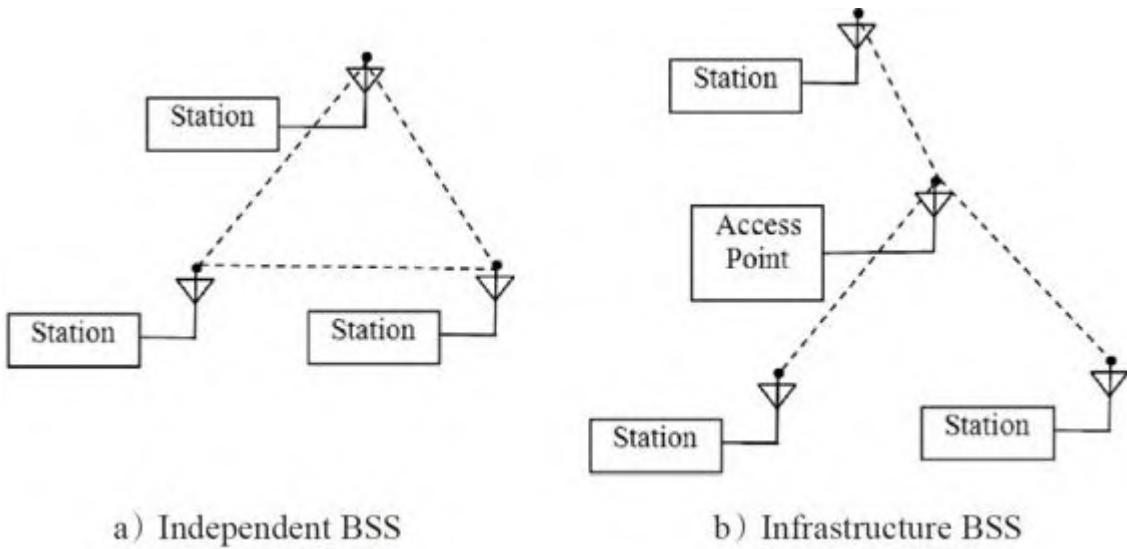


图3-10 BSS的两种方式

**提示** Independent BSS缩写为IBSS。而Infrastructure BSS没有对应的缩写。不过，一般用BSS代表Infrastructure BSS。根据前文所述，AP也是一个STA，但此处STA和AP显然是两个不同的设备。

由图3-10中BSS的结构可知，其网络覆盖范围由该BSS中的AP决定。在某些情况下，需要几个BSS联合工作以构建一个覆盖面更大的网络，这就是一个ESS（Extended Service Set，扩展服务集），如图3-11所示。

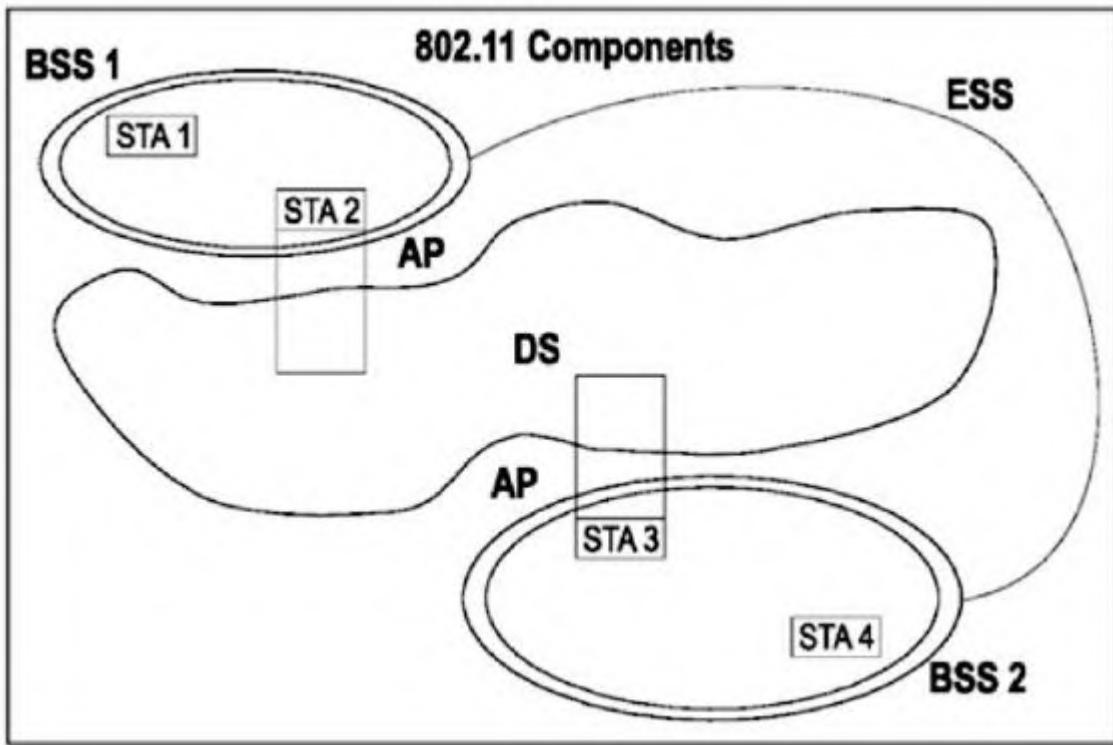


图3-11 ESS示意图

ESS在规范中的定义是"A set of one or more interconnected BSSs that appears as a single BSS to the LLC layer at any STA associated with one of those BSSs"。此定义包含几个关键点。

一个ESS包含一或多个BSS。如图3-11所示的BSS1和BSS2。

BSS1和BSS2本来各自组成了自己的小网络。但在ESS结构中，它们在逻辑上又构成了一个更大的BSS。这意味着最初在BSS2中使用的STA4（利用STA3，即BSS2中的AP上网）能跑到BSS1的范围内，利用它的AP（即STA2）上网而不用做任何无线网络切换之类的操作。此场景在手机通信领域很常见，例如在移动的汽车上打电话，此时手机就会根据情况在物理位置不同的基站间切换语音数据传输而不影响通话。

**注意** ESS中的BSS拥有相同的SSID（Service Set Identification，详细内容见下文），并且彼此之间协同工作。这和目前随着Wi-Fi技术的推广，家庭和工作环境中存在多个无线网络（即存在多个ESS）的情况有本质不同。在多个ESS情况下，用户必须手动选择才能切换到不同的ESS。由于笔者日常工作和生活中，ESS只包含一个BSS，当某个AP停

机时，笔者就得手动切换到其他无线网络中去了。另外，切换相关的知识点属于Roaming（漫游）范畴，读者可阅读Secure Roaming in 802.11 Networks一书来了解相关细节。

上述网络都有Identification，分别如下。

- BSSID (BSS Identification)：每一个BSS都有自己的唯一编号。在基础结构型网络中，BSSID就是AP的MAC地址，该MAC地址是真实的地址。IBSS中，其BSSID也是一个MAC地址，不过这个MAC地址是随机生成的。
- SSID (Service Set Identification)：一般而言，BSSID会和一个SSID关联。BSSID是MAC地址，而SSID就是网络名。网络名往往是一个可读字符串，因为网络名比MAC地址更方便人们记忆。

ESS包括一到多个BSS，而它对外看起来就像一个BSS。所以，对ESS的编号就由SSID来表达。只要设置其内部BSS的SSID为同一个名称即可。一般情况下，ESS的SSID就是其网络名（network name）。

### 规范阅读提示

1) 上述网络结构中，并未提及如何与有线网络（LAN）的整合。规范中其实还定义了一个名为portal的逻辑模块（logical component）用于将WLAN（Wireless LAN）和LAN结合起来。由于WLAN和LAN使用的MAC帧格式不同，所以portal的功能类似翻译，它在WLAN和LAN间转换MAC帧数据。目前，portal的功能由AP实现。

2) 规范中还定义了QoS BSS。这主要为了在WLAN中支持那些对QoS有要求的程序。由于无线网络本身固有的特性，WLAN中的QoS实现比较复杂，效果也不如LAN中的QoS。初学者可先不接触这部分内容。

### 3.3.4 802.11 Service介绍[13]

本节对802.11规范定义的和数据传输相关的服务进行介绍。规范在各服务的逻辑联系上往往一句带过，使得很难把它们之间的关系搞清楚。对数据传输来说，其逻辑关系如下。

- DSS用于完成数据的传输。
- 由于有线和无线网络的差异性，当数据需要传输到有线网络时候，就通过Integration Service在二者之间进行转换。
- 由于存在transition，为了保证DSS能找到对应的STA，所以就存在association、reassociation服务。
- 当STA不再使用DSS时，就通过disassociation服务离开DS。

802.11规范定义了很多Service。Service就是无线网络中对应模块应该具有的功能。本节内容主要来自802.11规范4.4节到4.8节，集中介绍了802.11无线网络应该提供的Service。

#### 1. 802.11 Service的分类

从模块考虑，802.11中的Service分为两大类别（Category），分别如下。

- SS (Station Service)：它是STA应该具有的功能。
- DSS (Distribution System Service)：它指明DS应具有的功能。

前文也提到过DS，其定义比较模糊。规范中还特别强调说802.11并不指明DS的具体实现，它只是从逻辑上指明DS应该具有的功能，也就是DSS。

**提示** 虽然规范未说明DS到底是什么，但目前普遍认为DS就是指以太网。

SS和DSS包含一些相同的服务。这是因为无线网络中，SS和DSS将协同工作。以数据传输为例，STA和DS都需要支持MAC帧数据的收发。所以二者必然包含类似的服务。另外，802.11中的Service还可从其所代表的功能出发进行划分以得到如下类别。

- 6种Service用于支持802.11 MAC层的数据传输（规范定义为MAC Service Data Unit Delivery）。
- 3种Service用于控制802.11 LAN的访问控制（Access Control）和数据机密性（Data Confidentiality）。
- 2种Service用于频谱管理（Spectrum Management）。
- 1种Service用于支持QoS。
- 1种Service用于支持时间同步。
- 1种Service用于无线电测量（Radio Measurement）。

接下来，本章将从功能划分角度出发来介绍其中所涉及的服务。

## 2. 数据传输相关服务

毋庸置疑的是，无线网络最重要的一个功能就是传输数据了。对802.11来说，此处的数据就是无线MAC帧数据。规范一共定义了6种Service支持数据传输。其中，QoS traffic Scheduling服务和QoS有关，主要目的是满足QoS的要求，本书不对这部分内容进行介绍。

### (1) Distribution Service和Integration Service

本节介绍数据传输服务中的前两位，Distribution Service (DS，分布式服务) 和Integration Service (IS，整合服务)。前文也提到过DSS，其目的其实非常简单。只要STA传送任何数据都会使用这项服务。当AP接收到数据帧后，就会使用该服务将帧传输到目的地。任何使用接入点的通信都会通过该服务进行传播。包括关联至同一个AP的两个STA之间的相互通信。规范还特别绘制了示意图来展示DS，如图3-12所示。

- BSS1中的STA1想发送数据给BSS2中的STA4。

- STA1先要把数据传输给BSS1中的AP，即STA2。
- STA2通过使用DSS将数据传输给DS。
- DS根据相关的信息找到位于BSS2的AP，即STA3。
- STA3把数据传递给STA4。

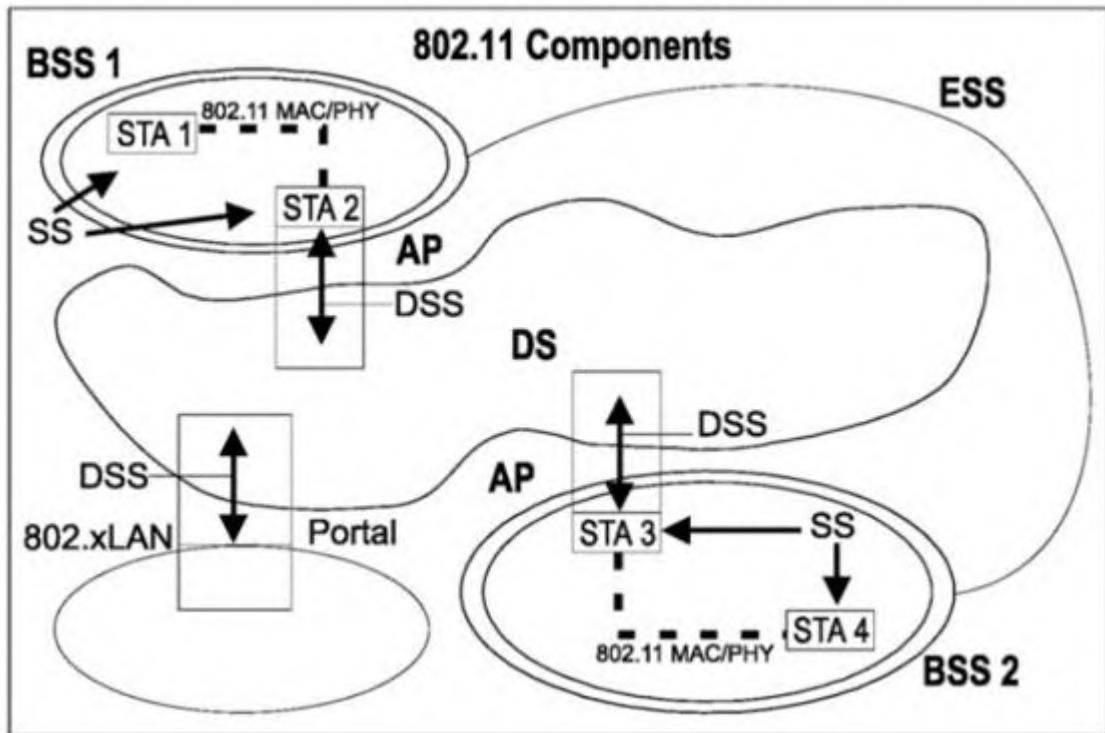


图3-12 DS示意图

在上述过程中，DSS的作用就是把来自STA2的数据传递到正确的STA3。规范并未说明DS如何实现DSS，它只要求数据发送端传递足够的参数使得STA3能被正确找到。

整合服务的使用也包含在图3-12中。当DSS发现数据的目的端是LAN时，就需要把数据传给Portal了。这时候，DS就会使用IS将无线数据进行必要的转换，以发送到LAN中去。

同样，规范并未指明IS的具体实现，不过读者可知家用的无线路由器就实现了该功能。

## (2) association、reassociation、disassociation服务

前文曾提到，规范要求使用DSS时，必须提供足够的信息给它以保证数据能传输成功。这些足够的信息就是由association（关联）、reassociation（重新关联）和disassociation（取消关联）服务提供的。这些服务是如何帮助DSS的呢？回答这个问题前，先介绍Transition Type。

Transition Type对STA在无线网络中移动的类型进行了分类，分别如下。

- No-Transition：即没有移动。它包括固定不动的情况以及在某个AP无线覆盖范围内移动。
- BSS-Transition：即从ESS中的一个BSS切换到另一个BSS。根据前面对ESS的介绍，我们希望这种移动不影响网络的使用。当然，要真正实现无缝切换，需要做一些其他工作。这部分内容由规范中的“Fast BSS Transition”一节描述。
- ESS-Transition：从一个ESS中的BSS切换到位于另外一个ESS的BSS。这种情况极有可能导致网络切换，影响用户使用。

当DS传输数据时，DSS需要知道和哪个AP建立联系。所以，规范要求STA在传输数据前，必须要和一个AP建立关联关系，这就需要使用association服务。关联服务的目的在于为AP和STA建立一种映射关系。例如当图3-12中的DS向BSS1发送数据时，它得知道STA1和STA2通过association服务建立了一种映射，这样数据才能传递给STA1。

在同一时刻，一个STA只能和一个AP建立关联关系。但AP可和多个STA建立关联关系。关联服务回答了这样一个问题：哪个AP为STA X服务？

当STA进行transition的时候（如BSS Transition），它就需要使用reassociation服务了。因为之前它和BSS1建立了关联关系，此后它需要和BSS2建立关联关系。这时就可以使用reassociation服务来完成该功能。reassociation服务只能由STA发起。

当STA不需要使用DSS，或者AP不再为某个STA服务时，就需要调用disassociation服务。这样STA就不能再使用DSS传输数据了。相比关

联和重新关联，STA和AP都可以调用取消关联服务。

**提示** 随着Wi-Fi安全要求的提高，上述场景在强健安全网络（Robust Security Network）中将有较大不同。

## 规范阅读提示

1) 上述内容在RSN中大有不同。RSN主要为了增强WLAN的数据加密和认证性能，并且针对WEP加密机制的各种缺陷做了多方面的改进。关于安全方面的内容，我们留待后文再介绍。

2) 介绍Service的时候，规范中还有一段对MAC帧类型进行了非常简单的介绍，即Service的具体调用是通过发送不同的MAC帧来触发的。规范中MAC帧分为三大类型，分别是data（数据帧）、management（管理帧）和control（控制帧）。其中和service直接相关的是management帧，而control帧用于帮助传输data和management帧。关于MAC帧的详细内容在3.3.5节。

## 3. 访问控制和数据机密性相关服务

访问控制和数据机密性（Access Control and Data Confidentiality）包括三个服务。其主要目的是解决无线网络中安全防护相关的工作。相比有线网络而言，安全性是无线网络中非常重要的部分。以访问控制为例，在有线网络中，例如你去一家公司参观，只有在经过对方同意的情况下，才能使用有线上网。有线网络控制由网管或相关人员实施。而无线网络中，只要你在该公司内部无线网络的覆盖范围内，都可以接入此网络。很明显，这时就需要对无线网络实施访问控制了。

在访问控制的进一步上，数据机密性用于对数据的内容进行加密保护。因为即使实施了访问控制，由于传输的数据还是通过无线网络，那么理论上无线网络覆盖内的所有人都可以接收到该数据。如果不加密的话，数据内容就能被不相关的人窥窃了。

规范定义了三个服务用于处理安全性，分别如下。

- Authentication和Deauthentication：这两个服务用于Access Control，译为身份验证以及解除身份验证。

- Confidentiality: 原本称为私密性 (privacy) 服务，后来对这部分内容实施了加强。目前规范中提到的数据加密方法有WEP、TKIP、CCMP。

#### 4. 频谱管理服务

频谱管理服务包括TPC (Transmit Power Control, 传输功率控制) 服务和DFS (Dynamic Frequency Selection, 动态频率选择) 服务。这两个服务的目的在于满足不同管制机构对无线电资源使用时的一些特定要求。

对TPC而言，当无线设备工作在5GHz频段时，必须限制其各信道的最大发射功率，以避免干扰卫星服务。其主要特点包括：

- 支持STA根据功率要求选择和不同的AP关联。
- 根据管制机构的要求，设定当前信道的最大传输功率。
- 根据传输过程中的损耗等信息来自动调整传输功率。

和TPC类似，DFS也是针对那些工作在5GHz频段的无线设备，使其能够根据情况动态选择传输信道以避免干扰雷达系统。DFS的主要特点是针对信道（即不同频率），包括：

- 根据STA支持的信道情况去选择合适的AP。
- 静默某个信道，以支持雷达的使用该信道。
- 在使用某个信道前，先检查是否有雷达系统已经使用该信道了。如果有，则必须停止在该信道工作以避免干扰雷达。

#### 5. QoS和时间同步服务

规范对这两个服务的介绍不多，故本书也不展开详细讨论。简单来说，802.11规范支持使用任何一种合适的QoS机制以预留一些资源，例如RRP (Resource Reservation Protocol)。

时间同步服务目的是支持一些对同步性要求较高的应用（如视音频播放等）。这些应用使用的同步方式可基于规范中定义的STA之间时间同

步 (Timing Synchronization Function, TSF) 机制之上。

提示 QoS来源于802. 11e。

## 6. 无线电测量服务

无线电测量 (Radio Measurement) 服务所提供的功能如下。

- 为上层应用提供查询无线电相关信息的接口。
- 通过无线电测量来获取周围AP的信息。
- 能够在所支持的信道上进行无线电测量。

提示 无线电测量服务来源于802. 11k。

## 7. 802. 11 Service相关知识总结

本节对802. 11提供的服务进行了基本介绍。服务其实是从无线网络所提供的功能的角度来考察无线网络技术的。所有服务中最关键的三个如下。

- 数据传输服务：这是无线网络的一个主要功能。
- 安全方面的服务：由于无线网络的特殊性，所以安全性是它必须要考虑的问题。
- 无线电测量服务：无线电测量用于无线网络组建。例如STA必须通过该服务去发现周围的AP。

提示 笔者认为，上述服务从文字描述上看起来并不复杂。但对软件工程师来说，很难把上述知识和编程/代码等联系起来。不过没有关系，下文的内容将更加具体。尤其是3. 3. 6节关于MLME (MAC Layer Management Entity) 的介绍，其描述方式就很接近编程中的API介绍。

### 3.3.5 802.11 MAC服务和帧

本节将向读者介绍802.11中MAC服务以及MAC帧方面的内容。首先来看MAC服务。

#### 1. MAC服务定义<sup>[14]</sup>

根据3.3.1节所示，MAC层定义了一些Service用于为上层LLC提供服务。其中，802.11中定义的MAC提供三种服务，分别如下。

- MA-UNITDATA. request: 供LLC层发送数据。
- MA-UNITDATA. indication: 通知LLC层进行数据接收。
- MA-UNITDATA-STATUS. indication: 通知LLC层自己（即MAC层）的状态。

规范中这三种服务的定义和编程语言中的函数非常像。根据前文所述，服务在规范中被称为原语（primitive），每个服务有特定的参数（parameter）。另外，规范对每条原语都做了非常详细的解释，例如原语使用场景（从编程角度看就是介绍什么时候用这些API），MAC层如何处理这些原语（从编程角度看即描述应该如何实现这些API）。

#### (1) MA-UNITDATA. request服务

request用于发送数据，其定义如下所示。

```
MA-UNITDATA.request(
    source address           // 指定本次数据发送端的MAC地址
    destination address     // 指定接收端的MAC地址。可以是组播或广播地址
    routing information      // 路由信息。对于802.11来说，该值为null（表示该值没有作用）
    data                     // MSDU，即上层需要发送的数据。对802.11来说，其大小不能超过2304字节
    priority                // priority和service
    class                    // class的解释见下文
    service class
)
```

request原语中，除priority和service class参数外，其他都很容易理解。而priority和service class参数与QoS有关系。

- priority的取值为0~15的整数。对于非QoS相关的操作，其取值为Contention和ContentionFree。802.11中，QoS设置了不同的用户优先级（User Priority, UP）。显然，UP高的数据将优先得到发送。Priority参数和QoS中的其他参数一起决定发送数据的优先级。
- service class的取值也针对是否为QoS而有所不同。对于非QoS，取值可为ReorderableGroupAddressed或StrictlyOrdered。这两个都和数据发送时是否重排（reorder）发送次序有关。非QoS情况下，STA发送数据时并不会主动去调整发送次序。而QoS情况下，很明显那些UP较高的数据会得到优先处理。但对于组播数据，发送者可以设置service class为ReorderableGroupAddressed进行次序调整。

对request原语来说，当LLC需要发送数据时，就会调用request。MAC层需要检查参数是否正确。如果不正确，将通过其他原语通知LLC层，否则将启动数据发送流程。

## (2) MA-UNITDATA.indication服务

下面来看MAC层收到数据后用于通知LLC层的原语indication，其定义如下。

```
MA-UNITDATA.indication(
    source address          // 代表数据源MAC地址。其值取值MAC帧中的
    SA (关于MAC帧格式详情见后文)
    destination address    // 代表目标MAC地址，取自MAC帧中的DA，可以是组
    播地址
    routing information    // 值为null
    data                   // MAC帧数据
    reception status       // 表示接收状态，例如成功或失败。对802.11来
    说，该状态永远返回success
                                // MAC层会丢弃那些错误的数据包
    priority               // 和request的取值略有不同，此处略过
    service class
)
```

802.11中，indication只有在MAC层收到格式完整、安全校验无误及没有其他错误的数据包时才会被调用以通知LLC层。

### (3) MA-UNITDATA-STATUS.indication服务

STATUS.indication用于向LLC层返回对应request原语的处理情况。其原型如下。

```
MA-UNITDATA-STATUS.indication(  
    // source和destination address和对应request的前两个参数一样  
    source address  
    destination address  
    transmission status // 返回request的处理情况，详情见下文  
    provided priority // 只有status返回成功，该参数才有意义。其值和调用request发送数据时的值一致  
    provided service class // 只有status返回成功，该参数才有意义  
)
```

indication最关键的一个参数是transmission status，可以是以下值。

- Successful：代表对应request发送数据成功。
- Undeliverable：数据无法发送。其原因有数据超长（excessive data length）、在request时指定了routing information（non-null source routing）、不支持的priority和service class、没有BSS（no BSS available）、没有key进行加密（cannot encrypt with anull key）等。

提示 本节对应于规范的第5节“MAC service definition”。和前面内容相比，本节内容更贴近于代码实现，当然也就更符合程序员的“审美观”。

接下来介绍和MAC帧相关的内容。

## 2. MAC帧<sup>[15]</sup><sup>[16]</sup><sup>[17]</sup>

802.11 MAC帧格式如图3-13所示。

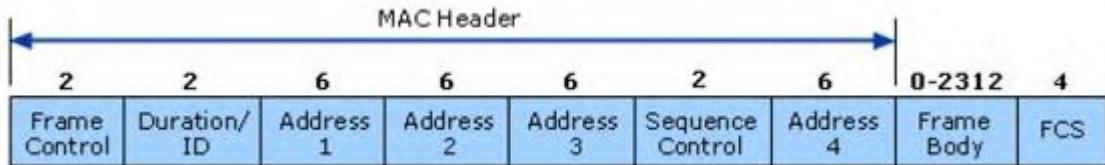


图3-13 802.11 MAC帧格式

由图3-13可知，MAC帧由以下三个基本域组成。

- MAC Header：包括帧控制（Frame Control）、时长（Duration）、地址（Address）等。注意，每个方框上面的数字表示该域占据的字节数。例如Frame Control需要2字节。
- Frame Body：代表数据域。这部分内容的长度可变，其具体存储的内容由帧类型（type）和子类型（sub type）决定。
- FCS：（Frame Check Sequence，帧校验序列）用于保障帧数据完整性。

#### 规范阅读提示

- 1) 关于MAC帧的格式。规范中还指出，如果是QoS数据帧，还需要附加QoS Control字段。如果是HT (High Throughput，一种用于提高无线网络传输速率的技术) 数据帧，还需要附加HT Control字段。
- 2) 关于Frame body长度。规范中指出其长度是7951字节，它和Aggregate-MPDU (MAC报文聚合功能) 以及HT有关。本文不拟讨论相关内容。故此处采用2312字节。

下面分别介绍MAC帧头中几个重要的域，首先是Frame Control域。

#### (1) Frame Control域

Frame Control域共2字节16位，其具体字段划分如图3-14所示。

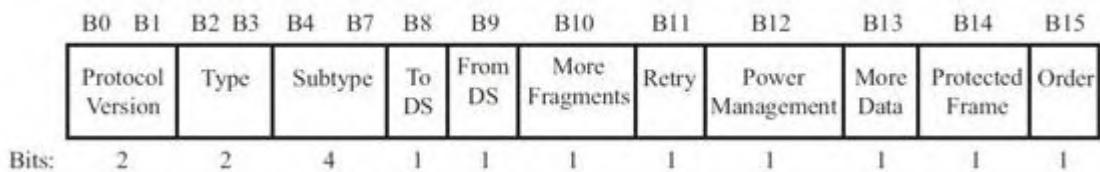


图3-14 Frame Control域的组成

- Protocol Version: 代表802.11 MAC帧的版本号。目前的值是0。
- Type和Subtype: 这两个字段用于指明MAC帧的类型。802.11中MAC帧可划分为三种类型，分别是control、data和management，每种类型的帧用于完成不同功能。详情见下文。
- To DS和From DS: 只用在数据类型的帧中。其意义见下文。
- More Fragments: 表明数据是否分片。只支持data和management帧类型。
- Retry: 如果该值为1，表明是重传包。
- Power Management: 表明发送该帧的STA处于活跃模式还是处于省电模式。
- More Data: 和省电模式有关。AP会为那些处于省电模式下的STA缓冲一些数据帧，而STA会定时查询是否有数据要接收。该参数表示AP中还有缓冲的数据帧。如果该值为0，表明STA已经接收完数据帧了（下节将介绍省电模式相关的内容）。
- Protected Frame: 表明数据是否加密。
- Order: 指明接收端必须按顺序处理该帧。

Type和Subtype的含义如表3-2所示。其中仅列出了部分Type和Subtype的取值。

表 3-2 MAC 帧 Type 和 Subtype

Type (b3b2)	类型描述	Subtype (b7-b4)	子类型描述
00	Management	0000/1	Association Request/Response
00	Management	0100/1	Probe Request/Response
00	Management	1000	Beacon 帧
00	Management	1101	Action 帧 <sup>②</sup>
01	Control	1010	PS-POLL 帧 (省电轮询)
01	Control	1011	RTS
01	Control	1100	CTS
01	Control	1101	ACK
10	Data	0000	Data
11	保留	0000-1111	保留

(一) 本书将在第7章详细介绍Action帧。

From DS和To DS的取值如表3-3所示。

表 3-3 From/To DS 含义

To DS 和 From DS 取值	含 义
To DS=0 From DS=0	1) 同一个 IBSS 中, 从一个 STA 到另一个 STA 2) 同一个 BSS 中, 从一个非 AP 的 STA 到另一个非 AP 的 STA 或者 BSS 外部
To DS=1 From DS=0	代表发送给 DS 或 AP 的 MAC 帧。当然, 发往 DS 的帧必须通过 AP 转发才行

(续)

To DS 和 From DS 取值	含 义
To DS=0 From DS=1	代表来自 DS 或 AP 的数据帧
To DS=1 From DS=1	仅 Mesh BSS 支持。本书不讨论此种情况

## 扩展阅读 省电模式<sup>[18]</sup>

无线网络的使用者常见于笔记本电脑、智能手机等设备, 它们最大的一个特点就是能够移动(无线技术的一个重要目的就是摆脱各种连接线的缠绕)。无线带来便捷的同时也引入了另外一个突出问题, 即电量消耗。在当前电池技术还没有明显进步的现实情况下, 802.11规范对电源管理也下了一番苦功。

规范为STA定义了两种和电源相关的状态, 分别是Active模式和PS(Power Save)模式。处于PS模式下, 无线设备将关闭收发器

(transceiver) 以节省电力。

STA为了节电而关闭数据收发无可厚非，怎么保证数据传输的连贯性呢？下面讨论基础结构型网络中省电模式的知识。在这种网络中，规范规定AP为了保证数据传输的连贯性，其有两个重要工作。

- 1) AP需要了解和它关联的STA的电源管理状态。当某个STA进入PS状态后，AP就要做好准备以缓存发给该STA的数据帧。一旦STA醒来并进入Active模式，AP就需要将这些缓存的数据发送给该STA。注意，AP无须PS模式，因为绝大多数情况下，AP是由外部电源供电（如无线路由器）。在AP中，每个和其关联的STA都会被分配一个AID（Association ID）。
- 2) AP需要定时发送自己的数据缓存状态。因为STA也会定期接收信息（相比发送数据而言，开启接收器所消耗的电力要小）。一旦STA从AP定时发送的数据缓存状态中了解到它还有未收的数据，STA则会进入Active模式并通过PS-POLL控制帧来接收它们。

现在来看MAC帧中Power Management的取值情况。

- 对于AP不缓存的管理帧，PM字段无用。
- 对于AP发送的帧，PM字段无用。
- STA发送给还未与之关联的AP的帧中，PM字段无用。
- 其他情况下，PM为1表示STA将进入PS状态，否则将进入Active状态。

配合PM使用的字段就是More Data。根据前文介绍，STA通过PS-POLL来获取AP端为其缓存的数据帧。一次PS-POLL只能获取一个缓存帧。STA怎么知道缓存数据都获取完了呢？原来More Data值为1表示还有缓存帧，否则为0。虽然MAC帧中没有字段说明AP缓存了多少帧，但通过简单的0/1来标示是否还有剩余缓存帧也不失为一种好方法。

提示 规范中的PS处理比较复杂，建议阅读参考资料[18]。

## (2) Duration/ID域

Duration/ID域占2字节共16位，其具体含义根据Type和Subtype的不同而变化，不过大体就两种，分别代表ID和Duration。

- 对于PS-POLL帧，该域表示AID的值。其中最后2位必须为1，而前14位取值为1~2007。这就是该域取名ID之意。
- 对于其他帧，代表离下一帧到来还有多长时间，单位是微秒。这就是该域取名Duration之意。

注意 Duration的用法和CSMA/CA的具体实现有关。本章不对它进行详细讨论，可阅读参考资料[8]。

### (3) Address域

讲解Address域的用法之前，先介绍MAC地址相关的知识。根据IEEE 802.3<sup>[19]</sup> 协议，MAC地址有如下特点。

- MAC地址可用6字节的十六进制来表示，如笔者网卡的MAC地址为1C-6F-65-8C-47-D1。
- MAC地址的组成包括两个部分。0~23位是厂商向IETF等机构申请用来标识厂商的代码，也称为“组织唯一标识符”（Organizationally Unique Identifier, OUI）。后24位是各个厂商制造的所有网卡的一个唯一编号。
- 第48位用于表示这个地址是组播地址还是单播地址。如果这一位是0，表示此MAC地址是单播地址；如果这位是1，表示此MAC地址是组播地址。例如01-XX-XX-XX-XX-XX为组播地址。另外，如果地址全为1，例如FF-FF-FF-FF-FF-FF，则为MAC广播地址。
- 第47位表示该MAC地址是全球唯一的还是本地唯一的。故该位也称为G/L位。

图3-15所示为MAC地址示意图<sup>[19][20]</sup>。

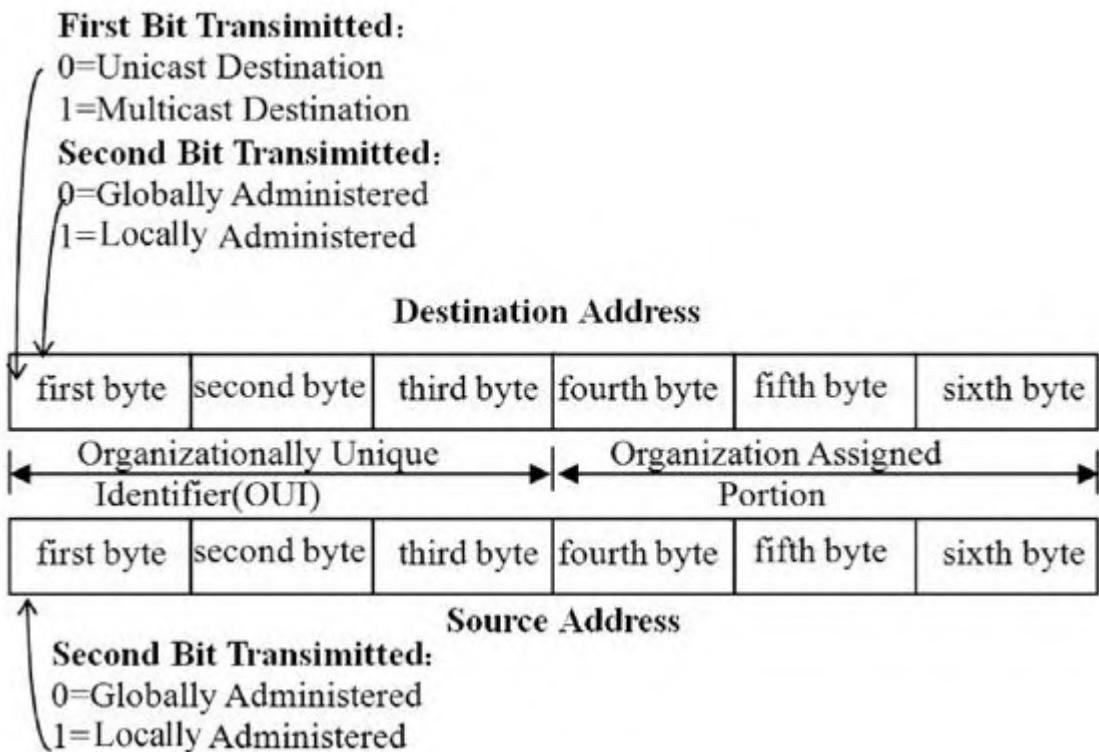


图3-15 MAC地址格式

注意图3-15中的字节序和比特序。

- 字节序为Big-endian，即最高字节在前。所以图中的first byte实际对应的MAC地址最左边一组。以Note 2 MAC地址90-18-7C-69-88-E2为例，first byte就是“90”（如果是Little-endian，则first byte是“E2”），其OUI是Samsung，代表三星电子。

- 比特序为Little-endian，即最低位在前。这就是01-XX-XX-XX-XX-XX为组播地址的原因了。因为“01”对应的二进制是“0000-0001”，其第0位是1，应该放在图中first byte最左边一格。

**提示** 上面的数值按从左至右为最高位到最低位。一般情况下，字节序和比特序是一致的。但是以太网数据传输时，字节序采用Big-endian，比特序为Little-endian。

参考资料[19]中原文是这样描述的。

The first bit (LSB) shall be used in the Destination Address field as an address type designation bit to identify the DA either as an individual or as a group address. If this bit is 0, it shall be "individual address", "Each octet of each address field shall be transmitted least significant bit first.

802.11 MAC帧头部分共包含四个Address域，但规范中却有五种地址定义，分别如下<sup>[15]</sup>。

- BSSID：在基础型BSS中，它为AP <sup>①</sup> 的地址。而在IBSS中则是一个本地唯一MAC地址。该地址由一个46位的随机数生成，并且U/M位为0，G/L位为1。另外对MAC广播地址来说，其名称为wildcard BSSID。
- 目标地址（Destination Address, DA）：用来描述MAC数据包最终接收者（final recipient），可以是单播或组播地址。
- 源地址（Source Address, SA）：用来描述最初发出MAC数据包的STA地址。一般情况下都是单播地址。
- 发送STA地址（Transmitter Address, TA）：用于描述将MAC数据包发送到WM中的STA地址。
- 接收STA地址（Receiver Address, RA）：用于描述接收MAC帧数据的。如果某个MAC帧数据接收者也是STA，那么RA和DA一样。但如果接收者不是无线工作站，而是比如以太网中的某台PC，那么DA就是该机器的MAC地址，而RA则是AP的MAC地址。这就表明该帧将先发给AP，然后由AP转发给PC。

上述无线地址中，一般只使用Address 1～Address 3三个字段，故图3-13中它们的位置排在一起。表3-4展示了Address域的用法。

表 3-4 Address 域用法说明

网络类型	Address 1 (接收端)	Address 2 (发送端)	Address 3	Address 4
IBSS	DA	SA	BSSID	未使用
To AP	BSSID	SA	DA	未使用
From AP	DA	BSSID	SA	未使用

由表3-4可知，Address 1用作接收端，Address 2用作发送端。Address 3携带其他信息用于帮助MAC帧的传输，针对不同类型，各个Address域的用法如下。

- IBSS网络中，由于不存在DS，所以Address 1指明接收者，Address 2代表发送者。Address 3被设置为BSSID，其作用是：如果Address 1被设置为组播地址，那么只有位于同一个BSSID的STA才有必要接收数据。
- 基础结构型网络中，如果STA发送数据，Address 1必须是BSSID，代表AP。因为这种网络中，所有数据都必须通过AP。Address 3代表最终的接收者，一般是DS上的某个目标地址。
- 基础结构型网络中，如果数据从AP来，那么Address 1指明STA的地址，Address 2代表AP的地址，Address 3则代表DS上的某个源端地址。
- Address 4只在无线网络桥接的时候才使用。本书不讨论这种情况。

图3-16描述了基础型结构网络和无线桥接情况下不同地址的使用场景。

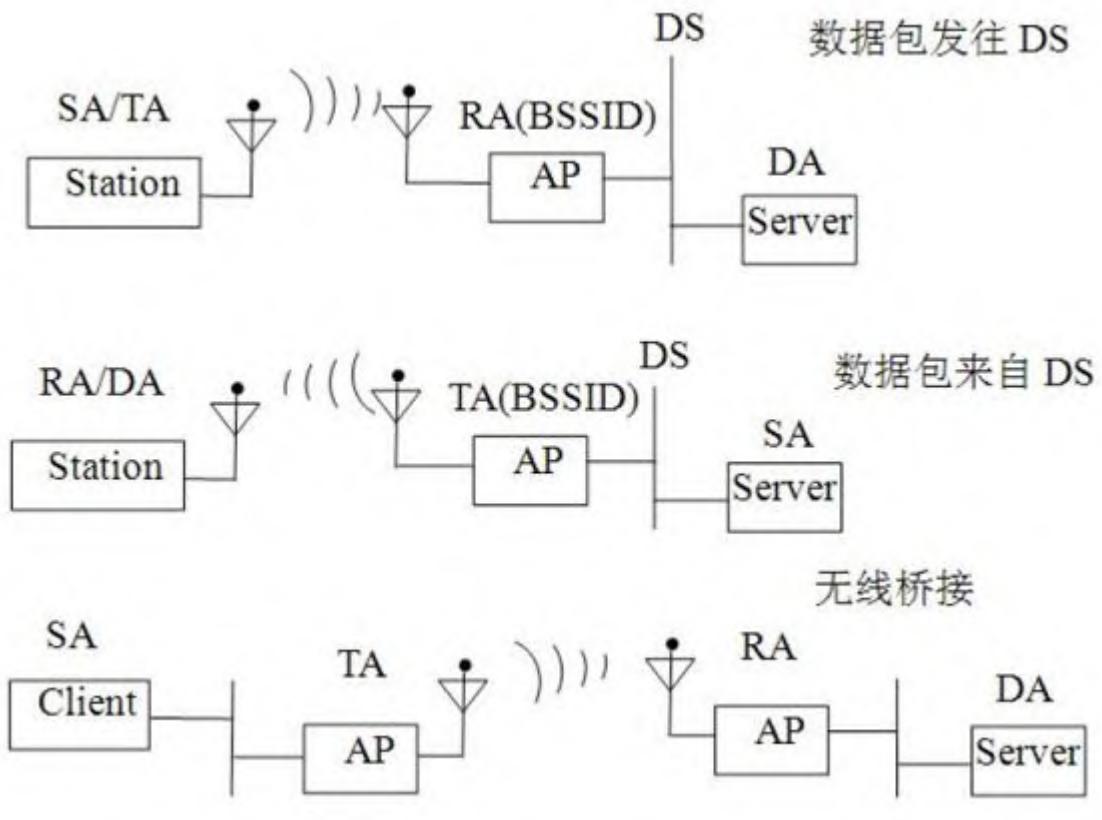


图3-16 地址的使用

如图3-16所示：

- 数据包发往DS的情况下，SA和TA一致，而RA和BSSID一致。
- 数据包来自DS的情况下，RA和DA一致，而TA和BSSID一致。
- 无线桥接的情况下，SA、TA、RA和DA四种地址都派上了用场。

**提示** 关于Address域的作用请读者务必清楚下面三个关键点。

1) MAC帧头中包含四个Address域，在不同的情况下，每个域中包含不同的地址。原则是Address 1代表接收地址，Address 2代表发送端地址，Address 3辅助用，Address 4用于无线桥接或Mesh BSS网络中。

2) 规范定义了五种类型的地址，即BSSID、RA、SA、DA和TA。每种地址有不同的含义。

3) 在某些情况下，有些类型地址的值相同，如图3-16所示。

#### (4) Sequence Control域

Sequence Control域长16位，前4位代表片段编号（Fragment Number），后12位为帧顺序编号（Sequence Number），域格式如图3-17所示。

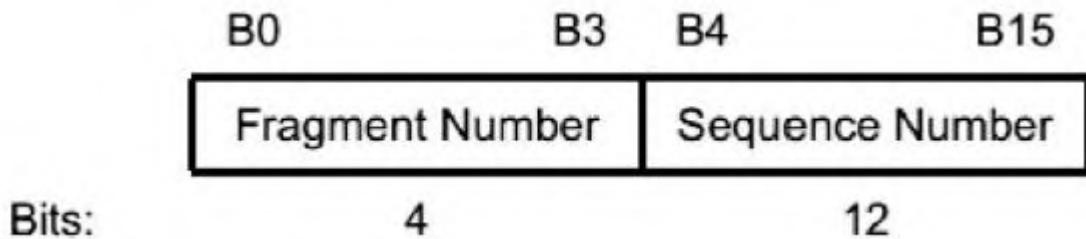


图3-17 Sequence Control域格式

- Sequence Number: STA每次发送数据帧时都会设置一个帧顺序编号。注意，控制帧没有帧顺序编号。另外，重传帧不使用新的帧顺序编号。
- Fragment Number: 用于控制分片帧。如果数据量太大，则MAC层会将其分片发送。每个分片帧都有对应的分片编号。

#### (5) MAC帧相关知识总结

本节对MAC帧格式进行介绍。这部分内容难度并不大，但读起来肯定有些枯燥。笔者在阅读参考资料[15]时也有同样的感觉。后来笔者利用公司的AirPcap无线网络分析设备截获无线网络数据并使用wireshark软件进行分析后，感觉MAC帧信息直观多了，如图3-18所示。

```
④ Frame 51039: 144 bytes on wire (1152 bits), 144 bytes captured (1152 bits) on interface 0
④ Radiotap Header v0, Length 20
③ IEEE 802.11 Probe Request, Flags: .......c
    Type/Subtype: Probe Request (0x04)
    Frame Control: 0x0040 (Normal)
        Version: 0
        Type: Management frame (0)
        Subtype: 4
    Flags: 0x0
        .... ..00 = DS status: Not leaving DS or network is operating in AD-HOC mode (To DS: 0 Fr
        .... .0.. = More Fragments: This is the last fragment
        .... 0... = Retry: Frame is not being retransmitted
        ....0 .... = PWR MGT: STA will stay up
        ..0. .... = More Data: No data buffered
        .0.. .... = Protected flag: Data is not protected
        0.... .... = Order flag: Not strictly ordered
    Duration: 0
    Destination address: Broadcast (ff:ff:ff:ff:ff:ff)
    Source address: SamsungE_69:88:e2 (90:18:7c:69:88:e2)
    BSS Id: Broadcast (ff:ff:ff:ff:ff:ff)
    Fragment number: 0
    Sequence number: 36
    ■ Frame check sequence: 0x68044408 [correct]
③ IEEE 802.11 wireless LAN management frame
③ Tagged parameters (96 bytes)
    ■ Tag: SSID parameter set: Test
        Tag Number: SSID parameter set (0)
        Tag length: 4
        SSID: Test
    ■ Tag: Supported Rates 1, 2, 5.5, 11, [Mbit/sec]
    ■ Tag: Extended Supported Rates 6, 9, 12, 18, 24, 36, 48, 54, [Mbit/sec]
    ■ Tag: HT Capabilities (802.11n D1.10)
    ■ Tag: DS Parameter set: Current Channel: 7
    ■ Tag: Vendor Specific: Broadcom
    ■ Tag: Vendor Specific: Epigram: HT Capabilities (802.11n D1.10)
```

图3-18 AirPcap使用

AirPcap设备比较贵，但该工具对于将来Wi-Fi知识的学习、工作中的难题解决相当有帮助。

注意 本书的资源共享文件中提供几个Wi-Fi数据包捕获文件，读者可直接利用它们进行分析。详情见1.3节。

接下来介绍常见的几种具体的MAC帧。首先从控制帧开始。

### 3. 控制帧<sup>[15]</sup><sup>[17]</sup>

控制帧的作用包括协助数据帧的传递、管理无线媒介的访问等。规范中定义的控制帧有好几种，本节将介绍其中的四种，它们的帧格式如图3-19所示。由图可知，控制帧不包含数据，故其长度都不大。

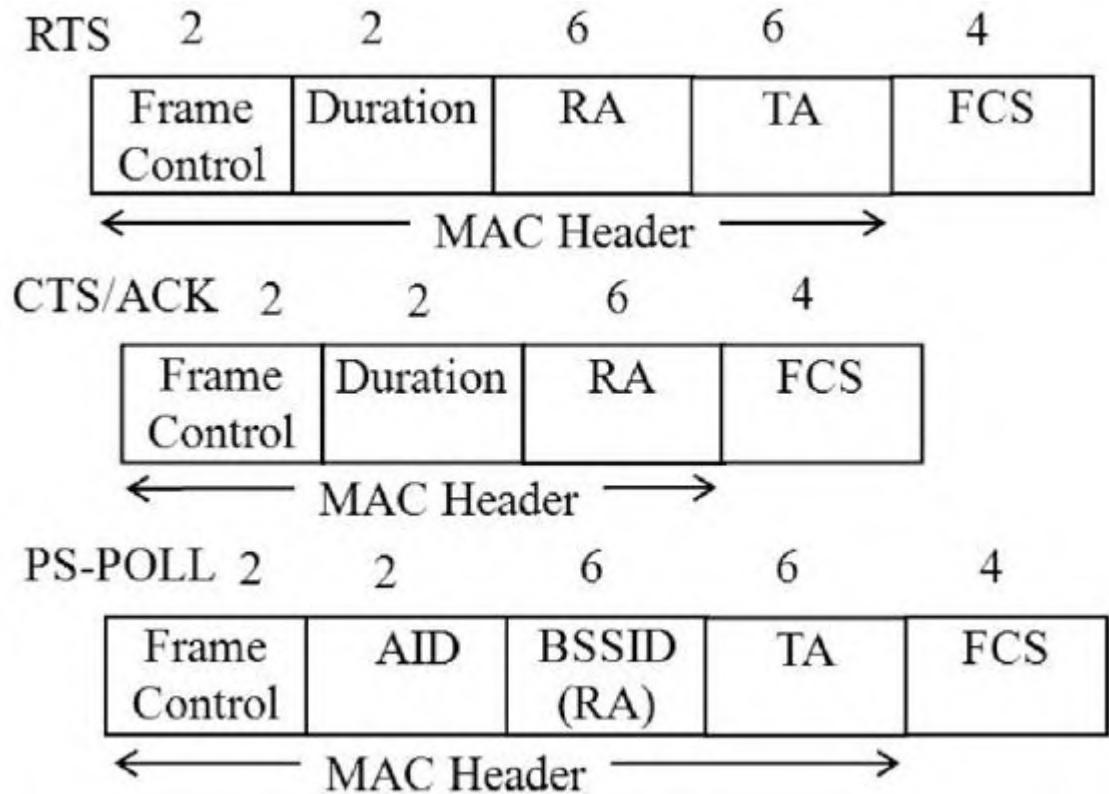


图3-19 控制帧格式

- **RTS (Request To Send) :** 根据3.3.3节关于CSMA/CA的介绍，RTS用于申请无线媒介的使用时间。值为Duration，单位为微秒。Duration的计算大致是：要发送的数据帧或管理帧所需时间+CTS帧所需时间+ACK帧所需时间+3 SIFS <sup>②</sup> 时间。
- **CTS (Clear To Send) :** 也和CSMA/CA有关，用于回复RTS帧。另外它被802.11g保护机制用来避免干扰旧的STA。
- **ACK:** 802.11中，MAC以及任何数据的传输都需要得到肯定确认。这些数据包括普通的数据传输、RTS/CTS交换之前帧以及分片帧。
- **PS-POLL:** 该控制帧被STA用于从AP中获取因省电模式而缓存的数据。其中AID的值是STA和AP关联时，由AP赋给该STA的。

#### 规范阅读提示

- 1) 控制帧还包括CF-End、CF-End+CF-Ack等，这部分内容请读者阅读规范8.3.1节。

2) 上述帧中Duration字段的计算也是一个比较重要的知识点，读者同样可阅读规范8.3.1节以了解其细节。

#### 4. 管理帧<sup>[15][17]</sup>

管理帧是802.11中非常重要的一部分。相比有线网络而言，无线网络管理的难度要大很多。MAC管理帧内容较为丰富，本节将介绍其中重要的几种管理帧，分别是Beacon（信标）帧、Association Request/Response（关联请求/回复）帧、Probe Request/Response（探测请求/回复）帧、Authentication/Deauthentication（认证/取消认证）帧。

介绍具体管理帧之前，先来看看管理帧的格式，如图3-20所示。

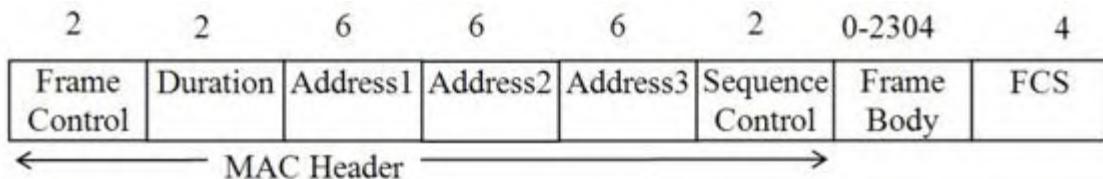


图3-20 管理帧格式

所有管理帧都包括MAC Header的6个域。不过，无线网络要管理的内容如此之多，这6个域肯定不能承担如此重任，所以管理帧中的Frame Body将携带具体的管理信息数据。

802.11的管理信息数据大体可分为两种类型。

- 定长字段：指长度固定的信息，规范称为Fixed Field。
- 信息元素：指长度不固定的信息。显然这类信息中一定会有一个参数用于指示最终的数据长度。

从程序员的角度来看，不同的管理信息数据就好像不同的数据类型或数据结构一样。下面我们来看看802.11为管理帧定义了哪些数据类型和数据结构。

##### (1) 定长字段

管理帧中固定长度的字段如下。

1) Authentication Algorithm Number : 该字段占2字节，代表认证过程中所使用的认证类型。其取值如下。

- 0: 代表开放系统身份认证（Open System Authentication）。
- 1: 代表共享密钥身份认证（Shared Key Authentication）。
- 2: 代表快速BSS切换（Fast BSS Transition）。
- 3: 代表SAE（Simultaneous Authentication of Equals）。用于两个STA互相认证的方法，常用于Mesh BSS网络。
- 65535: 代表厂商自定义算法。

提示 Authentication Algorithm Number的定义是不是可以在代码中用枚举类型来表示呢？

2) Beacon Interval field : 该字段占2字节。每隔一段时间AP就会发出Beacon信号用来宣布无线网络的存在。该信号包含了BSS参数等重要信息。所以STA必须要监听Beacon信号。Beacon Interval field字段用来表示Beacon信号之间间隔的时间，其单位为Time Units（规范中缩写为TU。注意，一个TU为1024微秒。这里采用2作为基数进行计算）。一般该字段会设置为100个TU。

3) Capability Information（性能信息） : 该字段长2字节，一般通过Beacon帧、Probe Request和Response帧携带它。该字段用于宣告此网络具备何种功能。2字节中的每一位（共16位）都用来表示网络是否拥有某项功能。该字段的格式如图3-21所示。

B0	B1	B2	B3	B4	B5	B6	B7
ESS	IBSS	CF Pollable	CF-Poll Request	Privacy	Short Preamble	PBCC	Channel Agility
B8	B9	B10	B11	B12	B13	B14	B15
Spectrum Mgmt	QoS	Short Slot Time	APSD	Radio Measurement	DSSS-OFDM	Delayed Block Ack	Immediate Block Ack

图3-21 Capability Information格式

图3-21中几个常用的功能位如下。

- ESS/IBSS：基础结构型网络中，AP将设置ESS位为1，IBSS位为0。相反，IBSS中，STA设置ESS位为0，而IBSS位为1。Mesh BSS中，这两个位都为0。
  - Privacy：如果传输过程中需要维护数据机密性（data confidentiality），则AP设置该位为1，否则为0。
  - Spectrum Mgmt：如果某设备对应MIB中dot11SpectrumManagementRequired为真，则该位为1。根据802.11 MIB对dot11SpectrumManagementRequired的描述，它和前面介绍的TPC（传输功率控制）和DFS（动态频率选择）功能有关。
  - Radio Measurement：如果某设备对应MIB中的dot11RadioMeasurementActivated值为真，则该位置为1，用于表示无线网络支持Radio Measurement Service。
- 4) Current AP Address：该字段长6字节，表示当前和STA关联的AP的MAC地址。其作用是便于关联和重新关联操作。
- 5) Listen Interval：该值长2字节，和省电模式有关。其作用是告知AP，STA进入PS模式后，每隔多长时间它会醒来接收Beacon帧。AP可以根据该值为STA设置对应的缓存大小，该值越长，对应的缓冲也相应较大。该值的单位是前文提到的另外一个定长字段Beacon Interval。
- 6) Association ID (AID)：该字段长2字节。STA和AP关联后，AP会为其分配一个Association ID用于后续的管理和控制（如PS-POLL帧的使用）。不过为了和帧头中的Duration/ID匹配，其最高2位永远为1，取值范围1~2007。
- 7) Reason Code：该字段长2字节。用于通知Disassociation、Deauthentication等操作（包括DELTS、DELBA、DLS Teardown等）失败的原因。Reason Code的详细取值情况可参考规范的8.4.1.7节的Table 8-36。此处为读者列出几个常见取值及含义，如表3-5所示。

表 3-5 Reason Code 取值

Reason Code 取值	含    义
0	保留
1	未指定
2	前一个身份验证已经无效
3	因为 STA 已经离开 IBSS 或 ESS, 故验证失效
4	因 STA 不活跃时间超时, 故取消关联
5	AP 没有资源再处理新的 STA, 故取消关联

注意 规范一共定义了67个不同的Reason Code值，其详细含义请参考规范的图8-36。

8) Status Code : 该字段长2字节，用于反馈某次操作的处理结果。0代表成功。该字段取值的细节请读者参考规范8.4.1.9节的Table 8-37。

管理帧中定长字段还有其他几个，此处就不一一列举了。读者可参考规范8.4.1节以获取详细信息。下面我们介绍管理帧定义的不定长数据结构，即信息元素（Information Elements, IE）。

## (2) 信息元素

信息元素（IE）包含数据长度不固定的信息，其标准结构如图3-22所示。其中，Element ID表示不同的信息元素类型。Length表示Information字段的长度。根据Element ID的不同，Information中包含不同的信息。

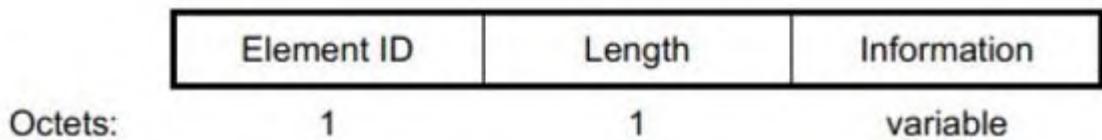


图3-22 IE标准结构

表3-6列出了几种Element ID及Length的取值情况。完整的取值列表请参考规范中Table 8-54。

表 3-6 Element ID/Length 取值

Element	Element ID	Length
SSID	0	0 ~ 32
Supported rates	1	1 ~ 8
Country	7	6 ~ 254
Supported Channels	36	2 ~ 254
Neighbor Report	52	13 ~ 255 (Subelements)
Location Parameters	82	0 ~ 255 (Subelements)
SSID List	84	0 ~ 255

注意，表3-6中最后一列的Subelements表示该项对应的Information采用如图3-23所示的格式来组织其数据。

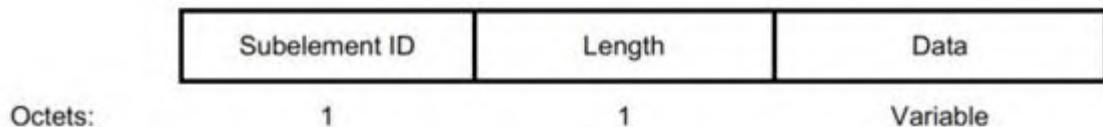


图3-23 Subelement格式

很显然，图3-23和图3-22所示格式完全一样，这也就是Subelement的由来。Subelement的具体内容介绍由Subelement ID来决定。

**规范阅读提示** 规范指出IE中的Length字段代表Information的长度。但规范给出的具体IE信息表（如Table 8-54）其Length一列却是IE的全长（即Element ID字段长+Length字段+Info字段长度）。本节所列表3-5中的Length长仅取Info字段长度。故比Table 8-54中的Length长度少2字节。

### (3) 常用管理帧

了解了定长字段以及信息元素后，下面介绍管理帧中几种重要类型的帧。

1) Beacon帧：非常重要，AP定时发送Beacon帧用来声明某个网络。这样，STA通过Beacon帧就知道该网络还存在。此外，Beacon帧还携带了网络的一些信息。简单来说，Beacon帧就是某个网络的心跳信息。Beacon帧能携带的信息非常多，不过并非所有信息都会包含在Beacon帧中。表3-7介绍了Beacon帧中必须包含的信息。

表 3-7 Beacon 帧必须包含的信息

顺序	信息名	说 明
1	Timestamp	时间戳信息，属于定长字段的一种，长 8 字节，用于同步 BSS 中的 STA。时间单位为微秒
2	Beacon Interval	见前面介绍定长字段时关于“Beacon Interval Field”的介绍
3	Capability	见前面介绍定长字段时关于“Capability Info”的介绍
4	SSID	网络名，用字符串表达

提示 如果读者手头有AirPcap工具，不妨尝试截获一下周围的Beacon帧。

2) Probe Request/Response帧：STA除了监听Beacon帧以了解网络信息外，还可以发送Probe Request帧用于搜索周围的无线网络。规范要求由送出Beacon帧的STA回复Probe Response帧，这样，在基础结构型网络中，Beacon帧只能由AP发送，故Probe Response也由AP发送。IBSS中，STA轮流发送Beacon帧，故Probe Response由最后一次发送Beacon帧的STA发送。

表3-8列出了Probe Request帧中必须包含的信息。

表 3-8 Probe Request 帧必须包含的信息

顺序	信息名	说 明
1	SSID	要搜索的网络名。如果 SSID 长度为 0，代表搜索周围的所有无线网络（这种类型的 SSID 也称为 wild SSID）
2	Supported Rates	STA 会列出自己所支持传输速率。如果 AP 支持这些速率，则它会允许 STA 加入网络
3	Extended Supported Rates	和 Supported Rates 类似，只不过包含的数据量多一些

当AP收到Probe Request时，会响应以Probe Response帧。Probe Response帧包含的内容绝大部分和Beacon帧一样，此处就不拟详细讨论了。读者可参考规范Table 8-27获取详细信息。

提示 图3-18所示为笔者利用AirPcap截获的SSID为“Test”的Probe Request帧，读者不妨回头看看此图。

3) Association Request/Response帧：当STA需要关联到某个AP时，将发送此帧。该帧携带的一些信息项如表3-9所示。

表 3-9 Association Request 帧信息

顺序	信息名	说 明
1	Capability	AP 将检查该字段用于判断 STA 是否满足要求
2	Listen Interval	AP 将根据该值分配 PS 时所需的缓冲
3	SSID	AP 检查 SSID 是否为自己所在的网络
4	Supported Rates	AP 将检查该字段是否满足要求

请读者参考规范Table 8-22获取Association Request帧包含的全部信息项。

针对Association Request帧，AP会回复一个Association Response帧用于通知关联请求处理结果。该帧携带的信息如表3-10所示。

表 3-10 Association Response 帧信息

顺序	信息名	说 明
1	Capability	AP 设置的 Capability
2	Status Code	AP 返回的关联请求处理结果
3	AID	AP 返回关联 ID 给 STA
4	Supported Rates	AP 支持的传输速率

请读者参考规范Table 8-23获取Association Response帧包含的全部信息项。

4) Authentication帧：用于身份验证，其包含的信息如表3-11所示。

表 3-11 Authentication 帧信息

顺序	信息名	说 明
1	Authentication Algorithm number	认证算法类型，见定长字段一节关于“Authentication Algorithm number”的介绍
2	Authentication transaction sequence number	认证过程可能需要好几次帧交换，所以每个帧都有自己的编号
3	Status Code	有些类型的认证会使用该值返回结果
4	Challenge Text	有些类型的认证会使用该字段

至此，我们对802.11 MAC管理帧的相关知识就介绍完毕。规范定义的管理帧类型并不多，一共15种。但管理帧携带的信息却相当复杂，其中定长字段有42种，IE有120种。此处建议读者先了解管理帧中信息的组织方式，然后对最重要的几种管理帧做简单了解。以后碰到不熟悉的情况，可直接查询802.11规范相关小节即可获取最完整的信息。

## 5. 数据帧<sup>[15]</sup><sup>[17]</sup>

图3-24展示了802.11中完整的数据帧格式。

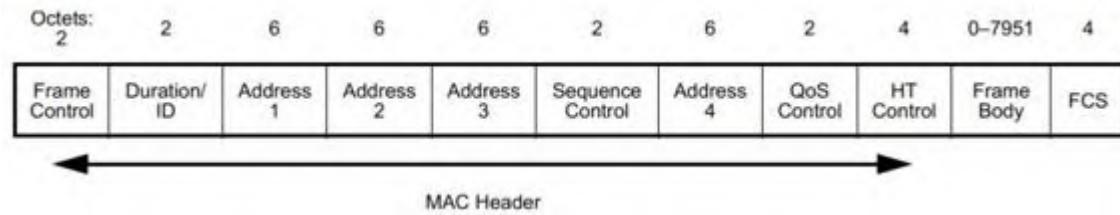


图3-24 数据帧格式

其中，QoS Control和HT（High Throughput）Control字段只在QoS和HT的情况下出现。本书不讨论它们的内容。其他情况下，Frame Body最大长度为2312字节。

提示 图3-24中4个Address字段的使用请参考表3-3和图3-16。

## 6. 802.11上层协议封装<sup>[21]</sup>

本节介绍802.11中如何对上层协议进行格式封装。和Ethernet不同，802.11使用LLC层来封装上层协议，如图3-25所示。

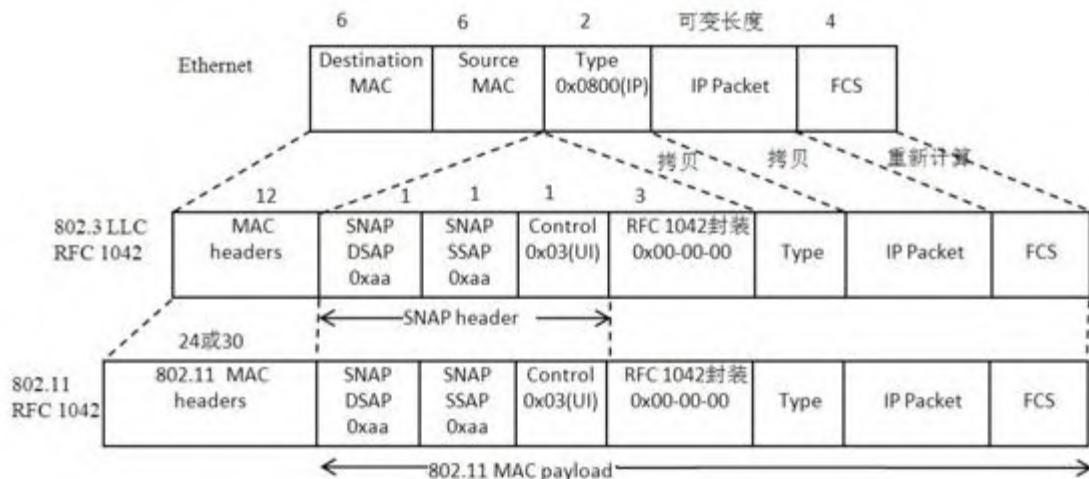


图3-25 802.11上层协议封装

图3-25中，最上层是Ethernet帧格式。前面12字节分别是目标MAC地址以及源MAC地址。Type字段可以为0x0800，代表后面的数据是IP包。

当Ethernet帧要在无线网络上传输时，必须先将其转换为LLC帧，如中间一层所示。这种转换方法由RFC 1042规定。它主要在MAC headers和Type之间增加了4个字段。它们统称为SNAP header（Sub Network Access Protocol，子网访问协议），分别如下。

- DSAP（Destination Service Access Point，目标服务接入点）。
- SSAP（Source Service Access Point，源服务接入点）。
- Control（控制字段，值被设定为0x03，代表Unnumbered Information，即未编号信息）。
- OUI（固定为0x000000）。

LLC数据最后被封装成802.11MAC帧。主要变化是帧头信息，因为802.11 MAC帧最多能携带4个MAC地址。其他信息则作为802.11 MAC帧的数据载荷。

注意 802.11除了可以用RFC 1024来封装Ethernet外，还支持802.1h的封装方法。详情见参考资料[21]。

笔者截获的802.11 MAC帧以及以太网帧IP包如图3-26所示。

```

# Frame 69: 1494 bytes on wire (11952 bits), 1494 bytes captured (11952 bits) on interface
# Ethernet II, Src: Tp-LinkT_1f:9c:5e (00:23:cd:1f:9c:5e), Dst: Giga-Byt_8c:47:d1 (1c:6f:6
#   Destination: Giga-Byt_8c:47:d1 (1c:6f:65:8c:47:d1)
#     Address: Giga-Byt_8c:47:d1 (1c:6f:65:8c:47:d1)
#       ....0. .... .... .... = LG bit: Globally unique address (factory default)
#       ....0. .... .... .... = IG bit: Individual address (unicast)
#   Source: Tp-LinkT_1f:9c:5e (00:23:cd:1f:9c:5e)
#     Address: Tp-LinkT_1f:9c:5e (00:23:cd:1f:9c:5e)
#       ....0. .... .... .... = LG bit: Globally unique address (factory default)
#       ....0. .... .... .... = IG bit: Individual address (unicast)
#     Type: IP (0x0800)
# Internet Protocol Version 4, Src: 180.149.135.224 (180.149.135.224), Dst: 192.168.1.100
# Transmission Control Protocol, Src Port: http (80), Dst Port: 64646 (64646), Seq: 40321,
# *****

# Frame 18107: 118 bytes on wire (944 bits), 118 bytes captured (944 bits) on interface
# Radiotap Header v0, Length 20
# IEEE 802.11 QoS Data, Flags: ...PR..T.
#   Type/Subtype: QoS Data (0x28)
#   Frame Control: 0x1988 (Normal)
#     .000 0001 0011 1010 - Duration: 314 microseconds
#     BSS Id: Ruckuswi_8a:a7:68 (c4:01:7c:8a:a7:68)
#     Source address: chimeico_10:83:f2 (cc:f3:a5:10:83:f2)
#     Destination address: Engineer_1d:f8:59 (00:0d:78:1d:f8:59)
#     Fragment number: 0
#     Sequence number: 261
#   Frame check sequence: 0x3dfa7b42 [incorrect, should be 0x3a7e39e4]
#   QoS Control
# Logical-Link Control
#   DSAP: SNAP (0xaa)
#   IG Bit: Individual
#   SSAP: SNAP (0xaa)
#   CR Bit: Command
#   Control field: U, func=UI (0x03)
#     Organization Code: Encapsulated Ethernet (0x000000)
#     Type: IP (0x0800)
# Internet Protocol Version 4, Src: 172.16.10.255 (172.16.10.255), Dst: 125.39.24.79 (12
# Transmission Control Protocol, Src Port: 36394 (36394), Dst Port: 10482 (10482), Seq:

```

图3-26 以太网帧和802.11 MAC帧

图3-26上半部分为Ethernet的IP帧封装情况，下半部分为IP包在802.11中的封装情况。

**注意** 图3-26中的黑框表示Radiotap头信息。它是网卡添加在802.11 MAC头部前的数据，记录了信号强度、噪声强度和传输速率等物理层信息。关于Radiotap更多信息，请读者参考  
<http://www.radiotap.org/>。

通过对MAC提供的服务以及MAC帧相关知识的介绍，读者会发现这部分难度主要集中在802.11 MAC帧上，尤其是管理帧包含的信息更是非常丰富。

从程序员角度看，可以认为本节为802.11定义了大量的数据结构和数据类型。从下一节开始，我们将介绍MAC层中的“类和函数”。

## 规范阅读提示

- 1) 本节内容主要取自规范的第8章“Frame Formats”。
  - 2) 由于篇幅问题，本书不介绍802.11 MAC层的功能。这部分内容主要集中在规范的第9章“MAC sublayer functional description”。
- ① 此条根据审稿专家的反馈意见修改。Infrastructure BSS中，BSSID一般为AP的MAC地址。

### 3.3.6 802.11 MAC管理实体<sup>[22][23]</sup>

802.11包括了MAC层和PHY层。根据图3-4可知这两层内部都对应有Entity，它们通过SAP（Service Access Point）为对应的上层提供服务。图3-27展示了802.11中的Entity和SAP。

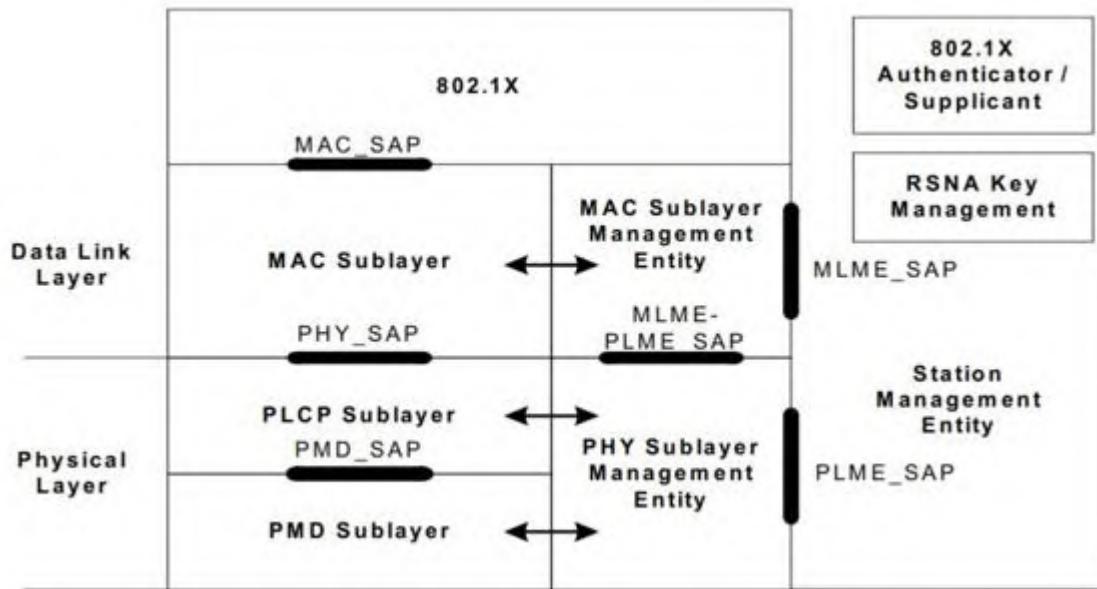


图3-27 802.11 Entity模型

由图3-27可知：

- **MAC\_SAP**为LLC层提供服务，其具体内容见3.3.5节“MAC服务定义”。
- **MAC子层**中还有专门负责管理的Entity，名为**MLME**（**MAC Sublayer Management Entity**），它对外提供的接口是**MLME\_SAP**。
- **PHY层**还可细分为**PLCP**和**PMD**两层（本书不讨论）。
- **物理层**对外提供的管理实体是**PLME**（**PHY Sublayer Management Entity**），对应的SAP缩写为**PLME\_SAP**。

- 为了方便对802.11 MAC及PHY层统一操作和管理，规范还定义一个SME（Station Management Entity）。该实体独立于MAC和PHY层。使用者可通过SME实体来操作MAC层和PHY层中的SAP。

本节重点关注MAC的管理Entity（即MLME）及其对应的SAP。规范中关于MLME\_SAP一共有82个原语，其中一部分如图3-28所示。

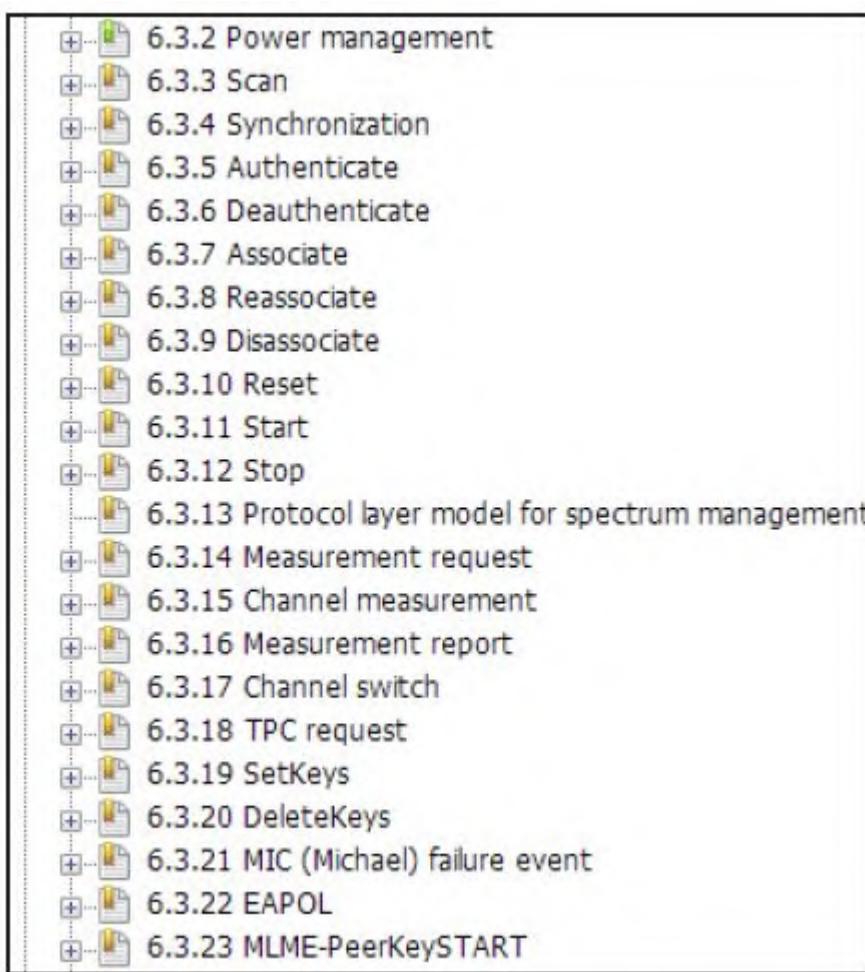


图3-28 MLME SAP部分内容

原语的定义曾在3.3.5节MAC服务定义中见过，它其实就是通过定义API来表达自己所具有的功能。MLME\_SAP非常多，本章不可能全部覆盖，此处仅介绍三个常用的原语。

- Scan：用于扫描周围的无线网络。

- **Authenticate**: 关联到某个AP前，用于STA的身份验证。
- **Associate**: 关联某个AP。关联成功后，STA就正式加入无线网络了。

**提醒** 请读者务必注意一点，理解MLME\_SAP原语对掌握Wi-Fi技术非常重要。从编程角度来说，MLME\_SAP相当于定义了一套接口函数，而后续章节将介绍的wpa\_supplicant只是对它们的实现。

## 1. Scan介绍

原语的参数比较多，但并不是所有参数都需要（由对应的MIB项控制），本书仅介绍比较重要的参数（用加粗字体表示）。

### (1) request

Scan.request原语用于扫描周围的无线网络，其原型如下。

```
MLME-SCAN.request(  
    BSSType, BSSID, SSID, ScanType, ProbeDelay,  
    ChannelList, MinChannelTime, MaxChannelTime,  
    RequestInformation, SSID List,  
  
    ChannelUsage, AccessNetworkType, HESSID, MeshID, VendorSpecificInfo  
)
```

Scan参数中，比较几个重要的分别如下。

- **BSSType**: 类型为枚举（Enumeration），可取值为INFRASTRUCTURE、INDEPENDENT、MESH、ANY\_BSS。
- **BSSID**: 类型为MAC地址。可以是某个指定的BSSID或者广播BSSID。
- **SSID**: 类型为字符串。0~32字节长。指定网络名，如果长度为0，则为wild ssid。
- **ScanType**: 类型为枚举，可取值为ACTIVE（主动）、PASSIVE（被动）。详情见下文。

- ProbeDelay: 类型为整型，单位为微秒。用于ACTIVE模式的扫描，详情见下文。
- ChannelList: 类型为有序整数列表（Ordered set of integers），扫描时使用。
- MinChannelTime和MaxChannelTime: 类型均为整型。用于指示扫描过程中在每个信道上等待的最小和最长时间。时间单位为TU。

由Scan.request的ScanType参数可知，802.11规定了两种扫描模式。

- ACTIVE模式：这种模式下，STA在每个Channel（信道）上都会发出Probe Request帧用来搜索某个网络。具体工作方式是，STA首先调整到某个信道，然后等待来帧指示（表明该信道有人使用）或超时时间ProbeDelay。两个条件满足任意一个，STA都将发送Probe Request帧，然后STA在该信道上等待最少MinChannelTime时间（如果在此时间内信道一直空闲），最长MaxChannelTime时间。
- PASSIVE模式：这种模式下，STA不发送任何信号，只是在ChannelList中各个信道间不断切换并等待Beacon帧。根据前述介绍可知，在基础结构型网络中AP会定时发送Beacon帧以宣告网络的存在。所以，PASSIVE模式下，STA能根据Beacon帧来了解周围所存在的无线网络。

## (2) confirm

Scan.confirm用于通知扫描结果，其原型定义如下。

```
MLME-SCAN.confirm (
    BSSDescriptionSet, BSSDescriptionFromMeasurementPilotSet,
    ResultCode, VendorSpecificInfo
)
```

- BSSDescriptionSet: 类型为BSSDescription列表。具体内容见下文。
- ResultCode: 类型为枚举，可取值为SUCCESS和NOT\_SUPPORTED。

BSSDescription包含很多内容，常见项如表3-12所示。

表 3-12 BSSDescription 内容

名 称	类 型	取 值 情 况	说 明
BSSID	MACAddress		BSS 的 BSSID 或 Mesh STA 的 MAC 地址
SSID	Octet String	0 ~ 32 字节	BSS 的 SSID
BSSType	Enumeration	INFRASTRUCTURE、INDEPENDENT、MESH	BSS 的网络类型
Beacon Period	Integer		见 3.3.5 节
IBSS Parameter Set	IBSS Parameter Set element		如果类型是 IBSS，用于说明 IBSS 的参数
CapabilityInformation	Capability Information field		见 3.3.5 节

获取周围的无线网络后，STA可以选择加入（join）其中的一个BSS。规范没有定义网络加入的原语，但实际上大部分无线网卡在实现时，都需要join相关的处理。因为如果周围存在多个无线网络时，需要用户参与来选择加入哪一个网络。BSSDescription全部取值列表请参考规范6.3.3.3.2节。

## 2. Authenticate介绍

关联到某个AP前，STA必须通过身份验证。该处理由Authenticate对应的原语来完成。由于身份验证涉及两个STA（以基础结构型网络为例，一个是STA，另一个是AP），所以Authenticate包含4个原语，分别如下。

- MLME-Authenticate.request: STA A向AP B发起身份验证请求。
- MLME-Authenticate.confirm: STA A收到来自AP B的身份验证处理结果。
- MLME-Authenticate.indication: AP B收到来自STA A的身份验证处理请求。
- MLME-Authenticate.response: AP B向STA A发送身份验证处理结果。

先来看request和confirm原语。

### (1) request和confirm

request和confirm原语原型如下。

```
MLME-AUTHENTICATE.request(
    PeerSTAAddress, AuthenticationType,
    AuthenticateFailureTimeout,
    Content of FT Authentication elements,
    Content of SAE Authentication Frame,
    VendorSpecificInfo
)
// confirm原型
MLME-AUTHENTICATE.confirm(
    PeerSTAAddress, AuthenticationType, ResultCode,
    Content of FT Authentication elements,
    Content of SAE Authentication Frame,
    VendorSpecificInfo
)
```

上述原语定义说明如下。

- PeerSTAAddress: 类型为MAC地址，代表对端STA的地址。以本例而言，则是AP B的MAC地址。
- AuthenticationType: 类型为枚举，可取值有OPEN\_SYSTEM、SHARED\_KEY、FAST\_BSS\_TRANSITION和SAE。用于表示认证过程中使用的认证类型。这部分内容见3.3.7节无线网络安全相关介绍。
- AuthenticateFailureTimeout: 类型为整型，代表认证超时时间，单位为TU。
- ResultCode: 代表认证处理结果。

MLME-AUTHENTICATE.request将触发STA A发送Authenticate帧。下面来看AP B如何处理收到的这个Authenticate帧呢？

## (2) indication和response

这两个原语定义和request以及confirm基本一样。

```
MLME-AUTHENTICATE.indication(
    PeerSTAAddress, AuthenticationType,
    Content of FT Authentication elements,
    Content of SAE Authentication Frame,
```

```

        VendorSpecificInfo
)
// response原型
MLME-AUTHENTICATE.response(
    PeerSTAAddress, AuthenticationType, ResultCode,
    Content of FT Authentication elements,
    Content of SAE Authentication Frame,
    VendorSpecificInfo
)

```

indication和对应的response参数与request以及confirm一样，此处不详述。

### 3. Associate介绍

STA通过身份验证后，就需要和AP关联。只有关联成功后，STA才正式成为无线网络的一员。

注意 对于RSN，关联成功后还需通过802.1X身份验证。相关内容留待3.3.7节介绍。

Associate包含的原语和Authentication一样，都有request、confirm、indication和response。我们先来看request和confirm。

#### (1) request和confirm

request和confirm原语定义如下。

```

MLME-ASSOCIATE.request(
    PeerSTAAddress,
    AssociateFailureTimeout, CapabilityInformation,
    ListenInterval, Supported Channels,
    RSN,
    QoS Capability, Content of FT Authentication
    elements, SupportedOperatingClasses,
    HT Capabilities, Extended Capabilities, 20/40 BSS Coexistence,
    QoSTrafficCapability,
    TIMBroadcastRequest, EmergencyServices, VendorSpecificInfo
)
// confirm原语
MLME-ASSOCIATE.confirm(
    ResultCode, CapabilityInformation, AssociationID, SupportedRates,

```

```
EDCAParameterSet, RCPI.request, RSNI.request, RCPI.response,  
RSNI.response, RMEnabledCapabilities, Content of FT  
Authentication elements,  
SupportedOperatingClasses, HT Capabilities, Extended  
Capabilities,  
20/40 BSS Coexistence,  
TimeoutInterval, BSSMaxIdlePeriod, TIMBroadcastResponse,  
QosMapSet, VendorSpecificInfo  
)
```

上述原语中包含的参数信息如下。

- PeerSTAAddress: 响应Association请求的STA的MAC地址，即AP的地址。
- AssociateFailureTimeout: 类型为整型，代表关联超时时间，单位为TU。
- CapabilityInformation: 指定AP的性能信息。
- ListenInterval: 用于告知AP，STA进入PS模式后，监听Beacon帧的间隔时间。
- RSN: 类型为RSNE，指示STA选设置的安全方面的信息。详情见3.3.7节。
- ResultCode: AP返回的处理结果。
- AssociationID: AP返回的关联ID。
- SupportedRates: AP返回的所支持的传输速率列表。速率以500kbps为单位。

## (2) indication和response

indication和response原语定义如下。

```
MLME-ASSOCIATE.indication(  
PeerSTAAddress, CapabilityInformation, ListenInterval, SSID, Suppor  
tedRates,  
RSN, QoS Capability, RCPI, RSNI, RMEnabledCapabilities,
```

```
    Content of FT Authentication
    elements, SupportedOperatingClasses,
    DSERegisteredLocation,
    HT Capabilities, Extended Capabilities, 20/40 BSS
    Coexistence, QoS Traffic Capability,
    TIM Broadcast Request, Emergency Services, Vendor Specific Info
)
// response原语
MLME-ASSOCIATE.response(
PeerSTAAddress, ResultCode, CapabilityInformation, AssociationID,
EDCAParameterSet, RCPI, RSNI, RMEnabledCapabilities,
Content of FT Authentication elements,
SupportedOperatingClasses, DSERegisteredLocation, HT Capabilities,
Extended Capabilities,
20/40 BSS Coexistence,
Timeout Interval, BSS Max Idle Period, TIM Broadcast Response,
QoS Map Set, Vendor Specific Info
)
```

上述原语和参数中，只有 indication 的 SSID 略有不同，它代表发起关联请求的 STA 的 MAC 地址。

#### 4. STA 状态转换<sup>[24]</sup>

上面原语操作成功后，STA 状态将发生变化。STA 从最初到最终的状态切换如图 3-29 所示。

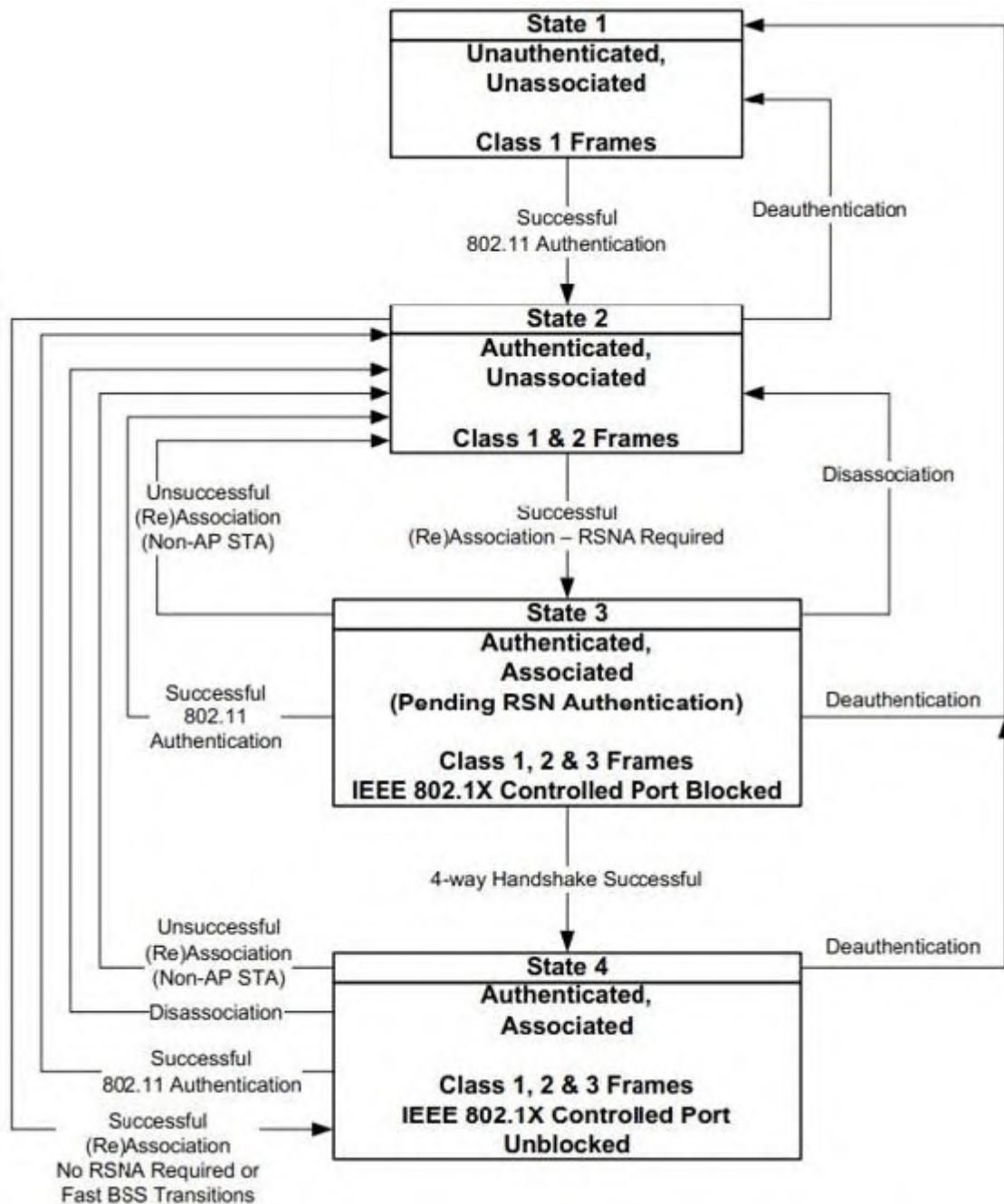


图3-29 STA状态切换

图3-29中包含很多知识点，下面一一介绍它们。首先是MAC帧Frame的类别（Class），规范定义MAC帧一共有三种类别，分别是Class 1、Class 2和Class 3，各自包含不同的MAC帧，如表3-13所示。

表 3-13 MAC 帧分类

类别	控制帧	管理帧	数据帧
Class 1	RTS、CTS、ACK 等	Probe Request/Response、Beacon、Authentication、Deauthentication、Public Action 等	无
Class 2	无	Association Request/Response、Reassociation Request/Response	无
Class 3	PS-POLL 等	除 Class 1 和 Class 2 包含的管理帧之外的其他管理帧	基础型结构网络中的所有数据帧

结合图3-29，可知STA在不同状态下发送的MAC帧类别也不同。这主要是和网络安全有关，没有通过身份验证的STA是不允许随意传输数据的。图3-29中，STA的状态转换如下（以基础结构型网络为例）。

- STA首先处于State 1，即未认证和未关联的情况。为了能加入无线网络，它需要发送Authentication帧给AP。如果认证成功，将转入State 2。State 1情况下不能发送数据帧。
- State 2是已认证，未关联状态。STA将发送Association帧给AP进行关联。成功后，进入State 3。State 2情况下也不能发送数据帧。
- State 3属于已认证，已经关联，但还未通过RSN（Robust Security Network，强健安全网络）认证的状态。RSN采用802.1X进行控制。由于未通过RSN认证，所以只能发送处理认证的数据帧，即4-Way Handshake（四次握手）帧。这部分内容在3.3.7节介绍。
- 4-Way Handshake成功后，STA进入State 4。此时它完全加入无线网络，所有数据帧就能正常传输。

MAC管理实体包含的功能很多，不过对程序员来说，其理解难度反而较小，因为它定义的原语类似代码中的API，而且每个参数的作用有详尽的解释。读者以后在分析wpa\_supplicant的时候，不妨多回顾本节内容。

### 3.3.7 无线网络安全技术知识点

安全是无线网络技术中一个很重要的部分，它主要有三个保护点。

- 数据的完整性（Integrity）：用于检查数据在传输过程中是否被修改。
- 数据的机密性（Confidentiality）：用于确保数据不会被泄露。
- 身份验证和访问控制（Authentication and Access Control）：用于检查受访者的身份。

**提示** 身份验证和访问控制的问题在无线网络中尤为明显。因为Wi-Fi以空气作为传播媒介，这意味着任何人通过类似AirPcap这样的工具就能获取其周围的无线通信数据。而在以太网环境中，虽然网络数据也是通过广播方式发送，但用户需要把网线插到网口中才能监听数据。相信没有哪个公司能随随便便允许一个外人把网线插到其公司内部的网口上。由于无线网络天然就不存在有线网络中这种类似的物理隔离，所以身份验证和访问控制是无线网络中非常重要的一个问题。

无线网络安全技术也是随着技术发展而逐渐在演变，下面介绍其主要发展历程。

- 802.11规则首先提出了WEP（Wired Equivalent Privacy，有线等效加密）。如其名所示，它是为了达到和有线网络同样的安全性而设计的一套保护措施。
- 经过大研究，人们发现WEP很容易破解。在802.11规范提出新解决方案之前，WFA提出了一个中间解决方案，即WPA（Wi-Fi Protected Access）。这个方案后来成为802.11i草案的一部分。
- 802.11i专门用于解决无线网络的安全问题。该规范提出的最终解决方案叫RSN（Robust Security Network，强健安全网络），而WFA把RSN称为WPA2。

本节先介绍WEP，然后介绍RSN中的数据加密，最后将介绍EAP、802.1X和RSNA密匙派生等内容。

注意 本书仅介绍数据加密及完整性校验的大致工作流程，而算法本身的内容请读者自行研究。

另外，安全知识点中会经常见到Key（密钥）和Password（密码，也叫Passphrase）这两个词。它们本质意思都一样，只不过Password代表可读（human readable，如字符串、数字等）的Key，而Key一般指由算法生成的不可读（如二进制、十六进制数据等）的内容。

## 1. WEP介绍

WEP是802.11规范的元老，在1999年规范刚诞生时就存在了。但随着技术的发展，如今WEP已经无力应付复杂险恶网络环境中的安全问题了。先来介绍WEP中的数据加密。

### (1) WEP中的数据加密<sup>[16][25][26]</sup>

保护数据Confidentiality的一个主要方法就是对所传输的数据进行加密，而WEP采用了流密码（Stream Cipher）对数据进行加密。这种加密方法的原理如图3-30所示。

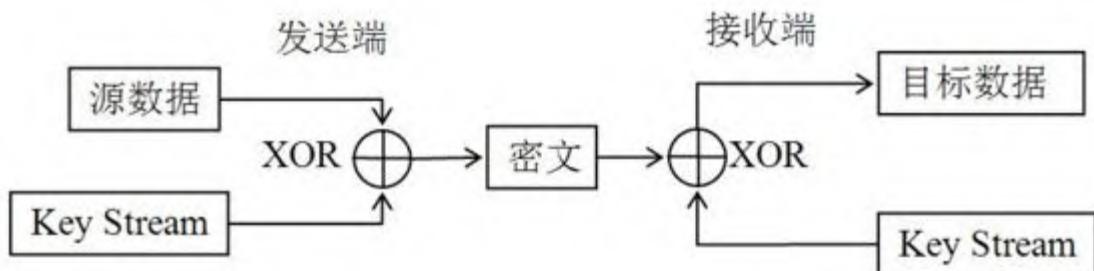


图3-30 流密码工作原理

流密码的工作原理如下。

- 源数据通过Key Stream（密钥流）进行加密，加密方式就是异或（XOR），得到密文。
- 接收端得到密文后，再用相同的Key Stream进行异或操作，这样就能得到目标数据。两次异或操作，Key Stream又一样，所以目标数据就是源数据。

- 整个过程中，只要Key Stream保证随机性，那么不知道Key Stream的人理论上就无法破解密文。

根据上述内容可知，流密码安全性完全依赖于Key Stream的随机性，那么怎么生成它呢？一般的做法是：用户输入一个Secret Key（即密码）到伪随机数生成器（PseudoRandom Number Generator, PRNG）中，而PRNG会根据这个Secret Key生成密钥流。

下面回到WEP，它的加密处理过程如图3-31所示。

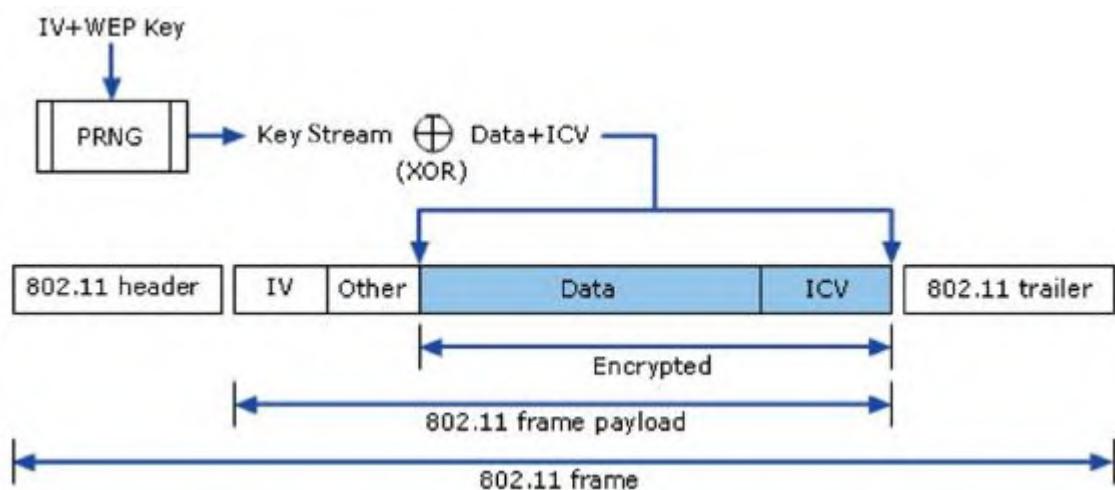


图3-31 WEP加密过程

WEP加密过程如下。

- 左上角PRNG的输入（即随机数种子），包括一个IV（Initialization Vector，初始向量，长24位）和WEP Key（Key长度有40、104和128位。位数越多，安全性越高）。IV的值由设备厂商负责生成。PRNG最终输出Key Stream。
- XOR（即加密）操作的一方是Key Stream，另一方是数据和ICV（Integrity Check Value，完整性校验值）。ICV长32位，其值由CRC-32方法对Data进行计算后得到。XOR操作后的数据即图3-29所示的Encrypted部分。可知Data和ICV都经过了加密。
- 最终的MAC帧数据（frame payload）包括IV以及加密后的Data以及ICV。

再来看WEP的解密过程，如图3-32所示。

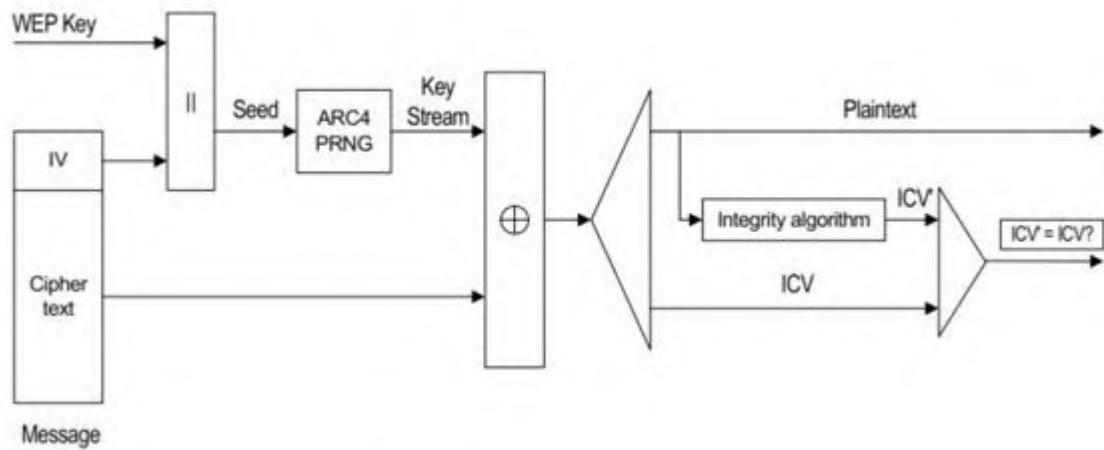


图3-32 WEP解密过程

WEP解密过程如下。

- 帧数据中的IV和WEP Key共同作为PRNG的输入以得到Key Stream。
- Key Stream和帧数据中的Cipher Text执行异或操作，得到明文(Plaintext)以及ICV。
- 接收端同时对自己解密得到的明文进行完整性校验，得到 $ICV'$ 。如果 $ICV$ 和 $ICV'$ 一样，则数据正确。

提醒 WEP弱点很多，其最主要的一个问题就是所有数据都使用同一个WEP Key进行加密。至于WEP存在的其他问题，读者可阅读参考资料[25]以获得更详细的信息。

## (2) WEP中的身份验证<sup>[27][28]</sup>

WEP其实并没有考虑太多关于身份验证方面的事情，根据IEEE 802.11-2012规范，本节真正的名字应该叫”Pre-RSNA Authentication”，即RSNA出现之前的身份验证。不过由于Pre-RSNA的认证方法和WEP有密切关系，所以一般就叫WEP身份验证。

WEP身份验证有以下两种。

• 开放系统身份验证（Open System Authentication）：这种验证其实等同于没有验证，因为无论谁来验证都会被通过。那它有什么用呢？规范规定，如果想使用更先进的身份验证（如RSNA），则STA在发起Authentication请求时，必须使用开放系统身份验证。由于开放系统身份验证总是返回成功，所以STA将接着通过Association请求进入图3-29中的State 3然后开展RSNA验证。

• 共享密钥身份认证（Shared Key Authentication）：Shared Key这个词以后还会经常碰到，它表示共享的密码。例如在小型办公及家庭网络（Small Office/Home Office, SOHO）环境中，AP的密码一般很多人（即很多STA）都知道。

**提示** 初看上去，共享密钥身份验证的安全性比开放系统要强，但实际却恰恰相反。因为使用了共享密钥身份验证就不能使用更为安全的RSNA机制。

由于笔者家庭网络就使用了共享密钥身份验证，故这里也简单向读者介绍一下其工作原理，如图3-33所示。

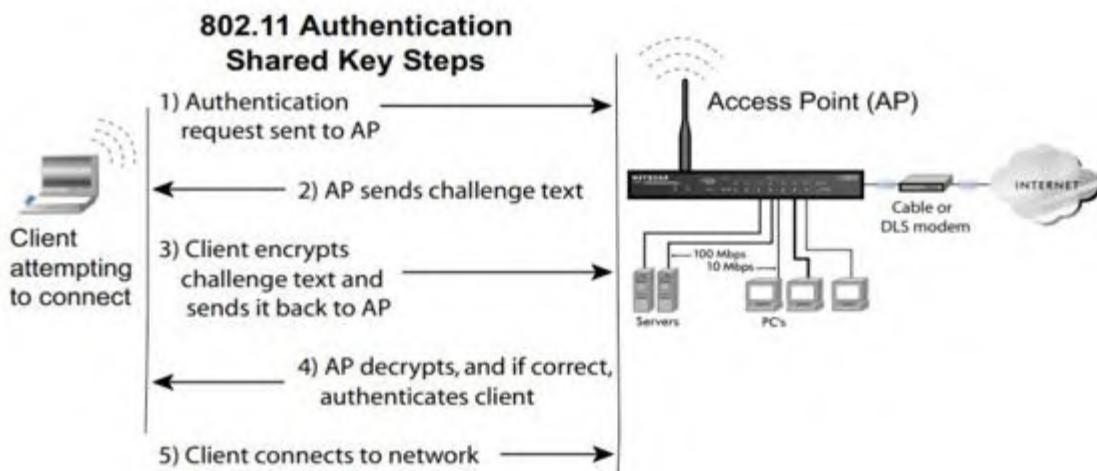


图3-33 共享密钥身份验证原理

由图3-33可知Shared Key验证的方法包括四次帧交互。

- STA（即图中的Client）发送Authentication请求给AP。
- AP收到请求后，通过Authentication Response发送一个质询明文（Challenge Text，长度为128字节）给STA。

- STA取出Challenge Text，通过图3-31所示的WEP加密方法利用自己设置的密钥对质询文进行加密，然后再次发送Authentication Request给AP。
- AP收到第二次Authentication Request帧后，会利用AP的密钥进行解密，同时还要验证数据完整性。如果ICV正确，并且解密后的译文等于之前发送的质询文，则可知STA拥有正确的密码，故STA身份验证通过。
- AP返回验证处理结果给STA。如果成功，STA后续将通过Association请求加入该无线网络。

WEP的介绍就到此为止，下面来介绍RSNA中的数据加密。

## 2. RSN数据加密及完整性校验<sup>[29][30]</sup>

RSN中数据加密及完整性校验算法有两种，分别是TKIP和CCMP。下面分别来介绍它们。

**规范阅读提示** 规范中还有一种广播/组播管理帧完整性校验的方法，叫BIP（Broadcast/Multicast Integrity Protocol）。请读者自行阅读规范11.4.4节以了解相关内容。

介绍TKIP前，先介绍WPA。WPA是802.11i规范正式发布前用于替代WEP的一个中间产物。相比WEP，WPA做了如下改动。

- WPA采用了新的MIC（Message Integrity Check，消息完整性校验）算法用于替代WEP中的CRC算法。新算法名为Michael。
- 采用TKIP（Temporal Key Integrity Protocol，临时密钥完整性协议）用于为每一个MAC帧生成不同的Key。这种为每帧都生成单独密钥的过程称为密钥混合（Key Mixing）。

### （1）TKIP加密

下面简单介绍TKIP的加密过程，如图3-34所示。

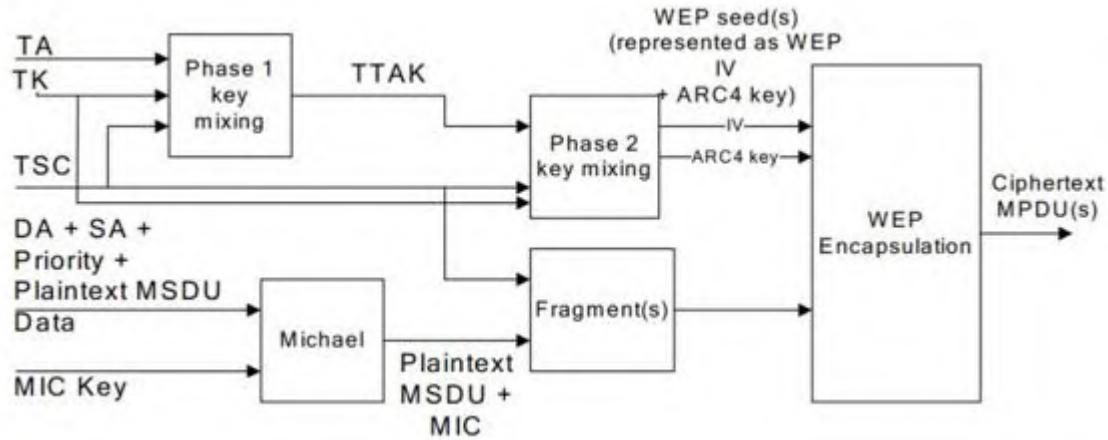


图3-34 TKIP加密过程

- 左下角：用于数据完整性校验的Michael算法的输入包括两部分，一部分是MIC Key（由厂商负责实现），另一部分是MAC帧头中的DA、SA、Priority和MAC帧数据。Michael算法的输出为Data和MIC。它们将作为后续加密算法的输入（等同于图3-31中加密前的Data+ICV）。
- 左上角：TKIP的密钥计算分为两部分。首先是第一阶段的密钥混合，其输入有TA（Transmitter Address, 32位）、TK（Temporal Key, 128位）和TSC（TKIP Sequence Counter, 序列号计数器。共48位，此处取其前32位）。此阶段密钥混合段的产物是TTAK（TKIP-mixed Transmit Address and Key, 长80位）。注意，TK的来历见后文关于密匙派生的知识。
- 中间部分：Phase 2 key mixing属于密钥计算的第二部分。本阶段计算的输入有TTAK和TSC（取其后16位），其产物可用作后续加密计算的WEP种子（包括一个128位的ARC4密钥以及IV，可参考图3-31）。
- 最后一步：利用WEP的加密方法进行数据加密，其过程即先利用WEP种子和PRNG生成密钥流（由于每一个待加密的帧都会有不同的TSC，导致在进行PRNG前，每次输入都有不同的WEP Key），然后使用XOR操作对Data和MIC进行异或。

由上述内容可知，TKIP将为每一帧数据都使用不同的密钥进行加密，故其安全性比WEP要高。另外，由于生成密钥和计算完整性校验时会把MAC地址（如DA、SA、TA）等信息都考虑进去，所以它可以抵抗几种不

同类型的网络攻击[30]。不过，从加密本身来考虑，TKIP和WEP一样都属于流密码加密。

## (2) CCMP加密

CCMP出现在WPA2中，它比TKIP更安全，因为它采用了全新的加密方式CCMP（Counter Mode with CBC-MAC Protocol，计数器模式及密码块链接消息认证码协议），这是一种基于AES（Advanced Encryption Standard，高级加密标准）的块的安全协议。

CCMP加密过程如图3-35所示。

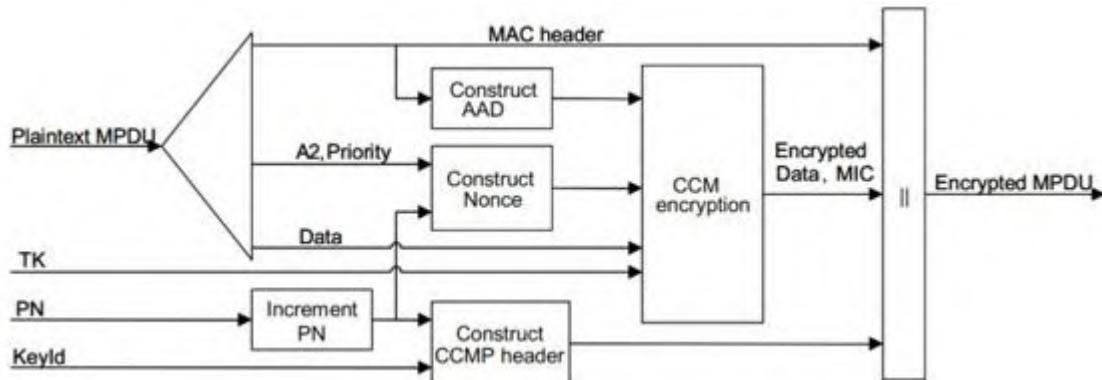


图3-35 CCMP加密过程

- 左下角：PN（Packet Number，帧编号，48位）和KeyId（Key Identifier，密钥标识符，每帧使用一个密钥）共同构成一个8字节的CCMP头（CCMP Header）。另外，每一帧的PN都会累加，所以图中有一个“Increment PN”处理框（注意，重传包的PN不累加）。
- 左上角：Plaintext MPDU（注意不是MSDU）意味着整个MAC帧（包括Head）都是输入参数。MPDU被拆分成三个部分，其中A2（Address 2字段）、Priority和PN共同构成Nonce（即临时的随机数，用一次后就丢弃不再使用）。MPDU中的MAC头信息将构成AAD（Additional Authentication Data，附加认证数据）。
- AAD、Nonce、MPDU中的MAC数据（Payload）以及TK（Temporary Key）共同作为加密算法的输入，最终得到加密后的数据及消息校验码（Message Integrity Check，MIC）。

- CCMP Header、MAC Header、加密后的Data以及MIC共同构成了MPDU包。

关于RSN中的加密算法TKIP及CCMP就介绍到此。由于加解密工作上由硬件来完成，读者仅需了解其中涉及的概念即可。

### 3. EAP和802.1X介绍

从本节开始，我们将重点关注安全主题中最后一个重要的部分，即身份验证。首先介绍EAP，然后介绍802.1X。

**提示** wpa\_supplicant很大一部分内容就是在处理身份验证，所以本节也将花费较多的笔墨来介绍相关内容。

#### (1) EAP<sup>[31]</sup><sup>[32]</sup><sup>[33]</sup>

目前身份验证方面最基础的安全协议就是EAP（Extensible Authentication Protocol），协议文档定义在RFC3748中。EAP是一种协议，更是一种协议框架。基于这个框架，各种认证方法都可得到很好的支持。下面来认识EAP协议。首先介绍其中涉及的几个基本概念。

- **Authenticator**（验证者）：简单点说，Authenticator就是响应认证请求的实体（Entity）。对无线网络来说，Authenticator往往是AP。
- **Supplicant**（验证申请者<sup>①</sup>）发起验证请求的实体。对于无线网络来说，Supplicant就是智能手机。
- **BAS**（Backend Authentication Server，后端认证服务器）：某些情况下（例如企业级应用）Authenticator并不真正处理身份验证，它仅仅将验证请求发给后台认证服务器去处理。正是这种架构设计拓展了EAP的适用范围。
- **AAA**（Authentication、Authorization and Accounting，认证、授权和计费）：另外一种基于EAP的协议。实现它的实体属于BAS的一种具体形式，AAA包括常用的RADIUS服务器等。在RFC3748中，AAA和BAS的概念可互相替代。

- EAP Server: 表示真正处理身份验证的实体。如果没有BAS，则EAP Server功能就在Authenticator中，否则该功能由BAS实现。

EAP架构如图3-36所示。

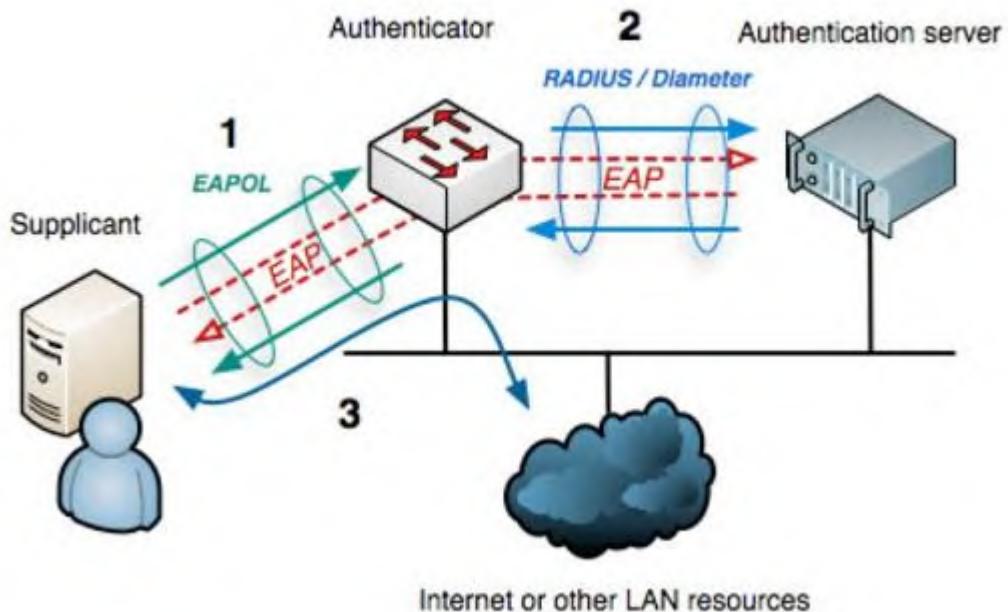


图3-36 EAP架构

由图3-36所示，Suplicant通过EAPOL（EAP Over LAN，基于LAN的扩展EAP协议）发送身份验证请求给Authenticator。图中的身份验证由后台验证服务器完成。如果验证成功，Suplicant就可正常使用网络了。

下面来认识EAP的协议栈，如图3-37所示。

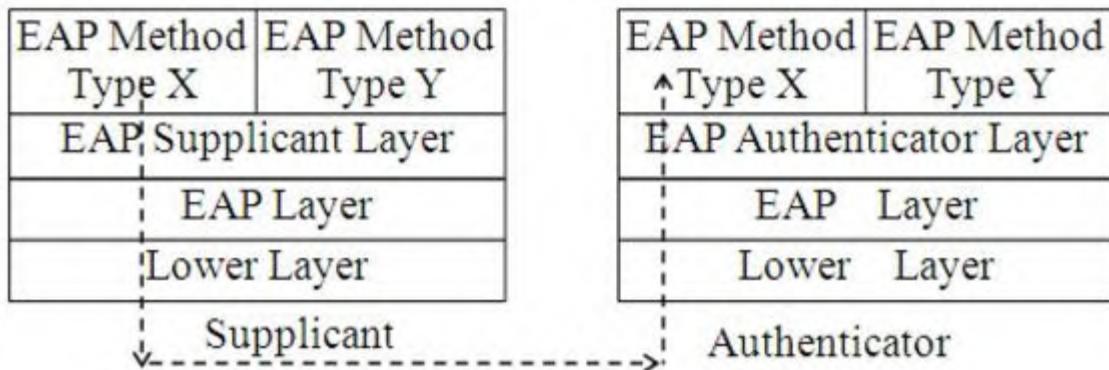


图3-37 EAP协议栈

由图3-37可知，EAP没有强制定义位于最底层（Lower Layer, LL）的传输层。目前支持EAP协议的网络有PPP、有线网（EAPOL，也就是802.1X协议）、无线网络（即802.11 WLAN）、TCP、UDP等。另外，LL本身的特性（例如以太网和无线网都支持数据重排及分片的特性）会影响其上一层EAP的具体实现。

- EAP Layer用于收发数据，同时处理数据重传及重复（Duplicate）包。
- EAP Supplicant/Authenticator Layer根据收到EAP数据包中Type字段（见下文介绍）的不同，将其转给不同的EAP Method处理。
- EAP Method属于具体的身份验证算法层。不同的身份验证方法有不同的Method Type。

EAP协议的格式非常简单，如图3-38所示。

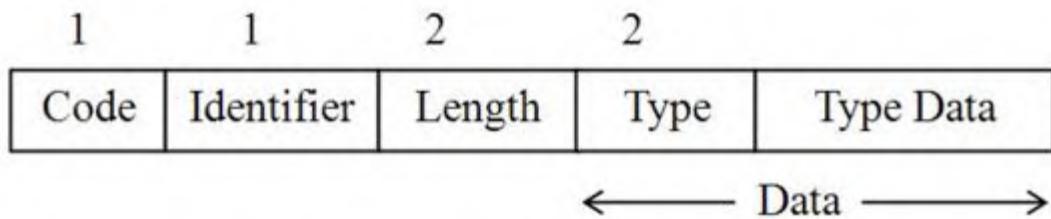


图3-38 EAP协议格式

- Code: EAP协议第一字节，目前仅有四种取值，分别为1（Request）、2（Response）、3（Success）、4（Failure）。
- Identifier: 消息编号（ID），用于配对Request和Response。
- Length: 2字节，用于表示EAP消息包长度（包括EAP头和数据的长度）。
- Data: EAP中具体的数据（Payload）。当Code为Request或Response的时候，Data字段还可细分为Type以及Type Data。Type就是图3-37中的EAP Method Type。

EAP Method Type取值如下。

- 1: 代表Identity。用于Request消息中。其Type Data字段一般将携带申请者的一些信息。一般简写为EAP-Request/Identity或者Request/Identity。
- 2: 代表Notification。Authenticator用它传递一些消息（例如密码已过期、账号被锁等）给Supplicant。一般简写为Request/Notification。
- 3: 代表Nak，仅用于Response帧，表示否定确认。例如Authenticator用了Supplicant不支持的验证类型发起请求，Supplicant可利用Response/Nak消息以告知Authenticator其支持的验证类型。
- 4: 代表身份验证方法中的MD5质询法。Authenticator将发送一段随机的明文给Supplicant。Supplicant收到该明文后，将其和密码一起做MD5计算得到结果A，然后将结果A返回给Authenticator。Authenticator再根据正确密码和MD5质询文做MD5计算得到结果B。A和B一比较就知道Supplicant使用的密码是否正确。
- 5: 代表身份验证方法为OTP（One Time Password，一次性密码）。这是目前最安全的身份验证机制。相信网购过的读者都用过它，例如网银付费时系统会通过短信发送一个密码，这就是OTP。
- 6: 代表身份验证方法为GTC（Generic Token Card，通用令牌卡）。GTC和OTP类似，只不过GTC往往对应一个实际的设备，例如许多国内银行都会给申请网银的用户一个动态口令牌。它就是GTC。
- 254: 代表扩展验证方法（Expanded Types）。
- 255: 代表其他试验性用途（Experimental Use）。

图3-39所示为一个简单的EAP交互。第三和第四步时，Authenticator要求使用MD5质询法进行身份验证，但Supplicant不支持，故其回复NAK消息，并通知Authenticator使用GTC方法进行身份验证。第六步中，如果Supplicant回复了错误的GTC密码时，Authenticator可能会

重新发送Request消息以允许Supplicant重新尝试身份验证。一般认证失败超过3次才会回复Failure消息。

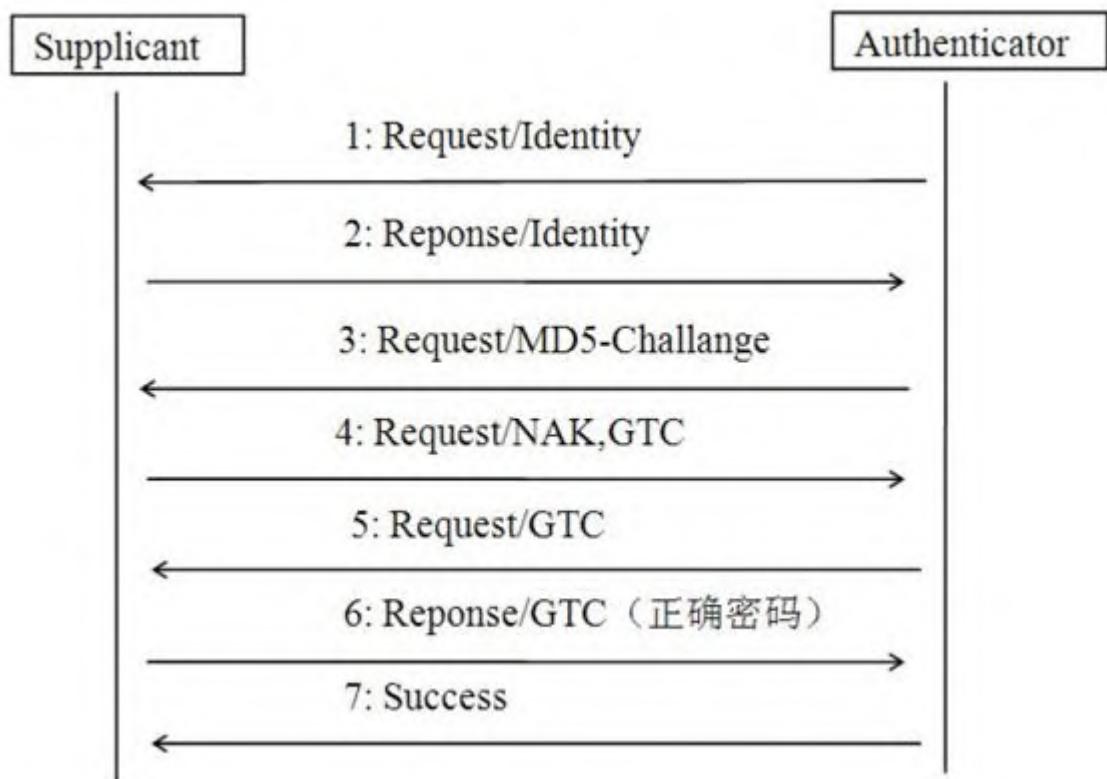


图3-39 EAP交互

EAP的基本情况就介绍到此，读者可通过RFC3748了解它的详细情况。下面介绍802.1X。

提示 可阅读参考资料[32]以了解EAP的其他几种验证方法。

## (2) 802.1X<sup>[34][35]</sup>

802.1X是802.1规范家族中的一员。简单来说，802.1X详细讨论了EAP在Wired LAN上的实现，即EAPOL。先来认识802.1X中的两个重要概念，如图3-40所示。

由图3-40可知，802.1X定义了：Controlled Port和UnControlled Port（受控和非受控端口）。

Port是802.1X的核心概念，其形象和我们常见的以太网中网口很类似。Port的定义和图3-4中所述的定义一样。802.1X定义了两种Port，对于UnControlled Port来说，只允许少量类型的数据流通（例如用于认证的EAPOL帧）。Controlled Port在端口认证通过前（即Port处于Unauthorized状态），不允许传输任何数据。

802.1X身份验证通过后，Control Port转入Port Authorized状态。这时，双方就可通过Controlled Port传递任何数据了。

**提示** 802.1X-2010文档只有200来页，但难度不小，引用的参考资料也非常多。建议读者看完本节后再结合实际需求去阅读它。另外802.1X中还定义了一个很重要对象叫PAE（Port Access Entity）。Entity的概念在3.3.1节曾介绍过，它代表封装了一组功能的模块。在802.1X中，PAE负责和PACP（Port Access Control Protocol）协议相关的算法及处理工作。PAE分为Supplicant PAE和Authenticator PAE两种。

下面来看EAPOL的数据格式，如图3-41所示。

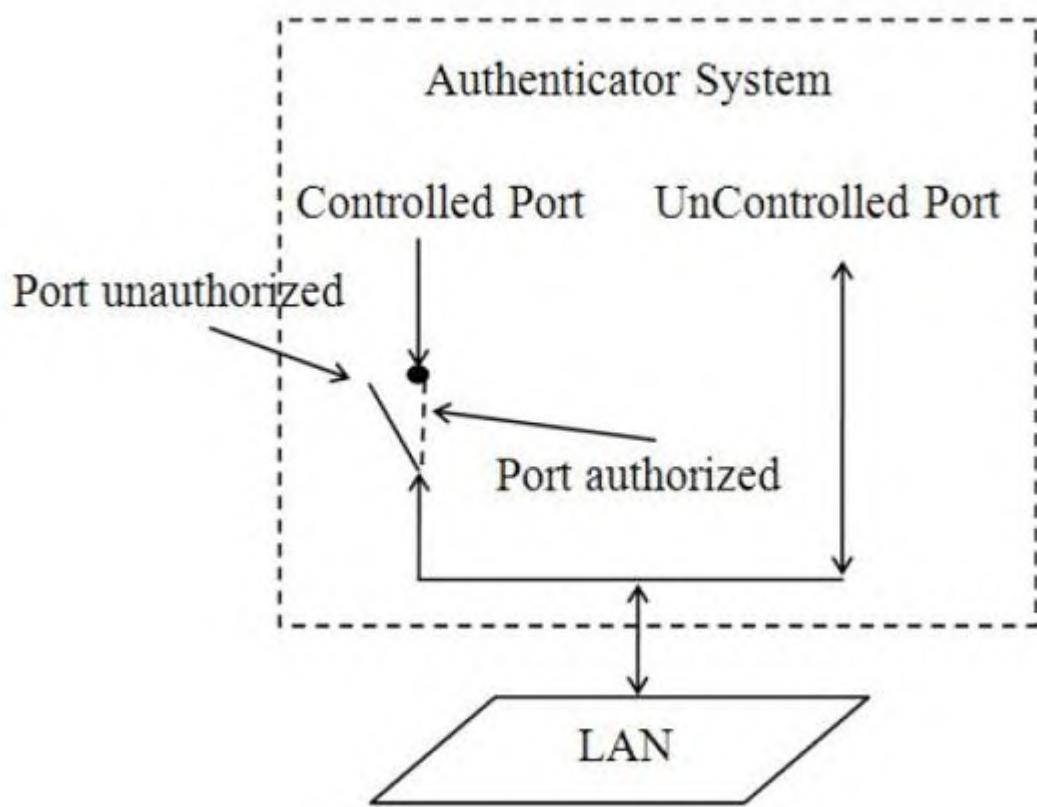


图3-40 802.1X中的Port

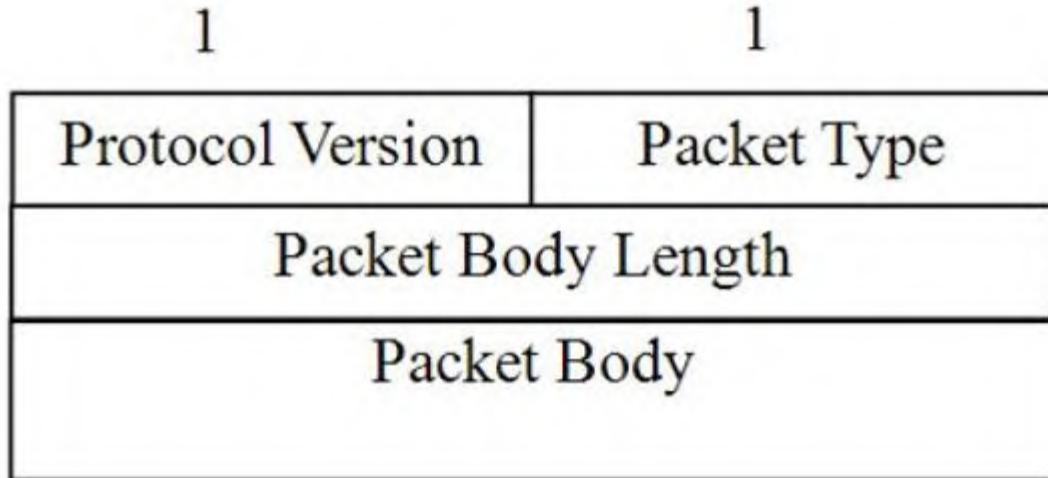


图3-41 EAPOL格式

由图3-41可知EAPOL消息格式如下。

- Protocol Version: 占1字节，代表当前使用的802.1X版本号。值为0x03代表802.1X-2010，值为0x02则表示802.1X-2004，值为0x01表示802.1X-2001。
- Packet Type: EAPOL对EAP进行了扩展，该字段取值详细内容见表3-14。
- Packet Body Length: 指明Packet Body的长度。

EAPOL消息中最重要的是Packet Type，目前规范定义的几种常见取值如表3-14所示。

表 3-14 EAPOL Packet Type 常见取值

Packet Type 取值 (二进制)	名 称	说 明
0000-0000	EAPOL-Packet	以前叫 EAP-Packet，用于携带 EAP 消息帧
0000-0001	EAPOL-Start	Supplicant 发起认证请求
0000-0010	EAPOL-Logoff	Supplicant 退出身份验证以停止使用网络
0000-0011	EAPOL-Key	用于交换加密密钥信息，其对应的数据格式见下文

EAPOL-Key用于交换身份验证过程中使用的密钥信息，其对应的Packet Body格式如图3-42所示。Descriptor Type表示后面Descriptor Body的类型。802.1X中，Descriptor Type值为2时，Descriptor Body的内容由802.11定义。此处给读者展示一个实际的例子，如图3-43所示。

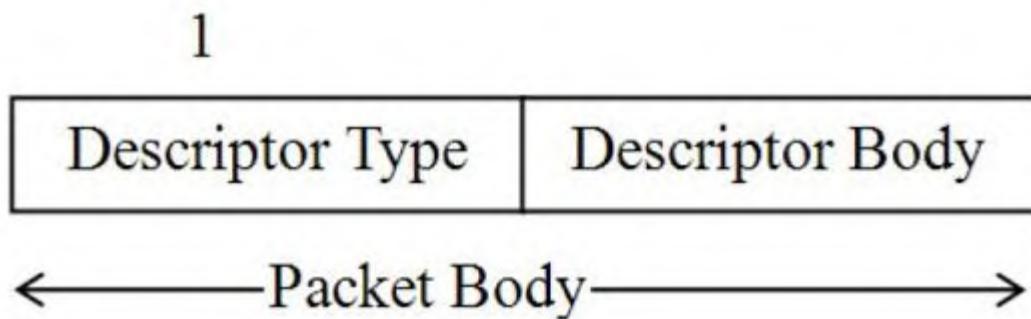


图3-42 EAPOL-Key帧对应的Body格式

```

Frame 22911: 157 bytes on wire (1256 bits), 157 bytes captured (1256 bits) on interface
Radiotap Header v0, Length 20
IEEE 802.11 QoS Data, Flags: .....F.C
Logical-Link Control
  DSAP: SNAP (0xaa)
  IG Bit: Individual
  SSAP: SNAP (0xaa)
  CR Bit: Command
  Control field: U, func=UI (0x03)
    Organization Code: Encapsulated Ethernet (0x0000000)
    Type: 802.1X Authentication (0x888e)
  802.1X Authentication
    Version: 802.1X-2004 (2)
    Type: Key (3)
    Length: 95
      Key Descriptor Type: EAPOL RSN Key (2)
      Key Information: 0x008a
        .... .... .... .010 = Key Descriptor version: AES Cipher, HMAC-SHA1 MIC (2)
        .... .... .... 1... = Key Type: Pairwise Key
        .... .... ..00 .... = Key Index: 0
        .... .... .0.. .... = Install: Not set
        .... .... 1.... .... = Key ACK: Set
        .... .... 0.... .... = Key MIC: Not set
        .... .... 0.... .... = Secure: Not set
        .... .... 0.... .... = Error: Not set
        .... 0.... .... .... = Request: Not set
        ...0 .... .... .... = Encrypted Key Data: Not set
      Key Length: 16
      Replay Counter: 1
      WPA KeyNonce: 40c8b47a82017251b581263ec2a96e079929087f97ec9c6e...
      Key IV: 00000000000000000000000000000000
      WPA Key RSC: 0000000000000000
      WPA Key ID: 0000000000000000
      WPA Key MIC: 00000000000000000000000000000000
      WPA Key Data Length: 0

```

Descriptor Body

图3-43 EAPOL-Key帧实例

图3-43中：

- 0x888e代表EAPOL在802.11帧中的类型。
- Key Descriptor Type值为2，表示EAPOL RSN Key。
- Descriptor Body的内容就是大括号中所包含的信息。其细节请读者阅读802.11第11.6.2节。

最后，看看用MD5质询算法作为身份验证的EAPOL认证流程，如图3-44所示。

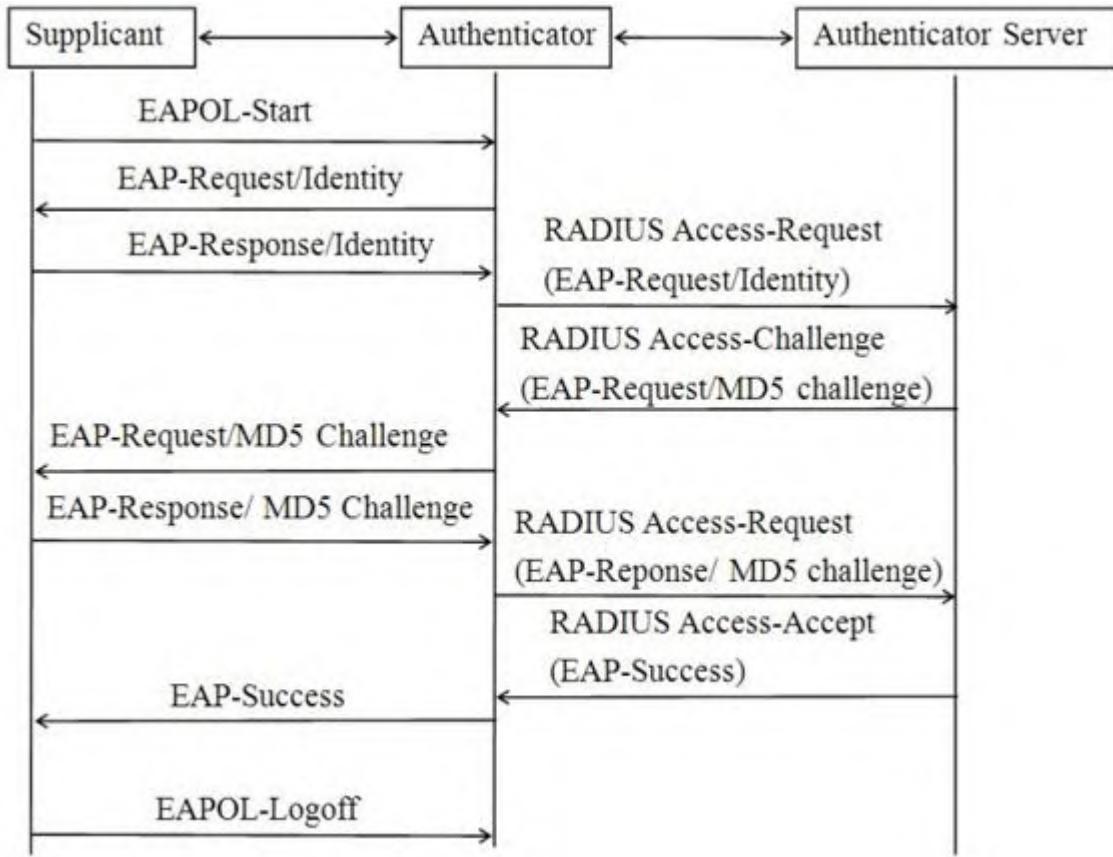


图3-44 EAPOL认证流程

- Supplicant主动向Authenticator发送EAPOL-Start消息来触发认证。注意，EAPOL-Start消息不是必需的。
- Authenticator收到EAPOL-Start消息后，将发出EAP-Request/Identity消息以要求Supplicant输入用户名。由于EAPOL-Start消息不是必需的，所以一个未经验证的Supplicant发送了其他数据包也将触发Authenticator发送EAP-Request/Identity消息。
- Supplicant将用户名信息通过EAP-Response/Identity消息发送给Authenticator。Authenticator再将它经过封装处理后（转换成RADIUS Access-Request消息）送给AS（Authenticator Server）进行处理。
- AS收到Authenticator转发的用户名信息后，将其与数据库中的用户名表对比，找到该用户名对应的密码信息。然后随机生成的一个MD5质

询文并通过RADIUS Access-Challenge消息发送给Authenticator，最后再由Authenticator转发给Supplicant。

- Supplicant收到EAP-Request/MD5 Challenge消息后，将结合自己的密码对MD5质询文进行处理，处理结果最终通过EAP-Response/MD5 Challenge消息经由Authenticator返回给AS。
- AS将收到RADIUS Access-Request消息和本地经过加密运算后的密码信息进行对比，如果相同，则认为该Supplicant为合法用户，然后反馈认证成功消息（RADIUS Access-Accept和EAP-Success）。
- Authenticator收到认证通过消息后将端口改为授权状态，允许Supplicant访问网络。注意，有些Supplicant还会定期向Supplicant发送握手消息以检测Supplicant的在线情况。如果Supplicant状态异常，Authenticator将禁止其上线。
- Supplicant也可以发送EAPOL-Logoff消息给Authenticator以主动要求下线。

至此，我们对EAP和802.1X进行了简单介绍，EAP和802.1X的目前是为了进行身份验证，二者有自己的数据格式。如果没有特殊需要，掌握上述知识就可以了。

**提示** 后续分析wpa\_supplicant时还将对802.1X进行更为详尽的介绍。

下面介绍安全部分最后一个内容，RSNA。

#### 4. RSNA介绍<sup>[36]</sup><sup>[37]</sup>

RSNA（Robust Secure Network Association，强健安全网络联合）是802.11定义的一组保护无线网络安全的过程，是一套安全组合拳。这套组合拳包含的过程如图3-45所示。RSNA包括如下过程。

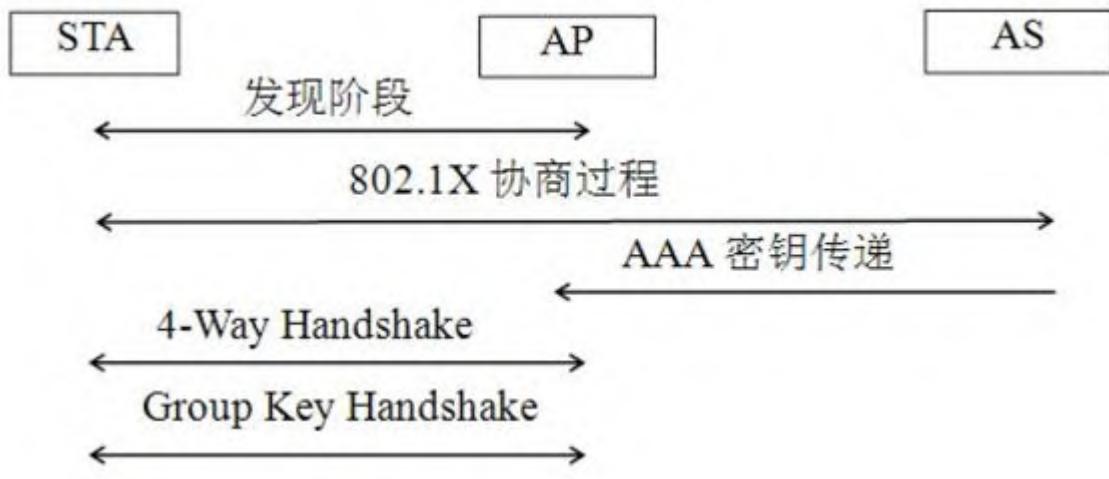


图3-45 RSNA过程

- 1) RSNA网络发现阶段。当STA扫描无线网络时，需检查Beacon帧或Probe Response帧中是否有RSNE信息元素。如果有，STA根据RSNE处理原则选择合适的AP并完成802.11 Authentication（设置认证类型为Open System Authentication）和Association。
- 2) 上一步建立的安全保护机制非常弱，所以RSNA的第二步工作是开展802.1X认证。目的是利用802.1X认证机制实现有效安全的认证用户身份，同时还需要分配该次会话所需要的一系列密钥用于后续通信的保护。
- 3) RSNA通过4-Way Handshake和Group Key Handshake协议完成RSNA中的密钥管理工作。密钥管理工作主要任务是确认上一阶段分配的密钥是否存在，以及确认所选用的加密套件，并生成单播数据密钥和组播密钥用于保护数据传输。

当上面三个步骤都完成后，RSNA网络就建立成功。由上所述，我们发现RSNA包括两个主要部分。

- 在数据加密和完整性校验方面，RSNA使用了前面章节提到的TKIP和CCMP。TKIP和CCMP中使用的TK（Temporary Key）则来自于RSNA定义的密钥派生方法。
- 密钥派生和缓存。RSNA基于802.1X提出了4-Way Handshake（四次握手协议，用于派生对单播数据加密的密钥）和Group Key

Handshake（组密钥握手协议，用于派生对组播数据加密的密钥）两个新协议用于密钥派生。另外，为了加快认证的速度，RSNA还支持密钥缓存。密钥派生和缓存的详情见下文。

RSNA中，我们仅介绍涉及的密钥派生和缓存。

为什么要进行密钥派生呢？它涉及密码安全方面的问题，不过其目的并不难理解。在WEP中，所有STA都使用同一个WEP Key进行数据加密，其安全性较差。而RSNA中要求不同的STA和AP关联后使用不同的Key进行数据加密，这也就是RSNA中Pairwise（成对）概念的来历。

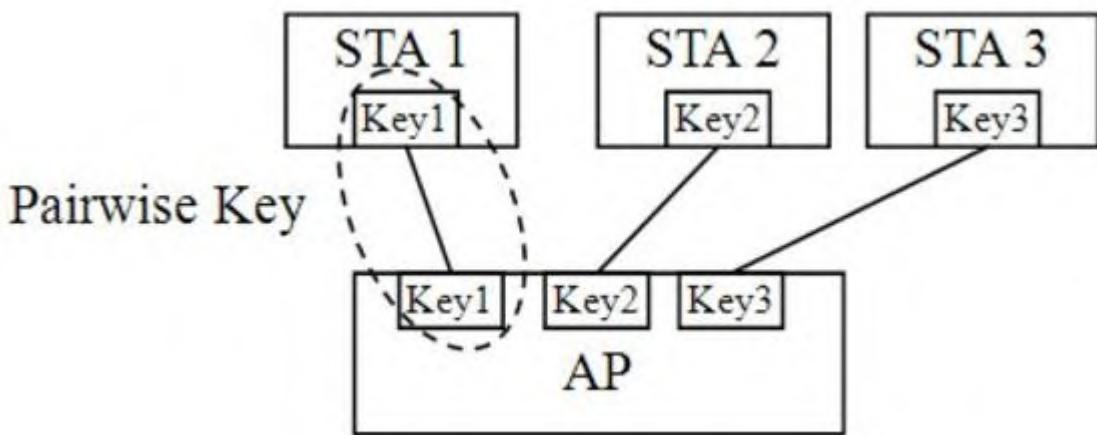


图3-46 Pairwise的概念

图3-46展示了Pairwise Key的含义，不过该图是否表明AP需要为不同的STA设置不同的密码呢？这显然和现实情况是违背的。因为现实生活中，我们关联多个STA到同一个AP时使用的都是相同的密码。

如何解决不同STA使用不同密码的问题呢？原来，我们在STA中设置的密码叫PMK（Pairwise Master Key，成对主密钥），其来源有两种。

- 在SOHO网络环境中，PMK来源于预共享密钥（Pre-Shared Key，PSK）。Shared Key概念在3.3.7节介绍WEP身份验证时提到过，即在家用无线路由器里边设置的密码，无须专门的验证服务器，对应的设置项为“WPA/WPA2-PSK”。由于适用家庭环境，所以有些无线路由器设置界面中也叫“WPA/WPA2-个人”（WFA认证名为“WPA/WPA2-Personal”）。

- 在企业级环境中，PMK和Authenticator Server（如RADIUS服务器）有关，需要通过EAPOL消息和后台AS经过多次交互来获取。这种情况多见于企业级应用。读者可在无线路由器设置界面中见到“WPA/WPA2-企业”（有时叫“WPA或WPA2”，WFA认证名为“WPA/WPA2-Enterprise”）的安全类型选项，并且会要求设置RADIUS服务器地址。

Personal模式和Enterprise模式中PMK的来源如图3-47所示。

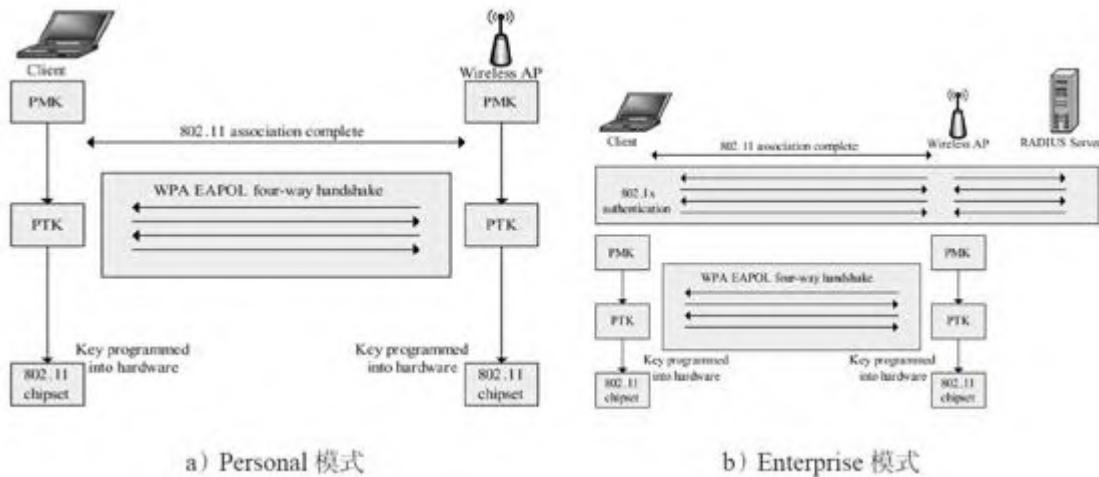


图3-47 PMK来源

图3-47中：

- Personal模式不需要RADIUS服务器参与。AP和STA双方的Key属于PSK，即事先就配置好了。
- Enterprise模式下，STA、AP和RAIDUS的PMK通过多次EAPOL消息来获取。获取的方法和具体的EAP Method有关。显然，相比Personal模式，这种方式更加安全，但其耗费的时间也较长。
- STA和AP得到PMK后，将进行密匙派生以得到PTK。最后，PTK被设置到硬件中，用于数据的加解密。
- 由于AP和STA都需要使用PTK，所以二者需要利用EAPOL Key帧进行信息交换。这就是4-Way Handshake的作用。其详情见下文。

有了PMK后，AP和STA将进行密钥派生，正是在派生的过程中，AP和不同STA将生成独特的密钥。这些密钥用于实际的数据加解密。由于STA每次关联都需要重新派生这些Key，所以它们称为PTK（Pairwise Transient Key，成对临时Key），PTK如图3-47所示。

提示 WFA要求目前生产的无线设备必须通过WPA2认证。

RSNA中，针对组播数据和单播数据有不同的派生方法，即单播数据和组播数据使用不同的Key进行加密。本书仅介绍单播数据密钥的派生方法，如图3-48所示。输入PMK长256位（如果用户设置的密钥过短，规范要求STA或AP将其扩展到256位），通过PRF（Pseudo Random Function，伪随机数函数）并结合S/A（Supplicant/Authenticator）生成的Nonce（这两个Nonce将通过4-Way Handshake协议进行交换）以及S/A MAC地址进行扩展，从而派生出TKIP和CCMP用的PTK。

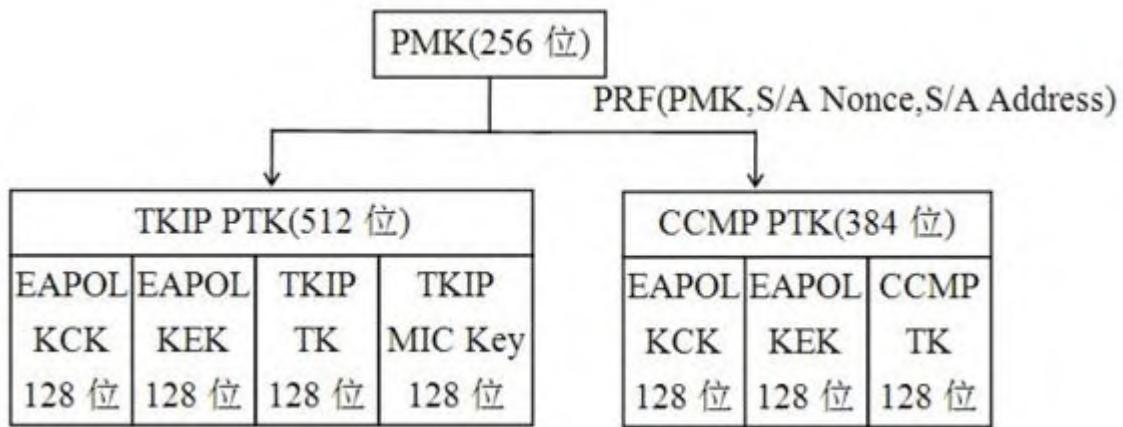


图3-48 单播数据密钥派生方法

图3-48中分别描述了TKIP PTK和CCMP PTK的组成。

- EAPOL KCK (Key Confirmation Key) 用于计算密钥生成消息的完整性校验值。
- EAPOL KEK (Key Encryption Key) 用来加密密钥生成消息。

这两个EAPOL Key将用于后续EAPOL Key帧中某些信息的加密。TKIP TK 和CCMP TK就是TKIP和CCMP算法中用到的密钥。

明白密钥派生的作用了吗？简单来说，WEP中，网络中传输的是用同一个Key加密后的数据，其破解的可能性较大。而TKIP和CCMP则把PMK保存在AP和STA中，实际数据加密解密时只是使用PMK派生出来的Key，即PTK。

提示 关于TKIP PTK和CCMP PTK等其他信息，请读者阅读参考资料[30]。

由于密钥派生时需要STA和AP双方的Nonce等其他一些信息，所以二者还需通过4-Way Handshake以交换双方的信息，其过程如图3-49所示[24]。

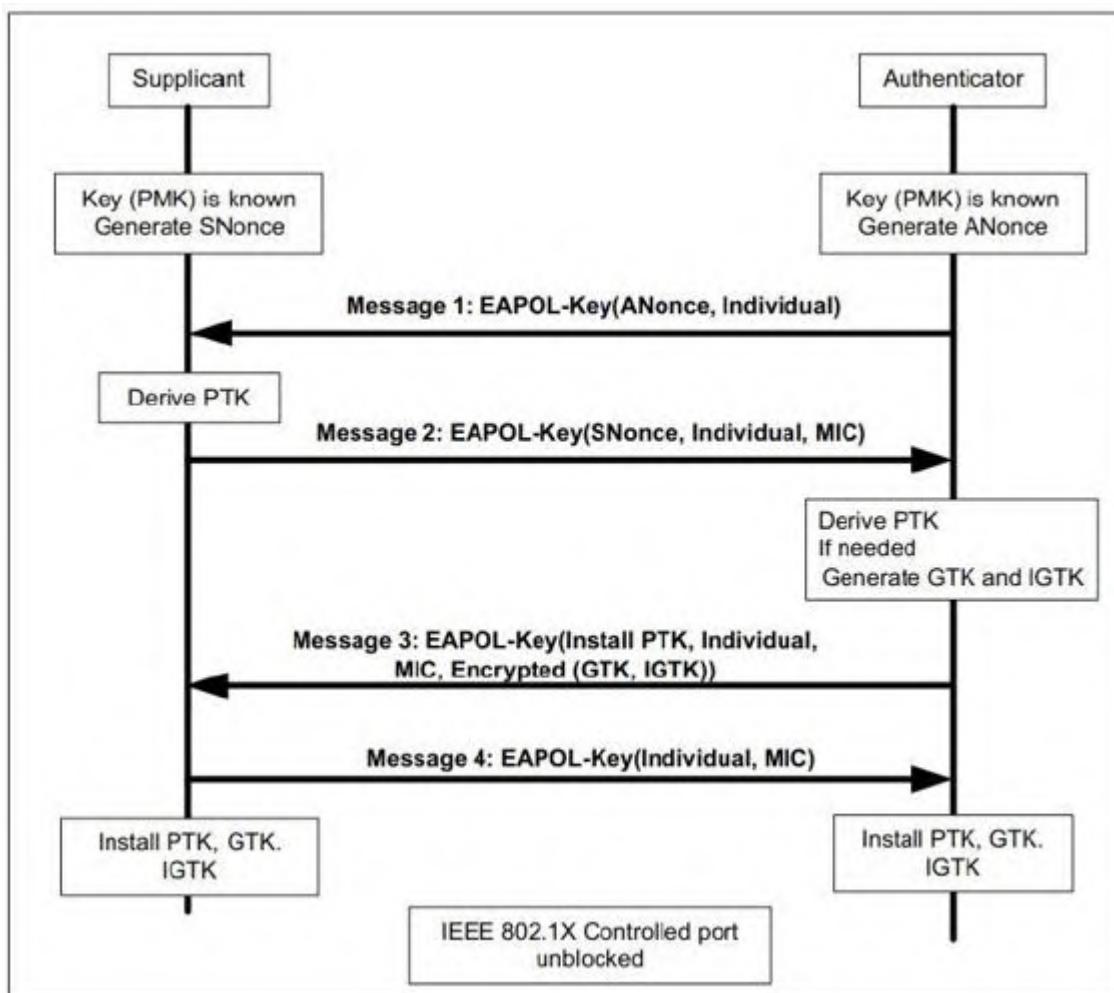


图3-49 4-Way Handshake流程

注意 4-Way Handshake除了用于交换STA和AP信息外，还可用于AP和STA之间相互判断对方是否有正确的PMK。

图3-49所示的4-Way Handshake工作过程如下。

1) Authenticator生成一个Nonce (ANonce)，然后利用EAPOL-Key消息将其发给Supplicant。

2) Supplicant根据ANonce、自己生成的一个Nonce (SNonce)、自己所设置的PMK和Authenticator的MAC地址等信息进行密钥派生。

Supplicant随后将SNonce以及一些信息通过第二个EAPOL-Key发送给Authenticator。Message 2还包含一个MIC值，该值会被图3-48中的KCK加密。接收端Authenticator取出Message 2中的SNonce后，也将进行和Supplicant中类似的计算来验证Supplicant返回的消息是否正确。如果不正确，这将表明Supplicant的PMK错误，于是整个握手工作就此停止。

3) 如果Supplicant密钥正确，则Authenticator也进行密钥派生。此后，Authenticator将发送第三个EAPOL-Key给Supplicant，该消息携带组临时密码 (Group Transient Key, GTK，用于后续更新组密钥，该密钥用图3-48中的KEK加密)、MIC (用KCK加密)。Supplicant收到Message 3后也将做一些计算，以判断AP的PMK是否正确。注意，图中的IGTK (Integrity GTK) 用于加解密组播地址收发的管理帧，本书不讨论。

4) Supplicant最后发送一次EAPOL-Key给Authenticator用于确认。

5) 此后，双方将安装 (Install) Key。Install的意思是指使用它们来对数据进行加密。

Supplicant和Authenticator就此完成密钥派生和组对，双方可以正常进行通信了。

提示 由于篇幅原因，请读者阅读参考资料[38]以了解GTK的产生和作用。另外，4.5.5节“EAPOL-Key交换流程分析”将结合wpa\_supplicant的代码详细介绍4-Way Handshake相关的内容。

另外，802.11还支持“混合加密”，即单播和组播使用不同的加密方法，例如单播数据使用CCMP而组播数据使用TKIP进行加密。注意，有些加密方法不能混合使用，例如组播数据使用CCMP加密，单播数据就不能使用TKIP。详情见参考资料[39]。

**提示** 混合加密的主要目的是为了解决新旧设备的兼容性问题。它对应的应用场景如下。

- 1) AP支持TKIP和CCMP，属于新一代的设备。
- 2) 一些老的STA只支持TKIP，而一些新的STA支持TKIP和CCMP。

对于单播数据来说，AP和STA的密钥是成对的。所以对于老STA，AP和它们协商使用TKIP。对于新STA，AP和它们协商使用CCMP。而对于组播数据来说，所有STA、AP都使用同一个密钥，加解密方法也只能是一样的。在这种情况下，大家只能使用TKIP而不能使用CCMP对组播数据进行加解密了。

下面我们来介绍密钥缓存。

根据前述内容可知，PMK是密钥派生的源，如果认证前，STA没有PMK，它将首先利用图3-47右图所示的802.1X协商步骤以获取PMK（当然，对于SOHO环境中所使用的PSK而言，则不存在这种情况）。由于802.1X协商步骤涉及多次帧交换，故其所花费时间往往较长。在这种情况下，STA缓存这个得来不易的PMK信息就可消除以后再次进行802.1X协商步骤的必要，从而大大提升整个认证的速度。

根据802.11规范，PMK缓存信息的名称叫PMKSA（PMK Security Association），它包括AP的MAC地址、PMK的生命周期（lifetime），以及PMKID（PMK Identifier，用于标示这个PMKSA，其值由PMK、AP的MAC地址、STA的MAC地址等信息用Hash计算得来）。

当STA和AP进行关联（或重关联）时：

- STA首先根据AP的MAC地址判断自己是否有缓存了的PMKSA，如果有则把PMKID放在RSNE中然后通过Association/Reassociation Request发送给AP。

- AP根据这个PMKID再判断自己是否也保持了对应的PMKSA。如果是，双方立即进入4-Way Handshake过程，从而避免802.1X协商步骤。

## 5. 无线网络安全相关知识总结

本节对无线网络安全技术进行一个总体介绍，其中所涉及的概念、缩略词较多，这里给读者简单总结一下其中的逻辑关系。

无线网络安全要解决的是数据的Confidentiality和Integrity以及使用者的Authentication这三个问题。

- 规范最早定义的WEP本来目的是解决数据完整和机密性的问题，后来WEP中附带完成了Authentication检查。不过整体保护机制非常弱。
- 在802.11i正式出台前，WFA提供了安全机制比WEP强的TKIP。注意，TKIP本身只能解决数据完整和机密性问题。而802.11i出台后，又增加了一个更强健的加密算法CCMP（有时候也叫AES）。AES和TKIP都用于解决数据完整和机密性问题。
- 为了解决Authentication问题，规范借鉴802.1X，从而引出RSNA，它包括密钥派生、缓存等内容。

提示 无线网络安全的内容非常多，读者不妨阅读参考资料[37][38]以加深理解。区分WPA/WPA2企业和个人用法的简单公式如下。

WPA-企业=802.1X+EAP+TKIP

WPA2-企业=802.1X+EAP+CCMP

WPA-个人=PSK+TKIP

WPA2-个人=PSK+CCMP

① RFC3748中也叫Peer，不过本文统一用Supplicant表示。

读累了记得休息一会儿哦~

公众号：古德猫宁李

- 电子书搜索下载
- 书单分享
- 书友学习交流

网站：[沉金书屋 https://www.chenjin5.com](https://www.chenjin5.com)

- 电子书搜索下载
- 电子书打包资源分享
- 学习资源分享

## 3.4 Linux Wi-Fi编程API介绍

前面一直在介绍Wi-Fi规范方面的内容。从本节开始，将向读者介绍Linux平台中Wi-Fi编程方面的知识。

**提示** 相比前文而言，本节内容较为轻松。但编程只不过是规范的某种实现，掌握规范才是理解无线网络技术的核心。这也是本书内容组织和编排的指导原则，希望读者能认真体会。

Linux平台上目前常用的专门针对无线网络设备编程的API有两套<sup>[40]</sup>。

- 最早的一套API由HP公司员工Jean Tourrilhes于1997年开发，全称为Linux Wireless Extensions。一般缩写为wex或wext。这套API使得用户空间的程序能通过ioctl函数来控制无线网卡驱动。
- 由于利用ioctl开展编程的方式不太符合Linux驱动开发的要求，后来Linux又提供了cfg80211和n180211两套编程接口用于替代wext。其中，cfg80211用于驱动开发，而n180211 API供用户空间进程使用以操作那些利用cfg80211 API开发的无线网卡驱动。

**注意** ioctl不符合Linux驱动开发要求主要体现在以下两方面。

1) ioctl的原型为int ioctl (int fd, unsigned long cmd, ...)，最后省略号代表它支持可变个数的参数。但对于一个经过严格定义的系统调用来说，支持可变个数参数的做法似乎显得有些随性。

2) ioctl的参数不仅个数不固定，其参数类型也无法通过函数原型来加以说明。这同样对于一个严谨的系统调用来说，也是不可接受的。

本节将重点介绍用户空间中的Wi-Fi API，wext和n180211。不过在介绍之前，请读者思考一个问题：为什么Wi-Fi需要在用户空间进行编程呢？

**答案** 目前的无线网卡分为两种，一种为SoftMAC。这类网卡中，MLME的处理基本上在软件层（即驱动或用户空间），这样可带来较大

的灵活性。另外，一些认证相关的操作，也可由软件来控制。另一种网卡称为FullMAC。这类网卡的MLME全在硬件处理。相比SoftMAC而言，其灵活性很小。所以目前市面上SoftMAC网卡占绝大多数，而cfg80211就仅支持SoftMAC类型的网卡。

### 3.4.1 Linux Wireless Extensions介绍

从开发者角度来说，wext的用法相当简单。Linux平台中，wext API定义于wireless.h文件。Android平台上，其文件位置在external/kernel-headers/original/linux目录下，主要供驱动开发者使用。

注意 bionic/libc/kernel/common/linux目录中也有一个wireless.h，不过此文件由工具程序根据Kernel中的wireless.h自动生成而来，供用户空间使用。两个文件的区别主要是bionic下的wireless.h包含很少的注释。所以本节将分析Kernel中的wireless.h。Android 4.2中的wext版本为20，由wireless.h中的宏WIRELESS\_EXT定义。

虽然前面提到说ioctl函数的一个缺点是其没有指明参数类型，但wext却比较严谨，提供了自己的数据类型。

#### 1. 常用数据结构

首先，所有用户空间发起的请求都统一包括在struct iwreq中，其原型如下。

```
[-->wireless.h: : struct iwreq]  
  
/*  
 * wext API在设计时参考了系统中现有数据结构及命名方式。作为区分，wext中几乎所有数据结构、类型、宏  
 * 等名字中都带一个w以代表wireless。如下面的iwreq结构体，其对应的普通数据结构类型是ifreq。  
 * 该结构体专门用于往socket句柄传递ioctrl控制参数。  
 */  
struct iwreq  
{  
    union  
    {  
        char ifrn_name[IFNAMSIZ];      // 用于指定要操作的网卡设备  
        name, 如wlan0  
    } ifr_ifrn;
```

```
    union      iwreq_data      u;          // 用于存储具体的参数信息
};
```

如iwreq结构所示，具体的参数信息存储在另外一个联合体iwreq\_data中，其原型如下。

[-->wireless.h: : union: iwreq\_data]

```
/*
iwreq_data是一个联合体，其最大size为16字节。
wext还自定义了一些小的数据结构，如iw_point、iw_param、iw_freq等。它们
的作用如下。
iw_point: 当参数信息的长度超过16字节时，就只能通过iw_point指向另外一块内
存区域，而参数就存储
在那个区域中。这个就是我们常用的指针方式。
iw_param: 当参数信息不超过16字节时，可以把信息存储在iw_param中。
iw_freq: 用于存储频率或信道值。其原型的介绍见本小节最后。
*/
union      iwreq_data
{
    char          name[IFNAMSIZ];
    struct iw_point      essid;           // 存储essid，也就是ssid
    struct iw_param      nwid;            // network id
    // 频率或信道。取值为0-1000时代表channel，大于1000则代表频率，单位
    // 为Hz
    struct iw_freq      freq;
    struct iw_param      sens;             // 信号强度阈值
    struct iw_param      bitrate;          // 码率
    struct iw_param      txpower;
    struct iw_param      rts;              // RTS阈值时间
    struct iw_param      frag;
    __u32                mode;              // 操作模式
    struct iw_param      retry;
    struct iw_point      encoding;
    struct iw_param      power;
    struct iw_quality    qual;

    struct sockaddr      ap_addr;          // AP地址
    struct sockaddr      addr;              // 目标地址

    struct iw_param      param;             // 其他参数
    struct iw_point      data;              // 其他字节数超过16的参数
};
```

当参数字节超过16的时候，wext还定义了和功能相关的参数类型，下面来看专门用于触发无线网卡发起扫描请求的数据结构iw\_scan\_req，其原型如下所示。

```
[-->wireless.h: : struct iw_scan_req]

struct     iw_scan_req
{
    __u8          scan_type;           // 可取值为
IW_SCAN_TYPE_{ACTIVE,PASSIVE}

    // 代表主动或被动扫描
    __u8          essid_len;          // essid字符串长度
    __u8          num_channels;       // 指明信道个数，如果为0，
则表示扫描所有可允许的信道
    __u8          flags;              // 目前仅用于字节对齐
    // bssid用于指明BSS的地址。如果全为FF则为广播BSSID，即wildcard
bssid
    struct sockaddr      bssid;

    __u8          essid[IW_ESSID_MAX_SIZE]; // essid

    /*
        min_channel_time:指示扫描过程中在每个信道等待到第一个回复的时间。
如果在此时间内没有等
        到回复，跳到下一个信道等待。如果等到一个回复，则一共在该信道等待的最大时间为max_channel_time。
        所有时间单位均为TU (Time Units)，即1024ms。
    */
    __u32         min_channel_time;
    __u32         max_channel_time;

    struct iw_freq      channel_list[IW_MAX_FREQUENCIES];//
IW_MAX_FREQUENCIES值为32
};
```

下面来看最后一个常见的数据结构iw\_freq，其原型如下。

```
[-->wireless.h: : struct: iw_freq]:
```

```
// 当频率小于109，m直接等于频率。否则m=f/(10e)
struct     iw_freq
{
    __s32         m;
```

```

    __s16      e;
    __u8       i;          // 该值表示此频率对象在channel_list数组中的
索引
    __u8       flags;     // 固定或自动
};


```

**提示** wext中的数据结构和定义还有许多，建议读者结合实际需要去学习wireless.h。wext API虽然简单，但相信读者已经体会到其背后所依赖的和802.11规范密切相关的理论知识了。

下面通过一个实际的例子来看看用户空间如何通过wext API来触发无线网卡扫描工作的。

## 2. wext API使用实例

本例来源于wpa\_supplicant，它是一个运行于用户空间的专门和无线网卡进行交互的程序。其详情将在下一章节进行介绍。本节仅通过一个函数看看wpa\_supplicant如何利用wext API和无线网卡交互。

```
[-->driver_wext.c: wpa_driver_wext_scan]

int wpa_driver_wext_scan(void *priv, struct
wpa_driver_scan_params *params)
{
    struct wpa_driver_wext_data *drv = priv;

    struct iwreq iwr;           // 定义一个iwreq对象
    int ret = 0, timeout;
    struct iw_scan_req req;     // 定义一个iw_scan_req对象

    // 获取调用者传递的ssid等参数
    const u8 *ssid = params->ssids[0].ssid;
    size_t ssid_len = params->ssids[0].ssid_len;
    .....
    os_memset(&iwr, 0, sizeof(iwr));
    // 为iwr的ifr_name传递需操作的网卡设备名
    os_strlcpy(iwr.ifr_name, drv->ifname, IFNAMSIZ);

    if (ssid && ssid_len) {
        os_memset(&req, 0, sizeof(req));
        // 设置iw_scan_req的信息
        req.essid_len = ssid_len;
        req.bssid.sa_family = ARPHRD_ETHER;
    }
}


```

```
// 设置bssid的MAC地址全为0xFF, 代表这是一个wildcard BSSID搜索
{
    os_memset(&req.bssid.sa_data, 0xff, ETH_ALEN);
    os_memcpy(&req.essid, ssid, ssid_len);
    // 通过data域指向这个iw_sca_req对象
    iwr.u.data.pointer = (caddr_t) &req;
    iwr.u.data.length = sizeof(req);
    // IW_SCAN_THIS_ESSID表示只扫描指定ESSID的无线网络
    iwr.u.data.flags = IW_SCAN_THIS_ESSID;
}
/*
    ioctl_sock指向一个socket句柄, 其创建时候的代码如下:
    ioctl_sock = socket(PF_INET, SOCK_DGRAM, 0)
    SIOCSIWSCAN用于通知驱动进行无线网络扫描。
*/
if (ioctl(drv->ioctl_sock, SIOCSIWSCAN, &iwr) < 0) {
    // 返回错误
}
.....// 其他处理
return ret;
}
```

### 3.4.2 n180211介绍

如上文所述，目前Linux平台中用于替代wext框架的是n180211和cfg80211。其中cfg80211供Kernel网卡驱动开发使用，而n180211 API则供用户空间进程使用。

相比wext，n180211的使用难度较大，因为在n180211框架中，用户进程和Kernel通信的手段没有使用wext中的ioctl，而是采用了netlink机制。所以，虽然n180211.h仅是定义了一些枚举值和有限的数据结构，但其操作却比较复杂。netlink是Linux平台上一种基于socket的IPC通信机制，它支持：

- 用户空间进程和Kernel通信。
- 用户空间中进程间的通信。

不过，相比其他IPC机制，netlink最常用之处还是用户空间进程和kernel模块间的通信。鉴于netlink的复杂性，开源世界提供了几个较为完备的基于netlink编程的框架，其中最著名的就是libnl。而Android也充分发扬拿来主义，在其system/core/libnl\_2目录中移植并精简了libnl项目的代码，得到一个小巧的libnl\_2工程。

本节首先介绍netlink基本知识，然后介绍libnl开源库的内容，最后结合实例介绍n180211的用法。

#### 1. netlink编程<sup>[41]</sup>

netlink是一种基于socket的IPC通信机制，所以它需要解决如下两个问题。

- 寻址：即如何定位通信对象。
- 双方通信的数据格式：socket编程中，通信双方传递的数据格式是由应用程序自己决定的。那么netlink是否有其自定义的数据格式呢？
  - (1) netlink socket创建及绑定

netlink使用前，需要通过socket调用创建一个socket句柄，而socket函数的原型如下。

```
int socket (int domain, int type, int protocol);
```

对于netlink编程来说，注意以下事项。

- **domain:** 必须设置为AF\_NETLINK，表示此socket句柄将用于netlink。
- **type:** netlink是基于消息的IPC通信，所以该值为SOCK\_DGRAM。注意，对netlink编程来说，内核并不区分type的值是SOCK\_DGRAM还是SOCK\_RAW。
- **protocol:** 该值可根据需要设为NETLINK\_ROUTE、NETLINK\_NFTABLES。不同的协议代表Kernel中不同的子系统。例如NETLINK\_ROUTE代表通信对象将是Kernel中负责route的子系统，而NETLINK\_NFTABLES代表通信对象将是Kernel中负责Netfilter的子系统，而第2章碰到的NETLINK\_KOBJECT\_UEVENT则代表应用程序希望接收来自Kernel中的uevent消息。

使用netlink通信时，如何保证确保通信双方能正确定位对方呢？原来，在netlink中，通信的另一方地址由一个数据类型为sockaddr\_nl的结构体来唯一标示。该结构体的原型如下。

[-->netlink.h]

```
struct sockaddr_nl {  
    // nl_family取值必须为AF_NETLINK或PF_NETLINK  
    sa_family_t nl_family;  
    unsigned short nl_pad;           // 该值暂时无用，必须设为0  
/*
```

nl\_pid看起来是用于存储进程pid的，但实际上它只是用于标示一个netlink socket。所以，用户空间

只要保证进程内该值的唯一性即可。另外，如果该值为0，表示通信的目标是Kernel。

```
*/  
    u32 nl_pid;  
/*
```

每一个netlink协议都支持最多32个多播组，加入多播组的成员都能接收到对应的多播消息。

例如NETLINK\_ROUTE协议就有RTMGRP\_LINK和RTMGRP\_IPV4\_IFADDR等多个多播组。每个多播组对应

不同的消息。之所以采用多播的方式，是因为它能减少消息发送/接收的次数。

nl\_groups为0，表示只处理单播消息。

```
*/  
__u32 nl_groups;  
};
```

设置好地址后，通过bind函数将该地址和socket句柄绑定，这样通信的另一方就正式确定了。

关于socket创建以及bind的使用，读者可参考下面的例子。

```
struct sockaddr_nl sa;  
  
memset(&sa, 0, sizeof(sa));  
sa.nl_family = AF_NETLINK;  
sa.nl_groups = RTMGRP_LINK | RTMGRP_IPV4_IFADDR;  
// sa.nl_pid已经通过memset函数置为0了，代表此次netlink通信的目标是  
Kernel  
fd = socket(AF_NETLINK, SOCK_RAW, NETLINK_ROUTE);  
bind(fd, (struct sockaddr *) &sa, sizeof(sa));
```

netlink中关于protocol和多播group的取值比较分散，而且Linux的man手册更新赶不上代码的更新速度。建议读者通过netlink.h寻找protocol的取值。通过protocol对应的头文件去寻找多播group的定义。例如rtnetlink.h就定义了RTMGRP\_LINK等多播组。

## (2) netlink消息类型和标志

netlink的消息及处理是netlink编程中较为复杂的一部分。netlink中，所有消息都有一个消息头，该头由结构体nlmsghdr表达，原型如下。

```
struct nlmsghdr {  
    __u32      nlmsg_len;      // 整个消息的长度，包括消息头  
    __u16      nlmsg_type;     // 消息类型，详情见后文  
    __u16      nlmsg_flags;    // 附加标志  
    __u32      nlmsg_seq;      // 消息序列号  
    __u32      nlmsg_pid;      // 发送该消息的nl_pid。该值为0，表示  
数据来自内核  
};
```

netlink定义了三种消息类型（type），分别如下。

- NLMSG\_NOOP：无操作，程序可直接丢弃这类消息。
- NLMSG\_ERROR：代表错误消息。这类消息携带的信息由结构体 struct nlmsgerr表达。
- NLMSG\_DONE：如果某个信息的数据量较大，需要分成多个netlink 消息发送的话，NLMSG\_DONE表示这是此信息最后一个消息分片包。

说实话，笔者觉得单从type值来区分netlink的消息有些不太好理解。如果从C/S角度来看，netlink消息可分为另外三种类型。

- Request消息：代表客户端向服务端发起的请求。这类消息必须为 nlmsg\_flags设置NLM\_F\_REQUEST标志位。同时，客户端最好为 nlmsg\_seq设置一个独一无二的值，以区分自己发送的不同请求。
- Response消息：该类消息作为服务端对客户端请求的回应。对应的 消息类型是NLMSG\_ERROR。注意，如果服务器处理请求成功，也返回该 值。只不过nlmsgerr的error变量值为0。
- Notification消息：用于服务端向客户端发送通知。由于它不对应 任何请求，故nlmsg\_seq一般取值为0。

基于上述讨论，读者会发现对于Response消息，无论服务端处理请求 是否成功，都会返回NLMSG\_ERROR消息，其对应的数据类型是 nlmsgerr，原型如下。

```
struct nlmsgerr {  
    int error;                                // 值为负代表错误码，值为0表示请求处  
    //理成功  
    // 携带对应请求消息的消息头  
    // 由于只返回消息头，故客户端必须根据消息头中的nlmsg_seq找到具体的消息  
    struct nlmsghdr msg;  
};
```

下面来看参数nlmsg\_flags，它比type复杂，常设的位值有（通 过“|”运算符将不同值“或”在一起）。

- NLM\_F\_REQUEST：代表请求消息。

- **NLM\_F\_MULTI**: 代表消息分片中的一个。理论上说，由于nlmsghdr中的nlmsg\_len为32位，其最大消息长度可达4GB，但内核实现时，最大消息的长度只有一个页面（一般为4KB）。如果有大于4KB的信息要传递，就需将其分成多个消息发送。除最后一个分片消息外，其余消息都需置NLM\_F\_MULTI位。当然，结合上文所述，最后一个分片消息需要设置nlmsg\_type为NLMSG\_DONE。

- **NLM\_F\_ACK**: 该标志将强制服务端接收并处理完请求后回复ACK给客户端。

提示 netlink中nlmsg\_flags的取值还有很多，感兴趣的读者可利用man 7 netlink命令阅读相关知识。

### (3) netlink消息的处理

对于socket编程来说，一般客户端会分配一个buffer用于接收数据，而后续解析该buffer中的netlink消息其实也是一件比较麻烦的事情。另外，由于netlink对消息的大小有字节对齐的要求，所以应用程序在构造自己的netlink消息时，也比较麻烦。不过好在netlink为我们提供了一些帮助宏，利用这些宏，netlink消息就不再麻烦了。以下代码为netlink消息处理时常用的一些宏。

```
NLMSG_ALIGN(len) // 获取len按4字节补齐后的长度。例如，如果len为1，则该宏返回的值应是4
// 整个消息包的长度，包括消息头和数据长度len。该值用于填充消息头中的nlmsg_len参数
NLMSG_LENGTH(len)
// 返回按4字节对齐后整个消息包的长度。它和NLMSG_LENGTH最大的不同是该宏将长度按4字节对齐
NLMSG_SPACE(len)                                // 该宏等于
NLMSG_ALIGN(NLMSG_LENGTH(len))
NLMSG_DATA(nlh)                                 // 获取消息中的数据起始地址
// 用于分片消息的处理。可获取下一条分片消息。其用法见下文例子
NLMSG_NEXT(nlh,len)
NLMSG_OK(nlh,len)                               // 用于判断接收到的数据是否包含一个完整的netlink消息
// 用于返回数据的长度。注意，由于netlink消息在创建时会按4字节补齐
// 所以其数据真正的长度不能通过nlmsg_len来判断
NLMSG_PAYLOAD(nlh,len)
```

下面来看一个netlink消息解析的例子，读者以后有需要，可以把它作为参考。

```
// 本例基于linux netlink手册。可通过man 7 netlink查阅
int len;
char buf[4096];
struct iovec iov = { buf, sizeof(buf) };
struct sockaddr_nl sa;
struct msghdr msg; // 用于recvmsg系统调用，和netlink没关系
struct nlmsghdr *nh;

msg = { (void *)&sa, sizeof(sa), &iov, 1, NULL, 0, 0 };
len = recvmsg(fd, &msg, 0); // recvmsg返回，可能存储了多条netlink消息

// 开始接受接收buffer，注意NLMSG_NEXT宏，其内部会对len长度进行修改，以调整到下一个消息的起始
for (nh = (struct nlmsghdr *) buf; NLMSG_OK(nh, len);
     nh = NLMSG_NEXT (nh, len)) {
    /* The end of multipart message. */
    if (nh->nlmsg_type == NLMSG_DONE) {
        return; // 分片消息处理
    }
    if (nh->nlmsg_type == NLMSG_ERROR) {
        struct nlmsgerr* pError = (struct nlmsgerr*)
NLMSG_DATA(nh); // 获取nlmsgerr的内容
        .... /* 错误或ACK处理*/
    }
    void* data = NLMSG_DATA(nh); // 获取数据的起始地址
    .... // 处理数据
}
```

netlink的消息收发和普通的socket消息收发一样，这里不赘述。

#### (4) netlink编程小结

netlink强制要求每个netlink消息都包含消息头，其实它还定义了一个名为nlattr的结构体，用于规范载荷（Payload）的数据类型。其目的是希望Payload以属性（attribute）的方式来描述自己。struct nlattr结构非常简单，如下所示。

```
struct nlaattr {  
    __u16      nla_len;           // 属性长度  
    __u16      nla_type;         // 属性类型  
};
```

netlink虽然复用了socket编程以方便应用程序和内核通信，但因为其文档很少，而且不同protocol往往还有自己特定的数据结构，所以实际使用过程中难度较大。

本书不讨论nlaattr及netlink其他的内容。建议读者考虑下文介绍的文档更加丰富的libnl开源库。

## 2. libnl开源库

根据前文介绍，netlink API用起来相对麻烦，建议读者考虑采用libnl开源库，其官方网站为<http://www.infradead.org/~tgr/libnl/>。libnl的内容也不少，其架构如图3-50所示。

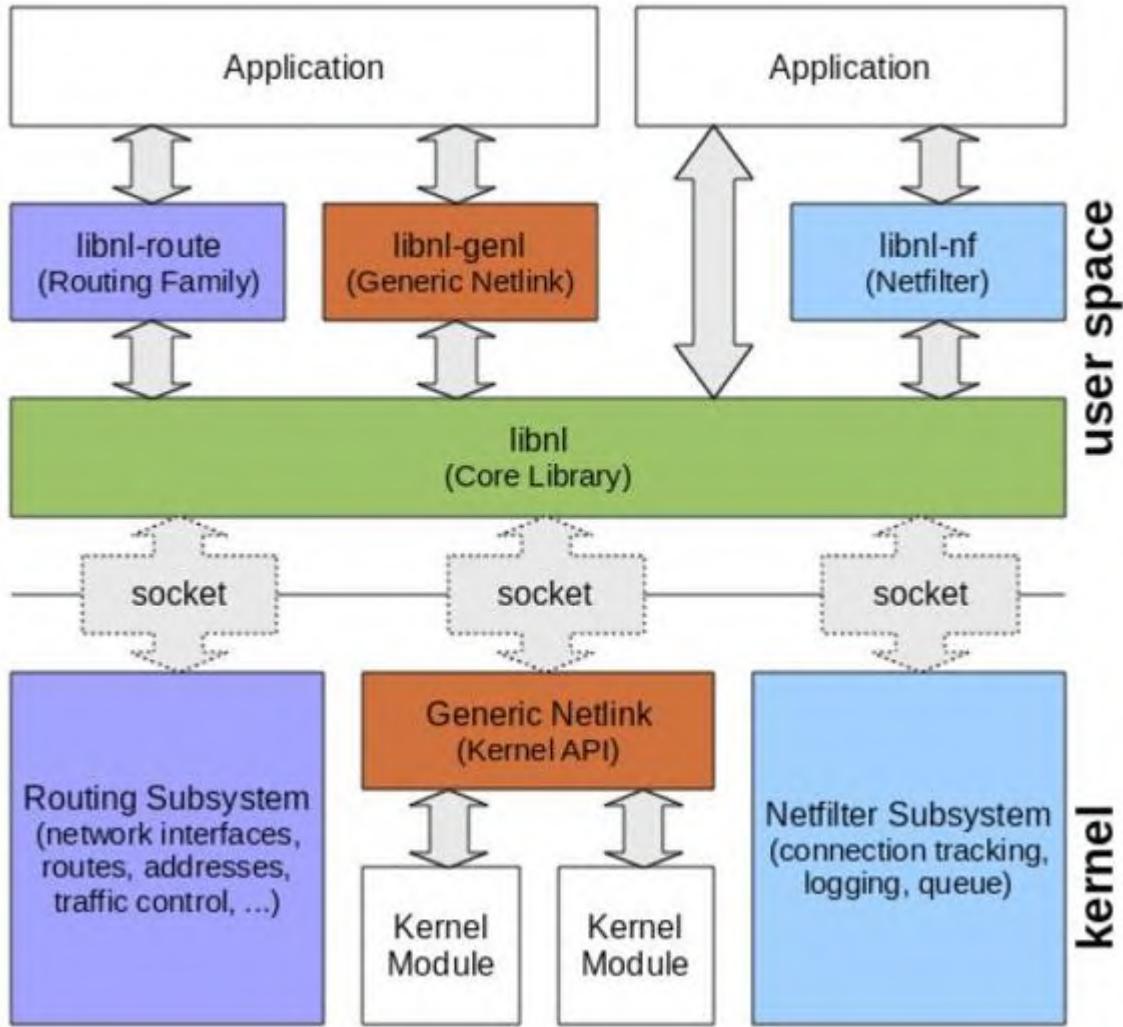


图3-50 libnl架构

由图3-50可知，以下三个库都基于其核心库libnl。

- **libnl-route**：用于和Kernel中的Routing子系统交互。
- **libnl-nf**：用于和Kernel中的Netfilter子系统交互。
- **libnl-genl**：用于和Kernel中的Generic Netlink模块交互。

**提示** 从图也可看出netlink使用的复杂性。

Android平台移植并精简了libnl和libnl-genl中的部分内容，得到了libnl\_2（2表示libnl工程的版本号，最新版为3），目录在

system/core/libnl\_2文件夹下。

本节介绍libnl中的一些常用API。详细内容还请读者参考其官方网站中的文档，地址为http://www.infradead.org/~tgr/libnl/doc/core.html。

### (1) nl\_sock结构体的使用

libnl以面向对象的方式重新封装了netlink原有的API。其使用时必须分配一个nl\_sock结构体。下面展示了和它相关的一些API及使用方法。

```
#include <netlink/socket.h>
// 分配和释放nl_sock结构体
struct nl_sock *nl_socket_alloc(void)
void nl_socket_free(struct nl_sock *sk)
// nl_connect内部将通过bind函数将netlink socket和protocol对应的模块进行绑定
int nl_connect(struct nl_sock *sk, int protocol)
```

linbl还可为每个nl\_sock设置消息处理函数，相关API如下。

```
// 为nl_sock对象设置一个回调函数，当该socket上收到消息后，就会回调此函数
// 进行处理
// 回调函数及参数封装在结构体struct nl_cb中
void nl_socket_set_cb(struct nl_sock *sk, struct nl_cb *cb);
// 获取该nl_sock设置的回调函数信息
struct nl_cb *nl_socket_get_cb(const struct nl_sock *sk);
```

注意，以上两个函数没有文档说明。建议使用另外一个控制力度更为精细的API。

```
/*
此API对消息接收及处理的力度更为精细，其中：
type类型包括NL_CB_ACK、NL_CB_SEQ_CHECK、NL_CB_INVALID等，可用于处理底层不同netlink消息的情况。
```

例如，当收到的netlink消息无效时，将调用NL\_CB\_INVALID设置的回调函数进行处理。

nl\_cb\_kinds指定消息回调函数的类型，可选值有NL\_CB\_CUSTOM，代表用户设置的回调函数，NL\_CB\_DEFAULT

代表默认的处理函数。

回调函数的返回值包括以下。

```

NL_OK: 表示处理正常。
NL_SKIP: 表示停止当前netlink消息分析，转而去分析接收buffer中下一条
netlink消息（消息分
片的情况）。
NL_STOP: 表示停止此次接收buffer中的消息分析。
*/
int nl_socket_modify_cb(struct nl_sock *sk,
                        enum nl_cb_type type, enum nl_cb_kind kind,
                        nl_recvmsg_msg_cb_t func, void *arg);

```

另外，netlink还可设置错误消息（即专门处理nlmsgerr数据）处理回调函数，相关API如下。

```

#include <netlink/handlers.h> // 必须包含此头文件
// 设置错误消息处理
int nl_cb_err(struct nl_cb *cb, enum nl_cb_kind kind,
              nl_recvmsg_err_cb_t func, void *arg);
typedef int(* nl_recvmsg_err_cb_t)(struct sockaddr_nl *nla,
                                    struct nlmsgerr *nlerr, void *arg);

```

## (2) libnl中的消息处理

libnl定义了自己的消息结构体struct nl\_msg。不过它也提供API直接处理netlink的消息。常用的API如下。

```

#include <netlink/msg.h> // 必须包含这个头
文件
// 下面这两个函数计算netlink消息体中对应部分的长度
int nlmsg_size(int payloadlen); // 请参考图来理解这两个函数
返回值的意义
int nlmsg_total_size(int payloadlen);

```

关于netlink消息的长度如图3-51所示。

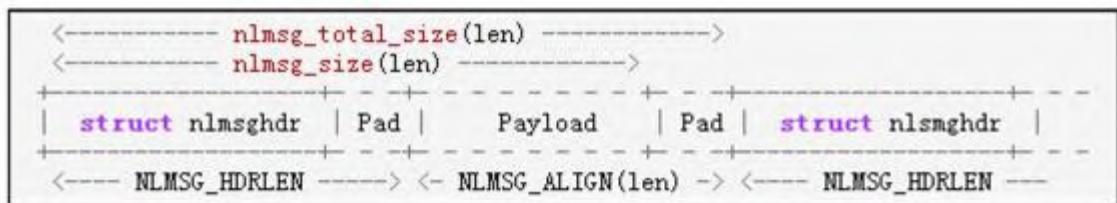


图3-51 nlmsg消息长度的计算

其他可直接处理netlink消息的API如下。

```
struct nlmsghdr *nlmsg_next(struct nlmsghdr *hdr, int
*remaining);
int nlmsg_ok(const struct nlmsghdr *hdr, int remaining);
/*定义一个消息处理的for循环宏，其值等于
for (int rem = len, pos = head; nlmsg_ok(pos, rem); \
     pos = nlmsg_next(pos, &rem))
*/
#define nlmsg_for_each(pos, head, len)
```

开发者也可以通过libnl定义的消息结构体nl\_msg进行相关操作，和nl\_msg有关的API如下。

```
struct nl_msg *nlmsg_alloc(void);
void nlmsg_free(struct nl_msg *msg);
// nl_msg内部肯定会指向一个netlink消息头实例，下面这个函数用于填充
netlink消息头
struct nlmsghdr *nlmsg_put(struct nl_msg *msg,
                           uint32_t port, uint32_t seqnr,
                           int nlmsg_type, int payload, int
nlmsg_flags);
```

### (3) libnl中的消息发送和接收

netlink直接利用系统调用（如send、recv、sendmsg、recvmsg等）进行数据收发，而libnl封装了自己特有的数据收发API。其中和发送有关的几个主要API如下。

```
// 直接发送netlink消息
int nl_sendto (struct nl_sock *sk, void *buf, size_t size)
// 发送nl_msg消息
int nl_send (struct nl_sock *sk, struct nl_msg *msg)
int nl_send_simple(struct nl_sock *sk, int type,
                   int flags,void *buf, size_t size);
```

常用的数据接收API如下。

```
// 核心接收函数。nla参数用于存储发送端的地址信息。creds用于存储权限相关的信息
int nl_recv(struct nl_sock *sk, struct sockaddr_nl *nla,
            unsigned char **buf, struct ucred **creds)
// 内部通过nl_recv接收消息，然后通过cb回调结构体中的回调函数传给接收者
int nl_recvmsgs (struct nl_sock *sk, struct nl_cb *cb)
```

#### (4) libnl-genl API介绍<sup>[41]</sup>

由图3-50可知，libnl-genl封装了对generic netlink模块的处理，它基于libnl。Linux中关于generic netlink的说明几乎没有，建议大家参考libnl中的说明。一条genl消息的结构如图3-52所示。

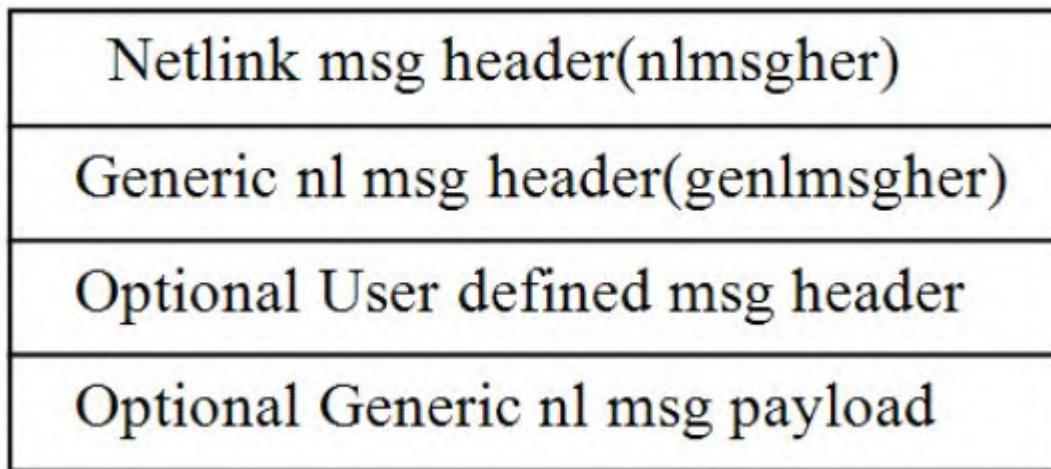


图3-52 genl消息结构

其中，genlmsghdr的原型如下。

```
struct genlmsghdr {  
    __u8 cmd;           // cmd和version都和具体的案例有关  
    __u8 version;  
    __u16 reserved;    // 保留  
};
```

genl常用的API如下。

```
// 和libnl的nl_connect类型，只不过协议类型为GENERIC_NETLINK  
int genl_connect (struct nl_sock *sk)  
// genlmsg_put用于填充图中的nlmsghdr、genlmsghdr和用户自定义的消息  
头。详细内容见下文  
void* genlmsg_put (struct nl_msg *msg, uint32_t port,  
                   uint32_t seq, int family, int hdrlen,  
                   int flags, uint8_t cmd, uint8_t version)  
// 用于获取genl消息中携带的nla_attr内容  
struct nla_attr* genlmsg_attrdata(const struct genlmsghdr  
*gnlh, int hdrlen)
```

另外，genl还有几个比较重要的API，它们和genl机制的内核实现有关，这里仅简单介绍其中几点内容。为实现genl机制，内核创建了一个虚拟的Generic Netlink Bus。所有genl的使用者（包含内核模块或用户空间进程）都会注册到此Bus上。这些使用者注册时，都需要填充一个名为genl\_family的数据结构，该结构是一种身份标示。所以某一方只要设置好genlmsg\_put中的family参数，数据就能传递到对应的模块。

family是一个整型，可读性较差，所以genl使用者往往会指定一个字符串作为family name。而family name和family的对应关系则由genl中另外一个重要模块去处理。这个模块就是genl中的Controller，它也是Generic Bus使用者。其family name为“nlctrl”，只不过它的family是固定的，目前取值为16（一般为它定义一个NETLINK\_GENERIC宏）。Controller的一个重要作用就是为其他注册者建立family name和family之间关系，也就是动态为其他注册者分配family编号。另外，Controller也支持查询，即返回当前Kernel中注册的所有genl模块的family name和family的值。

对用户空间程序来说，只要知道family的值，就可和指定模块进行通信了。libnl-genl封装了上述操作，并提供了几个常用的API。

```
// 根据family name字符串去查询family，该函数内部实现将发送查询消息给
Controller
int genl_ctrl_resolve (struct nl_sock *sk, const char *name)
/*
如果每次都向Controller去查询family编号将严重影响效率，所以libnl-genl
会把查询到的信息
缓存起来。
下面这个函数将分配一个nl_cache列表，其内容存储了当前注册到Generic
Netlink Bus上所有注
册者的信息。
*/
int genl_ctrl_alloc_cache (struct nl_sock *sk, struct nl_cache
**result)
// 根据family name从缓存中获取对应的genl_family信息
struct genl_family * genl_ctrl_search_by_name
                (struct nl_cache *cache, const char *name)
```

提示 相比直接使用netlink API，libnl对开发者更加友好，即使libnl封装得再好，netlink编程依然不是一件轻松的事情。目前为

止，笔者还没有找到一篇文档能全面描述netlink中的protocol及对应的多播组、genl中Controller模块所支持的命令等至关重要的知识点。当年在Windows平台做开发时，微软为开发者提供的编程文档中不仅有原理性说明，还有很多编程技巧。这些内容对开发者而言都是无价之宝。不过，指望Linux重新修订、增补文档无疑是一件异想天开的事情。在此笔者也只能希望读者们在学习过程中注意收集资料并和大家一起分享了。

### 3. n180211实例

了解netlink和libnl之后，现在来看n180211。简单来说，n180211的核心就是通过netlink机制向Kernel中的无线网卡驱动发送特定的消息。只不过这些消息的类型、参数等都由n180211.h定义。此处通过一个案例，学习如何通过n180211触发网卡进行无线网络扫描。

```
[-->driver_n180211.c: : wpa_driver_n180211_scan]

static int wpa_driver_n180211_scan(void *priv,
                                    struct wpa_driver_scan_params *params)
{
    struct i802_bss *bss = priv;
    struct wpa_driver_n180211_data *drv = bss->drv;
    int ret = -1, timeout;
    struct nl_msg *msg, *rates = NULL; // 定义两个nl_msg对象,
rates和P2P有关, 读者可忽略它

    drv->scan_for_auth = 0;
    // 创建n180211消息, 其中NL80211_CMD_TRIGGER_SCAN是N180211定义的
命令, 用于触发网络扫描
    msg = n180211_scan_common(drv, NL80211_CMD_TRIGGER_SCAN,
                               params);
    .....// P2P 处理
    // 发送netlink消息
    ret = send_and_recv_msgs(drv, msg, NULL, NULL);
    msg = NULL;
    if (ret)
        goto nla_put_failure;
    .....// wpa_supplicant其他处理
    .....// 错误处理
    return ret;
}
```

上面代码中构造无线网络扫描nl\_msg的重要函数nl80211\_scan\_common  
代码如下所示。

[-->driver\_nl80211.c: : nl80211\_scan\_common]

```
static struct nl_msg * nl80211_scan_common
    (struct wpa_driver_nl80211_data *drv, u8 cmd,
     struct wpa_driver_scan_params *params)
{
    struct nl_msg *msg;
    int err;
    size_t i;
    // 分配一个nl_msg对象
    msg = nlmsg_alloc();
    /*
        调用nl80211_cmd函数填充nl_msg中的信息，其内部代码如下。
        static void * nl80211_cmd(struct wpa_driver_nl80211_data
*drv,
        struct nl_msg *msg, int flags, uint8_t cmd) {
            return genlmsg_put(msg, 0, 0, drv->global-
>nl80211_id, 0, flags, cmd, 0);
        }
    */
    nl80211_cmd(drv, msg, 0, cmd);
    /*
        nl80211消息的参数通过netlink中的nla来存储。
        NL80211_ATTR_IFINDEX代表
            此次操作所指定的网络设备编号。
        */
    nla_put_u32(msg, NL80211_ATTR_IFINDEX, drv->ifindex);

    if (params->num_ssids) {
        struct nl_msg *ssids = nlmsg_alloc();
        for (i = 0; i < params->num_ssids; i++) {
            nla_put(ssids, i + 1, params-
>ssids[i].ssid_len, params->ssids[i].ssid);
            .....
        }
        // netlink支持消息嵌套，即属性中携带的数据可以是另外一个nl_msg
        // 消息
        err = nla_put_nested(msg, NL80211_ATTR_SCAN_SSIDS,
        ssids);
        nlmsg_free(ssids);
        .....
    }
    .... // 其他处理
```

```
    return msg;
    .....// 错误处理
}
```

由上面的例子可知，nl80211其实就是利用netlink机制将一些802.11相关的命令和参数发送给驱动去执行。这些命令和参数信息可通过nl80211头文件查询。

**提示** 本书后续章节将分析wpa\_supplicant8，读者可参考external/wpa\_supplicant8/wpa\_supplicant/src/drivers/nl80211\_copy.h。

首先，nl80211\_copy.h定义其支持的命令，如下所示。

[-->nl80211\_copy.h]

```
enum nl80211_commands {
    NL80211_CMD_UNSPEC,
    NL80211_CMD_GET_WIPHY,
    NL80211_CMD_SET_WIPHY,
    .....
    NL80211_CMD_GET_INTERFACE,
    NL80211_CMD_SET_INTERFACE,
    .....
    NL80211_CMD_SET_BSS,
    NL80211_CMD_SET_REG,
    .....// 一共定义了94条命令
}
```

然后定义属性的取值，如下所示。

```
enum nl80211_attrs {
    NL80211_ATTR_UNSPEC,
    NL80211_ATTR_WIPHY,
    NL80211_ATTR_WIPHY_NAME,
    NL80211_ATTR_IFINDEX,
    NL80211_ATTR_IFNAME,
    NL80211_ATTR_IFTYPE,
    NL80211_ATTR_MAC,
    .....// 一共定义了155条属性
}
```

头文件中对命令、属性等信息的注释都非常详细。本节不赘述，请读者自行阅读该文件。

**提示** 相比wext而言，n180211的使用难度明显要复杂，其中重要原因是它是基于netlink编程的。而且，如果没有libnl的支持，相信使用难度会更大。但从Wi-Fi角度来看，n180211和wext没有太大区别，二者都是紧紧围绕MAC层service来设计数据结构的。

## 3.5 本章总结和参考资料说明

### 3.5.1 本章总结

本章是全书关于Wi-Fi技术方面的一篇基础文章，涉及面很广，内容也很杂，需要读者耐心阅读并理解。从大体上来说，本章按如下逻辑开展。

- 首先简单介绍无线频谱资源和802.11发展历程。
- 为了帮助读者真正看懂802.11规范，本章介绍了OSI/RM模型以及其中关于Entity、SAP、MIB等基本概念。
- 接着正式介绍802.11规范中的一些重要内容，包括无线网络组件、网络结构、无线网络提供的服务等。
- 在上述基础上，着重对802.11 MAC层进行了介绍，包括MAC帧格式、MLME等。这部分内容是理解后续章节关于wpa\_supplicant介绍的核心。
- 然后解决802.11的安全性问题。建议读者从机密性、完整性和身份验证三个方面来理解其中的各种安全保护方法。
- 本章最后对Linux Wi-Fi API进行了介绍，目前用得最多的应该是nl80211。不过读者可先从简单的wext开始学习。

**提示** 本章是笔者从事Android写作以来耗时最长的一章（包括学习时间长达3个多月，而且还有很多技术点未能覆盖）。在这个过程中，笔者也经历过烦恼和痛苦，感觉比纯粹的代码分析难度要大。这也是笔者希望读者把注意力放到代码背后的理论上的初衷。从下一章开始，在分析wpa\_supplicant的同时，通过背景知识介绍的方式来补充本节没有涵盖的内容。

### 3.5.2 参考资料说明

#### 1. 概述

[1] 《Real 802.11 Security: Wi-Fi Protected Access and 802.11i》7.1节“Relationship Between Wi-Fi and IEEE 802.11”

说明：本书是笔者找到的关于802.11无线网络安全技术方面知识面最完整的书籍。是英文版，需要掌握Wi-Fi的基础知识后才能真正理解。

#### 2. 无线电频谱知识

[2] 《802.11无线网络权威指南（第2版）》P16-P17，“无线电频谱：关键资源”

说明：此书是目前市面上关于802.11无线网络书籍的“圣经”。不过其排版奇特，连章节号都没有，读起来着实有些费劲。读者可首先阅读这本书以对Wi-Fi有个基本了解。另外，该书涉及物理层的内容可以略去不读。

[3] 《802.11无线网络权威指南（第2版）》“无线网络导论”一节，P20-P22

[4] <http://baike.baidu.com/view/345218.htm>

说明：百度百科词条“802.11”。笔者对其内容做了必要的增删改。

#### 3. OSI基本参考模型及相关基本概念

[5] ISO/IEC 7498-1“Basic Reference Model: The Basic Model”

说明：OSI模型的官方文档，不过和其他官方文档一样，极难理清楚其间的逻辑关系，做手册使用还可以。

[6] <http://baike.baidu.com/view/486949.htm>

说明：百度百科词条“开放系统互连参考模型”。

[7] ISO/IEC 8802-2 "Part 2 Logical Link Control"

说明：Logical Link Control层的官方标准。读者只需阅读第一节"Overview"。

#### 4. CSMA/CA介绍

[8] <http://baike.baidu.com/view/645723.htm>

说明：百度百科词条"CSMA/CA"，以通俗的方式解释了CSMA/CA的工作原理。也可参考《802.11无线网络权威指南（第2版）》第3章，P47-P55。

#### 5. MAC层Service及其他概念

[9] IEEE 802.1X-2010 "Port-Based Network Access Control", Annex D "Basic architectural concepts and terms"

说明：大名鼎鼎的802.1X规范。不过这些规范引用的其他标准非常多，所以802.1X在附录D中对一些基本概念和术语进行了一番介绍。建议读者阅读此文档。

[10] <http://www.doc88.com/p-696270935777.html>

说明：ISO/IEC 15802-1电子文档，由doc88提供。全名为"Part 1: Medium Access Control (MAC) service definition"，定义了MAC service和相关的原语。

#### 6. MIB介绍

[11]

[http://en.wikipedia.org/wiki/Management\\_information\\_base](http://en.wikipedia.org/wiki/Management_information_base)

说明：维基百科词条"Management Information Base"。英文介绍，比较容易理解。

#### 7. 802.11组件

[12] 《802.11-2012》4.1节“General description of the architecture”、4.2节“How WLAN systems are different”、4.3节“Components of the IEEE 802.11 architecture”

## 8. 802.11 Service介绍

[13] 《802.11-2012》4.5节“Overview of the Services”

说明：注意，《802.11-2012》第4章非常重要，里边的许多基本概念都需要了解。该节包括的内容非常多，读者应有选择地阅读。

## 9. 802.11 MAC服务和帧

[14] 《802.11-2012》第5章“MAC service definition”

说明：对802.11 MAC服务进行了完整说明。

[15] 《802.11-2012》第8章“Frame formats”

说明：802.11 MAC帧完整说明。读者可当做手册来用。

[16] [http://technet.microsoft.com/en-us/library/cc757419\(v=ws.10\).aspx](http://technet.microsoft.com/en-us/library/cc757419(v=ws.10).aspx)

说明：微软技术文章“How 802.11 Wireless Works”。通俗易懂，包含的知识面也比较全。读者可通过它对802.11有一个大致的认识。

[17] 《802.11无线网络权威指南（第2版）》第4章“802.11成帧细节”，P78-P127

[18] 《802.11无线网络权威指南（第2版）》第8章中的“节省电力”，P200-P208

说明：非常详细地介绍了Power Save的原理和过程。

[19] 《802.3-2008》“CSMA/CD Access Method and Physical Layer Specifications”3.2.3节“Address fields”

说明：LAN中MAC层的官方文档。其中有对MAC地址格式的说明。

[20] <http://www.doc88.com/p-905531556977.html>

说明： doc88上关于MAC组播地址的中文说明。建议读者阅读此文档。

[21] 《802.11无线网络权威指南（第2版）》第3章中的“802.11对上层协议的封装”，P65-P66

## 10. 802.11 MAC管理实体

[22] 《802.11-2012》第6章“Layer management”

说明： MLME的官方说明，非常详细。请读者当手册使用。

[23] 《802.11无线网络权威指南（第2版）》第8章“管理操作”，P182-P224

说明： 逻辑还算清晰，建议读者结合参考资料[22]一起阅读。

[24] 《802.11-2012》4.10节“IEEE Std 802.11 and IEEE Std 802.1X-2004”

说明： 介绍802.1X如何与802.11相结合。

## 11. WEP介绍

[25] 《802.11无线网络权威指南（第2版）》第5章“有线等效加密”，P127-P142

说明： 关于WEP的详细介绍。但有些内容用得非常少（例如关于动态WEP密匙的说明）。

[26] 《802.11-2012》11.2.2节“Wired Equivalent Privacy (WEP) ”

说明： 官方对WEP的说明，非常详细。

[27] 《802.11-2012》11.2.3节“Pre-RSNA authentication”

说明： 官方对WEP中身份验证方法的说明。

[28]

<http://documentation.netgear.com/reference/ita/wireless/pdfs/FullManual.pdf>

说明：“Wireless Networking Basics”，Netgear公司提供的关于Wi-Fi安全方面的一些简单介绍。

## 12. RSN数据加密及完整性校验

[29] 《802.11-2012》11.4节“RSNA confidentiality and integrity protocols”

说明：官方文档关于TKIP和CCMP的介绍。

[30] 《802.11无线网络权威指南（第2版）》第7章“802.11：RSN、TKIP与CCMP”，P162-P176

说明：请读者结合参考资料[29]研究。

## 13. EAP和802.1X介绍

[31] RFC3748“Extensible Authentication Protocol (EAP) ”

说明：EAP的官方文档，共68页，难度不是特别大，建议读者阅读全文。

[32] [http://en.wikipedia.org/wiki/IEEE\\_802.1X](http://en.wikipedia.org/wiki/IEEE_802.1X)

说明：维基百科词条“IEEE 802.1X”。图文并茂，读者可仔细阅读此文。

[33] 《802.11无线网络权威指南（第2版）》第6章“802.1X用户身份验证”，P142-P154

[34]

[http://www.h3c.com.cn/Products\\_\\_Technology/Technology/Security\\_Encrypt/Other\\_technology/Technology\\_recommend/200812/624138\\_30003\\_0.htm](http://www.h3c.com.cn/Products__Technology/Technology/Security_Encrypt/Other_technology/Technology_recommend/200812/624138_30003_0.htm)

说明：H3C公司关于802.1X的介绍，非常详细，难度较小。读者可先阅读此文档。

[35] 《802.1X-2010》第11章“EAPOL PDUs”

说明：官方对EAPOL格式的详细说明。

#### 14. RSNA介绍

[36] <http://www.docin.com/p-439759696.html>

说明：“一种针对RSNA无线网络的安全等级回滚攻击研究”，来自豆丁网。3页内容，比较容易理解。

[37] 《Real 802.11 Security: Wi-Fi Protected Access and 802.11i》第7章“WPA, RSN, and IEEE 802.11i”和第9章“WPA and RSN Key Hierarchy”

说明：此书是目前笔者找到的关于Wi-Fi安全性方面覆盖面最齐全的资料。建议读者深入阅读。

[38] 《802.11-2012》11.6节“Keys and key distribution”

说明：官方文档关于密匙派生的说明。

[39] 《802.11无线网络权威指南（第2版）》第7章“802.11: RSN、TKIP与CCMP”，P176-P182

说明：介绍了RSN的运作方式，对密匙派生和缓存有较为详细的说明。

#### 15. Linux Wi-Fi编程API

[40]

<http://wireless.kernel.org/en/developers/Documentation/Wireless-Extensions>

说明：Linux Wireless Kernel官方网站，内容非常丰富。建议读者仔细阅读。

## 16. netlink编程

[41] <http://www.infradead.org/~tgr/libnl/>

说明：libnl官方网站，文档较为丰富。读者可阅读其中关于libnl的文档。

# 第4章 深入理解wpa\_supplicant

本章所涉及的源代码文件名及位置

- BoardConfig.mk build/target/board/generic/BoardConfig.mk
- android.cfg  
externel/wpa\_supplicant\_8/wpa\_supplicant/android.cfg
- wpa\_supplicant.conf  
externel/wpa\_supplicant\_8/wpa\_supplicant/wpa\_supplicant.conf
- main.c externel/wpa\_supplicant\_8/wpa\_supplicant/main.c
- wpa\_supplicant.c  
externel/wpa\_supplicant\_8/wpa\_supplicant/wpa\_supplicant.c
- wpa\_debug.c  
externel/wpa\_supplicant\_8/wpa\_supplicant/wpa\_debug.c
- eap\_register.c  
externel/wpa\_supplicant\_8/wpa\_supplicant/eap\_register.c
- eap\_i.h externel/wpa\_supplicant\_8/src/eap\_peer/eap\_i.h
- eloop.h externel/wpa\_supplicant\_8/src/utils/eloop.h
- eloop.c externel/wpa\_supplicant\_8/src/utils/eloop.c
- if.h externel/kernel-headers/original/linux/if.h
- config\_ssid.h  
externel/wpa\_supplicant\_8/wpa\_supplicant/config\_ssid.h
- config\_file.c  
externel/wpa\_supplicant\_8/wpa\_supplicant/config\_file.c

- config.c  
external/wpa\_supplicant\_8/wpa\_supplicant/config.c
- drivers.mk  
external/wpa\_supplicant\_8/src/drivers/drivers.mk
- driver\_n180211.c  
externel/wpa\_supplicant\_8/src/drivers/driver\_n180211.c
- rfkill.c      externel/wpa\_supplicant\_8/src/drivers/rfkill.c
- wpas\_glue.c  
externel/wpa\_supplicant\_8/wpa\_supplicant/wpas\_glue.c
- ctrl\_iface\_unix.c  
externel/wpa\_supplicant\_8/wpa\_supplicant/ctrl\_iface\_unix.c
- bss.c      externel/wpa\_supplicant\_8/wpa\_supplicant/bss.c
- wpa.c      externel/wpa\_supplicant\_8/src/rsn\_supp/wpa.c
- eap\_defs.h  
external/wpa\_supplicant\_8/src/eap\_common/eap\_defs.h
- defs.h      external/wpa\_supplicant\_8/src/common/defs.h
- eap.h      external/wpa\_supplicant\_8/src/eap\_peer/eap.h
- eapol\_supp\_sm.c  
external/wpa\_supplicant\_8/src/eapol\_supp/eapol\_supp\_sm.c
- ctrl\_iface.c  
externel/wpa\_supplicant\_8/wpa\_supplicant/ctrl\_iface.c
- scan.c      externel/wpa\_supplicant\_8/wpa\_supplicant/scan.c
- wpa\_i.h      externel/wpa\_supplicant\_8/src/rsn\_supp/wpa\_i.h
- wpa.c      externel/wpa\_supplicant\_8/src/rsn\_supp/wpa.c

- eapol\_supp\_sm.c  
externel/wpa\_supplicant\_8/src/eapol\_supp\_sm.c

## 4.1 概述

wpa\_supplicant<sup>①</sup>是一个开源软件项目，它实现了Station对无线网络进行管理和控制的功能。根据官方描述，wpa\_supplicant所支持的功能非常多，此处列举其中几个重要的功能点。

1) 支持WPA和IEEE 802.11i所定义的大部分功能。

这部分功能集中在安全方面，包括以下。

- 支持WPA-PSK（即WPA-Personal）和WPA-Enterprise（即利用RAIDUS认证服务器来完成身份认证的情况）。
- 数据加密方面支持CCMP、TKIP、WEP104和WEP40。注意，WEP104和WEP40中的数字代表密钥的长度。104表示密钥长度为104个二进制位（如以ASCII字符个数来计算的话，WEP104支持的密钥长度为13个ASCII字符）。
- 完全支持WPA和WPA2，包括PMKSA缓存，预认证（pre-authentication）等功能。
- 支持IEEE 802.11r和802.11w，其中802.11r规范定义了快速基础服务转移（Fast Transition）功能，而802.11w则新增了对管理帧的安全保护机制。
- 支持WFA制定的Wi-Fi Protected Setup功能、P2P、TDLS等。

2) 支持多种EAP Method。

主要和802.1X中Supplicant的功能有关，wpa\_supplicant支持多达25种EAP Method，包括以下。

- EAP-TLS：TLS（Transport Layer Security）本身是一种传输层安全协议，它利用密钥算法提供端点身份认证与通讯保密，其基础是公钥基础设施（Public Key Infrastructure，PKI）。EAP-TLS定义于RFC 5216。

- EAP-PEAP：PEAP（Protected Extensible Authentication Protocol，可扩展EAP）由微软、思科以及RSA Security三个公司共同开发，是一种利用证书加用户名和密码来进行身份验证的方法。
- EAP-TTLS：TTLS（Tunneled Transport Layer Security，隧道传输层安全协议）是TLS的拓展，相比TLS，它简化了认证过程中客户端的工作。
- EAP-SIM、EAP-PSK、EAP-GPSK等其他认证方法。

提示 可阅读参考资料[1]以了解更多EAP方法的知识。

### 3) 对各种无线网卡和驱动的支持。

- 支持n180211/cfg80211驱动和Linux Wireless Extension驱动<sup>②</sup>。
- 支持Windows平台中的NDIS驱动。

提示 wpa\_supplicant虽然支持Windows平台，但相信绝大多数读者使用的是Windows自带的无线网络管理程序（或者Intel芯片相关软件提供的无线网络管理程序）。从功能角度来说，读者可认为wpa\_supplicant是这些私有程序的一种开源实现。

Android作为开源世界的集大成者，在无线网络管理和控制方面直接使用了wpa\_supplicant。Android 4.1中，external目录下有两个和wpa\_supplicant相关的目录，分别是wpa\_supplicant\_6和wpa\_supplicant\_8。6和8分别代表对应wpa\_supplicant的版本号为0.6.10和2.0-devel。

提示 关于wpa\_supplicant的发布历史，请读者参考<http://hostap.epitest.fi/releases.html>。

本书的分析目标是wpa\_supplicant\_8，它包含三个主要子目录，分别如下。

- hostapd：当手机进入Soft AP模式时，手机将扮演AP的角色，故需要hostapd来提供AP的功能。

- wpa\_supplicant : Station模式，也叫Managed模式。本书分析的重点。
- src : hostapd和wpa\_supplicant中都包含了一些通用的数据结构和处理方法，这些内容都放在此src目录中。注意，hostapd/src和wpa\_supplicant/src子目录均链接到此src目录。

wpa\_supplicant是Android用户空间中无线网络部分的核心模块，所有Framework层中和Wi-Fi相关的操作最终都将借由wpa\_supplicant来完成。另外，wpa\_supplicant本身对802.11、802.1X和Wi-Fi Alliance定义的一些规范都有极好的支持。所以，分析它将是加深理解802.11及相关理论知识的一个非常重要的途径。

本章分两条路线来分析wpa\_supplicant和相关的功能模块。

- 路线一：首先介绍wpa\_supplicant的初始化过程。这条路线将帮助读者了解wpa\_supplicant中常见的数据结构及之间的关系。这条路非常难走，请读者做好心理准备。
- 路线二：通过命令行发送命令的方式触发wpa\_supplicant进行相关工作，使手机加入一个利用WPA-PSK进行认证的无线网络。这条路线将帮助读者了解wpa\_supplicant中的命令处理、scan、association、4-Way Handshake等相关处理流程。

**提示** 后续章节还将围绕Android中无线网络技术开展更多的讨论。第5章将介绍Android Framework中的WifiService及其相关模块。第6、7章将继续wpa\_supplicant之旅，其内容和WPS、Wi-Fi P2P以及WifiP2pService有关。

为了行文方便，本书将用WPAS来表示wpa\_supplicant。另外，后文代码分析中还能见到一种重要的数据结构，也叫wpa\_supplicant。请读者根据上下文信息来理解wpa\_supplicant的含义。

正式开始分析之旅前，先简单了解wpa\_supplicant。

**①** 注意，wpa\_supplicant项目中还包含一个名为hostapd程序的代码，它实现了AP的功能，本书不讨论。官方地址为<http://hostap.epitest.fi/>。

② 根据审稿专家的反馈，wpa\_supplicant仅支持Linux Wireless Extension V19以后的版本。

## 4.2 初识wpa\_supplicant

本节介绍WPAS一些外围知识，包括软件结构、编译配置、控制命令和对应控制API的用法。其中，控制命令的格式和API的用法将在介绍WifiService相关模块时见到。另外，在研究WPAS时，能熟练使用git查询历史版本信息也非常关键。首先从WPAS软件架构开始。

#### 4.2.1 wpa\_supplicant架构

wpa\_supplicant是一个比较庞大的开源软件项目，包含500多个文件，20万行代码，其内部模块构成如图4-1所示<sup>[2]</sup>。

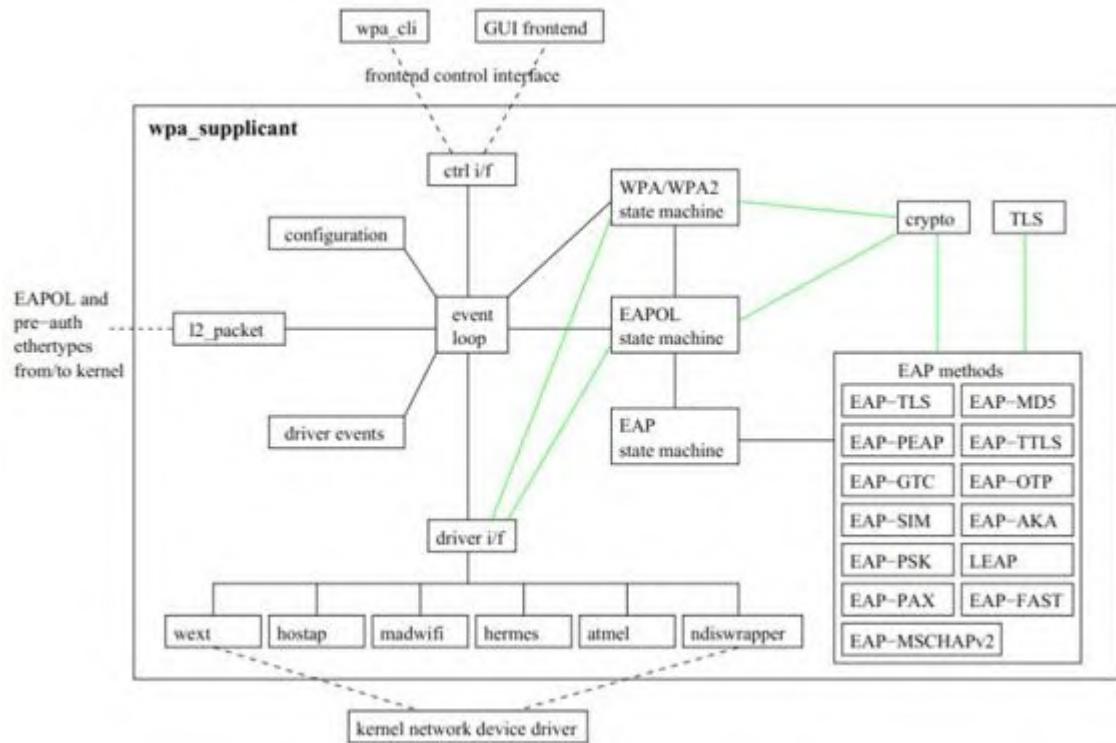


图4-1 wpa\_supplicant软件架构

图4-1所示的WPAS软件架构包括如下重要模块。

- WPAS所有工作都围绕事件（event loop模块）展开。它是基于事件驱动的。事件驱动和消息驱动类似，主线程等待事件的发生并处理它们。WPAS没有使用多线程编程，所有事件处理都在主线程中完成。从这一点看，WPAS的运行机制很简单。
- 位于event loop模块下方的driver i/f (i/f代表interface) 接口模块用于隔离和底层驱动直接交互的那些driver控制模块（如wext、ndiswrapper等，WPAS中称为driver wrapper）。这些driver wrapper

和平台以及芯片所使用的驱动相关。不过，由于driver i/f的隔离作用，WPAS中其他模块将能最大程度保持平台以及驱动无关性。

- driver wrapper经常要返回一些信息给上层。WPAS中，这些信息将通过driver events的方式反馈给WPAS其他模块进行处理。
- 上一章曾介绍过EAP以及EAPOL协议。除了定义消息格式外，RFC4137文档定义了EAP状态机，而802.1X文档中还定义了EAPOL状态机。WPAS根据这两个协议分别实现了EAP和EAPOL状态机。本章后续将详细分析这两个状态机以及背后的协议。除此之外，WPAS还定义了自己的状态机（即WPA/WPA2 State Machine）。
- WPAS实现了多种EAP方法，如EAP method模块。另外它还包含了TLS模块和crypto模块用于支持对应的EAP方法。
- EAPOL以及EAP消息都属于LLC层数据，所以WPAS的12\_packet模块用于收发EAPOL和EAP消息。
- WPAS支持较多的配置参数，这些参数的处理由configuration模块完成。
- WPAS是C/S结构中的Server端，它通过ctrl i/f模块向客户端提供通信接口。Linux/UNIX平台中，Client端利用Unix域socket与其通信。目前常用的Client端wpa\_cli（无界面的命令行程序）和wpa\_gui（UI用Qt实现）。

WPAS支持众多功能，使用前往往需根据平台或驱动的特性进行编译配置，下面通过一个实例来介绍如何在Android中编译wpa\_supplicant。

读累了记得休息一会儿哦~

公众号：古德猫宁李

- 电子书搜索下载
- 书单分享
- 书友学习交流

网站：[沉金书屋 https://www.chenjin5.com](https://www.chenjin5.com)

- 电子书搜索下载
- 电子书打包资源分享
- 学习资源分享

## 4.2.2 wpa\_supplicant编译配置

先介绍本实例的背景情况。有一台三星Galaxy Note 2手机，其OS为Android 4.1.2。现在，编译一个AOSP（Android Open Source Project）的wpa\_supplicant程序以替换Note 2中原有的wpa\_supplicant。

**提示** AOSP即Google公版Android源码。几乎所有手机厂商都会根据芯片、硬件以及厂商自定义的特性去修改它。由于Note 2源码不公开，所以笔者只能编译AOSP版的wpa\_supplicant。

假设读者已经按第1章要求部署Android 4.1源码和开发环境，接下来要做的事情如下。

```
cd 4.1source #首先进入4.1源码根目录  
source build/envsetup #建立Android源码编译环境  
lunch #选择要编译的设备和版本。笔者选择了1，代表full-eng。eng代表工程版，该选项对应的目标设备类型  
# (TARGET_PRODUCT) 为generic，其编译出来的镜像文件可由模拟器加载并运行
```

由上述配置可知，笔者将使用generic版本编译一个wpa\_supplicant以运行在真实的机器上。

**提示** 通过执行lunch命令可知，不同的设备应有对应的编译配置项。由于笔者没有Note 2的源码，所以只能尝试编译generic版本。

接下来要为generic平台定制所使用的wpa\_supplicant版本，通过修改BoardConfig.mk来完成的。

[-->BoardConfig.mk]

```
#在此文件最后添加如下内容  
WPA_SUPPLICANT_VERSION := VER_0_8_X #表明使用wpa_supplicant_8  
BOARD_WPA_SUPPLICANT_DRIVER := NL80211 #表明驱动使用NL80211  
BOARD_WLAN_DEVICE := bcmdhd #表明Kernel中的Wi-Fi设备为博通公司的bcmdhd  
#编译博通公司驱动相关的静态库，该库对应的代码也在AOSP源码中，位置是
```

```
#hardware/broadcom/wlan/bcmdhd/wpa_supplicant_8_lib/  
BOARD_WPA_SUPPLICANT_PRIVATE_LIB := lib_driver_cmd_bcmdhd
```

巧合的是，Note 2使用的wlan芯片刚好为bcmdhd。

除了修改BoardConfig.mk外，WPAS也定义了自己的编译配置文件 android.config，其内容如下。

```
.....#该文件主要定义了编译时生成的宏，各平台根据自己的硬件情况去设置需要  
编译的内容  
# Driver interface for generic Linux wireless extensions  
CONFIG_DRIVER_WEXT=y #可注释这一条以取消编译WEXT相关代码  
# Driver interface for Linux drivers using the nl80211 kernel  
interface  
CONFIG_DRIVER_NL80211=y #可去掉此行的注释符号以增加对Nl80211的支持  
CONFIG_LIBNL20=y  
.....#其他很多编译配置项都可在此文件中修改  
#注意，此文件中对CONFIG_DRIVER_NL80211的修改和BoardConfig.mk中的  
BOARD_WPA_SUPPLICANT_DRIVER  
#相重合。BoardConfig.mk的优先级较高，所以请读者先修改它
```

配置完毕后，开始编译。

```
#首先要编译wpa_supplicant依赖的静态库lib_driver_cmd_bcmdhd  
mmm hardware/broadcom/wlan/bcmdhd/wpa_supplicant_8_lib/  
mmm external/wpa_supplicant_8 #生成wpa_supplicant，同时也会生成  
wpa_cli
```

将编译后的wpa\_supplicant替换Note 2 的/system/bin/wpa\_supplicant并设置其为可运行（通过chmod命令设置其权限位0755）。同时，把wpa\_cli”push”到/system/bin下为后续测试做准备。

经过测试发现，AOSP的wpa\_supplicant以及wpa\_cli均能正常工作在 Note 2上。这也间接表明Note 2并未对wpa\_supplicant以及博通芯片相关的代码做较大改动。

注意 严格来说，android.cfg应该是唯一的编译控制文件。但由于底层wlan芯片不同，WPAS可能还依赖其他模块。所以，在具体实施时，BoardConfig.mk（或其他文件，视具体情况而定）也需要做修改。

### 4.2.3 wpa\_supplicant命令和控制API

由图4-1可知，WPAS对外通过控制接口模块与客户端通信。在Android平台中，WPAS的客户端是位于Framework中的WifiService。用户在Settings界面进行Wi-Fi相关的操作最终都会经由WifiService通过发送命令的方式转交给wpa\_supplicant去执行。

#### 1. 命令

WPAS定义了许多命令，常见命令如下。

- PING：心跳检测命令。客户端用它判断WPAS是否工作正常。WPAS收到“PING”命令后需要回复“PONG”。
- MIB：客户端用该命令获取设备的MIB信息。
- STATUS：客户端用该命令来获取WPAS的工作状态。
- ADD\_NETWORK：为WPAS添加一个新的无线网络。它将返回此新无线网络的ID（从0开始）。注意，此network id非常重要，客户端后续将通过它来指明自己想操作的无线网络。
- SET\_NETWORK<network id><variable><value>：network id是无线网络的ID。此命令用于设置指定无线网络的信息。其中variable为参数名，value为参数的值。
- ENABLE\_NETWORK<network id>：使能某个无线网络。此命令最终将促使WPAS发起一系列操作以加入该无线网络。

除了接收来自Client的命令外，WPAS也会主动给Client发送命令。例如，WPAS需用户为某个无线网络输入密码。这类命令称为Interactive Request，其格式如下。

WPAS向客户端发送的命令遵循以下格式。

CTRL-REQ-<field name>-<network id>-<human readable text>

如以下命令表示需要用户为0号网络输入密码。

CTRL-REQ-PASSWORD-0-Passwork needed for SSID test-network

客户端处理完后，需回复：

CTRL-RSP-<field name>-<network id>-<value>

目前支持的field包括PASSWORD、IDENTITY（EAP中的identity或者用户名）、PIN等。

最后，WPAS还可通过形如以下命令向客户端通知一些事情。

CTRL-EVENT-<field>-<text>

提示 除了“CTRL-EVENT-XXX”之外，WPAS还支持形如“WPA:XXX”和“WPS-XXX”的通知事件。这些事件和WPA和WPS有关。下一章分析WifiService时还能见到它们。

图4-2所示为利用wpa\_cli测试status命令得到的结果。图中最后几行显示WPAS向wpa\_cli返回了两个CTRL-EVENT信息。

```
Interactive mode
> status
bssid=5c:63:bf:bd:87:ae
ssid=Test
id=3
mode=station
pairwise_cipher=CCMP
group_cipher=CCMP
key_mgmt=WPA2-PSK
wpa_state=COMPLETED
ip_address=192.168.1.123
address=90:18:7c:69:88:e2
<3>CTRL-EVENT-STATE-CHANGE id=3 state=9 BSSID=00:00:00:00:00:00 SSID=Test
<3>CTRL-EVENT-CONNECTED - connection to 5c:63:bf:bd:87:ae completed (reauth) [id=3 id str=]
```

图4-2 wpa\_cli命令测试

## 2. 控制API

Android平台中WifiService是WPAS的客户端，它和WPAS交互时必须使用wpa\_supplicant提供的API。这些API声明于wpa\_ctrl.h中，其用法

如下。

```
// 必须包含此头文件，链接时需包含libwpa_client.so动态库
#include "wpa_ctrl.h"
```

客户端使用wpa\_ctrl时首先要分配控制对象。下面两个API用于创建和销毁控制对象wpa\_ctrl。

```
// 创建一个wpa控制端对象wpa_ctrl。Android平台中，参数ctrl_path代表
// unix域socket的位置
struct wpa_ctrl * wpa_ctrl_open(const char *ctrl_path);
void wpa_ctrl_close(struct wpa_ctrl *ctrl);           // 注销wpa_ctrl
// 控制对象
```

下面这个函数用于发送命令给WPAS。

```
// 客户端发送命令给wpa_supplicant，回复的消息保存在reply中
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));
```

msg\_cb是一个回调函数，该参数的设置和WPAS中C/S通信机制的设计有关。

从Client角度来看，它发送给WPAS的命令所对应的回复属于solicited event（有请求的事件），而前面所提到的CTRL-EVENT事件（用于通知事件）对应为unsolicited event（未请求的事件）。当Client在等待某个命令的回复时，WPAS同时可能有些通知事件要发送给客户端，这些通知事件不是该命令的回复，所以不能通过wpa\_ctrl\_request的reply参数返回。

为了防止丢失这些通知事件，wpa\_cli设计了一个msg\_cb回调用于客户端在等待命今回复的时候处理那些unsolicited event。

这种一个函数完成两样完全不同的功能的设计实在有些特别，所以wpa\_supplicant规定只有打开通知事件监听功能的wpa\_ctrl对象，才能在wpa\_ctrl\_request中通过msg\_cb获取通知事件。而打开通知事件监听功能相关的API如下所示。

```
// 打开通知事件监听功能  
int wpa_ctrl_attach(struct wpa_ctrl *ctrl);  
// 打开通知事件监听功能的wpa_ctrl对象能直接调用下面的函数来接收  
unsolicited event  
int wpa_ctrl_recv(struct wpa_ctrl *ctrl, char *reply, size_t  
*reply_len);
```

如果客户端并不发送命令，而只是想接收Unsolicited event，可通过wpa\_ctrl\_recv函数来达到此目的。

综上所述，单独使用wpa\_ctrl\_recv和wpa\_ctrl\_request都不方便。所以，一种常见的用法是：客户端创建两个wpa\_ctrl对象来简化自己的逻辑处理。

- 一个打开了通知事件监听功能的wpa\_ctrl对象将只通过wpa\_ctrl\_recv来接收通知事件。
- 另外一个wpa\_ctrl专职用于发送命令和接叔回复。由于没有调用wpa\_ctrl\_attach，故它不会收到通知事件。

提示 下一章分析WifiService时将见到这种创建两个wpa\_ctrl对象的做法。

#### 4.2.4 git的使用

WPAS难度较大的一个重要原因是其注释较少，很多变量的含义没有任何解释。笔者也为此大伤脑筋。不得已，只能通过查看WPAS代码的历史版本来寻根溯源。经过实践，笔者总结了利用git来查询WPAS历史版本信息的一些步骤，分别如下。

用git clone命令下载WPAS官方代码。

```
git clone git://w1.fi/srv/git/hostap.git
```

以下命令的含义是查询use\_monitor在driver\_nl80211.c中的变化情况。

```
git blame src/drivers/driver_nl80211.c | grep use_monitor
```

因为use\_monitor定义于该文件中，所以用git blame查看。得到的结果如图4-3所示。

```
root@linnpost:/ /wpa_supplicantOfficial/hostap# git blame src/drivers/driver_nl80211.c | grep use_monitor
a11241fa (Johannes Berg 2011-12-06 18:24:00 +0200 254) unsigned int use_monitor;
73a3c0ff (Felipe Fletkau 2012-09-23 13:28:31 +0300 2967) drv->use_monitor = linfo.poll_command_supported || linfo.data_tx_status;
536062f2 (Jouni Malinen 2011-12-23 18:13:01 +0200 2969) if (drv->device.ap_sme && drv->use_monitor) {
536062f2 (Jouni Malinen 2011-12-23 18:13:01 +0200 2976)     wpa_printf(MSG_DEBUG, "nl80211: Disable use_monitor "
536062f2 (Jouni Malinen 2011-12-23 18:13:01 +0200 2979)     drv->use_monitor = 0;
a11241fa (Johannes Berg 2011-12-06 18:24:00 +0200 2988)     if (!drv->use_monitor && linfo.data_tx_status)
a11241fa (Johannes Berg 2011-12-06 18:24:00 +0200 3332)         if (!drv->use_monitor)
a11241fa (Johannes Berg 2011-12-06 18:24:00 +0200 5648)         if (drv->use_monitor)
536488c05 (Jouni Malinen 2011-12-23 18:10:59 +0200 6723)             "use_monitor=0", drv->device.ap_sme, drv->use_monitor);
a11241fa (Johannes Berg 2011-12-06 18:24:00 +0200 6733)     if (!drv->device.ap_sme && !drv->use_monitor)
536cc0602 (Jouni Malinen 2011-12-23 18:15:07 +0200 6737)         if (drv->device.ap_sme && !drv->use_monitor)
a11241fa (Johannes Berg 2011-12-06 18:24:00 +0200 6741)         if (!drv->device.ap_sme && drv->use_monitor &&
536cc0602 (Jouni Malinen 2011-12-23 18:15:07 +0200 6763)             if (!drv->use_monitor)
536cc0602 (Jouni Malinen 2011-12-23 18:15:07 +0200 6765)         } else if (drv->use_monitor)
a11241fa (Johannes Berg 2011-12-06 18:24:00 +0200 6816)             if (drv->device.ap_sme || !drv->use_monitor)
```

图4-3 git blame结果

图4-3中的第一行显示了use\_monitor最早出现的patch的情况，其对应的commit id是a11241fa。接着，再通过命令“git log a11241fa”可查看当时的commit信息。

```
commit a11241fa114923b47892ad3279966839e9c2741d
Author: Johannes Berg <johannes.berg@intel.com>
Date:   Tue Dec 6 18:24:00 2011 +0200

nl80211: Use nl80211 for mgmt TX/RX in AP mode

To achieve this, multiple things are needed:
1) since hostapd needs to handle *all* action frames,
   make the normal registration only when in a non-AP
   mode, to be able to do this use the new socket
2) store the frequency in each BSS to be able to give
   the right frequency to nl80211's mgmt-tx operation
3) make TX status processing reject non-matched cookie
   only in non-AP mode

The whole thing depends on having station-poll support
in the kernel. That's currently a good indicator since
the kernel patches are added together.

Signed-hostap: Johannes Berg <johannes.berg@intel.com>
```

图4-4 git log结果

图4-4展示了a11241fa对应的commit消息。由于提交者一般会在该消息中添加注释性内容，所以可通过研究这些内容来了解代码中某些变量的含义。

下面正式开始WPAS的代码分析之旅。首先分析WPAS的初始化流程。

## 4.3 wpa\_supplicant初始化流程

Android系统中，WPAS启动是通过“setprop ctrl.start wpa\_supplicant”来触发init进程去fork一个子进程来完成的。WPAS在init配置文件中被定义为一个service。图4-5所示为Note 2 init.smdk4x12.rc文件中关于wpa\_supplicant的定义。

```
service wpa_supplicant /system/bin/wpa_supplicant \
    -Dnl80211 -iwlan0 -e/data/misc/wifi/entropy.bin -c/data/misc/wifi/wpa_supplicant.conf
    # we will start as root and wpa_supplicant will switch to user wifi
    # after setting up the capabilities required for WEXT
    # user wifi
    # group wifi inet keystore
    class main
    socket wpa_wlan0 dgram 660 wifi wifi
    disabled
    oneshot
```

图4-5 init配置文件中的wpa\_supplicant

图4-5中的黑框展示了wpa\_supplicant的启动参数①。其众多参数中，最重要的是通过“-c”参数指定的WPAS启动配置文件（图4-5中，该配置文件全路径名为/data/misc/wifi/wpa\_supplicant.conf）。

**提示** wpa\_supplicant源代码中包含一个启动配置文件的模板，该文件对各项配置参数都有说明。其文件路径为  
external/wpa\_supplicant\_8/wpa\_supplicant/wpa\_supplicant.conf  
。

Note 2中该配置文件的内容如图4-6所示。

```
ctrl_interface=wlan0
update_config=1
device_name=t03gzc
manufacturer=samsung
model_name=GT-N7100
model_number=GT-N7100
serial_number=4df13b8d0429af5b
device_type=10-0050F204-5
config_methods=physical_display.virtual_push_button.keypad
p2p_listen_reg_class=81
p2p_listen_channel=1
p2p_oper_reg_class=115
p2p_oper_channel=48
bss_expiration_scan_count=1

network={
    → ssid="TP-LINK_1F9C5E"
    → key_mgmt=NONE
    → auth_alg=OPEN.SHARED
    → wep_key0="xxxx"
    → priority=4
}

network={
    → ssid="Tenda_487A08"
    → psk="oeqn2b8tzxzk3"
    → proto=RSN
    → key_mgmt=WPA-PSK
    → pairwise=CCMP
    → auth_alg=OPEN
}
```

一个无线网络的配置项

图4-6 wpa\_supplicant.conf文件内容

- ctrl\_interface指明控制接口unix域socket的文件名。
- update\_config表示如果WPAS运行过程中修改了配置信息，则需要把它们保存到此wpa\_supplicant.conf文件中。
- 从device\_name到config\_method都和WPS设置有关。后续章节介绍其作用。

- p2p等选项和Wi-Fi P2P有关。后续章介绍它们的作用。
- WPAS运行过程中得到的无线网络信息都会通过一个“network”配置项保存到此配置文件中。如果该信息完整，一旦WPAS找到该无线网络就会尝试用保存的信息去加入它（这也是为什么用户在settings中打开无线网络后，手机能自动加入周围某个曾经登录过的无线网络的原因）。
- network项包括的内容非常多。图中第二个network项展示了该无线网络的ssid、密钥管理方法（key management）、身份认证方法及密码等信息。network中的priority表示无线网络的优先级。其作用是，如果同时存在多个可用的无线网络，WPAS优先选择priority高的那一个。

下面正式进入WPAS的代码，先来看其入口函数main。

① 关于init.rc文件的解析及setprop的实现，读者可阅读《深入理解Android：卷 I》第3章。

### 4.3.1 main函数分析

Android平台中，main函数定义于main.c中，代码如下所示。

[-->main.c: : main]

```
int main(int argc, char *argv[])
{
    int c, i;
    struct wpa_interface *ifaces, *iface;
    int iface_count, exitcode = -1;
    struct wpa_params params;
    struct wpa_global *global;
    /*

Android平台中，下面这个函数的实现在os_unix.c中。Android对其做了一些修改，主要是权限方面的设置防止某些情况下被破解者利用权限漏洞以获取root权限。
*/
    if (os_program_init())
        return -1;

    os_memset(&params, 0, sizeof(params));
    params.wpa_debug_level = MSG_INFO;

    iface = ifaces = os_zalloc(sizeof(struct wpa_interface));
    .....
    iface_count = 1;
    wpa_supplicant_fd_workaround();           // 输入输出重定向
到/dev/null设备

    for (;;) { // 参数解析，由图4-3所知，Note 2中WPAS启动只使用了4个参数
        c = getopt(argc, argv,
        "b:Bc:C:D:d:e:f:g:hi:KLNo:O:p:P:qstuvW");
        if (c < 0)
            break;
        switch (c) {
        .....
        case 'c':
            // 指定配置文件名。注意，该参数赋值给了wpa_interface中的
变量
            iface->confname = optarg;
            break;
```

```
.....
case 'D':
    // 指定driver名称。注意，该参数赋值给了wpa_interface中的
变量
    iface->driver = optarg;
    break;
.....
case 'e':
    // 指定初始随机数文件，用于后续随机数的生成
```

①

```
        params.entropy_file = optarg;           break;
.....
case 'i':
    iface->ifname = optarg; // 指定网络设备接口名，本例
是"wlan0"
    break;
.....
}
}

exitcode = 0;
// 关键函数①：根据传入的参数，创建并初始化一个wpa_global对象
global = wpa_supplicant_init(&params);
.....
for (i = 0; exitcode == 0 && i < iface_count; i++) {
.....
// 关键函数②：WPAS支持操作多个无线网络设备，此处需将它们一一添
加到WPAS中
    // WPAS内部将初始化这些设备
    if (wpa_supplicant_add_iface(global, &ifaces[i]) ==
NULL)
        exitcode = -1;
}
// Android平台中，wpa_supplicant通过select或epoll方式实现多路I/O
复用。相关解释见下文
```

```

if (exitcode == 0)
    exitcode = wpa_supplicant_run(global);

wpa_supplicant_deinit(global);
.....// 退出
return exitcode;
}

```

main函数中出现了几个重要的数据结构和两个关键函数。

注意 虽然WPAS代码遵循C语法，但笔者也将称结构体实例称为对象。

先来认识这几个重要数据结构，如图4-7所示。



图4-7 main函数中重要的数据结构

图4-7中：

- wpa\_interface用于描述一个无线网络设备。该参数在初始化时用到。
- wpa\_global是一个全局性质的上下文信息。它通过ifaces变量指向一个wpa\_supplicant对象（以后介绍wpa\_supplicant时，读者将发现系统内的所有wpa\_supplicant对象将通过单向链表连接在一起。所以，严格意义上来说，ifaces变量指向一个wpa\_supplicant对象链表）。drv\_priv包含driver wrapper所需要的全局上下文信息。其drv\_count代表当前编译到系统中的driver wrapper个数（详情见下文）。另外，wpa\_global有一个全局控制接口，如果设置该接口，其他wpa\_interface设置的控制接口将被替代。

- wpa\_supplicant是WPAS的核心数据结构。一个interface对应有一个wpa\_supplicant对象，其内部包含非常多的成员变量（图4-7并未画出，下文详细介绍）。另外，系统中所有wpa\_supplicant对象都通过next变量链接在一起。
- ctrl\_iface\_global\_priv是全局控制接口的信息，内部包含一个用于通信的socket句柄。

**提示** 由于篇幅原因，笔者将根据情况略去数据结构中部分成员变量的介绍。

下面分析关键函数wpa\_supplicant\_init。

**①** 读者可阅读《深入理解Android：卷II》3.3节以了解Android平台中更多和随机数有关的知识。

### 4.3.2 wpa\_supplicant\_init函数分析

wpa\_supplicant\_init代码如下所示。

```
[-->wpa_supplicant.c: : wpa_supplicant_init]

struct wpa_global * wpa_supplicant_init(struct wpa_params
*params)
{
    struct wpa_global *global;
    int ret, i;
    .....
#ifdef CONFIG_DRIVER_NDIS
    .....// windows driver支持
#endif
#ifndef CONFIG_NO_WPA_MSG
    // 设置全局回调函数，详情见下文解释
    wpa_msg_register_ifname_cb(wpa_supplicant_msg_ifname_cb);
#endif /* CONFIG_NO_WPA_MSG */
    // 输出日志文件设置，本例未设置该文件
    wpa_debug_open_file(params->wpa_debug_file_path);
    .....
    ret = eap_register_methods(); // ①注册EAP方法
    .....
    global = os_zalloc(sizeof(*global)); // 创建一个wpa_global对
象
    ..... // 初始化global中的其他参数
    wpa_printf(MSG_DEBUG, "wpa_supplicant v" VERSION_STR);
    // ②初始化事件循环机制
    if (eloop_init()) {.....}
    // 初始化随机数相关资源，用于提升后续随机数生成的随机性
    // 这部分内容不是本书的重点，感兴趣的读者请自行研究
    random_init(params->entropy_file);

    // 初始化全局控制接口对象。由于本例中未设置全局控制接口，故该函数的处
理非常简单，请读者自行阅读该函数
    global->ctrl_iface =
    wpa_supplicant_global_ctrl_iface_init(global);
    .....
    // 初始化通知机制相关资源，它和dbus有关。本例没有包括dbus相关内容，略
    if (wpas_notify_supplicant_initialized(global)) {.....}
    // ③wpa_driver是一个全局变量，其作用见下文解释
    for (i = 0; wpa_drivers[i]; i++)
```

```

    global->drv_count++;
    .....
    // 分配全局driver wrapper上下文信息数组
    global->drv_priv = os_zalloc(global->drv_count *
sizeof(void *));
    .....
    return global;
}

```

wpa\_supplicant\_init函数的主要功能是初始化wpa\_global以及一些与整个程序相关的资源，包括随机数资源、eloop事件循环机制以及设置消息全局回调函数。

此处先简单介绍消息全局回调函数，一共有两个。

- wpa\_msg\_get\_ifname\_func：有些输出信息中需要打印出网卡接口名。该回调函数用于获取网卡接口名。
- wpa\_msg\_cb\_func：除了打印输出信息外，还可通过该回调函数进行一些特殊处理，如把输出信息发送给客户端进行处理。

上述两个回调函数相关的代码如下所示。

[-->wpa\_debug.c]

```

// wpa_msg_ifname_cb用于获取无线网卡接口名
// WPAS为其设置的实现函数为wpa_supplicant_msg_ifname_cb
// 读者可自行阅读此函数
static wpa_msg_get_ifname_func wpa_msg_ifname_cb = NULL;
void wpa_msg_register_ifname_cb(wpa_msg_get_ifname_func func) {
    wpa_msg_ifname_cb = func;
}
// WPAS中，wpa_msg_cb的实现函数是
wpa_supplicant_ctrl_iface_msg_cb，它将输出信息发送给客户端
// 图4-2最后两行的信息就是由此函数发送给客户端的。而且前面的"<3>"也是由它
添加的
static wpa_msg_cb_func wpa_msg_cb = NULL;
void wpa_msg_register_cb(wpa_msg_cb_func func) {
    wpa_msg_cb = func;
}

```

现在来看wpa\_supplicant\_init中列出的三个关键点，首先是eap\_register\_method函数。

### 1. eap\_register\_methods函数

该函数本身非常简单，它主要根据编译时的配置项来初始化不同的eap方法。其代码如下所示。

```
[-->eap_register.c: : eap_register_methods]

int eap_register_methods(void)
{
    int ret = 0;
#ifdef EAP_MD5                                // 作为supplicant端, 编译时将定义
EAP_MD5
    if (ret == 0)
        ret = eap_peer_md5_register();
#endif /* EAP_MD5 */
    .....
#ifdef EAP_SERVER_MD5                          // 作为Authenticator端, 编译时将定
义EAP_SERVER_MD5
    if (ret == 0)
        ret = eap_server_md5_register();
#endif /* EAP_SERVER_MD5 */
    .....
    return ret;
}
```

如上述代码所示，eap\_register\_methods函数将根据编译配置项来注册所需的eap method。例如，MD5身份验证方法对应的注册函数是eap\_peer\_md5\_register，该函数内部将填充一个名为eap\_method的数据结构，其定义如图4-8所示。

图4-8所示的struct eap\_method结构体声明于eap\_i.h中，其内部一些变量及函数指针的定义和RFC4137有较大关系。此处，我们暂时列出其中一些简单的成员变量。4.4节将详细介绍RFC4137相关的知识。

来看第二个关键函数eloop\_init，它和图4-1所示WPAS软件架构中的event loop模块有关。

### 2. eloop\_init函数及event loop模块

eloop\_init函数本身特别简单，它仅初始化了WPAS中事件驱动的核心数据结构体eloop\_data。WPAS事件驱动机制的实现非常简单，它就是利用epoll（如果编译时设置了CONFIG\_ELOOP\_POLL选项）或select实现了I/O复用。

提醒 select（或epoll）是I/O复用的重要函数，属于基础知识范畴。请不熟悉的读者自行学习相关内容。

从事件角度来看，WPAS的事件驱动机制支持5种类型的event。

- read event：读事件，例如来自socket的可读事件。
- write event：写事件，例如socket的可写事件。
- exception event：异常事件，如果socket操作发生错误，则由错误事件处理。
- timeout event：定时事件，通过select的等待超时机制来实现定时事件。
- signal：信号事件，信号事件来源于Kernel。WPAS允许为一些特定信号设置处理函数。

以上这些事件相关的信息都保存在eloop\_data结构体中，如图4-9所示。

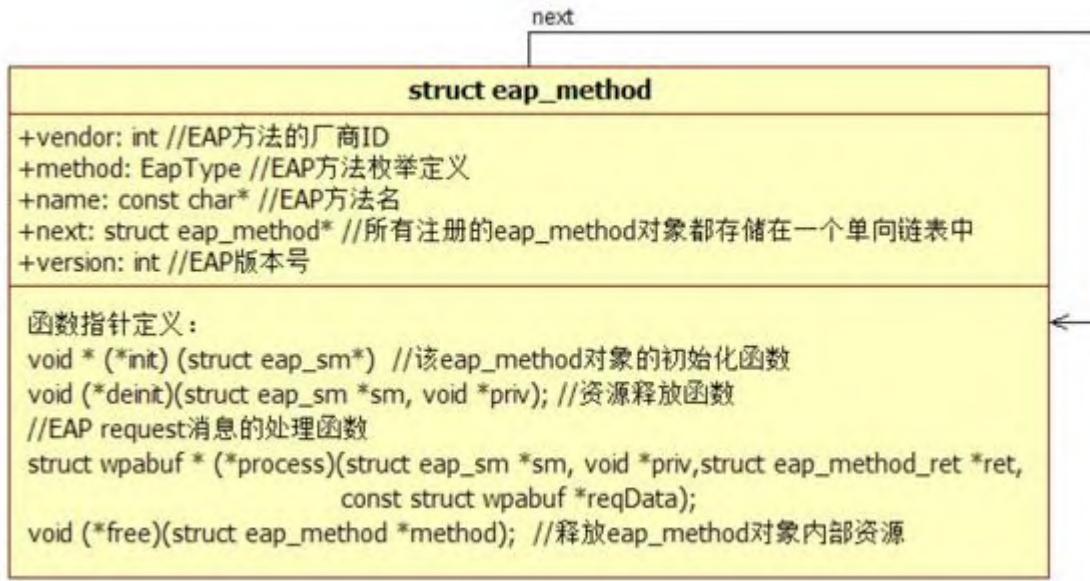


图4-8 eap\_method数据结构

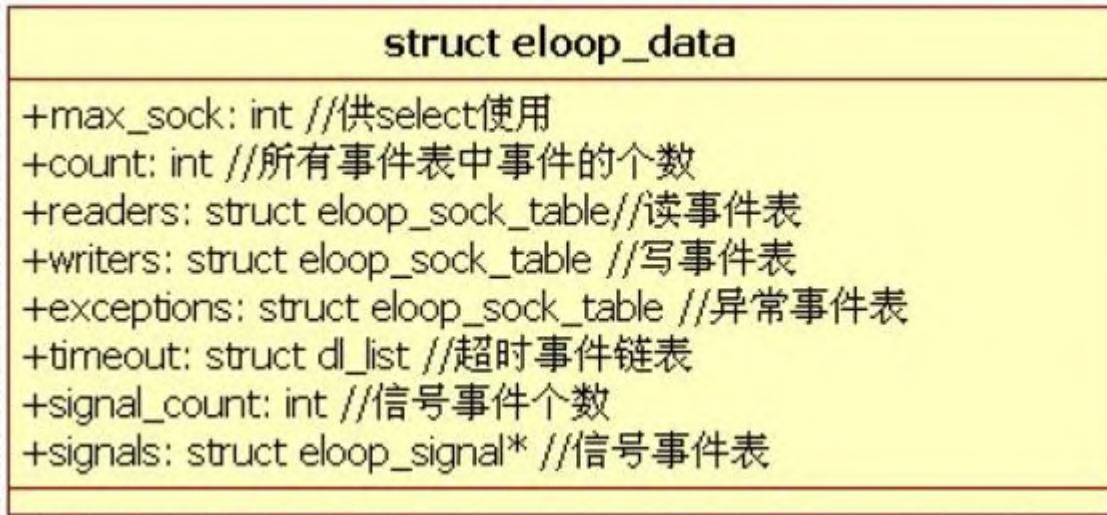


图4-9 eloop\_data结构体

简单介绍一下eloop提供的事件注册API及eloop事件循环核心处理函数eloop\_run。首先是事件注册API函数，相关代码如下所示。

[-->eloop.h]

```
// 注册socket读事件处理函数，参数sock代表一个socket句柄。一旦该句柄上有
读事件发生，则handler函数
```

```

// 将被事件处理循环（见下文eloop_run函数）调用
int eloop_register_read_sock(int sock, eloop_sock_handler
handler,
                             void *eloop_data, void *user_data);
// 注册socket事件处理函数，具体是哪种事件（只能是读、写或异常）由type参数
决定
int eloop_register_sock(int sock, eloop_event_type type,
                        eloop_sock_handler handler, void *eloop_data, void
*user_data);
// 注册超时事件处理函数
int eloop_register_timeout(unsigned int secs, unsigned int
usecs,
                           eloop_timeout_handler handler, void *eloop_data,
                           void *user_data);
// 注册信号事件处理函数，具体要处理的信号由sig参数指定
int eloop_register_signal(int sig, eloop_signal_handler
handler, void *user_data);

```

最后，向读者展示一下WPAS事件驱动机制的运行原理，其代码在eloop\_run函数中，如下所示。

[-->eloop.c : : eloop\_run]

```

void eloop_run(void)
{
    fd_set *rfds, *wfds, *efds; // fd_set是select中用到的一种参数类
型
    struct timeval _tv;
    int res;
    struct os_time tv, now;
    // 事件驱动循环
    while (!eloop.terminate &&
           (!dl_list_empty(&eloop.timeout) ||
eloop.readers.count > 0 ||
eloop.writers.count > 0 || eloop.exceptions.count > 0))
    {
        struct eloop_timeout *timeout;
        // 判断是否有超时事件需要等待
        timeout = dl_list_first(&eloop.timeout, struct
eloop_timeout,list);
        if (timeout) {
            os_get_time(&now);
            if (os_time_before(&now, &timeout->time))
                os_time_sub(&timeout->time, &now, &tv);
            else

```

```

        tv.sec = tv.usec = 0;
        _tv.tv_sec = tv.sec;
        _tv.tv_usec = tv.usec;
    }
    // 将外界设置的读事件添加到对应的fd_set中
    eloop_sock_table_set_fds(&eloop.readers, rfds);
    .....// 设置写、异常事件到fd_set中
    // 调用select函数
    res = select(eloop.max_sock + 1, rfds, wfds,
    efds, timeout ? &_tv : NULL);
    if(res < 0) {.....// 错误处理}
    // 先处理信号事件
    eloop_process_pending_signals();
    // 判断是否有超时事件发生
    timeout = dl_list_first(&eloop.timeout, struct
eloop_timeout,list);
    if (timeout) {
        os_get_time(&now);
        if (!os_time_before(&now, &timeout->time)) {
            void *eloop_data = timeout->eloop_data;
            void *user_data = timeout->user_data;
            eloop_timeout_handler handler = timeout-
>handler;
            eloop_remove_timeout(timeout);           // 注意
意，超时事件只执行一次
            handler(eloop_data, user_data);         // 处理
超时事件
        }
    }
    .....// 处理读/写/异常事件。方法和下面这个函数类似
    eloop_sock_table_dispatch(&eloop.readers, rfds);
    .....// 处理wfds和efds
}
out:
    return;
}

```

eloop\_run中的while循环是WPAS进程的运行中枢。不过其难度也不大。

下面来看wpa\_supplicant\_init代码中的第三个关键点，即wpa\_drivers变量。

### 3. wpa\_drivers数组和driver i/f模块

wpa\_drivers是一个全局数组变量，它通过extern方式声明于main.c中，其定义却在drivers.c中，如下所示。

[-->drivers.c: : wpa\_drivers定义]

```
struct wpa_driver_ops *wpa_drivers[] =  
{  
    #ifdef CONFIG_DRIVER_WEXT  
        &wpa_driver_wext_ops,  
    #endif /* CONFIG_DRIVER_WEXT */  
    #ifdef CONFIG_DRIVER_NL80211  
        &wpa_driver_nl80211_ops,  
    #endif /* CONFIG_DRIVER_NL80211 */  
    .....// 其他driver接口  
}
```

wpa\_drivers数组成员指向一个wpa\_driver\_ops类型的对象。

wpa\_driver\_ops是driver i/f模块的核心数据结构，其内部定义了很多函数指针。而正是通过定义函数指针的方法，WPAS能够隔离上层使用者和具体的driver。

注意 此处的driver并非通常意义所指的那些运行于Kernel层的驱动。读者可认为它们是Kernel层wlan驱动在用户空间的代理模块。上层使用者通过它们来和Kernel层的驱动交互。为了避免混淆，本书后续将用driver wrapper一词来表示WPAS中的driver。而driver一词将专指Kernel里对应的wlan驱动。

另外，wpa\_drivers数组包含多少个driver wrapper对象也由编译选项来控制（如代码中所示的CONFIG\_DRIVER\_WEXT宏，它们可在android.cfg中被修改）。

此处先列出wpa\_driver\_nl80211\_ops的定义。

[-->driver\_nl80211.c: : wpa\_driver\_nl80211\_ops]

```
const struct wpa_driver_ops wpa_driver_nl80211_ops = {  
    .name = "nl80211", //  
    // driver wrapper的名称  
    .desc = "Linux nl80211/cfg80211", // 描述  
    // 信息  
    .get_bssid = wpa_driver_nl80211_get_bssid, // 用于
```

```

获取bssid
.....
.scan2 = wpa_driver_nl80211_scan, // 扫描
函数
.....
.get_scan_results2 = wpa_driver_nl80211_get_scan_results,
// 获取扫描结果
.....
.disassociate = wpa_driver_nl80211_disassociate, // 触发
disassociation操作
.authenticate = wpa_driver_nl80211_authenticate, // 触发
authentication操作
.associate = wpa_driver_nl80211_associate, // 触发
association操作
// driver wrapper全局初始化函数，该函数的返回值保存在wpa_global成
员变量drv_pri数组中
.global_init = nl80211_global_init,
.....
.init2 = wpa_driver_nl80211_init, // 
driver wrapper初始化函数
.....
#endif ANDROID // Android平台定义了该宏
.driver_cmd = wpa_driver_nl80211_driver_cmd, // 该函数用于处理
和具体驱动相关的命令
#endif
};

```

本节介绍了main函数中第一个的关键点wpa\_supplicant\_init，其中涉及的知识有：几个重要数据结构，如wpa\_global、wpa\_interface、eap\_method、wpa\_driver\_ops等；event loop的工作原理；消息全局回调函数和wpa\_drivers等内容。

下面来分析main中第二个关键函数wpa\_supplicant\_add\_iface。

### 4.3.3 wpa\_supplicant\_add\_iface函数分析

wpa\_supplicant\_add\_iface用于向WPAS添加接口设备。所谓的添加(add iface)，其实就是初始化这些设备。该函数代码如下所示。

```
[-->wpa_supplicant.c: : wpa_supplicant_add_iface]
struct wpa_supplicant * wpa_supplicant_add_iface(struct
wpa_global *global,
                                                struct wpa_interface *iface)
{
    struct wpa_supplicant *wpa_s;
    struct wpa_interface t_iface;
    struct wpa_ssid *ssid;
    .....
    wpa_s = wpa_supplicant_alloc();
    .....
    wpa_s->global = global;
    t_iface = *iface;
    .....// 其他一些处理。本例未涉及它们
    // wpa_supplicant_init_iface为重要函数，下面单独用一节来分析它
    if (wpa_supplicant_init_iface(wpa_s, &t_iface)) {.....}
    .....
    // 通过dbus通知外界有新的iface加入。本例并未使用DBUS
    if (wpas_notify_iface_added(wpa_s)) {.....}
    // 通过dbus通知外界有新的无线网络加入
    for (ssid = wpa_s->conf->ssid; ssid; ssid = ssid->next)
        wpas_notify_network_added(wpa_s, ssid);

/*
    还记得图4-7中wpa_global数据结构吗？wpa_global的ifaces变量指向一个
    wpa_supplicant对象，而wpa_supplicant又通过next变量将自己链接到
    一个单向链表中。
*/
    wpa_s->next = global->ifaces;
    global->ifaces = wpa_s;
    return wpa_s;
}
```

wpa\_supplicant\_add\_iface的内容非常丰富，包括两个重要数据结构(wpa\_supplicant和wpa\_ssid)以及一个关键函数

wpa\_supplicant\_init\_iface。由于这些数据结构涉及较多背景知识，故本节先来介绍它们。

提示 wpa\_supplicant\_init\_iface内容也比较丰富，本章将在4.3.4节中单独介绍。

### 1. wpa\_ssid结构体

wpa\_ssid用于存储某个无线网络的配置信息（如所支持的安全类型、优先级等）。它其实是图4-6所示wpa\_supplicant.conf中无线网络配置项在代码中的反映（conf文件中每一个network项都对应一个wpa\_ssid对象）。它的一些主要数据成员如图4-10所示。



图4-10 wpa\_ssid数据结构

图4-10所示中的一些数据成员非常重要，下面分别介绍它们。

#### (1) 安全相关成员变量及背景知识

和安全相关的成员变量如下所示。

1) passphrase：该变量只和WPA/WPA2-PSK模式有关，用于存储我们输入的字符串密码。而实际上，规范要求使用的却是图4-10中的psk变量。结合3.3.7节中关于key和password的介绍可知，用户一般只设置字符串形式的password。而WPAS将根据它和ssid进行一定的计算以得到最终使用的PSK。参考资料[3]中有PSK计算方法。

2) pairwise\_cipher和group\_cipher：这两个变量和规范中的cipher suite（加密套件）定义有关。cipher suite用于指明数据收发两方使用的数据加密方法。pairwise\_cipher和group\_cipher分别代表为该无线网络设置的单播和组播数据加密方法。标准说明请阅读参考资料[4]。WPAS中的定义如下。

```
// 位于defs.h中
#define WPA_CIPHER_NONE BIT(0) // 不保护。BIT(N)是一个宏，代表左移N位后的值
#define WPA_CIPHER_WEP40 BIT(1) // WEP40（即5个ASCII字符密码）
#define WPA_CIPHER_WEP104 BIT(2) // WEP104（即13个ASCII字符密码）
#define WPA_CIPHER_TKIP BIT(3) // TKIP
#define WPA_CIPHER_CCMP BIT(4) // CCMP
// 系统还定义了两个宏用于表示默认支持的加密套件类型：（位于config_ssid.h中）
#define DEFAULT_PAIRWISE (WPA_CIPHER_CCMP | WPA_CIPHER_TKIP)
#define DEFAULT_GROUP (WPA_CIPHER_CCMP | WPA_CIPHER_TKIP | \
                    WPA_CIPHER_WEP104 | WPA_CIPHER_WEP40)
```

3) key\_mgmt：该成员和802.11中的AKM suite相关。

AKM（Authentication and Key Management，身份验证和密钥管理）suite定义了一套算法用于在Supplicant和Authenticator之间交换身份和密匙信息。标准说明见参考资料[5]，WPAS中定义的key\_mgmt可取值如下。

```
// 位于defs.h中
#define WPA_KEY_MGMT_IEEE8021X BIT(0) // 不同的AKM suite有对应的流程与算法。不详细介绍
#define WPA_KEY_MGMT_PSK BIT(1)
#define WPA_KEY_MGMT_NONE BIT(2)
#define WPA_KEY_MGMT_IEEE8021X_NO_WPA BIT(3)
#define WPA_KEY_MGMT_WPA_NONE BIT(4)
#define WPA_KEY_MGMT_FT_IEEE8021X BIT(5) // FT (Fast
```

Transition) 用于ESS中快速切换BSS

```
#define WPA_KEY_MGMT_FT_PSK BIT(6)
#define WPA_KEY_MGMT_IEEE8021X_SHA256 BIT(7) // SHA256表示key派
生时使用SHA256做算法
#define WPA_KEY_MGMT_PSK_SHA256 BIT(8)
#define WPA_KEY_MGMT_WPS BIT(9)
// 位于config_ssid.h中
#define DEFAULT_KEY_MGMT (WPA_KEY_MGMT_PSK |
WPA_KEY_MGMT_IEEE8021X)
// 默认的AKM suite
```

4) proto : 代表该无线网络支持的安全协议类型。其可取值如下。

```
// 位于defs.h中
#define WPA_PROTO_WPA BIT(0)
#define WPA_PROTO_RSN BIT(1) // RSN
其实就是WPA2
// 位于config_ssid.h中
#define DEFAULT_PROTO (WPA_PROTO_WPA | WPA_PROTO_RSN) // 默认
支持两种协议
```

5) auth\_alg : 表示该无线网络所支持的身份验证算法，其可取值如下。

```
// 位于defs.h中
#define WPA_AUTH_ALG_OPEN BIT(0) // Open System, 如果要使
用WPA或RSN, 必须选择它
#define WPA_AUTH_ALG_SHARED BIT(1) // Shared Key算法
#define WPA_AUTH_ALG_LEAP BIT(2) // LEAP算法, LEAP是思科公
司提出的身份验证方法
#define WPA_AUTH_ALG_FT BIT(3) // 和FT有关, 此处不详细介绍, 读者可阅读参考资料[6]
```

6) eapol\_flags : 和动态WEP Key有关（本书不讨论，读者可阅读参考资料[7]），其取值包括如下。

```
// 位于config_ssid.h中
#define EAPOL_FLAG_REQUIRE_KEY_UNICAST BIT(0)
#define EAPOL_FLAG_REQUIRE_KEY_BROADCAST BIT(1)
```

上述变量的取值将影响wpa\_supplicant的处理逻辑。本章后续代码分析将见识到它们的实际作用。

## (2) 其他成员变量及背景知识

图4-10中其他三个重要成员变量介绍如下。

1) proactive\_key\_caching : 该变量和OPC (Opportunistic PMK Caching) <sup>①</sup> 技术有关。该技术虽还未正式被标准所接受，但很多无线设备厂商都支持它。其背景情况是，一组AP和一个中心控制器 (central controller) 共同组建一个所谓的mobility zone (移动区域)。zone中的所有AP都连接到此控制器上。当STA通过zone中的某一个AP (假设是AP\_0) 加入到无线网络后，STA和AP0完成802.1X身份验证时所创建的PMKSA (假设是PMKSA\_0) 将由controller发送到zone中的其他AP。其他AP将根据此PMKSA\_0来生成PMKSA\_i。当STA切换到zone中的AP\_i时，它将根据PMKSA\_0计算PMKID\_i (不熟悉的读者请阅读3.3.7节RSNA介绍)，并试图和AP\_i重新关联 (Reassociation)。如果此AP\_i属于同一个zone，因为之前它已经由controller发送的PMKSA\_0计算出了PMKSA\_i，所以STA可避过802.1X认证流程而直接进入后续的 (如4-Way Handshake) 处理流程。802.1X验证的目的就是得到PMKSA，所以，如果AP\_i已经有PMKSA\_i，就无须费时费力开展802.1X认证工作了。proactive\_key\_caching默认值为0，即不支持此功能。另外，OPC功能需要AP支持。关于OPC的信息请阅读参考资料[8]和[9]。

2) disable : 该变量取值为0 (代表该无线网络可用)、1 (代表该无线网络被禁止使用，但可通过命令来启用它)、2 (表示该无线网络和P2P有关)。

3) mode : wpa\_ssid结构体内部还定义了一个枚举型变量，其可取值如图4-10底部所示。此处要特别指出的是，基础结构型网络中，如果STA和某个AP成功连接的话，STA也称为Managed STA (对应枚举值为WPAS\_MODE\_INFRA)。

### 2. wpa\_supplicant结构体

wpa\_supplicant结构体定义的成员变量非常多，图4-11列出了其中一部分内容。

```

next

struct wpa_supplicant

+global: struct wpa_global* //指向wpa_global对象
+next: struct wpa_supplicant* //进程中所有wpa_supplicant对象都保存在一个单向链表中
+l2: struct l2_packet_data* //用于处理EAP和EAPOL消息，L2是Link Layer的简写
+own_acdr: unsigned char数组 //元素个数为ETH_ALEN, 即16个
+ifname: char数组 //元素个数为100
+confname: char* //运行时配置文件名, 本例是/data/misc/wifi/wpa_supplicant.conf
+conf: struct wpa_conf* //解析运行时配置文件后得到的配置信息
+countermeasures: int //见下文解释
+bssid: u8数组 //元素各位为ETH_ALEN, 表示此supplicant连接到的无线网络的BSSID
+pending_bssid: u8数组 //当supplicant还处于关联过程中时, 该变量保存目标无线网络的BSSID,
+reassociate: int //是否重新关联
+disconnected: int //此supplicant是否被禁止连接无线网络
+current_ssid: struct wpa_ssid* //当前使用的wpa_ssid对象
+current_bss: struct wpa_bss* //当前使用的wpa_bss对象。见下文解释
+pairwise_cipher: int //此supplicant选择的单播数据加密类型, 类似变量还有其他几个。见下文解释
+drv_priv: void* //驱动对应的上下文信息
+global_drv_priv: void* //驱动对应的全局上下文信息, 见下文解释
+sched_scan_timeout: int //和scheduled(计划)扫描功能有关, 见下文解释
+bss: struct dl_list: //保存此supplicant搜索到的周围的无线网络 (由wpa_bss对象表示)
+drv: struct wpa_driver_ops* //此supplicant对应的驱动对象
+wpa: struct wpa_sm* //WPA状态机
+eapol: struct wapol_sm* //eapol状态机
+ctrl_interface: struct ctrl_iface_priv* //此supplicant对应的控制接口对象
+wpa_state: enum wpa_states: //supplicant当前的状态
+blacklist: struct wpa_blacklist* //黑名单。supplicant将不会连接黑名单中的无线网络
+bgscan: const struct bgscan_ops * //background扫描功能, 见下文解释
+gas: struct gas_query* //和GAS功能有关, 见下文解释

enum wpa_states {
    WPA_DISCONNECTED, WPA_INTERFACE_DISABLED, WPA_INACTIVE,
    WPA_SCANNING, WPA_AUTHENTICATING, WPA_ASSOCIATING,
    WPA_ASSOCIATED, WPA_4WAY_HANDSHAKE, WPA_GROUP_HANDSHAKE,
    WPA_COMPLETED
}; //wpa_supplicant的状态切换见下文解释

```

图4-11 wpa\_supplicant结构体

此处先解释几个比较简单的成员变量。

- `drv_priv` 和 `global_drv_priv` : WPAS 为 driver wrapper 一共定义了两个上下文信息。这是因为 driver i/f 接口定义了两个初始化函数 (以 n180211 driver 为例, 它们分别是 `global_init` 和 `init2`)。其中, `global_init` 返回值为 driver wrapper 全局上下文信息, 它将保存在 `wpa_global` 的 `drv_priv` 数组中 (见图 4-7)。每个 `wpa_supplicant` 都对应有一个 driver wrapper 对象, 故它也需要保存对应的全局上下文信息。`init2` 返回值则是 driver wrapper 上下文信息, 它保存在 `wpa_supplicant` 的 `drv_priv` 中。

- current\_bss : 该变量类型为wpa\_bss。wpa\_bss是无线网络在wpa\_supplicant中的代表。wpa\_bss中的成员主要描述了无线网络的bssid、ssid、频率(freq, 以MHz为单位)、Beacon心跳时间(以TU为单位)、capability信息(网络性能, 见3.3.5节定长字段介绍)、信号强度等。wpa\_bss的作用很重要, 不过其数据结构相对比较简单, 此处不介绍。以后用到它时再来介绍。

现在, 来看wpa\_supplicant结构体中其他更有“料”的成员变量。

### (1) 安全相关成员变量及背景知识

wpa\_supplicant也定义了一些和安全相关的成员变量。

- pairwise\_cipher、group\_cipher、key\_mgmt、wpa\_proto、mgmt\_group\_cipher: 这几个变量表示该wpa\_supplicant最终选择的安全策略。其中mgmt\_group\_cipher和IEEE 802.11w(定义了管理帧加密的规范)有关。为节约篇幅, 图4-11中仅列出pairwise\_cipher一个变量。
- countermeasures: 该变量名可译为“策略”, 和TKIP的MIC(Message Integrity Check, 消息完整性校验)有关。因为TKIP MIC所使用的Michael算法在某些情况下容易被攻破, 所以规范特别定义了TKIP MIC countermeasures用于处理这类事情。例如, 一旦检测到60秒内发生两次以上MIC错误, 则停止TKIP通信60秒。这部分内容请阅读参考资料[10]和[11]。

### (2) 功能相关成员变量及背景知识

wpa\_supplicant结构体中有一些成员变量和功能相关。

- sched\_scan\_timeout (还有一些相关变量未在图4-11中列出): 该变量和计划扫描(scheduled scan)功能有关。计划扫描即定时扫描, 需要Kernel(版本必须大于3.0)的Wi-Fi驱动支持。启用该功能时, 需要为驱动设置定时扫描的间隔(以毫秒为单位)。
- bgscan (还有其他相关成员变量未在图4-11中列出): 该变量和后台扫描及漫游(background scan and roaming)技术有关。当STA在ESS(假设该ESS由多个AP共同构成)中移动时, 有时候因为信号不好

(例如STA离之前所关联的AP距离过远等），它需要切换到另外一个距离更近（即信号更好）的AP。这个切换AP的工作就是所谓的漫游。为了增强切换AP时的无缝体验（扫描过程中，STA不能收发数据帧。从用户角度来看，相当于网络不能使用），STA可采用background scan（定时扫描一小段时间或者当网络空闲时才扫描，这样可减少对用户正常使用的干扰）技术来监视周围AP的信号强度等信息。一旦之前使用的AP信号强度低于某个阈值，STA则可快速切换到某个信号更强的AP。除了background scan外，还有一种on-roam scan也能提升AP切换时的无缝体验。关于background scan和roaming，请阅读参考资料[12]和[13]。

- gas：该变量是GAS（Generic Advertisement Service，通用广告服务）的小写，和802.11u协议有关。该协议规定了不同网络间互操作的标准，其制定的初衷是希望Wi-Fi网络能够像运营商的蜂窝网络一样，方便终端设备接入。例如，人们用智能手机可搜索到数十个、甚至上百个无线网络。在这种情况下如何选择正确的无线网络呢？

802.11u协议使用GAS和ANQP（Access Network Query Protocol，接入网络查询协议）来帮助设备自动选择合适的无线网络。其中，GAS是MLME SAP中的一种（见规范6.3.71节），它使得STA在通过认证前（prior to authentication）就可以向AP发送和接收ANQP数据包。STA则使用ANQP协议向AP查询无线网络运营商的信息，然后STA根据这些信息来判断自己可以加入哪一个运营商的无线网络（例如中国移动手机卡用户可以连接中国移动架设的无线网络）。802.11u现在还不是特别完善，详细信息可阅读参考资料[14]和[15]。

- CONFIG\_SME：该变量是一个编译宏，用于设置WPAS是否支持SME。我们在3.3.6节“802.11 MAC管理实体”中曾介绍过SME（Station Management Entity）。如果该功能支持，则driver wrapper可直接利用SME定义的SAP，而无须使用MLME的SAP了。Android平台中如果定义了CONFIG\_DRIVER\_NL80211宏，则CONFIG\_SME也将被定义（参考drivers.mk文件）。不过SME的功能是否起作用，还需要看driver是否支持。Galaxy Note 2 wlan driver不支持SME，故本书不讨论。

### （3）wpa\_states的取值

wpa\_states的取值如下。

- WPA\_DISCONNECTED：表示当前未连接到任何无线网络。

- WPA\_INTERFACE\_DISABLED : 代表当前此wpa\_supplicant所使用的网络设备被禁用。
- WPA\_INACTIVE : 代表当前此wpa\_supplicant没有可连接的无线网络。这种情况包括周围没有无线网络，以及有无线网络，但是因为没有配置信息（如没有设置密码等）而不能发起认证及关联请求的情况。
- WPA\_SCANNING、WPA\_AUTHENTICATING、WPA\_ASSOCIATING : 分别表示当前wpa\_supplicant正处于扫描无线网络、身份验证、关联过程中。
- WPA\_ASSOCIATED : 表明此wpa\_supplicant成功关联到某个AP。
- WPA\_4WAY\_HANDSHAKE : 表明此wpa\_supplicant处于四次握手处理过程中。当使用PSK（即WPA/WPA2-Personal）策略时，STA收到第一个EAPOL-Key数据包则进入此状态。当使用WPA/WPA2-Enterprise方法时，当STA完成和RAIDUS身份验证后则进入此状态。
- WPA\_GROUP\_HANDSHAKE : 表明STA处于组密钥握手协议处理过程中。当STA完成四次握手协议并收到组播密钥交换第一帧数据后即进入此状态（或者四次握手协议中携带了GTK信息，也会进入此状态。详情见4.5.5节EAPOL-Key交换流程分析）。
- WPA\_COMPLETED : 所有认证过程完成，wpa\_supplicant正式加入某个无线网络。

介绍完上述几个重要数据结构后，下面将分析  
wpa\_supplicant\_add\_iface中一个关键函数  
wpa\_supplicant\_init\_iface。

(1) 有些书上也叫Opportunistic Key Caching，简写为OKC。

#### 4.3.4 wpa\_supplicant\_init\_iface函数分析

wpa\_supplicant\_init\_iface内容非常多，我们将通过逐步展示代码段的方法，分五部分介绍。

[-->wpa\_supplicant.c: : wpa\_supplicant\_init\_iface代码段一]

```
static int wpa_supplicant_init_iface(struct wpa_supplicant
*wpa_s,
                                     struct wpa_interface
*iface)
{
    const char *ifname, *driver;
    struct wpa_driver_capa capa;
    if (iface->confname) {
        .....// CONFIG_BACKEND_FILE处理，此宏指明WPAS使用的配置项信息
来源于文件
        // Android定义了它
        wpa_s->conf = wpa_config_read(wpa_s->confname);
    }
    .....
```

由上述代码可知，init\_iface初始化的第一个工作是解析运行时配置文件。其中，wpa\_s->confname的值为"/data/misc/wifi/wpa\_supplicant.conf"，解析函数是wpa\_config\_read。

##### 1. wpa\_supplicant\_init\_iface分析之一

这个函数本身没有特别之处，仅是把配置文件中的信息转换成对应的数据结构。

[-->config\_file.c: : wpa\_config\_read]

```
struct wpa_config * wpa_config_read(const char *name)
{
    FILE *f;
    char buf[256], *pos;
    int errors = 0, line = 0;
    struct wpa_ssid *ssid, *tail = NULL, *head = NULL;
    struct wpa_config *config; // 配置文件在代码中对应的数据结构
```

```

int id = 0;

config = wpa_config_alloc_empty(NULL, NULL);
.....
f = fopen(name, "r");
.....
while (wpa_config_get_line(buf, sizeof(buf), f, &line,
&pos)) {
    if (os_strcmp(pos, "network={") == 0) {
        // 读取配置文件中的network项，并将其转化成一个wpa_ssid类型
        的对象
        ssid = wpa_config_read_network(f, &line, id++);
        .....
        // 根据图4-10所示，wpa_ssid通过next成员变量构成了一个单向
        链表
        if (head == NULL) { head = tail = ssid; }
        else { tail->next = ssid; tail = ssid; }
        // network项属于配置文件的一部分，故wpa_ssid对象也包含在
        wpa_config对象中
        if (wpa_config_add_prio_network(config, ssid))
    .....
    .....
    // CONFIG_NO_CONFIG_BLOBS，blob是配置文件中的一个
    字段，用于存储有些身
    // 份认证算法需要用的证书之类的信息。本例没有使用blob配
    置项
    // 解析其他项
} else if (wpa_config_process_global(config, pos, line)
< 0) { .....
}
fclose(f);
config->ssid = head;
.....
return config;
}

```

wpa\_config和wpa\_ssid这两个数据结构都是配置文件中的信息在代码中的反映。读者可查看wpa\_supplicant.conf配置模板文件来了解各个配置项的含义。

上述代码中，wpa\_config\_process\_global的实现有一些特别，它通过宏的方式来定义解析项及对应的解析函数。由于解析函数最终结果就是设置wpa\_config中对应项的值，故本章不讨论其细节，感兴趣的读者不妨自行阅读它们。

## 2. wpa\_supplicant\_init\_iface分析之二

wpa\_supplicant\_init\_iface函数代码段二如下所示。

[-->wpa\_supplicant.c: : wpa\_supplicant\_init\_iface代码段二]

```
.....// 接wpa_supplicant_init_iface代码段一
if (os_strlen(iface->ifname) >= sizeof(wpa_s->ifname))
{.....}
// 将wpa_interface中的ifname复制到wpa_supplicant的ifname变量中
os_strlcpy(wpa_s->ifname, iface->ifname, sizeof(wpa_s-
>ifname));
.....
// 下面这两个函数和EAPOL状态机相关，我们将在4.4节介绍
eapol_sm_notify_portEnabled(wpa_s->eapol, FALSE);
eapol_sm_notify_portValid(wpa_s->eapol, FALSE);

driver = iface->driver;
next_driver:
    if (wpa_supplicant_set_driver(wpa_s, driver) < 0)  return
-1;
```

wpa\_supplicant\_set\_driver将根据driver wrapper名（本例是”nl80211”）找到wpa\_driver数组中nl80211指定的driver wrapper对象wpa\_driver\_nl80211\_ops，然后调用其global\_init函数。直接来看global\_init函数的实现。

提示 global\_init函数将返回全局driver wrapper上下文信息，它保存在wpa\_global的drv\_priv数组中。

### (1) global\_init函数分析

global\_init是wpa\_driver\_ops结构体中的一个类型为函数指针的成员变量。nl80211对应的driver wrapper将其设置为nl80211\_global\_init，代码如下所示。

[-->driver\_nl80211.c: : nl80211\_global\_init]

```
static void * nl80211_global_init(void)
{
    struct nl80211_global *global;
```

```

struct netlink_config *cfg;

global = os_zalloc(sizeof(*global));
global->ioctl_sock = -1;
dl_list_init(&global->interfaces);
global->if_add_ifindex = -1;

cfg = os_zalloc(sizeof(*cfg));
.....
cfg->ctx = global;
/*

```

下面这三条语句用于创建netlink socket来接收来自内核的网卡状态变化事件（如UP、DORMANT、

REMOVED），然后通过eloop\_register\_read\_sock注册一个netlink\_recv函数用于处理接收

到的socket消息。

netlink\_recv函数内部将根据消息的类别来调用newlink\_cb和dellink\_cb以处理网卡状态变

化事件。这两个回调函数处理比较简单，读者可在阅读完本章后再自行研究它们。

```

*/
cfg->newlink_cb = wpa_driver_nl80211_event_rtm_newlink;
cfg->dellink_cb = wpa_driver_nl80211_event_rtm_dellink;
global->netlink = netlink_init(cfg);
// 将加入netlink中AF_NETLINK协议中的RTMGRP_LINK组播组
.....
// nl80211利用netlink机制和wlan driver交互
if (wpa_driver_nl80211_init_nl_global(global) < 0) .....//错误处理

global->ioctl_sock = socket(PF_INET, SOCK_DGRAM, 0);
.....
return global;
}

```

上面代码涉及一个比较重要的数据结构，即代表nl80211 driver wrapper全局上下文信息的nl80211\_global，其结构如图4-12所示。

struct nl80211_global
+interfaces: struct dl_list //该链表保存后文要介绍的wpa_driver_nl80211_data对象
+if_add_ifindex: int
+netlink: struct netlink_data* //见代码中解释
+nl_cb: struct nl_cb* //netlink回调对象
+nl: struct nl_handle* //nl_handle被定义成nl_socket。nl成员变量用于发送netlink event
+nl80211_id: int //nl80211模块的netlink family值
+ioctl_sock: int //用于ioctl函数的socket句柄
+nl_event: struct nl_handle* //用于接收netlink event

图4-12 nl80211\_global结构体

需要注意的是nl80211\_global包含两个nl\_handle对象。nl\_handle的真实类型就是libnl定义的nl\_socket。其中，nl用于发送netlink消息，nl\_event用于接收netlink消息。

这两个nl\_handle对象的初始化由wpa\_driver\_nl80211\_init\_nl\_global函数完成，马上来看它。

## (2) wpa\_driver\_nl80211\_init\_nl\_global函数分析

wpa\_driver\_nl80211\_init\_nl\_global是global\_init的核心函数，其代码如下所示。

```
[-->driver_nl80211.c: : wpa_driver_nl80211_init_nl_global]

static int wpa_driver_nl80211_init_nl_global(struct
nl80211_global *global)
{
    // 此函数利用了第3章中介绍的API
    int ret;
    // 创建一个netlink回调对象
    global->nl_cb = nl_cb_alloc(NL_CB_DEFAULT);
    /*
        nl_create_handle返回值的类型为nl_handle*, 而nl_handle在
        driver_nl80211.c中
        就是nl_socket (代码中的定义: #define nl_handle nl_sock)。
        nl_create_handle内部调用genl_connect连接到内核对应的模块。注意,
        该函数最后的字符串参数
        (如此处的"nl") 仅用于输出调试信息。
    */
}
```

```

global->nl = nl_create_handle(global->nl_cb, "nl");
/*
    向netenlink中的"nl"模块查询"nl80211"模块的编号。注意,
genl_ctrl_resolve函数本
    来由libnl2定义，但driver_nl80211.c通过
#define genl_ctrl_resolve android_genl_ctrl_resolve
    宏将其指向android_genl_ctrl_resolve。该函数内部通过发送查询消息
来获取"nl80211"
    模块的family值。请读者自行阅读android_genl_ctrl_resolve
函数。
*/

```

```

global->n180211_id = genl_ctrl_resolve(global->nl,
"nl80211");

```

```

.....
// 创建另外另一个nl_sock对象，其用途是接收netlink消息
global->n1_event = nl_create_handle (global->nl_cb,
"event");

```

```

.....
/*
```

下面这几个函数的作用如下。

`nl_get_multicast_id`: 先从n180211模块中获得对应的组播组编号，如"scan"、"mlme"以及

"regulatory"组播组的编号。

`nl_socket_add_membership`: 加入某个组播组。这样，当某个组播有消息发送时，`n1_event`就能收到了。

```

*/
ret = nl_get_multicast_id(global, "n180211", "scan");
ret = nl_socket_add_membership(global->n1_event, ret);
ret = nl_get_multicast_id(global, "n180211", "mlme");
ret = nl_socket_add_membership(global->n1_event, ret);
ret = nl_get_multicast_id(global, "n180211", "regulatory");
ret = nl_socket_add_membership(global->n1_event, ret);
```

```

nl_cb_set(global->n1_cb, NL_CB_SEQ_CHECK, NL_CB_CUSTOM,
          no_seq_check, NULL); // 设置序列号检查函数为

```

```

no_seq_check
nl_cb_set(global->n1_cb, NL_CB_VALID, NL_CB_CUSTOM,
          process_global_event, global); // 设置netlink消
息回调处理函数
```

```

/*
```

将`n1_event`对应的socket注册到eloop中，回调函数为

`wpa_driver_n180211_event_receive`,

该函数内部将调用`nl_recvmsg`，而`nl_recvmsg`又会调用`process_global_event`。所以，我们只

```

要关注process_global_event就可以了。
*/
    eloop_register_read_sock(nl_socket_get_fd(global-
>nl_event),
                           wpa_driver_nl80211_event_receive, global->nl_cb,
                           global->nl_event);
    return 0;
.....
}

```

wpa\_driver\_nl80211\_init\_nl\_global内容比较多，此处总结一下其工作内容。

- 创建了两个nl\_handle对象，分别是global->n1和gobal->event。nl\_handle内部定义一个socket句柄。所以，两个nl\_handle等同于两个socket句柄。global->event用于接收netlink消息。n180211定义了几个组播组，此处选择加入其中的“scan”、“mlme”和“regulatory”三个组播组，它们分别对应于扫描信息、mlme信息及管制信息。wlan driver内部会往这三个组播发送相关的消息。这样，global->event就能收到它们。
- 接着将global->event对应的socket注册到eloop读事件队列中。如此，内核发送的netlink消息就能被wpa\_driver\_nl80211\_event\_receive处理。wpa\_driver\_nl80211\_event\_receive内部将调用libnl API中的nl\_recv\_msg来接收消息，而它又会触发最重要的process\_global\_event函数被调用。
- global->n1用来向wlan driver发送netlink消息。根据第3章对genlmsg的介绍，其内部有一个变量用于指明family，而n180211对应的family编号则保存在global->n180211\_id中。

**提示** 根据笔者的心得，读者大可不必对libnl等进行深入细致的源码分析。对WPAS的来说，仅了解libnl2 API的用法即可。

### 3. wpa\_supplicant\_init\_iface分析之三

介绍完wpa\_supplicant\_set\_driver后，现在回到wpa\_supplicant\_init\_iface，继续看第三段代码。

[-->wpa\_supplicant.c: : wpa\_supplicant\_init\_iface代码段三]

```
.....// 接wpa_supplicant_set_driver代码段
// 又是一个关键函数
wpa_s->drv_priv = wpa_drv_init(wpa_s, wpa_s->ifname);
.....
// 设置driver参数，本例没有使用这一项功能
if (wpa_drv_set_param(wpa_s, wpa_s->conf->driver_param) <
0) {.....}
// 从driver中获取网卡名
ifname = wpa_drv_get_ifname(wpa_s);
if (ifname && os_strcmp(ifname, wpa_s->ifname) != 0) {
    // 如果不一致则替换配置文件中设置的网卡设备名
    os_strlcpy(wpa_s->ifname, ifname, sizeof(wpa_s->ifname));
}
```

上一节初始化driver wrapper的全局上下文信息后（通过调用global\_init来完成），接着要处理的就是单个driver wrapper了。该工作由wpa\_drv\_init函数完成。其内部将调用driver wrapper的init2函数（注意，如果driver wrapper定义了init2函数，init2将唯一被调用，否则将调用其定义的init函数）。

直接来看driver\_nl80211实现的init2函数，其代码如下所示。

[-->driver\_nl80211.c: : wpa\_driver\_nl80211\_init]

```
static void * wpa_driver_nl80211_init(void *ctx, const char
*ifname, void *global_priv)
{
    struct wpa_driver_nl80211_data *drv;
    struct rfkill_config *rcfg;  struct i802_bss *bss;
    .....
    drv = os_zalloc(sizeof(*drv));
    .....
    drv->global = global_priv;
    drv->ctx = ctx; // ctx的真正类型是wpa_supplicant
    bss = &drv->first_bss; bss->drv = drv;
    os_strlcpy(bss->ifname, ifname, sizeof(bss->ifname));
    drv->monitor_ifidx = -1;  drv->monitor_sock = -1;  drv-
>eapol_tx_sock = -1;
    // ap_scan_as_station变量和hostapd有关
    drv->ap_scan_as_station = NL80211_IFTYPE_UNSPECIFIED;
    // ①下面两个关键函数见后文解释
    if (wpa_driver_nl80211_init_nl(drv)) {.....}
```

```
    if (nl80211_init_bss(bss)) goto failed;
/*
```

下面这个函数将读取/sys/class/net/wlan0/phy80211/name文件的内容，并将其保存到

wpa\_driver\_nl80211\_data->phyname变量中。该文件存储了Wi-Fi物理设备的名称，如phy0等。

它由wifi wlan注册时动态生成，所以其值有可能变化。

注意，/sys/class/net/wlan0中的wlan0为无线网络设备名，它由wpa\_supplicant -i参数指明。

```
 */
nl80211_get_phy_name(drv);
rcfg = os_zalloc(sizeof(*rcfg));
rcfg->ctx = drv;
os_strlcpy(rcfg->ifname, fname, sizeof(rcfg->ifname));
// 和rfkill相关，见下文解释
rcfg->blocked_cb = wpa_driver_nl80211_rfkill_blocked;
rcfg->unblocked_cb = wpa_driver_nl80211_rfkill_unblocked;
drv->rfkill = rfkill_init(rcfg);
.....
// 关键函数②
if (wpa_driver_nl80211_finish_drv_init(drv)) goto failed;
// 见下文关于PF_PACKET的解释
drv->eapol_tx_sock = socket(PF_PACKET, SOCK_DGRAM, 0);

if (drv->data_tx_status) { .... }

if (drv->global) {
    // 把自己加到nl80211_global中的interfaces链表中去
    dl_list_add(&drv->global->interfaces, &drv->list);
    drv->in_interface_list = 1;
}
return bss; // wpa_driver_nl80211_init返回的是一个i802_bss结构体对象
.....
}
```

上述代码包含的知识点较多，涉及rfkill以及PF\_PACKET背景知识，以及三个关键函数，wpa\_driver\_nl80211\_init\_nl、nl80211\_init\_bss和wpa\_driver\_nl80211\_finish\_drv\_init。

### (1) rfkill背景知识

rfkill代表radio frequency (RF) connector kill switch support，它是Kernel中的一个子系统 (subsystem)。其功能是控制

系统中射频设备的电源（包括Wi-Fi、GPS、BlueTooth、FM等设备。注意，这些设备驱动只有把自己注册到rfkill子系统中后，rfkill才能对它们起作用）的工作以避免浪费电力。rfkill有软硬两种方式来禁止（block）RF设备。

- hard block：不能通过软件来重新启用RF设备。据观察，Android手机还没有hard block功能。不过笔者猜测某些笔记本有这个功能。例如，笔者的Dell笔记本上有一个特殊的开关，一旦把它关上，Wi-Fi模块就不能工作<sup>①</sup>。
- soft block：可以用软件来重新启用RF设备。

rfkill对用户空间提供了相应的控制接口，主要是通过/dev/rfkill设备文件来完成相关操作。我们通过wpa\_driver\_nl80211\_init中调用的一个名为rfkill\_init的函数来认识如何使用rfkill。该函数代码如下所示。

```
[-->rfkill.c: : rfkill_init]

struct rfkill_data * rfkill_init(struct rfkill_config *cfg)
{
    /*
        rfkill_data 是WPAS自定义的一个数据结构，主要用于设置两个回调函数用于处理block
        和unblock的情况。
        由上面一段代码可知，这两个回调函数分别是
        wpa_driver_nl80211_rfkill_blocked和
        wpa_driver_nl80211_rfkill_unblocked。
    */
    struct rfkill_data *rfkill;
    struct rfkill_event event;                                // rfkill_event
    代表rfkill事件
    ssize_t len;

    rfkill = os_zalloc(sizeof(*rfkill));
    rfkill->cfg = cfg;
    // O_RDONLY标志表示driver_nl80211只读取rfkill事件，而不会去操作
    rfkill模块
    rfkill->fd = open("/dev/rfkill", O_RDONLY);
    .....
    // 设置I/O操作为非阻塞式
    if (fcntl(rfkill->fd, F_SETFL, O_NONBLOCK) < 0) {.....}
```

```

for (;;) { // 读者知道为什么这里是一个for无限循环吗?
    // 读取/dev/rfkill中已有的事件信息。rfkill事件信息保存在
    rfkill_event结构体中
    len = read(rfkill->fd, &event, sizeof(event));
    if (len < 0) {
        if (errno == EAGAIN) break; // 无数据可读，则
        跳出循环
        break; // 其他错误也跳出循环
    }
    .....
/*
    rfkill_event的op变量代表rfkill事件的类型，目前可取值有
    RFKILL_OP_ADD（代表一
    个设备添加到了rfkill子系统）、RFKILL_OP_DEL等。
    rfkill_event的type变量代表该rfkill事件所对应设备的类型。目前可
    取值有RFKILL_
        TYPE_WLAN（无线网卡设备）、RFKILL_TYPE_BLUETOOTH（蓝牙设备）
    等。
*/
    if (event.op != RFKILL_OP_ADD || event.type !=
    RFKILL_TYPE_WLAN)
        continue;
    if (event.hard) { // 表示是否为hard block
        rfkill->blocked = 1;
    } else if (event.soft) { // 表示
    是否为soft block
        rfkill->blocked = 1;
    } // 如果hard和soft均未被设置，则表示该设备属于unblock状态，即
    设备允许被使用
}
// 为eloop注册一个读事件，一旦rfkill有新的事件到来，则eloop会触发
rfkill_receive函数被调用
eloop_register_read_sock(rfkill->fd, rfkill_receive,
rfkill, NULL);
return rfkill;
.....// 错误处理
}

```

从上述代码可知，WPAS只是监控rfkill设备以获取发生在其上的rfkill\_event，而它并不操作rfkill以关闭或启用无线设备。

**提示** 关于rfkill更多的信息请阅读参考资料[16][17]。

## (2) PF\_PACKET背景知识

PF\_PACKET有时也被称为AF\_PACKET，是socket域（domain）中的一种，用于直接在OSI/RM的数据链路层（Data Link Layer）上收发数据。所以，通过AF\_PACKET，用户空间可直接实现在物理层之上的协议，如EAP和EAPOL等。

提醒 由3.3.1节可知，DLL层还可细分为LLC子层和MAC子层。

下面将通过一些具体代码段来展示PF\_PACKET的使用。

### [AF\_PACKET用法示例]

```
/*
socket函数的第二个参数叫socket_type。AF_PACKET中可以使用SOCK_DGRAM
和SOCK_RAW，二者
略有区别，主要体现在如何处理物理层地址信息上。使用AF_PACKET时，需要为数据
包设置物理层地址，
它由结构体struct sockaddr_ll来表达。当socket_type设置为：
SOCK_RAW：用户接收到的数据包也将包含物理层地址，并且发送数据时，驱动将使
用用户指定的物理层
地址来填充数据包。
SOCK_DGRAM：它比SOCK_RAW要高级一点。用户接收的数据包将不包括物理层地址
信息，而用户发送时
指定的物理层地址也仅是一个参考，kernel会根据实际情况来填充一个更为合适的
物理层地址。
另外，程序可以通过bind函数指定接收某个网卡设备上的数据包。
*/
int fd = socket(PF_PACKET, SOCK_DGRAM, htons(ETH_P_EAPOL)); // 最后
一个参数代表EAPOL协议类型
struct sockaddr_ll ll; // sockaddr_ll
结构体代表地址信息
memset(&ll, 0, sizeof(ll));
ll.sll_family = PF_PACKET; // 该变量必须被设
置成AF_PACKET
ll.sll_ifindex = ifindex; // 网络
设备的索引号
ll.sll_protocol = htons(ETH_P_EAPOL);
bind(fd, (struct sockaddr *) &ll, sizeof(ll)); // 绑定到指定的网络
设备
.....// 其他处理
```

```

// 发送数据
struct sockaddr_ll ll2;// 目标地址
memset(&ll2, 0, sizeof(ll2));
ll.sll_family = AF_PACKET;
ll.sll_ifindex = ifindex
ll.sll_protocol = htons(ETH_P_EAPOL); // 帧类型，此处代表EAPOL帧
ll.sll_halen = ETH_ALEN; // 目标MAC地址长度
memcpy(ll.sll_addr, dst_addr, ETH_ALEN); // sll_addr用于表示目标物理层地址（即MAC地址）

// 发送EAPOL帧
ret = sendto(fd, buf, len, 0, (struct sockaddr *) &ll2, sizeof(ll2));
.....
// 接收数据
struct sockaddr_ll ll3;
socklen_t fromlen;
memset(&ll3, 0, sizeof(ll3));
fromlen = sizeof(ll3);
int res = recvfrom(fd, buf, sizeof(buf), 0, (struct sockaddr *) &ll3, &fromlen);

```

关于PF\_PACKET更为详细的信息，请读者通过man 7 packet查询Linux手册。接着来看wpa\_driver\_nl80211\_init中的三个重要函数，首先是wpa\_driver\_nl80211\_init\_nl和nl80211\_init\_bss。

### (3) wpa\_driver\_nl80211\_init\_nl与nl80211\_init\_bss函数分析

这两个函数都使用了libnl创建了回调对象，代码如下所示。

```
[-->driver_nl80211.c: : wpa_driver_nl80211_init_nl]

static int wpa_driver_nl80211_init_nl(struct wpa_driver_nl80211_data *drv)
{
    drv->nl_cb = nl_cb_alloc(NL_CB_DEFAULT);
    .....
    nl_cb_set(drv->nl_cb, NL_CB_SEQ_CHECK,
NL_CB_CUSTOM,no_seq_check, NULL);
    nl_cb_set(drv->nl_cb, NL_CB_VALID, NL_CB_CUSTOM,
process_drv_event, drv);
    return 0;
}
```

```

static int nl80211_init_bss(struct i802_bss *bss)
{
    bss->nl_cb = nl_cb_alloc(NL_CB_DEFAULT);
    .....
    nl_cb_set(bss->nl_cb, NL_CB_SEQ_CHECK,
NL_CB_CUSTOM,no_seq_check, NULL);
    nl_cb_set(bss->nl_cb, NL_CB_VALID, NL_CB_CUSTOM,
process_bss_event, bss);

    return 0;
}

```

不过，它们仅创建了nl\_cb对象，却并未创建nl\_handle（即没有创建nl\_socket）。没有和socket绑定，这些回调对象就不可能真正被用上。它们什么时候用呢？此处先提前介绍一下使用它们的代码。

[-->driver\_nl80211.c: : nl80211\_alloc\_mgmt\_handle]

```

static int nl80211_alloc_mgmt_handle(struct i802_bss *bss)
{
    struct wpa_driver_nl80211_data *drv = bss->drv;
    if (bss->nl_mgmt) {...../*重复注册*/return -1; }

    bss->nl_mgmt = nl_create_handle(drv->nl_cb, "mgmt"); // 注意该函数的第一个参数
    eloop_register_read_sock(nl_socket_get_fd(bss->nl_mgmt),
                           wpa_driver_nl80211_event_receive, bss->nl_cb,
bss->nl_mgmt);
    return 0;
}
static void wpa_driver_nl80211_event_receive(int sock, void
*eloop_ctx, void *handle)
{
    struct nl_cb *cb = eloop_ctx;
    nl_recvmsgs(handle, cb); // cb是
bss->nl_cb
}

```

注意，上述代码有一个非常奇怪的地方。bss->nl\_mgmt创建时使用了drv->nl\_cb对象，该回调对象由wpa\_driver\_nl80211\_init\_nl创建，其对应的回调函数是process\_drv\_event。nl\_create\_handle返回的实际上是一个nl\_socket对象，其内部有一个s\_cb变量指向nl\_create\_handle的第一个参数（本例中即是drv->nl\_cb）。注册到eloop模块中的wpa\_driver\_nl80211\_event\_receive函数，在处理回调

的时候却使用了bss->nl\_cb，该回调对象对应的是process\_bss\_event函数。

也就是说，上述函数一共使用了两个回调对象，一个是drv->nl\_cb，另外一个是bss->nl\_cb。什么时候调用drv->nl\_cb，什么时候调用bss->nl\_cb呢？

根据笔者对比Android中libnl2和libnl2官方代码的结果，nl\_recvmsgs将使用指定的nl\_cb对象进行回调（即它的第二个参数，本例中的bss->nl\_cb），而nl\_recvmsgs\_default将使用nl\_socket中s\_cb指定的回调对象（即本例中的drv->nl\_cb）。不过，Android的libnl2并没有nl\_recvmsgs\_default函数。所以，drv->nl\_cb实际上永远不会被用到。

注意 综合4.3.4节对wpa\_driver\_nl80211\_init\_nl\_global的分析，WPAS中实际上真正使用到的回调对象就是两个：一个是bss->nl\_cb，对应的回调函数是process\_bss\_event，另一个是global->nl\_cb，对应的回调函数是process\_global\_event。

另外，作为练习，请通过以下命令查看上一节提到的与drv->nl\_cb和bss->nl\_cb使用有关的信息。

```
git blame src/drivers/driver_nl80211.c | grep process_drv_event  
git show d6c9aab8
```

#### (4) wpa\_driver\_nl80211\_finish\_drv\_init函数分析

wpa\_driver\_nl80211\_init中的最后一个关键函数代码如下所示。

```
[-->driver_nl80211.c: : wpa_driver_nl80211_finish_drv_init]  
  
static int  
wpa_driver_nl80211_finish_drv_init(struct  
wpa_driver_nl80211_data *drv)  
{  
    struct i802_bss *bss = &drv->first_bss;  
    int send_rfkill_event = 0;  
  
    drv->ifindex = if_nametoindex(bss->ifname); // 获取网卡设备的索
```

引，属于netdevice编程范畴

```
    drv->first_bss.ifindex = drv->ifindex;
```

```
#ifndef HOSTAPD                                // hostapd是另外一个程序,  
本书不讨论
```

```
    if (drv->ifindex != drv->global->if_add_ifindex &&  
        /*
```

①设置接口类型为NL80211\_IFTYPE\_STATION，见下文解释。注意，这个函数内容非常丰富，

其中包含很多和P2P相关的信息。本章暂时不考虑它。另外，此函数内部会调用到上一节提到的

```
    nl80211_alloc_mgmt_handle.  
    */
```

```
    wpa_driver_nl80211_set_mode(bss,  
NL80211_IFTYPE_STATION) < 0) {.....}
```

```
/*
```

linux\_set\_iface\_flags通过ioctl方式启动ifname对应的网卡设备。  
该函数使用了netdevice API，请读者回顾表2-2。

其使用的参数为SIOCSIFFLAGS和IFF\_UP。

```
*/
```

```
    if (linux_set_iface_flags(drv->global->ioctl_sock, bss-  
>ifname, 1)) {
```

// 注意，如果linux\_set\_iface\_flags返回非0值（即启动设备失败）

// 要判断是不是rfkill禁止了该设备

```
    if (rfkill_is_blocked(drv->rfkill)) {
```

.....// 如果是因为rfkill原因导致设备被禁止，则需要通知  
wpa\_supplicant

drv->if\_disabled = 1;// 设置if\_disabled为1，表示该设备  
被rfkill禁止了

send\_rfkill\_event = 1; // 该值表示需要设置WPAS的  
状态

```
    } else {.....}
```

```
}
```

// ②设置Wi-Fi设备工作状态为, IF\_OPER\_DORMANT，见下文解释

```
    netlink_send_oper_ifla(drv->global->netlink, drv-  
>ifindex, 1, IF_OPER_DORMANT);
```

```
#endif /* HOSTAPD */
```

// ③获取Wi-Fi设备的capability，见下文解释

```
if (wpa_driver_nl80211_capa(drv)) return -1;
```

// 通过ioctl方式获取指定网卡的MAC地址，也属于netdeivce编程范  
畴，回顾表2-2

```
if (linux_get_ifhwaddr(drv->global->ioctl_sock, bss-  
>ifname, bss->addr)) return -1;
```

```

if (send_rfkill_event) {
    /*
     * 添加一个超时任务，超时时间为0秒。超时处理函数为
     * wpa_driver_nl80211_send_rfkill，该
     * 函数内部将设置wpa_states为WPA_INTERFACE_DISABLED。
     * 可参考4.3.3节了解WPA_INTERFACE_DISABLED状态。
     */
    eloop_register_timeout(0, 0,
wpa_driver_nl80211_send_rfkill, drv, drv->ctx);
}
return 0;
}

```

wpa\_driver\_nl80211\_finish\_drv\_init代码不长，但内容却比较丰富，先简单总结一下其工作流程。

- 1) 调用wpa\_driver\_nl80211\_set\_mode函数设置Wi-Fi设备类型为NL80211\_IFTYPE\_STATION。下文将详细介绍Wi-Fi设备类型的知识。
- 2) 调用linux\_set\_iface\_flags通过netdevice API启用该Wi-Fi设备。如果失败，则需要判断该设备是否被rfkill block。
- 3) 调用netlink\_send\_oper\_ifla函数设置网卡的工作状态（Interface Operational Status, IfOperStatus）为IF\_OPER\_DORMANT。关于IfOperStatus详情见下文解释。
- 4) 调用wpa\_driver\_nl80211\_capa获取Wi-Fi设备的处理能力(capability)。详情见下文解释。
- 5) 最后，调用linux\_get\_ifhwaddr获取Wi-Fi设备的MAC地址，并判断是否需要设置超时函数wpa\_driver\_nl80211\_send\_rfkill。

上述内容中有两个背景知识（设备类型以及工作状态）和一个重要函数wpa\_driver\_nl80211\_capa。此处先来认识设备类型。

一般而言，一块网络接口设备只有一个MAC地址，但现在许多设备都支持多个所谓的虚拟设备（Virtual Interface），每一个虚拟设备都对应有一个虚拟MAC地址。例如，图4-13所示为Wi-Fi P2P中一种名为并发设备（Concurrent Device）的示意图。

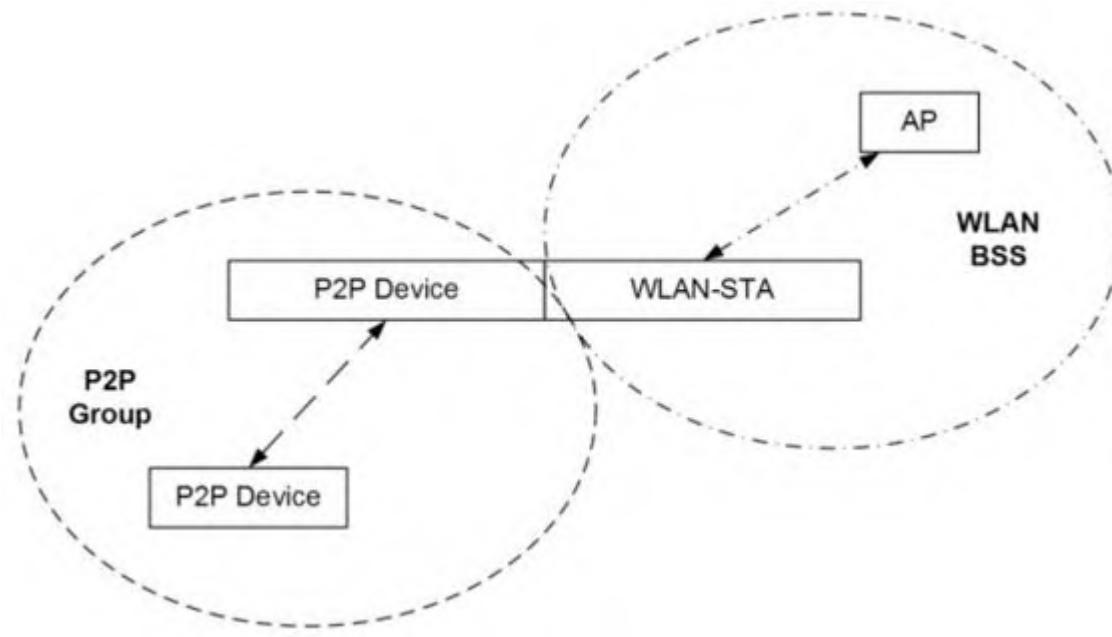


图4-13 P2P Concurrent设备

图4-13中，位于中间的Wi-Fi设备一方面以P2P Device的身份和左下角的另一个P2P Device相连，另一方面又以STA的身份和右边的AP相连。P2P Device和STA的工作方式不尽相同，怎么实现这种并发设备呢？解决方法就是通过这种虚拟设备的方式，使P2P Device和STA分别使用不同的Virtual Interface和Virtual MAC。

**提示** 目前，市面上许多Android手机打开Wi-Fi P2P功能后就必须关闭STA功能，而笔者的Note 2就能做到P2P和STA同时工作。

NL80211定义了多种Virtual Interface类型，如下所示。

```
enum nl80211_iftype {
    NL80211_IFTYPE_UNSPECIFIED,
    NL80211_IFTYPE_ADHOC,                                // IBSS类型
    NL80211_IFTYPE_STATION,                             // 基础结构型网络中的STA
    NL80211_IFTYPE_AP,                                  // 基础结构型网络中的AP
    NL80211_IFTYPE_AP_VLAN,                            // 和VLAN有关，本书不讨论
    NL80211_IFTYPE_WDS,                                // 无线桥接。本书不讨论
    NL80211_IFTYPE_MONITOR,                           // 可接收无线网络所有的数据包，它提供
    // 类似AirPcap这样的功能
    NL80211_IFTYPE_MESH_POINT,                         // Mesh网络节点，本书不讨论
    NL80211_IFTYPE_P2P_CLIENT,                         // P2P相关，见后续章节
    NL80211_IFTYPE_P2P_GO,                            // P2P相关，见后续章节
}
```

```
    NUM_NL80211_IFTYPES,
    NL80211_IFTYPE_MAX = NUM_NL80211_IFTYPES - 1
};
```

下面来看接口设备的工作状态IfOperStatus，其定义来自RFC2863。

[-->if.h]

```
enum {
    IF_OPER_UNKNOWN,           // 未知状态
    // 下面这个状态表示因为系统缺乏该接口设备所依赖的模块（一般是硬件模
    // 块）而导致接口设备不能工作
    IF_OPER_NOTPRESENT, // 
    IF_OPER_DOWN,           // 接口不能工作
    /* 相比DOWN状态， LOWERLAYERDOWN指出了接口设备不能工作的原因是其所依赖
     * 的更低一层的设备不
     * 能正常工作。
     */
    IF_OPER_LOWERLAYERDOWN,
    IF_OPER_TESTING,          // 接口处于测试状态中
    /* 接口处于休眠或暂停（pending）状态中。这种状态表示接口设备在等待某个
     * 事情的发生（例如上层
     * 有数据要发送，则会将DORMANT状态设置为UP状态）。
     */
    IF_OPER_DORMANT,
    IF_OPER_UP,               // 接口可工作
};
```

关于IfOperStatus，请读者阅读参考资料[18]。

最后，看关键函数wpa\_driver\_nl80211\_capa，其功能是用于获取无线网络设备的capability。代码如下所示。

[-->driver\_nl80211.c: : wpa\_driver\_nl80211\_capa]

```
static int wpa_driver_nl80211_capa(struct
wpa_driver_nl80211_data *drv)
{
    struct wiphy_info_data info;
    // 发送netlink命令NL80211_CMD_GET_WIPHY来获取Wi-Fi设备的信息。下
    // 文将单独用一节来介绍此函数
    if (wpa_driver_nl80211_get_info(drv, &info)) return -1;
```

```

drv->has_capability = 1;
/*
    drv->capa变量的类型是struct wpa_driver_capa，用于表示设备的
capability，这些capa如下。
    key_mgmt: 该设备支持的密钥管理类型。默认支持WPA、WPA-PSK、WPA2
和WPA2-PSK。
    enc: 支持的加密算法类型。默认支持WEP40、WEP104、TKIP和CCMP。
    auth: 支持的身份验证类型：默认支持Open System、Shared和LEAP。
*/
drv->capa.key_mgmt = WPA_DRIVER_CAPA_KEY_MGMT_WPA |
WPA_DRIVER_CAPA_KEY_MGMT_WPA_PSK |
WPA_DRIVER_CAPA_KEY_MGMT_WPA2 |
WPA_DRIVER_CAPA_KEY_MGMT_WPA2_PSK;
drv->capa.enc = WPA_DRIVER_CAPA_ENC_WEP40 |
WPA_DRIVER_CAPA_ENC_WEP104 |
WPA_DRIVER_CAPA_ENC_TKIP | WPA_DRIVER_CAPA_ENC_CCMP;
drv->capa.auth = WPA_DRIVER_AUTH_OPEN |
WPA_DRIVER_AUTH_SHARED | WPA_DRIVER_AUTH_LEAP;

/*
    WPA_DRIVER_FLAGS_SANE_ERROR_CODES选项主要针对associate操作。当
关联操作失败后，
    如果driver支持该选项，则表明driver能处理失败之后的各种收尾工作
(key、timeout等工作）。
    否则，WPAS需要自己处理这些事情。
*/
drv->capa.flags |= WPA_DRIVER_FLAGS_SANE_ERROR_CODES;
/*
    WPA_DRIVER_FLAGS_SET_KEYS_AFTER_ASSOC_DONE标志标明
association成功后，Kernel
    driver需要设置WEP key。这个标志出现的原因是由于Kernel API发生了变
动，使得只能在关联
    成功后才能设置key。
*/
drv->capa.flags |=
WPA_DRIVER_FLAGS_SET_KEYS_AFTER_ASSOC_DONE;
/*
    下面这两个标志表示Kernel中的driver是否能反馈EAPOL数据帧发送情况以及
Deauthentication/
    Disassociation帧发送情况（TX Report）。
*/
drv->capa.flags |= WPA_DRIVER_FLAGS_EAPOL_TX_STATUS;
drv->capa.flags |= WPA_DRIVER_FLAGS_DEAUTH_TX_STATUS;

/*

```

以下几个选项都和设备做AP使用有关，也就是和hostapd相关。此处简单介绍一下它们。

device\_ap\_sme表示AP集成了SME。读者还记得SME吗？3.3.6节曾介绍过。

```
/*
drv->device_ap_sme = info.device_ap_sme;
*/
poll_command_supported: hostapd需要判断STA是否还活跃，即心跳检测。检测方法是发送null数据帧（即不带任何数据的无线MAC数据帧），如果STA还活跃的话，一定会回复ACK给AP（读者还记得CSMA/CA机制吗？）。发送null数据帧的工作可以由Kernel driver完成，也可以由hostapd来完成。如果Kernel driver支持poll_command_supported，hostapd只要发送netlink命令NL80211_CMD_PROBE_CLIENT给Kernel驱动，所有工作就由Kernel驱动完成。
```

否则，hostapd需要自己构造一个null数据帧，然后再发送出去。

```
/*
drv->poll_command_supported = info.poll_command_supported;
*/
和WPA_DRIVER_FLAGS_EAPOL_TX_STATUS有关。如果wlan驱动支持的话，EAPOL帧TX
```

Report将通知给用户空间的driver wrapper，即此处的driver\_n180211。

```
/*
drv->data_tx_status = info.data_tx_status;
*/
use_monitor也和AP心跳检测STA有关。如果Kernel driver不支持poll_command_supported的话，hostapd可通过创建一个NL80211_IFTYPE_MONITOR类型的接口设备用于监控STA的活跃情况。
```

```
/*
drv->use_monitor = !info.poll_command_supported;

if (drv->device_ap_sme && drv->use_monitor) {
    // monitor_supported表示kernel driver是否支持创建NL80211_IFTYPE_MONITOR类
    // 型的接口设备
    if (!info.monitor_supported) drv->use_monitor = 0;
}
*/
经过测试，Galaxy Note 2机器中上述变量取值情况如下。
device_ap_sme为1，poll_command_supported为0，data_tx_status为
```

```

0, use_monitor为1,
    capa.flags取值情况见下文。
 */
if (!drv->use_monitor && !info.data_tx_status)
    drv->capa.flags &= ~WPA_DRIVER_FLAGS_EAPOL_TX_STATUS;

return 0;
}

```

Galaxy Note 2手机中得到的flags为0x2c0c0，它是如下几个选项的组合。

```

#define WPA_DRIVER_FLAGS_AP
0x00000040
#define WPA_DRIVER_FLAGS_SET_KEYS_AFTER_ASSOC_DONE
0x00000080
#define WPA_DRIVER_FLAGS_SANE_ERROR_CODES
0x00004000
#define WPA_DRIVER_FLAGS_OFFCHANNEL_TX
0x00008000
#define WPA_DRIVER_FLAGS_DEAUTH_TX_STATUS
0x00020000

```

上述wpa\_driver\_nl80211\_capa函数中，先调用wpa\_driver\_nl80211\_get\_info函数，从wlan driver获取设备信息，然后wpa\_driver\_nl80211\_capa再做一些处理。

此处向读者展示一下wpa\_driver\_nl80211\_get\_info的内容，请读者关注其中和nl80211用法有关的部分。

提醒 WPAS中类似wpa\_driver\_nl80211\_get\_info的函数还有很多，仅以wpa\_driver\_nl80211\_get\_info为代表进行介绍。

## (5) wpa\_driver\_nl80211\_get\_info函数分析

wpa\_driver\_nl80211\_get\_info代码如下所示。

[-->driver\_nl80211.c: : wpa\_driver\_nl80211\_get\_info]

```

static int wpa_driver_nl80211_get_info(struct
wpa_driver_nl80211_data *drv,
            struct wiphy_info_data *info)
{

```

```

struct nl_msg *msg;

os_memset(info, 0, sizeof(*info));
info->capa = &drv->capa;
msg = nlmsg_alloc();
.....
// 构造一个NL80211_CMD_GET_WIPHY命令以获取设备信息
nl80211_cmd(drv, msg, 0, NL80211_CMD_GET_WIPHY);
// NL80211_CMD_GET_WIPHY命令需要携带ifindex参数以指明要查询哪个设备
NLA_PUT_U32(msg, NL80211_ATTR_IFINDEX, drv->first_bss.ifindex);
// 发送命令并等待回复，回复消息将由wiphy_info_handler函数处理
if (send_and_recv_msgs(drv, msg, wiphy_info_handler, info) == 0) return 0;
.....
}

```

在driver\_nl80211.c中，wpa\_driver\_nl80211\_get\_info函数非常具有典型性。当driver wrapper和wlan driver通信时，需要构造一个nl\_msg消息，然后往其中填写对应的参数。发送该消息时，如果需要等待driver的回复，还可以设置一个回复消息处理函数用于解析接收到的回复消息。

上述代码中，wiphy\_info\_handler就是这个回调函数。其内容非常长。不过，绝大部分代码都是在解析netlink消息。因此，我们仅看其中与接口类型解析相关的代码片段即可窥斑见豹。

[-->driver\_nl80211.c: : wiphy\_info\_handler]

```

static int wiphy_info_handler(struct nl_msg *msg, void *arg)
{
    struct nla_attr *tb[NL80211_ATTR_MAX + 1];
    struct genlmsghdr *gnlh = nlmsg_data(nlmsg_hdr(msg));
    struct wiphy_info_data *info = arg;
    .....
    struct wpa_driver_capa *capa = info->capa;
    static struct nla_policy
        .....// 其他信息解析
    if (tb[NL80211_ATTR_SUPPORTED_IFTYPES]) {
        struct nla_attr *nl_mode;
        int i;
        nla_for_each_nested(nl_mode,
                           // 遍历
                           netlink attribute信息

```

```

        tb[NL80211_ATTR_SUPPORTED_IFTYPES], i) {
    switch (nla_type(nl_mode)) {
        case NL80211_IFTYPE_AP:// wlan driver支持设置接口类型
为AP
            capa->flags |= WPA_DRIVER_FLAGS_AP;      //
Galaxy Note 2支持此项
            break;
        .....
        case NL80211_IFTYPE_MONITOR:
            info->monitor_supported = 1;
            break;
    }
}
....// 其他信息解析
return NL_SKIP;
}

```

关于driver\_n180211.c中n180211使用最典型的wpa\_driver\_n180211\_get\_info即介绍到此，读者可阅读n180211头文件来了解NL80211\_CMD\_GET\_WIPHY和NL80211\_ATTR\_SUPPORTED\_IFTYPES相关的信息，其注释非常详尽。另外，读者可阅读参考资料[19]以了解更多和n180211命令相关的知识。

通过上面内容可知，wpa\_supplicant\_iface代码段三中最主要的函数是wpa\_drv\_init，下面总结它的相关知识。

### (6) wpa\_drv\_init相关知识总结

本节对wpa\_drv\_init函数进行了详细分析，其中涉及的两个重要数据结构如图4-14所示。

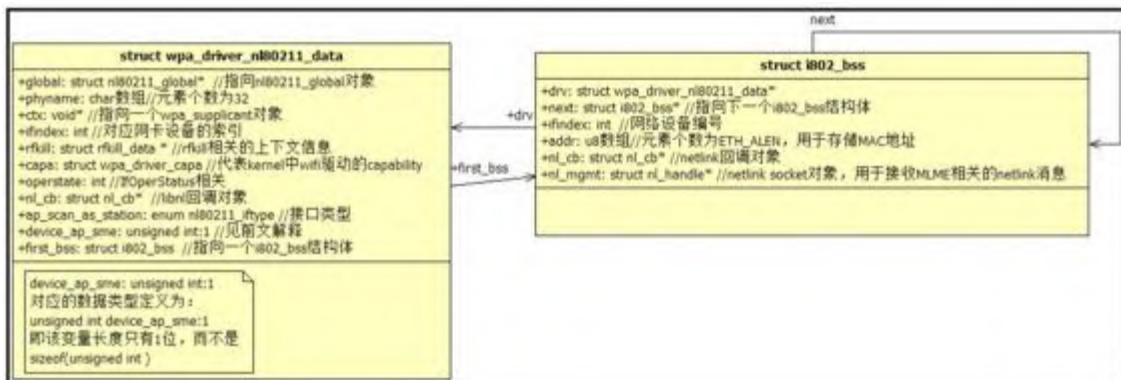


图4-14 wpa\_driver\_n180211\_data和i802\_bss结构体

图4-14中：

- wpa\_driver\_n180211\_data通过first\_bss成员包含一个i802\_bss结构体对象，而i802\_bss内部通过next指针构成一个单向链表。
- wpa\_driver\_n180211\_init最后返回的是一个i802\_bss对象，它就是driver wrapper上下文信息。i802\_bss通过drv变量指向一个wpa\_driver\_n180211\_data对象。

提醒 根据前文的分析，global\_init函数返回的是全局driver wrapper上下文信息。对于n180211 driver wrapper来说，这个全局上下文信息就是一个n180211\_global对象。

图4-15所示为wpa\_drv\_init中一些重要函数的调用流程。

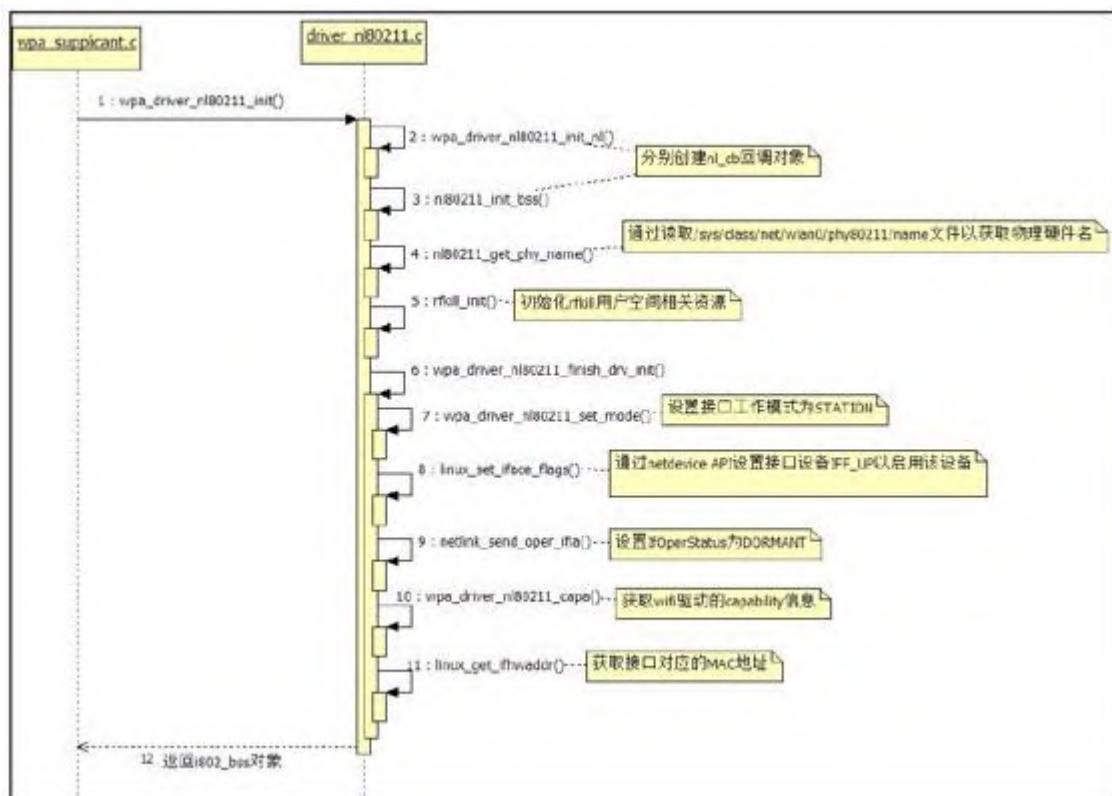


图4-15 wpa\_drv\_init流程

图4-15所示函数较多，内容也比较丰富，请读者注意其中涉及的背景知识。

#### 4. wpa\_supplicant\_init\_iface分析之四

继续来看wpa\_supplicant\_init\_iface函数，这次要分析的代码片段如下所示。

[-->wpa\_supplicant.c : : wpa\_supplicant\_init\_iface代码段四]

```
.....// 接wpa_drv_init
// ①初始化wpa上下文信息。见下文解释
if (wpa_supplicant_init_wpa(wpa_s) < 0) return -1;
    // 设置wpa_s->wpa指向一个wpa_sm对象，下面这两个函数用于设置wpa_sm中的一些成员变量
    wpa_sm_set_ifname(wpa_s->wpa, wpa_s->ifname,
                      wpa_s->bridge_ifname[0] ? wpa_s->bridge_ifname :
NULL);
    wpa_sm_set_fast_reauth(wpa_s->wpa, wpa_s->conf->fast_reauth);

/*
如果运行时配置文件(即wpa_supplicant.conf)设置了
dot11RSNAConfigPMKLifetime、
dot11RSNAConfigPMKReauthThreshold和
dot11RSNAConfigSATimeout，则使用配置文件中的值
来替换wpa_sm中的默认值。下文将详细介绍这个三个变量的含义。
*/
if (wpa_s->conf->dot11RSNAConfigPMKLifetime &&
    wpa_sm_set_param(wpa_s->wpa, RSNA_PMK_LIFETIME,
                      wpa_s->conf->dot11RSNAConfigPMKLifetime))
{.....}
.....// 处理dot11RSNAConfigPMKReauthThreshold和
dot11RSNAConfigSATimeout
    // ②获取Wi-Fi设备的hardware特性
    wpa_s->hw.modes = wpa_drv_get_hw_feature_data(wpa_s, &wpa_s-
>hw.num_modes,
                                                &wpa_s->hw.flags);

    // wpa_drv_get_capa函数已经见识过了，但这里出现了上一节没有介绍的新
成员
    if (wpa_drv_get_capa(wpa_s, &capa) == 0) {
        // ③capability信息，见下文解释
        wpa_s->drv_capa_known = 1;
        // 笔者的Note 2中，capa.flags的值为0x2c0c0
```

```

    wpa_s->drv_flags = capa.flags;
    wpa_s->probe_resp_offloads = capa.probe_resp_offloads;
    wpa_s->max_scan_ssids = capa.max_scan_ssids;
    wpa_s->max_sched_scan_ssids =
capa.max_sched_scan_ssids;
        wpa_s->sched_scan_supported =
capa.sched_scan_supported;
        wpa_s->max_match_sets = capa.max_match_sets;
        wpa_s->max_remain_on_chan = capa.max_remain_on_chan;
        wpa_s->max_stations = capa.max_stations;
    }
    if (wpa_s->max_remain_on_chan == 0)
        wpa_s->max_remain_on_chan = 1000;
}

```

上述代码片段共有三个关键点，分别如下。

- `wpa_supplicant_init_wpa`函数用于初始化`wpa_sm`相关的资源。
- `wpa_drv_get_hw_feature_data`函数用于获取hw特性。其中一些变量涉及较深的背景知识。
  - `wpa_drv_get_capa`是获取driver的capability。这个函数在上一节已经介绍过了，但本节出现了一些新的capability信息。

### (1) `wpa_supplicant_init_wpa`函数分析

`wpa_supplicant_init_wpa`函数代码并不复杂，主要完成以下两件事情。

- 创建一个`wpa_sm_ctx`对象并填充其中的函数指针成员。
- 初始化`wpa_sm`状态机。

代码如下所示。

[-->`wpas_glue.c`: : `wpa_supplicant_init_wpa`]

```

int wpa_supplicant_init_wpa(struct wpa_supplicant *wpa_s)
{
#ifndef CONFIG_NO_WPA
    struct wpa_sm_ctx *ctx;
    ctx = os_zalloc(sizeof(*ctx));
    .....

```

```

ctx->ctx = wpa_s;
ctx->msg_ctx = wpa_s;
ctx->set_state = _wpa_supplicant_set_state;
.....// 其他成员变量设置
wpa_s->wpa = wpa_sm_init(ctx);

#endif /* CONFIG_NO_WPA */
    return 0;
}

```

wpa\_sm\_init的代码如下所示。

[-->wpa.c: : wpa\_sm\_init]

```

struct wpa_sm * wpa_sm_init(struct wpa_sm_ctx *ctx)
{
    struct wpa_sm *sm;
    sm = os_zalloc(sizeof(*sm));
    dl_list_init(&sm->pmksa_candidates);
    sm->renew_snounce = 1;
    sm->ctx = ctx;
    // 下面这三个MIB相关成员变量的解释见下文
    sm->dot11RSNAConfigPMKLifetime = 43200;
    sm->dot11RSNAConfigPMKReauthThreshold = 70;
    sm->dot11RSNAConfigSATimeout = 60;
    // 创建PMKSA缓存，用于存储PMKSA
    sm->pmksa = pmksa_cache_init(wpa_sm_pmksa_free_cb, sm, sm);
    .....
    return sm;
}

```

上述两段代码中涉及的函数指针暂且先略过，先介绍其中的几个重要数据结构，它们如图4-16所示。

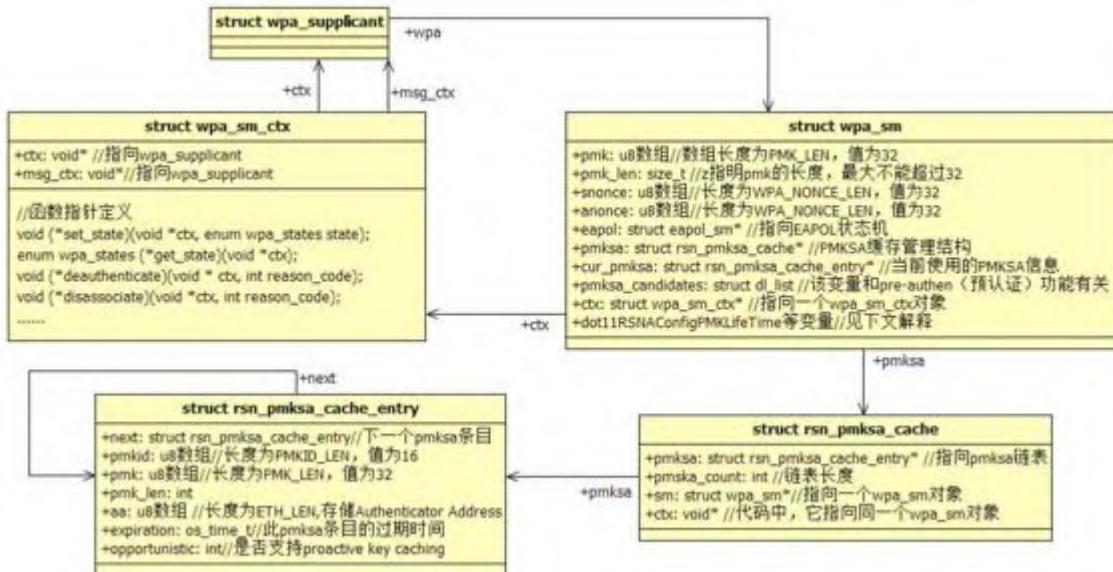


图4-16 wpa\_sm\_ctx和wpa\_sm等结构体

图4-16显示了四个重要数据结构的内容。

- **struct wpa\_sm\_ctx** 定义一些函数指针。这些函数的作用留待后续用到时再介绍。
- **struct wpa\_sm** 结构体名为状态机（SM代表State Machine），但和WPAS中其他状态机比起来，它更像是一个存储状态的上下文信息。该结构体内部通过eapol变量指向一个**struct eapol\_sm**对象。4.4节将详细分析eapol\_sm。
- **struct rsn\_pmksa\_cache**、**struct rsn\_pmksa\_cache\_entry**与PMKSA缓存有关。每一个**rsn\_pmksa\_cache\_entry**代表一个PMKSA条目。注意，**rsn\_pmksa\_cache\_entry**中有一个名为aa的数组，其存储的是Authenticator的Address。一般情况下它和AP的bssid相同。

PMKSA还和几个MIB选项有关，它们被定义成**wpa\_sm**中的同名成员变量（数据类型都是unsigned int），分别如下。

- **dot11RSNAConfigPMKLifetime**：表示每一个PMKSA条目的有效时间（单位为秒），默认是43200秒。过了有效时间后，需要重新计算PMKSA。

- dot11RSNAConfigPMKReauthThreshold：用于指明PMKSA条目有效时间过去百分之多少后，需要重新进行身份认证。默认是70%。
- dot11RSNAConfigSATimeout：指明supplicant和Authenticator双方进行身份验证的最长时间。默认是60秒。在此时间内没有完成身份验证，则认为验证失败。
- dot11RSNA4WayHandshakeFailures：用于保存4-Way Handshake失败的次数。

下面来看代码中的第二个关键函数wpa\_drv\_get\_hw\_feature\_data及相关的背景知识。

## (2) wpa\_drv\_get\_hw\_feature\_data函数分析

该函数内部将通过wpa\_driver\_ops结构体中的get\_hw\_feature\_data指针调用driver\_n180211实现的wpa\_driver\_n180211\_get\_hw\_feature\_data函数以获取wifi hw特性。此处不讨论其函数实现，而是看看hw特性都有哪些内容。hw特性由数据结构hostapd\_hw\_modes来表达，如图4-17所示。

<b>struct hostapd_hw_modes</b>
<pre>+mode: enum hostapd_hw_mode //指明此hostapd_hw_modes结构体所描述的是哪种模式 +num_channels: int //指明channels数组的个数 +channels: struct hostapd_channel_data * //存储channel相关的信息，包括channel编号，中心频率，最大功率等 +num_rates: int //指明rates数组的个数 +rates: int* //所支持的传输速率信息，以100kbs为单位 +ht_capab: u16 //和802.11n有关，本书不讨论 +mcs_set: u8数组//元素个数为16，和802.11n有关，本书不讨论 +a_mpdu_params: u8//和802.11n有关，本书不讨论 +flags: unsigned int//目前唯一定义的标志为HOSTAPD_MODE_FLAG_HT_INFO_KNOWN，其值为1</pre>
<pre>enum hostapd_hw_mode {     HOSTAPD_MODE_IEEE80211B,     HOSTAPD_MODE_IEEE80211G,     HOSTAPD_MODE_IEEE80211A,     NUM_HOSTAPD_MODES }; //枚举变量hostapd_hw_mode定义</pre>

图4-17 hostapd\_hw\_modes数据结构

wpa\_drv\_get\_hw\_feature\_data返回的是一个hostapd\_hw\_modes数组，其内容已经在图4-17中标记出来。这里展示一个实例，图4-18所示为修改WPAS后打印的Note 2 hw特性的一部分。

```

D/wpa_supplicant( 8319): =====Dump Mode[2]=====
D/wpa_supplicant( 8319): mode = 80211b
D/wpa_supplicant( 8319): supported channels = 13
D/wpa_supplicant( 8319): Channel info: id=[1] freq=[2412MHz],max_tx_power=[20dBm]
D/wpa_supplicant( 8319): Channel info: id=[2] freq=[2417MHz],max_tx_power=[20dBm]
D/wpa_supplicant( 8319): Channel info: id=[3] freq=[2422MHz],max_tx_power=[20dBm]
D/wpa_supplicant( 8319): Channel info: id=[4] freq=[2427MHz],max_tx_power=[20dBm]
D/wpa_supplicant( 8319): Channel info: id=[5] freq=[2432MHz],max_tx_power=[20dBm]
D/wpa_supplicant( 8319): Channel info: id=[6] freq=[2437MHz],max_tx_power=[20dBm]
D/wpa_supplicant( 8319): Channel info: id=[7] freq=[2442MHz],max_tx_power=[20dBm]
D/wpa_supplicant( 8319): Channel info: id=[8] freq=[2447MHz],max_tx_power=[20dBm]
D/wpa_supplicant( 8319): Channel info: id=[9] freq=[2452MHz],max_tx_power=[20dBm]
D/wpa_supplicant( 8319): Channel info: id=[10] freq=[2457MHz],max_tx_power=[20dBm]
D/wpa_supplicant( 8319): Channel info: id=[11] freq=[2462MHz],max_tx_power=[20dBm]
D/wpa_supplicant( 8319): Channel info: id=[12] freq=[2467MHz],max_tx_power=[20dBm]
D/wpa_supplicant( 8319): Channel info: id=[13] freq=[2472MHz],max_tx_power=[20dBm]
D/wpa_supplicant( 8319): supported rates = 4
D/wpa_supplicant( 8319): rates info: rates[0]=10*100kbps
D/wpa_supplicant( 8319): rates info: rates[1]=20*100kbps
D/wpa_supplicant( 8319): rates info: rates[2]=55*100kbps
D/wpa_supplicant( 8319): rates info: rates[3]=110*100kbps
D/wpa_supplicant( 8319): =====Dump Mode=====

```

图4-18 Note 2 dump信息

图4-18所示为hostapd\_hw\_modes数组中第三个元素的信息，它展示了硬件中和802.11b相关的特性。共13个信道，以及每个信道的中心频率以及最大功率，支持四种传输速率。

现在来看最后一个关键点。

### (3) capability信息及含义

wpa\_supplicant\_init\_wpa代码片段最后还显示了一些capability信息，它们的含义如下。

- probe\_resp\_offloads：当设备做AP使用时（即运行hostapd），它需要发送Probe Response帧以回复其他STA的Probe Request帧。Probe Response帧（或者AP发送的Beacon帧）的内容需要hostapd来填充。这个变量用于指明哪些vendor specific的内容将由Wi-Fi驱动或者硬件去填充。目前NL80211.h通过枚举类型nl80211\_probe\_resp\_offload\_support\_attr来定义所能支持的协议，包括WPSv1、WPSv2、P2P和802.11u。

- max\_scan\_ssids：一个Probe Request要么指定wildcard ssid以扫描周围所有的无线网络，要么指定某个ssid以扫描特定无线网络。为

了方便wpa\_supplicant的使用，driver新增了一个功能，使得上层可通过一次scan请求来扫描多个不同ssid的无线网络。注意，此功能只是方便了WPAS内部的使用。由于规范定义的Probe Request帧只能携带一个ssid参数。所以，上层即使想一次scan多个ssid，硬件实际上还是要为每一个ssid发送一个Probe Request帧。

- max\_sched\_scan\_ssids和sched\_scan\_supported：与计划扫描有关。max\_sched\_scan\_ssids和max\_scan\_ssids作用类似，是方便wpa\_supplicant同时扫描多个ssid而设置的。
- max\_match\_sets：使用计划扫描时，可以给驱动指定一个ssid过滤列表。只有扫描结果符合ssid过滤列表的那些无线网络才会通知wpa\_supplicant以开展后续处理。由于该过滤功能可由Wi-Fi硬件来完成，所以它可以节省一部分电力（即无须软件去执行过滤功能）。
- max\_remain\_on\_chan：该变量和off-channel transmission功能有关。该功能使得Wi-Fi硬件能在某个特定信道（channel）上保持awake状态一定时间用于传输某些MAC帧（例如管理帧中的一种名为Action的帧）。该功能叫off-channel的原因是，STA实际上在另一个信道（此channel叫on-channel）上和AP保持连接。举一个简单的例子，假设STA和所关联的AP工作在2.4GHz第6频段。在某些时候，STA会转移到2.4GHz其他频段以接收或处理其他STA（P2P的情况）或AP发送的MAC帧。上述例子中，6频段就是on-channel，而其他频段则是off-channel。max\_remain\_on\_chan变量用于指明STA在off-channel中工作的最长时间，以毫秒为单位。为什么要限制off-channel时间呢？还是以上述例子为例，STA和AP工作在第6频段，二者数据传输也是在第6频段。当STA转移到其他频段时，它将无法接收第6频段所发送的数据。如果max\_remain\_on\_chan时间过长，用户将发现数据传输率大幅降低②。
- max\_stations：当手机做AP使用时（即无线网络接口设备的类型为NL80211\_IFTYPE\_AP），该变量表示最多支持多少个STA与之关联。

下面接着分析wpa\_supplicant\_init\_iface如下所示最后一部分代码。

## 5. wpa\_supplicant\_init\_iface分析之五

wpa\_supplicant\_init\_iface最后一段代码如下所示。

[-->wpa\_supplicant.c: : wpa\_supplicant\_init\_iface代码段五]

```
// ①初始化driver wrapper模块最后一部分内容
if (wpa_supplicant_driver_init(wpa_s) < 0) return -1;
.....// TDLS相关, 本书不讨论
.....// 设置country
    // 初始化WPS相关模块, 本章不讨论
if (wpas_wps_init(wpa_s)) return -1;
    // ②初始化EAPOL模块。这部分内容4.4节介绍
if (wpa_supplicant_init_eapol(wpa_s) < 0) return -1;
    wpa_sm_set_eapol(wpa_s->wpa, wpa_s->eapol);
    // ③初始化ctrl i/f模块
    wpa_s->ctrl_iface = wpa_supplicant_ctrl_iface_init(wpa_s);
    .....
    wpa_s->gas = gas_query_init(wpa_s); // GAS相关, 本书不讨论
#endif CONFIG_P2P
    if (wpas_p2p_init(wpa_s->global, wpa_s) < 0) {.....// P2P
        模块初始化, 见第7章分析}
#endif /* CONFIG_P2P */
    // ④bss相关, 详情见下文
    if (wpa_bss_init(wpa_s) < 0) return -1;

    return 0;// wpa_supplicant_init_iface终于成功返回
```

上述代码包括四个关键函数, 其中第二个关键点和EAPOL模块相关, 其内容4.4节再介绍。

### (1) wpa\_supplicant\_driver\_init函数分析

先来看第一个关键函数wpa\_supplicant\_driver\_init, 代码如下所示。

[-->wpa\_supplicant.c: : wpa\_supplicant\_driver\_init]

```
int wpa_supplicant_driver_init(struct wpa_supplicant *wpa_s)
{
    static int interface_count = 0;
    // 关键函数, 见下文代码分析
    if (wpa_supplicant_update_mac_addr(wpa_s) < 0) return -1;

    if (wpa_s->bridge_ifname[0]) {.....// 桥接相关内容, 本书不讨论}
```

```

// 清除driver中保存的key相关的信息
wpa_clear_keys(wpa_s, NULL);
// 设置TKIP countermeasure值为0
wpa_drv_set_countermeasures(wpa_s, 0);
// 清空drive wrapper及driver中保存的pmkid信息。
wpa_drv_flush_pmkid(wpa_s);
// 设置wpa_supplicant结构体中的一些变量的初值
wpa_s->prev_scan_ssid = WILDCARD_SSID_SCAN;
wpa_s->prev_scan_wildcard = 0;

// 判断wpa_conf中是否有使能的网络
if (wpa_supplicant_enabled_networks(wpa_s->conf)) {
    .....// 当前配置文件中没有使能任何一个网络，故此段代码略去
} else // 设置状态为WPA_INACTIVE。该函数比较简单，请读者自行阅读
    wpa_supplicant_set_state(wpa_s, WPA_INACTIVE);

return 0;
}

```

上述代码中唯一需要介绍的就是wpa\_supplicant\_update\_mac\_addr，因为它和图4-1中的12\_packet模块初始化有关，其代码如下所示。

```

[-->wpa_supplicant.c: : wpa_supplicant_update_mac_addr]

int wpa_supplicant_update_mac_addr(struct wpa_supplicant
*wpa_s)
{
    if (wpa_s->driver->send_eapol) { .....// nl80211 driver
wrapper没有定义该函数
    } else if (!(wpa_s->drv_flags &
        WPA_DRIVER_FLAGS_P2P_DEDICATED_INTERFACE)) {
        // WPA_DRIVER_FLAGS_P2P_DEDICATED_INTERFACE和P2P有关,
本例不支持该参数
        12_packet_deinit(wpa_s->12);           // 先释放之前创建的
12_packet模块
        // 初始化12_packet
        wpa_s->12 = 12_packet_init(wpa_s->ifname,
            wpa_drv_get_mac_addr(wpa_s),      // 获取
接口的MAC地址
            ETH_P_EAPOL,
            // 收到的EAPOL及EAP帧将由此函数负责处理
            wpa_supplicant_rx_eapol, wpa_s, 0);
        .....
    } else { .....}
}

```

```

    // 将l2_packet_data中的own_addr内容复制到wpa_supplicant的
    own_addr成员变量
    if (wpa_s->l2 && l2_packet_get_own_addr(wpa_s->l2, wpa_s-
    >own_addr)) {....}
    // 再把wpa_supplicant的own_addr复制到wpa_sm中的own_addr中
    wpa_sm_set_own_addr(wpa_s->wpa, wpa_s->own_addr);
    .....
    return 0;
}

```

l2\_packet\_init内部就是创建一个PF\_PACKET域的socket。注意，l2\_packet\_init最后一个参数为0，这样，socket的类型将是SOCK\_DGRAM。l2\_packet\_init返回值类型为l2\_packet\_data，其成员如图4-19所示。



图4-19 l2\_packet\_data成员

l2\_packet\_init通过eloop\_register\_read\_sock函数为图4-19中的socket句柄fd注册一个读事件回调函数l2\_packet\_receive，而该函数将接收socket数据，然后回调rx\_callback。该函数对于4-Way Handshake非常重要，后文将详细介绍此处设置的回调函数（wpa\_supplicant\_rx\_eapol）。

下面来看第三个关键函数wpa\_supplicant\_ctrl\_iface\_init。

## (2) wpa\_supplicant\_ctrl\_iface\_init函数分析

该函数内部将创建一个unix域socket，然后向eloop注册一个读事件处理函数。Android平台对此函数进行了定制，主要是利用图4-3中init配置文件中wpa\_supplicant的socket选项。init在fork出一个wpa\_supplicant子进程时将创建一个socket，并通过环境变量传给wpa\_supplicant子进程。

提示 对socket选项感兴趣的读者可阅读《深入理解Android：卷I》3.2.3节“启动Zygote”。

```
[-->ctrl_iface_unix.c: : wpa_supplicant_ctrl_iface_init]

wpa_supplicant_ctrl_iface_init(struct wpa_supplicant *wpa_s)
{
    struct ctrl_iface_priv *priv;
    struct sockaddr_un addr;
    .....
    priv = os_zalloc(sizeof(*priv));
    dl_list_init(&priv->ctrl_dst);
    priv->wpa_s = wpa_s;
    priv->sock = -1;
    buf = os_strdup(wpa_s->conf->ctrl_interface);
    .....
#ifdef ANDROID                               // Android平台定义了此编译宏
    // addr.sun_path的值为wpa_wlan0。该值和图4-5中socket选项指定的
    // 值一样
    os_snprintf(addr.sun_path, sizeof(addr.sun_path), "wpa_%s",
                wpa_s->conf-
                >ctrl_interface);
    priv->sock = android_get_control_socket(addr.sun_path); // 
    // 获取socket句柄
    if (priv->sock >= 0)
        goto haveSock;                      // 直接跳转
#endif /* ANDROID */
    .....
haveSock:
#endif /* ANDROID */
    // 客户端发送命令都由wpa_supplicant_ctrl_iface_receive处理
    eloop_register_read_sock(priv->sock,
                            wpa_supplicant_ctrl_iface_receive,
                            wpa_s, priv);
    // 读者还记得4.3.2节wpa_supplicant_init分析中提到的消息全局回调函
    // 数吗
    wpa_msg_register_cb(wpa_supplicant_ctrl_iface_msg_cb);
```

```
    os_free(buf);
    return priv;
```

上述代码中，客户端发送的命令将由  
wpa\_supplicant\_ctrl\_iface\_receive函数处理。

提示 后文分析线路二中用户发送的WPAS命令时，就将直接分析此函数。

### (3) wpa\_bss\_init函数分析

最后来看wpa\_bss\_init函数。

[-->bss.c: : wpa\_bss\_init]

```
int wpa_bss_init(struct wpa_supplicant *wpa_s)
{
    // bss和bss_id是wpa_supplicant结构体中的成员变量，它们通过链表的方式
    // 来保存wpa_bss信息
    dl_list_init(&wpa_s->bss);
    dl_list_init(&wpa_s->bss_id);
    // 注册一个超时任务，超时时间为WPA_BSS_EXPIRATION_PERIOD，值为10
    // 秒
    eloop_register_timeout(WPA_BSS_EXPIRATION_PERIOD,
0, wpa_bss_timeout, wpa_s, NULL);
    return 0;
}
```

wpa\_supplicant注册了一个定时任务用于定时更新其保存的wpa\_bss信息，一旦某个无线网络在一定时间内没有更新或使用，则需要从链表中把它去掉。

超时任务的函数代码如下。

[-->bss.c: : wpa\_bss\_timeout]

```
static void wpa_bss_timeout(void *eloop_ctx, void *timeout_ctx)
{
    struct wpa_supplicant *wpa_s = eloop_ctx;
    // bss_expiration_age默认是1800秒
    // 下面这个函数将更新wpa_bss链表以删除一些无用的wpa_bss对象
    wpa_bss_flush_by_age(wpa_s, wpa_s->conf-
```

```
>bss_expiration_age);
    eloop_register_timeout(WPA_BSS_EXPIRATION_PERIOD,
0,wpa_bss_timeout, wpa_s, NULL);
}
```

## 6. wpa\_supplicant\_add\_iface流程总结

看到这里，读者一定会感慨，线路一走下来比较艰难。其中所调用函数之多、每个变量背后含义之丰富都是初学者要面临的挑战。在此，通过图4-20展示wpa\_supplicant\_add\_iface中所涉及的几个重要函数的调用流程。

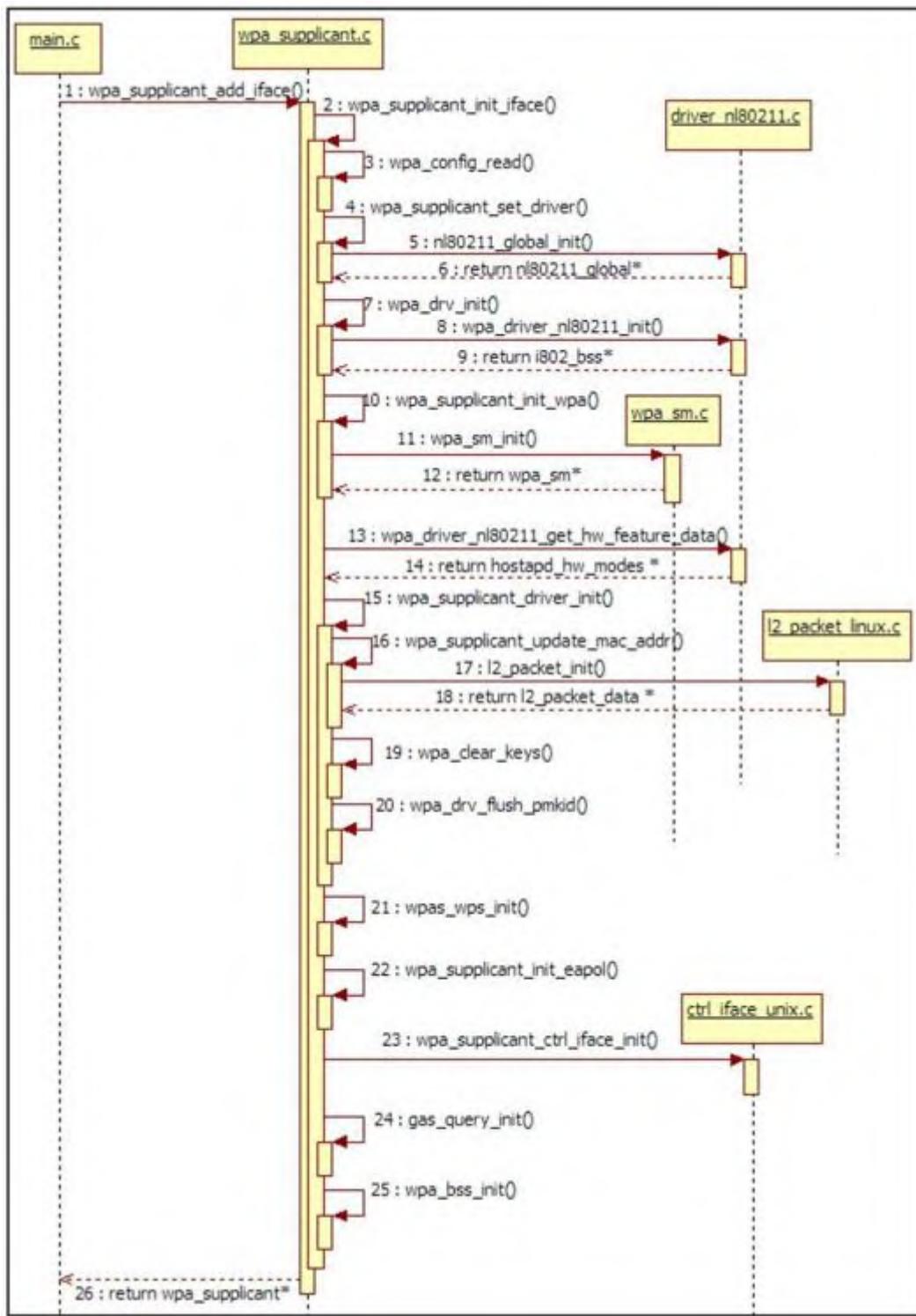


图4-20 wpa\_supplicant\_add\_iface重要流程

图4-20中的第8个函数调用`wpa_driver_nl80211_init`的内容在图4-15中。请读者结合这两个图来学习调用流程。

即使用了如此之多的笔墨，`wpa_supplicant_init`初始化所涉及的内容依然不能全部覆盖到。下一节介绍非常重要的两个模块：EAP和EAPOL。

① 注意，此结论为笔者根据笔记本的表现形式进行的猜测。如有读者知道其工作原理不妨与大家分享。

② `max_remain_on_chan`的官方解释可参考n180211关于NL80211\_CMD\_REMAIN\_ON\_CHANNEL的定义，其原文是”Request to remain awake on the specified channel for the specified amount of time. This can be used to do off-channel operations like transmit a Public Action frame and wait for a response while being associated to an AP on another channel”。请了解该功能的读者和大家分享相关知识。

## 4.4 EAP和EAPOL模块

我们在第3章曾介绍过EAP和EAPOL方面的知识，它们主要和EAP/EAPOL数据包格式以及数据包交互流程有关。在此基础上，本节将进一步讨论WPAS中图4-1所涉及的EAP State Machine和EAPOL State Machine这两个模块。

**提示** 虽然图4-1所示这两个模块名字中都带State Machine一词，但笔者更愿意称它们为EAP模块和EAPOL模块，其原因我们后续将会影响到。另外，为了行文方便，以后将用SM代表State Machine。

本节先来介绍EAP模块，它和RFC4137协议有关。

# 读累了记得休息一会儿哦~

公众号：古德猫宁李

- 电子书搜索下载
- 书单分享
- 书友学习交流

网站：[沉金书屋 https://www.chenjin5.com](https://www.chenjin5.com)

- 电子书搜索下载
- 电子书打包资源分享
- 学习资源分享

## 4.4.1 EAP模块分析<sup>[20]</sup>

RFC4137协议的全称是“State Machine for Extensible Authentication Protocol (EAP) Peer and Authenticator”，它描述了Peer端（即Supplicant端）和Authenticator端通过状态机（State Machine）这种方式来实现EAP处理流程的具体步骤和相关细节。本节将重点介绍Supplicant端SM的设计原理。为了行文方便，本节将使用SUPP代替Supplicant。

### 1. Supplicant端SM设计原理

对状态机来说，最重要的是其状态切换图。RFC4137中SUPP SM状态切换如图4-21所示。

图4-21的内容极为丰富，此处先介绍其中三个知识点。

- SUPP SM一共定义了13个状态，每个状态用一个框表示。框顶部所示为状态名，如INITIALZE、IDLE等。
- 每个状态都可以有自己的Entry Action（以后简称EA）。进入这个状态后，EA将被执行。EA由状态框中状态名下边的伪代码（采用了类C++语法）表示。以FAILURE状态为例，当状态机进入该状态后，将执行“eapFail=TRUE”伪代码，eapFail是SUPP SM定义的变量，下文将详细介绍图中涉及的变量和相关函数。
- 图中的UCT代表Unconditional Transition，即无条件状态转换。以DISCARD状态和IDLE状态为例，由于UCT的存在，当SUPP SM在DISCARD状态中执行完其EA后，将直接转换到IDLE状态。

对一个状态机而言，其状态的转换是因为外界条件发生变化导致。在规范中，这些外界条件由变量来表达。图4-21中出现了很多变量和EA所包括的一些函数，它们都由RFC4137文档定义。了解它们的作用对真正理解SUPP SM有直接和重要帮助。接下来的章节就将介绍这些变量和函数。

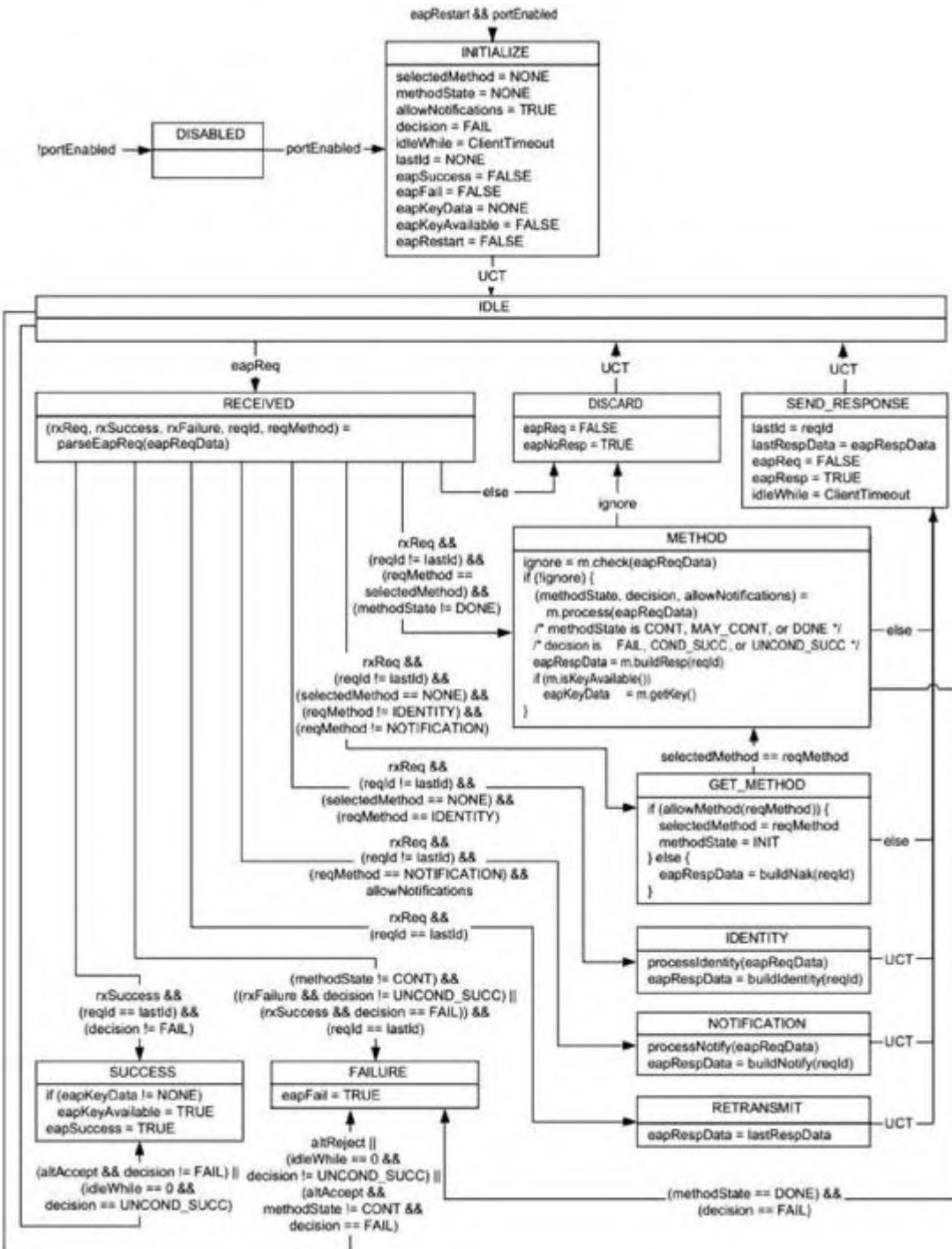


图4-21 SUPP SM状态切换

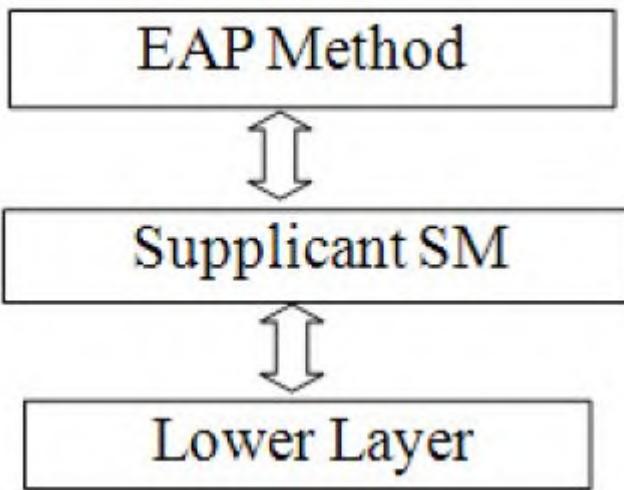


图4-22 RFC4137 SUPP SM模块划分

RFC4137将和SUPP SM相关的模块分为三层，如图4-22所示。图中最底层是Lower Layer（LL），这一层的作用是接收和发送EAP包。位于中间的SUPP SM层实现了Supplican状态机。最上层是EAP Method（EM）层，它实现了具体的EAP方法。

SUPP SM将与EM层和LL层交互。一个最典型的交互例子就是LL收到EAP数据包后，将该数据包交给SUPP SM层去处理。如果该EAP包需要EM层处理（例如具体的验证算法需要EM完成），则SUPP SM层将该包交给EM。EM处理完的结果将由SUPP SM转交给LL去发送。

**提示** RFC4137中，三层之间交互的手段可以是设置变量，或者是调用函数。

先来看SUPP SM与LL交互时所使用的变量。

### (1) LL层和SUPP SM层交互变量

LL层和SUPP SM层的交互比较简单，主要包括三个步骤。

- 1) LL层收到EAP数据包后，将其保存在eapReqData变量中，然后设置eapReq变量为TRUE。这个变量的改变对SUPP SM层来说是一个触发信号(signal)。SUPP SM可能会发生状态转换。

2) SUPP SM层从eapReqData中取出数据后进行处理。如果有需要回复的数据，则设置eapResp值为TRUE，否则设置eapNoResp值为TRUE。回复数据存储在eapRespData中。LL层将发送此回复包。

3) 如果SUPP SM完成身份验证后，它将设置eapSuccess或eapFailure变量以告知LL层其验证结果。eapSuccess为TRUE，表明验证成功。eapFailure为TRUE，则验证失败。

上述描述中所涉及的变量及其类型如表4-1所示。注意，此处的数据类型属于伪代码。

**提示** 在WPAS中，LL层并非那些直接利用socket进行数据收发的模块，而是EAPOL模块。EAP和EAPOL模块的关系将留待下一节再介绍。

表 4-1 LL 层和 SUPP SM 层交互变量

变 量 名	数据类型	变 量 说 明
eapReq	boolean	当 LL 收到 EAP Request 包时，将设置该为 TRUE
eapReqData	EAP Packet	当 eapReq 为 TRUE 时，该变量用于保存一个 EAP Packet
portEnabled	boolean	LL 层设置该值为 TRUE，用以通知 SUPP SM 底层 Port 已经使能，可以开展认证工作。该值如若设为 FALSE，SUPP SM 直接进入 DISABLED 状态（参考图 4-21）
idleWhile	integer	Supplicant 需要等待来自 Authenticator 的 EAP Request 包。对此，规范使用了一个超时时间（见本表最后的 ClientTimeout 变量）。idleWhile 表示还剩多少时间（单位为秒）。该值一旦为 0，则 Supplicant 认证超时
eapRestart	boolean	LL 层通知 SUPP SM 需要重新开始认证工作（restart authentication）。注意前文的提醒，在 WPAS 中，LL 层其实是 EAPOL 模块
altAccept	boolean	Alternate indication of success。见下文解释
altReject	boolean	Alternate indication of failure。见下文解释
eapResp	boolean	SUPP SM 通知 LL 层，表示有一个 EAP Response 包需要发送

( 续 )

变 量 名	数据类型	变 量 说 明
eapNoResp	boolean	SUPP SM 通知 LL 层，表示请求已处理，但没有 EAP Response 包要发送
eapSuccess	boolean	值为 TRUE 表示 supplicant 进入 SUCCESS 状态
eapFail	boolean	值为 FALSE，表示 supplicant 进入 FAILURE 状态
eapRespData	EAP packet	当 eapResp 为 TRUE 时，指向 EAP Response 数据
eapKeyData	EAP Key	用于存储和 Key 相关的数据
eapKeyAvailable	boolean	该值只在 SUCCESS 状态设置。如果有 Key 相关的信息则置为 TRUE
ClientTimeout	integer	表示 Supplicant 等待 EAP Request 消息的超时时间。它和 idleWhile 相对应。ClientTimeout 的值是固定的，而 idleWhile 表示还剩多少时间

表4-1中altAccept和altReject两个变量的命名非常晦涩难懂。RFC4137指出这两个变量的定义在RFC3748中。实际上，RFC3748从头至尾都没有出现过这两个变量。经过仔细研究，笔者发现<http://lists.frascone.com/pipermail/eap/msg02578.html>对此有一个说法，内容如下。

这两个变量取名为lowerLayerSuccess和lowerLayerFailure更合适，它们用于通知LL层Success或Failure信息。结合上述资料，笔者查阅了RFC3748 7.12节，在802.11网络中，Indication（通知）Success和Failure的可能场景如下。

- 当supplicant收到Disassociate帧或者Deauthenticate帧时，表示lowerLayerFailure。
- 当收到4-Way Handshake第一个Message时，表示lowerLayerSuccess。

规范阅读提示 RFC4137中，上述变量还可分为两种类型。

1) 由LL层暴露给SUPP SM层的变量（Variables from Lower Layer to Peer）。它们从表4-1中的eapReq开始，到altReject结束。原则上，LL层和SUPP SM层都可以修改这些变量。

2) 由SUPP SM层暴露给LL层的变量（Variables from Peer to Lower Layer）。它们从表4-1中的eapResp开始，到eapKeyAvailable结束。原则上，LL层和SUPP SM层都可以修改这些变量。另外，这些变量也可由SUPP SM层暴露给EM层来使用。

接着来看SUPP SM层和EM层交互变量。

## (2) SUPP SM层和EM层交互变量

SUPP SM层和EM层的交互也是通过变量来完成的。这些变量如表4-2所示。

表 4-2 SUPP SM 和 EM 层交互变量

变 量 名	数据类型	变量说明
eapReqData	EAP Packet	存储了 EAP Request 数据包，SUPP SM 需将其交给 EM 处理
ignore	boolean	取值情况见 methodState 一栏
allowNotifications	boolean	取值情况见 methodState 一栏
decision	枚举类型	取值情况见 methodState 一栏

( 续 )

变 量 名	数据类型	变量说明
methodState	枚举类型	methodState 用于记录 EM 层的状态。其不同取值有不同含义
		EM 在此状态中要完成以下工作： 1) 初始化 EM 内部资源 2) 判断是否需要处理 EAP 包或丢弃它。碰到错误情况时可丢弃数据包，并设置 ignore 为 TRUE，表示忽略此次数据包处理 3) 如果需要处理 EAP 包，则将其交给具体的 Method 去处理。处理过程中，如果有 Key 要生成（由 EM 层中具体的 Method 决定），则将 Key 数据保存在 eapKeyData 中。处理完后生成的回复包保存在 eapRespData。最后，设置 ignore 为 FALSE 4) 根据处理结果，更新 methodState 的值
		CONT 之意，decision 被设置为 FAIL，表示 EM 层还需要继续工作
		Method 根据和 Authenticator 交互的信息来决定是否继续执行后面的流程（设置 decision 为 COND_SUCC）还是结束此次认证（设置 decision 为 FAIL）。另外，如果 EM 发现有消息（根据和 Authenticator 交互的信息来判断）需要通知用户的话，则会设置 allowNotifications 为 TRUE
	DONE	EAP Method 的最终状态。如果认证失败，则设置 decision 为 FAIL。如果确定认证成功（即下一个 EAP 包将是 EAP Success），则设置 decision 为 UNCOND_SUCC。如果 Method 不确定服务端的认证情况，但在服务器允许的情况下，Supplicant 愿意尝试，则设置 decision 为 COND_SUCC

注意，methodState 和 decision 的值由具体的认证方法（即 Method）来确定。

**提示** 本书不讨论所有 EAP 方法的具体实现。感兴趣的读者可以深入研究 EAP 模块。不过对 EAP SUPP SM 来说，methodState 和 decision 的取值情况才是最重要的，因为它们会直接影响 SUPP SM 的状态切换。

### (3) SUPP SM 其他变量和处理函数

RFC4137 还为 SUPP SM 定义了其他一些变量（Peer State Machine Local Variables）及函数。变量定义见表 4-3。

表 4-3 SUPP SM 内部变量定义

变 量 名	数据类型	作 用
selectMethod	EAP Type	表示所选的 EAP 方法类型
lastId	integer	取值范围为 0 – 255 或 NONE。表示上一次 (last)EAP 包的标记 (identifier)
lastRespData	EAP packet	表示上一次发送的 EAP Packet
rxReq	boolean	表示当前接收的 EAP 包类型为 EAP Request 包
rxSuccess	boolean	表示当前接收的 EAP 包类型为 EAP Success 包
rxFailure	boolean	表示当前接收的 EAP 包类型为 EAP Failure 包
reqId	integer	表示当前 EAP 请求的 ID
reqMethod	EAP type	标示当前 EAP 包请求的认证方法 (EAP Method)
ignore	boolean	在 METHOD 状态设置，表示是否丢弃当前 EAP 包

表4-4所示为SUPP SM处理函数的定义。

表 4-4 SUPP SM 函数定义

函 数 名	作 用
parseEapReq	解析 EAP 数据包。该函数的使用案例如下： (rxReq,rxSuccess,rxFailure,reqId,reqMethod)=parseEapReq(eapReqData)
processNotify	处理 Notification Request 请求。其使用案例如下： processNotify(eapReqData)
buildNotify	创建 notification response。其使用案例如下： eapRespData=buildNotify(reqId)
processIdentity	处理 Identity Request 消息。其使用案例如下： processIdentity(eapReqData)
buildIdentity	创建 Identity Response。其使用案例如下： eapRespData=buildIdentity(reqId)
m.check	m 表示 EM 层。用来监测 EAP 数据包消息是否。其使用案例如下： ignore=m.check(eapReqData)
m.process	处理认证请求。其使用案例如下： (methodState.decision.allowNotifications)=m.process(eapReqData)
m.buildResp	构造回复消息。其使用案例如下： eapRespData=m.buildResp(reqId)
m.getKey	返回认证过程中生成的和 Key 相关的信息。其使用案例如下： eapKeyData=m.getKey()

注意，表4-4中用伪代码展示了这些函数的使用案例，它们并不遵守 C++语法。例如：

```
(rxReq, rxSuccess, rxFailure, reqId, reqMethod)
=parseEapReq (eapReqData)
```

等号左边为parseEapReq函数的返回值，等号右边括号中的“eapReqData”为parseEapReq的输入参数。如果使用案例中没有等号，则表示该函数无返回值（具体实现时，可设置该函数的返回值为 void）。

注意 图4-21中还包含一些其他函数，奇怪的是规范中并没有列举它们。不过，相信读者很容易理解这些数作用，此处不详述。现在，读者能看懂图4-21所示的状态图了吗？

#### (4) SUPP SM定义的状态

SUPP SM定义的13个状态如表4-5所示。

表 4-5 SUPP SM 状态

状态名	作用
DISABLED	portEnabled 为 FALSE 时，SUPP SM 将直接跳转至此状态
INITIALIZE	初始化状态
IDLE	空闲状态。SUPP SM 等待事情发生
RECEIVED	收到一个 EAP 包将进入此状态。该状态中，SUPP SM 需要解析 EAP 包头

(续)

状态名	作用
GET_METHOD	该状态对应的情况是：当 Peer 和 Authenticator 协商的 EAP Method 不一致时，Supplicant 需申请换一种 Method 方法。详情见图 3-38
METHOD	具体的 EAP 消息处理在此状态中完成
SEND_RESPONSE	该状态下，SUPP SM 通知 LL 层有一个 response 包需要发送
DISCARD	该状态下，SUPP SM 丢弃请求包，并且没有数据包需要发送
IDENTITY	用于处理 Identity Request 包
NOTIFICATION	用于处理 Notification 请求，并返回一个 response
RETRANSMIT	重新发送一次 response 包
SUCCESS	认证成功
FAILURE	认证失败

前面展示了RFC4137中和SUPP SM相关的内容。笔者觉得这个状态机定义太过烦琐。不过，本着简单、明了并且没有歧义的原则，这种做法似乎又无可厚非。从笔者经验来看，读者只要能看懂图4-21所示SUPP SM状态切换图即算掌握了EAP模块的精髓了。WPAS中EAP SUPP SM该如何实现呢？请看下节。

## 2. EAP SUPP SM代码分析

上文中曾提到过，RFC4137定义的状态机非常烦琐，而具体实现可以根据情况进行裁剪。不过，WPAS中的EAP SUPP SM却较为严格得遵循了RFC4137。先来看其定义的数据类型和数据结构。

#### (1) 相关数据结构与数据类型

图4-23所示为EAP SUPP SM定义的枚举变量类型。

```
//下面这两个枚举变量定义于eap_i.h
typedef enum {
    DECISION_FAIL, DECISION_COND_SUCC,
    DECISION_UNCOND_SUCC} EapDecision;

typedef enum {
    METHOD_NONE, METHOD_INIT, METHOD_CONT,
    METHOD_MAY_CONT, METHOD_DONE} EapMethodState;

//定义于Defs.h
typedef enum { FALSE = 0, TRUE = 1 } Boolean

//对应RFC4137中boolean类型，定义于eap.h中
enum eapol_bool_var {
    EAPOL_eapSuccess,EAPOL_eapRestart,EAPOL_eapFail,
    EAPOL_eapResp,EAPOL_eapNoResp,EAPOL_eapReq,
    EAPOL_portEnabled,EAPOL_altAccept,EAPOL_altReject
};

//对应于RFC4137中integer类型，定义于eap.h中
enum eapol_int_var {
    EAPOL_idleWhile
};

//右图
typedef enum {
    EAP_TYPE_NONE = 0,
    EAP_TYPE_IDENTITY = 1 /* RFC 3748 */,
    EAP_TYPE_NOTIFICATION = 2 /* RFC 3748 */,
    EAP_TYPE_NAK = 3 /* Response only, RFC 3748 */,
    EAP_TYPE_MDS = 4, /* RFC 3748 */
    EAP_TYPE OTP = 5 /* RFC 3748 */,
    EAP_TYPE_GTC = 6, /* RFC 3748 */
    EAP_TYPE_TLS = 13 /* RFC 2716 */,
    EAP_TYPE LEAP = 17 /* Cisco proprietary */,
    EAP_TYPE SIM = 18 /* RFC 4186 */,
    EAP_TYPE TTLS = 21 /* RFC 5281 */,
    EAP_TYPE AKA = 23 /* RFC 4187 */,
    .....
} EapType //定义于eap_defs.h
```

图4-23 EAP SUPP SM枚举类型定义

图4-23中，左图定义了EapDecision、EapMethodState、Boolean、eapol\_bool\_var和eapol\_int\_var枚举类型，它们都和上节介绍的变量及类型有关。右图定义了EapType枚举变量，代表不同的EAP Method。

图4-24所示为WPAS中和SUPP SM相关的数据结构。

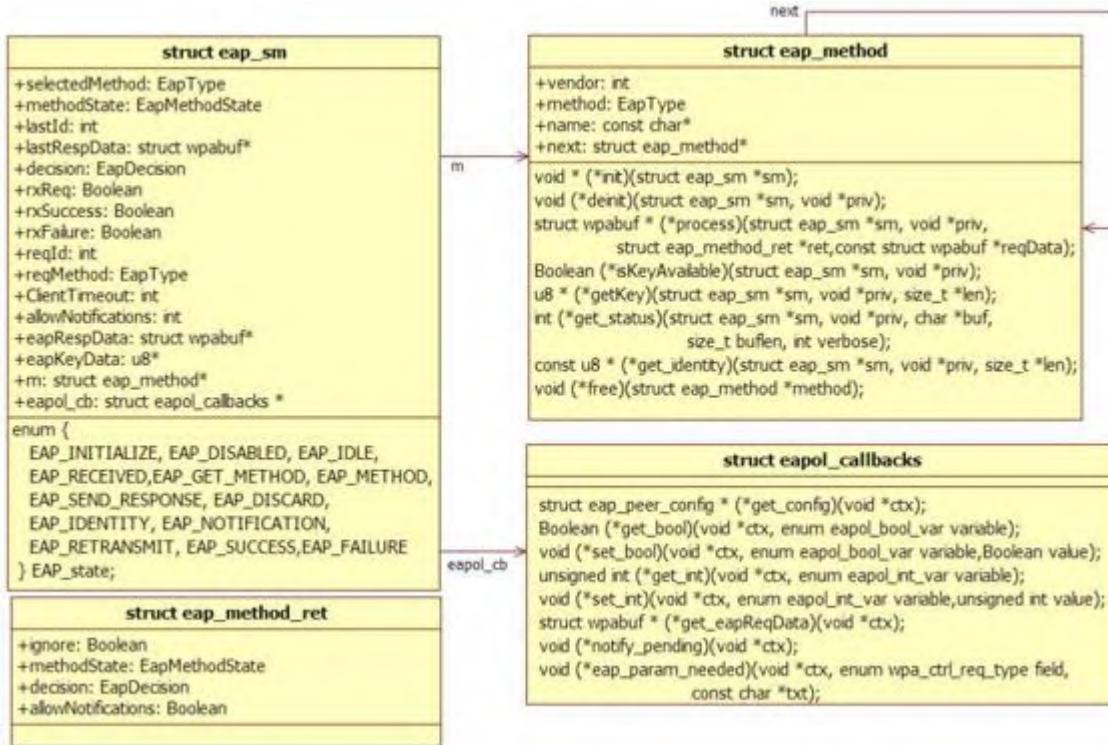


图4-24 EAP SUPP SM相关数据结构定义

图4-24中，`eap_sm`是RFC4137 EAP SUPP SM的代表。从其成员变量的命名可知，它几乎完全是按照RFC4137来实现的。`eap_sm`定义了一个名为`EAP_state`的枚举类型成员变量。

`eap_sm`通过`m`成员变量指向一个`eap_method`链表。`eap_method`是一个由`next`指针链接起来的单向链表，每一个`eap_method`对象代表一种具体的EAP Method (EAP Method的注册请回顾4.3.2节“`eap_register_methods`函数分析” )。`eap_method`最重要的是其处理函数，WPAS对其略有修改。例如，`process`函数实际上完成了表4-4中`m.check`、`m.process`和`m.buildResp`的功能。

`eap_method`中，`process`函数第三个参数的类型是`eap_method_ret*`，代表一个`eap_method_ret`对象。由图4-24可知，它包括了`ignore`、`methodState`、`decision`以及`allowNotifications`变量。

`eap_sm`的`eapol_cb`对象指向一个`eapol_callbacks`对象，它是LL层的代表。不过，`eapol_callbacks`的定义看起来和RFC4137关系不大。

注意 图4-24中eap\_sm第一个成员变量EAP\_State是一个枚举类型，其枚举值就是图4-21中SUPP SM的各个状态。

EAP SUPP SM初始化时，eap\_sm的eapol\_callbacks被设置为eapol\_cb对象。代码如下所示。

[-->eapol\_supp\_sm.c: : eapol\_cb定义]

```
static struct eapol_callbacks eapol_cb = {  
    eapol_sm_get_config, eapol_sm_get_bool, eapol_sm_set_bool, eapol_sm_get_int,  
    eapol_sm_set_int, eapol_sm_get_eapReqData, eapol_sm_set_config_blob,  
    eapol_sm_get_config_blob, eapol_sm_notify_pending, eapol_sm_eap_param_needed,  
    eapol_sm_notify_cert  
};
```

上述函数相关代码我们留待碰到它们时再介绍。

## (2) WPAS状态机通用宏

WPAS中有许多状态机，所以它定义了一些通用宏来帮助实现状态机相关的代码。这些宏的定义如下所示。

[-->state\_machine.h]

```
/*  
 * 定义一个状态的EA，它是一个函数声明。  
 * 而STATE_MACHINE_DATA也是一个宏。对于EAP SSM来说，其类型是struct  
 * eap_sm。  
 * global代表触发该状态的原因是否为UCT。  
 */  
#define SM_STATE(machine, state) \  
    static void sm_## machine ## _## state ##  
_Enter(STATE_MACHINE_DATA *sm, int global)  
  
// 每个状态进入后执行的一段代码。一般是打印一些信息，并设置新的状态  
#define SM_ENTRY(machine, state) \  
if (!global || sm->machine ## _state != machine ## _## state)
```

```

{ \
    sm->changed = TRUE; \// changed变量用于记录状态机的状态是否发生变化
    wpa_printf(MSG_DEBUG, STATE_MACHINE_DEBUG_PREFIX ": "
#machine \
                " entering state " #state); \
} \
sm->machine ## _state = machine ## _ ## state; \// 设置状态机的状态

// SM_ENTER宏对应一次函数调用，调用的是SM_STATE宏定义的函数
#define SM_ENTER(machine, state) \
sm ## machine ## _ ## state ## _Enter(sm, 0)
// 对应一次函数调用，表示因UCT而直接进入某个状态
#define SM_ENTER_GLOBAL(machine, state) \
sm ## machine ## _ ## state ## _Enter(sm, 1) \// 这个函数由SM_STATE宏声明

// 运行状态机。该宏定义一个函数
#define SM_STEP(machine) \
static void sm ## machine ## _Step(STATE_MACHINE_DATA *sm)
// 该宏对应一次函数调用，即sm##machine##Step(sm)，sm参数由调用函数内声明
#define SM_STEP_RUN(machine) sm ## machine ## Step(sm)

```

下面通过SUPP SM的实现代码来认识下上述通用宏的用法。

### (3) EAP SUPP SM的实现

SUPP SM的状态比较多，此处仅列举DISABLED状态的实现代码以帮助读者理解通用宏的作用。对状态机来说，其状态对应的EA非常重要。如下代码所示为DISABLED状态对应的EA。根据上节对通用宏的介绍，EA由SM STATE宏来定义。

```
[-->eap.c: : SM_STATE(EAP, DISABLED) ]  
  
/*  
该宏对应的代码是  
static void sm_EAP_DISABLED_Enter(STATE_MACHINE_DATA *sm, int  
global)  
*/  
SM_STATE(EAP, DISABLED)  
{  
    SM_ENTRY(EAP, DISABLED); // 每个状态的EA都会执行  
    SM_ENTRY代码段
```

```

/*
SM_ENTRY宏对应的代码是:
if (!global || sm->EAP_state != EAP_DISABLED) {
    sm->changed = TRUE;
    wpa_printf(MSG_DEBUG, STATE_MACHINE_DEBUG_PREFIX ":" "
"EAP" \
                " entering state " "DISABLED");           // 这段日志对了解
SUPP SM当前处于哪个状态非常重要
}
// EAP_state是eap_sm中成员变量。读者可参考图4-24
sm->EAP_state = EAP_DISABLED;
*/
sm->num_rounds = 0;
}

```

SM\_STATE只是定义了状态机某个状态的EA，那么状态机是如何运作的呢？根据图4-21以及前文所述，状态机的状态切换主要是通过判断条件是否满足来完成。SM\_STEP定义的函数就是用于检查状态机的这些条件变量，然后根据情况进行状态转换的。SUPP SM的SM\_STEP宏对应的代码如下所示。

```

[-->eap.c: : SM_STEP (EAP) ]

/*
SM_STEP宏对应的函数定义为:
static void sm_EAP_Step(STATE_MACHINE_DATA *sm)
*/
SM_STEP (EAP)
{
    /*
    对应UCT的处理。eapol_get_bool函数将调用eapol_callbacks对象中的
    eapol_sm_get_
    bool函数其内部返回eap_sm中eapRestart成员变量的值。
    */
    if (eapol_get_bool(sm, EAPOL_eapRestart) &&
        eapol_get_bool(sm, EAPOL_portEnabled))
    /*
    调用由SM_STATE (EAP, INITIALIZE) 定义的函数，以进入EAP_INITIALIZE状
    态。
    GLOBAL的意思是UCT。请读者注意此处的判断条件：如果eap_sm的eapRestart
    和
    portEnabled成员变量都为true，则直接进入INITIALIZE状态。它完全和图4-
    21
    SUPP SM状态图一样。
    */
}

```

```

        */
        SM_ENTER_GLOBAL(EAP, INITIALIZE);
        else if (!eapol_get_bool(sm, EAPOL_portEnabled) || sm-
>force_disabled)
            SM_ENTER_GLOBAL(EAP, DISABLED);
        else if (sm->num_rounds > EAP_MAX_AUTH_ROUNDS) {
            if (sm->num_rounds == EAP_MAX_AUTH_ROUNDS + 1) {
                .....// 有一些EAP方法在认证错误时会有很多消息往来，WPAS对
此做了一个限制
                // 一旦这些错误消息往来超过50次（由EAP_MAX_AUTH_ROUNDS），
则直接进入FAILURE状态
                sm->num_rounds++;
                SM_ENTER_GLOBAL(EAP, FAILURE);           // GLOBAL代表UCT
的情况
            }
        } else    eap_peer_sm_step_local(sm);          // 对应其他非UCT
的情况
    }
}

```

此处简单看一下eap\_peer\_sm\_step\_local的代码。

[-->eap.c: : eap\_peer\_sm\_step\_local]

```

static void eap_peer_sm_step_local(struct eap_sm *sm)
{
    switch (sm->EAP_state) {
        .....
        case EAP_IDLE:
            // 图4-21中, idle状态可依据条件不同而跳转到其他多个状态
            // 下面这个函数用于选择目标状态及跳转到它
            eap_peer_sm_step_idle(sm); // 根据图4-21的idle状态跳转，读者
能想象出该函数的代码实现吗
            break;
        case EAP_RECEIVED:
            eap_peer_sm_step_received(sm);
            break;
        case EAP_GET_METHOD:
            if (sm->selectedMethod == sm->reqMethod)
                SM_ENTER(EAP, METHOD);           // 直接进入
METHOD状态
            else
                SM_ENTER(EAP, SEND_RESPONSE);   // 直接进入
SEND_RESPONSE状态
            break;
    }
}

```

```
    ....  
}
```

eap\_peer\_sm\_step\_local用于处理那些非UCT导致的状态切换。

**提示** EAP SUPP SM的代码虽不复杂，但由于SUPP SM状态和触发条件（即定义的那些变量）太多，想通过看代码去跟踪SUPP SM间的状态跳转是一件非常困难的事情。相比而言，图4-21比代码要直观，更加容易把注意力集中在目标状态以及它对应的EA上。另外，SM\_STATE代码段中包含的那段wpa\_printf输出将告知EAP模块当前的状态。读者以后在分析WPAS日志时千万要注意。

### 3. EAP SUPP SM总结

EAP SUPP SM基于RFC4137而实现，其内部变量的定义以及状态切换逻辑都来源于规范。以笔者的经验来看，掌握RFC4137是理解WPAS中EAP SUPP SM实现的基石。另外，对具体的处理逻辑而言，SUPP SM最重要的内容还是各个状态对应的EA。正如前文所述，图4-21对SUPP SM的运行极为重要，希望读者认真学习。

另外，对WPAS中状态机的实现来说，SM\_STATE用于定义某个状态的EA（即一个函数）。每个EA都会执行SM\_ENTRY宏定义的一段代码。SM\_ENTER和SM\_ENTER\_GLOBAL宏用于调用SM\_STATE定义的函数。GLOBAL代表UCT的情况。SM\_STEP宏用于运行整个状态机。请读者注意它和SM\_ENTER的区别。SM\_ENTER宏将直接调用某个指定状态（由SM\_ENTER宏的参数决定）的EA，而SM\_STEP则将根据SM中的变量情况来决定下一个要跳转的状态，然后调用它的SM\_ENTER。

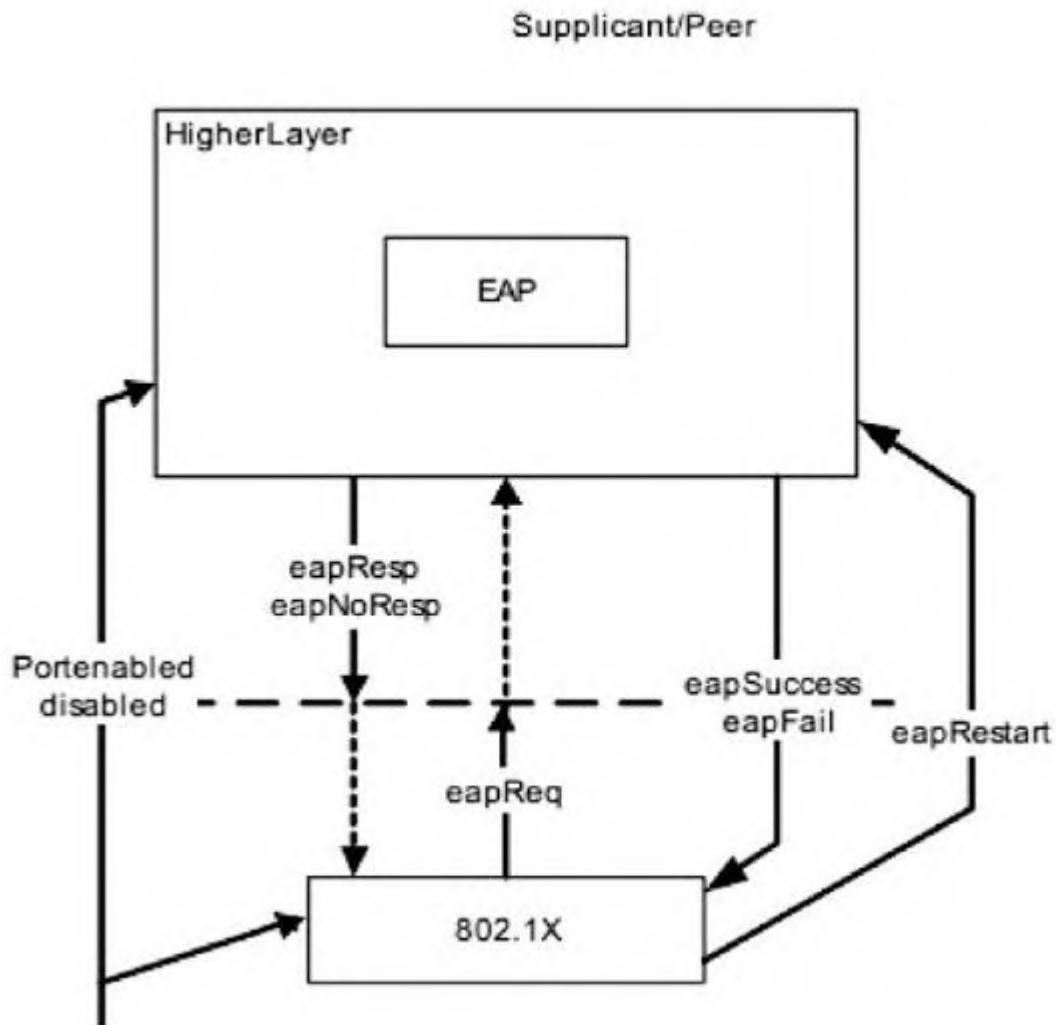
下面来看EAPOL模块的实现。

#### 4.4.2 EAPOL模块分析[21]

同EAP模块类似，EAPOL模块的实现参考了另外一个规范，即IEEE 802.1X。

注意 参考IEEE 802.1X 2004版规范的主要原因是，WPAS中EAPOL模块也基于该版本的规范。另外，笔者比较了2004版和2010版的802.1X，发现2004版的内容组织相对清晰易读。

在介绍802.1X前，先来看其描述的EAP和EAPOL之间的关系，如图4-25所示。



## 图4-25 EAP和EAPOL的关系

根据上一节对EAP SUPP SM的介绍，读者会发现图4-25中所示的eapResp、eapSuccess等变量就是RFC4137中定义的用于LL层和EAP SUPP SM层交互的变量。很明显，802.1X模块（在WPAS中，它就是EAPOL模块）是EAP SUPP的LL层（参考图4-22）。

另外一个可能会让读者感到惊奇的是，802.1X规范为EAPOL Supplicant定义了5个不同的状态机，分别如下。

- Port Timers SM：Port超时控制状态机。Port的概念请参考3.3.7节802.1X介绍。
- Supplicant PAE SM：PAE是Port Access Entity的缩写。该状态机用于维护Port的状态。
- Supplicant Backend SM：规范并没有明示该状态机的作用。但笔者觉得它主要用于给Authenticator发送EAPOL回复消息。
- The Key Receiver SM：用于处理Key（指EAPOL-Key帧）相关流程的状态机。
- The Supplicant Key Transmit SM：该状态机非必选项，所以WPAS未实现它。

说实话，EAPOL Supplicant定义5个状态机确实有些复杂。主要原因是这5个状态机相互之间都有关联，这些关联体现在它们可能都受同一个变量的影响，从而导致各自的状态发生变化。例如Port Timers SM修改了一些变量后，就有可能使得其他状态机的状态发生变化。规范中把这些变量成为全局变量（Global Variables）。

### 规范阅读提示

1) 除了SUPP包含的这五个状态机外，规范还为Authenticator定义了四个状态机。Authenticator也需要实现Port Timers SM和The Key Receiver SM。

2) 规范中将这些状态机统称为PACP（Port Access Control Protocol）State Machine。

下面，先来认识这些全局变量。

## 1. EAPOL SUPP全局变量

802.1X定义了一些全局变量，它们被多个状态机使用。这些全局变量的定义如表4-6所示。

表 4-6 SUPP 全局变量定义

变 量 名	类 型	说 明
eapolEap	boolean	收到一个类型为 EAPOL 类型为 EAP-Packet 包时，该值为 TRUE
initialize	boolean	当该值为 TRUE，所有状态机将被强制恢复初始状态
portControl	枚举	该枚举变量有三个值可选： <ul style="list-style-type: none"><li>• ForceUnauthorized：强制 controlled port 处于未授权状态</li><li>• ForceAuthorized：强制 controlled port 处于授权状态</li><li>• Auto：根据 supplicant 认证的结果，controlled port 处于授权或未授权状态</li></ul>
portValid	boolean	802.1X 没有说明这个值什么时候会被设置成 TRUE 或 FALSE。但对 802.11 无线网络来说，当 STA 和 AP 关联成功，并且成功交换了 key 信息后（即密钥派生成功），该值为 TRUE。该变量为 TRUE 时，表示通过此 port 往来的数据都是安全的（即加密了）。它和 keyDone 联合使用
keyDone	boolean	该变量和 portValid 变量关系密切，二者组合起来对应四种情况： <ul style="list-style-type: none"><li>• portValid 和 keyDone 都为 TRUE：port 处于有效状态（valid state）</li><li>• portValid 为 TRUE，keyDone 为 FALSE：port 处于 unknown state，原因是 key 相关的 SM 还未运行</li><li>• portValid 为 FALSE，keyDone 为 TRUE：port 处于 unknown state，原因是 key 处理失败</li><li>• portValid 为 FALSE，keyDone 为 FALSE：port 处于 unknown state，原因是 key 相关的 SM 还未运行</li></ul>
keyRun	boolean	外部设置该值为 TRUE 以触发 Transmit Key SM。否则设置它为 FALSE 以停止 Transmit Key SM
suppAbort	boolean	PAE SM 设置该值为 TRUE 以通知 BE SM 停止认证流程
suppFail	boolean	BE SM 设置该值为 FALSE，表示认证失败
suppPortStatus	枚举	PAE SM 用该值表示当前的授权状态，可取值有 Unauthorized 和 Authorized
suppStart	boolean	PAE SM 设置该值为 TRUE 用来通知 BE SM 开始认证工作
suppSuccess	boolean	BE SM 成功完成认证工作后设置该值为 TRUE
suppTimeout	boolean	BE SM 认证超时的话将设置该值为 TRUE

注意，表4-6省略了部分和Authenticator相关的全局变量。另外，规范还定义了一些全局超时变量，它们将在Port Timers SM中介绍。

## 2. SUPP PACP状态机

### (1) Port Timers SM

Port Timers SM (PT SM) 对应的状态切换如图4-26所示。

PT SM的功能比较简单，就是每一秒触发一次以从ONE\_SECOND状态进入TICK状态。TICK状态的EA中，它将递减（图4-26中的dec函数）某些变

量的值。

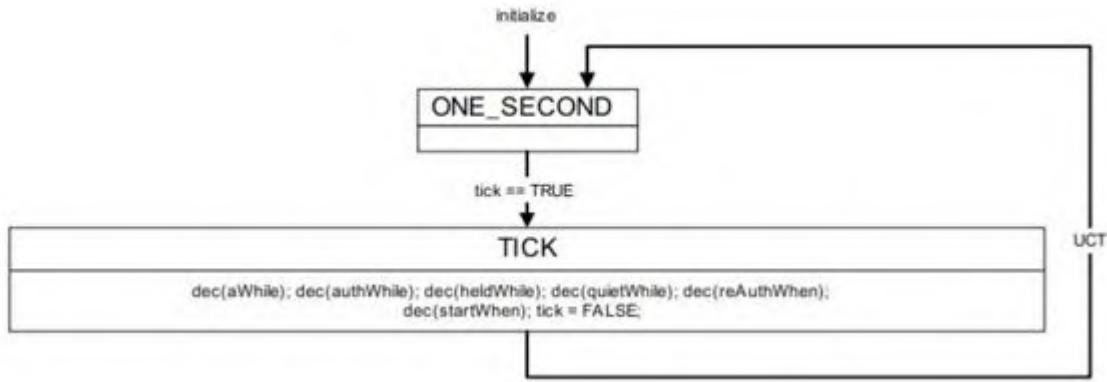


图4-26 PT SM状态切换

注意，PT SM在SUPP和AUTH两端都有。所以，图4-26中的一些变量只用于AUTH端。这些变量的含义如表4-7所示。

表 4-7 PT SM 变量

变 量 名	类 型	说 明
aWhile	integer	初始值为 serverTimeout，和 BE Authentication SM 有关，该值递减到 0 时，表明认证超时
authWhile	integer	初始值为 authPeriod，该值递减到 0 时，PAE SM 等待来自 Authenticator EAPOL-Request 包将超时
heldWhile	integer	在 heldWhile 时间内，Supplicant 将暂时不会与 Authenticator 交互。初始值为 heldPeriod
quietWhile	integer	和 heldWhile 相对应，quietWhile 用于 Authenticator 端。初始值为 quietPeriod
reAuthWhen	integer	用于 Authenticator 的 Reauthentication Timer SM
startWhen	integer	初始值为 startPeriod，该值递减到 0 时将触发 SUPP PAE SM 发送 EAPOL-Start 包

虽然规范定义了PT SM，但WPAS中，PT SM的功能并不是通过状态机宏来实现的，而仅仅是向eloop模块注册了一个超时时间为1秒的函数eapol\_port\_timers\_tick，其代码如下所示。

[-->eapol\_supp\_sm.c: : eapol\_port\_timers\_tick]

```
static void eapol_port_timers_tick(void *eloop_ctx, void *timeout_ctx)
{
    struct eapol_sm *sm = timeout_ctx;
    if (sm->authWhile > 0) { // 处理authWhile
        sm->authWhile--;
        if (sm->authWhile == 0)
```

```

        wpa_printf(MSG_DEBUG, "EAPOL: authWhile --> 0");
    }
    // 处理heldWhile, startWhen, idleWhile (idleWhile见表4-1)
    .....
    if (sm->authWhile | sm->heldWhile | sm->startWhen | sm-
>idleWhile) {
        // 重新注册超时处理函数, 相当于切换到图4-26中的ONE_SECOND状态
        eloop_register_timeout(1, 0, eapol_port_timers_tick,
        eloop_ctx, sm);
    } else {
        sm->timer_tick_enabled = 0;
    }
    eapol_sm_step(sm); // 处理其他状态机的状态切换, 此函数内容下文会介绍
}

```

上述代码中，eapol\_port\_timers\_tick除了递减相关变量外，最后还需要调用eapol\_sm\_step函数以判断其他状态机是否需要切换状态。这是PT SM和其他状态机联动的关键纽带，而这个纽带在规范中并不能直接体现出来（规范中，PT SM只是修改某些变量，至于其他状态机到底怎么被触发，则没有说明。而eapol\_port\_timers\_tick函数修改完变量后，直接调用eapol\_sm\_step函数完成了对其他状态机的检查）。

下面来看第二个状态机The Key Receiver SM。

## (2) The Key Receiver SM

图4-27所示为The Key Receiver SM（以后简称TKR SM）状态切换图。主要有两点值得关注。

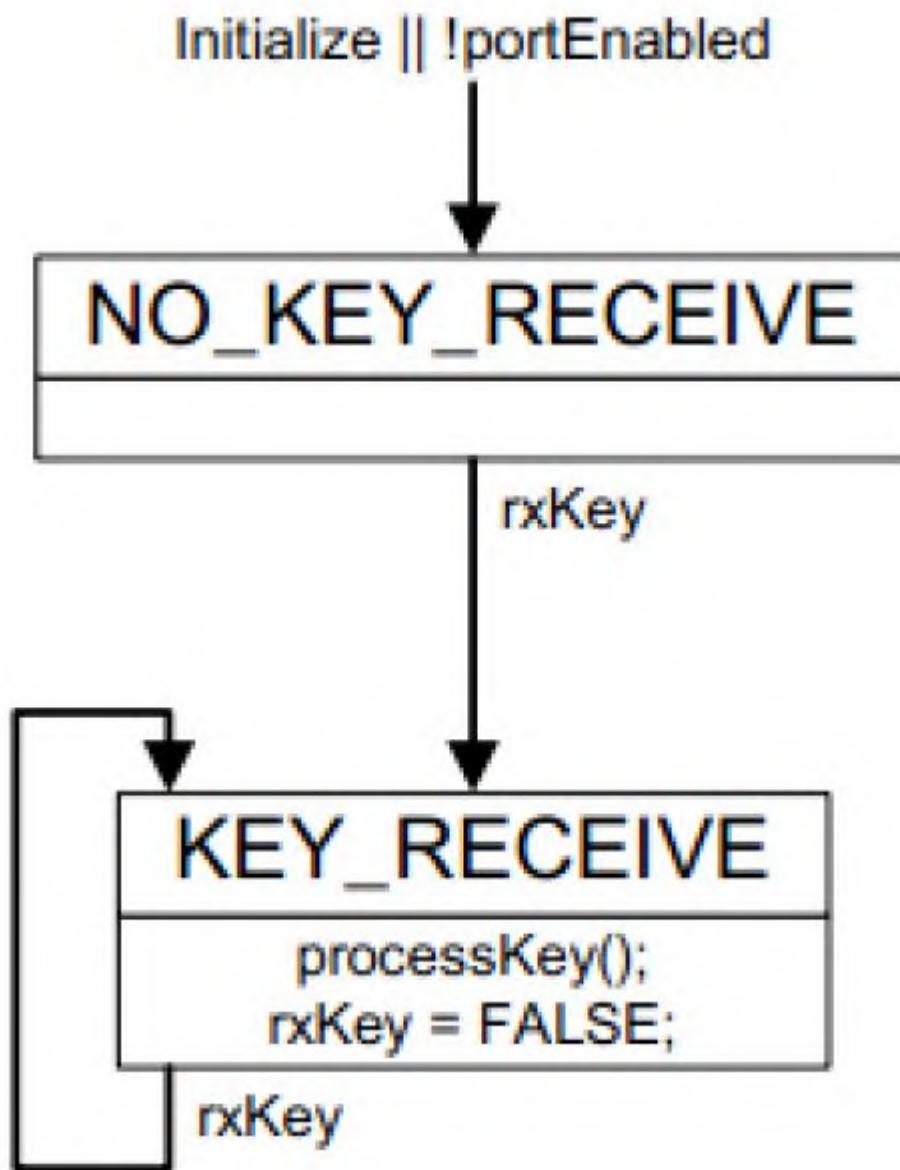


图4-27 TKR SM状态切换

- TKR SM包含两个状态。第一个是NO\_KEY\_RECEIVE状态。当rxKey (boolean型变量, 当Supplicant收到EAPOL Key帧后, 该值为TRUE) 变为TRUE时, TKR进入KEY\_RECEIVE状态。
- TKR在KEY\_RECEIVE状态时需要调用processKey函数处理EAPOL Key消息。

WPAS中, TKR的代码也非常简单, 如下所示。

[-->eapol\_supp\_sm.c: : TRK SM相关函数]

```
SM_STATE(KEY_RX, NO_KEY_RECEIVE)
{
    SM_ENTRY(KEY_RX, NO_KEY_RECEIVE);
}

SM_STATE(KEY_RX, KEY_RECEIVE)
{
    SM_ENTRY(KEY_RX, KEY_RECEIVE);
    eapol_sm_processKey(sm); // 对应图4-27所示
    的processKey函数
    sm->rxKey = FALSE;
}

SM_STEP(KEY_RX) // TKR状态机状态切换函数
{
    if (sm->initialize || !sm->portEnabled)
        SM_ENTER_GLOBAL(KEY_RX, NO_KEY_RECEIVE); // 直接
    进入NO_KEY_RECEIVE状态
    switch (sm->KEY_RX_state) {
        case KEY_RX_UNKNOWN:
            break;
        case KEY_RX_NO_KEY_RECEIVE:
            if (sm->rxKey)
                SM_ENTER(KEY_RX, KEY_RECEIVE);
            break;
        case KEY_RX_KEY_RECEIVE:
            if (sm->rxKey)
                SM_ENTER(KEY_RX, KEY_RECEIVE);
            break;
    }
}
```

TKR SM的代码非常简单，此处不详述。下面来看PAE SM。

### (3) PAE SM

PAE SM比较复杂，其状态切换如图4-28所示。

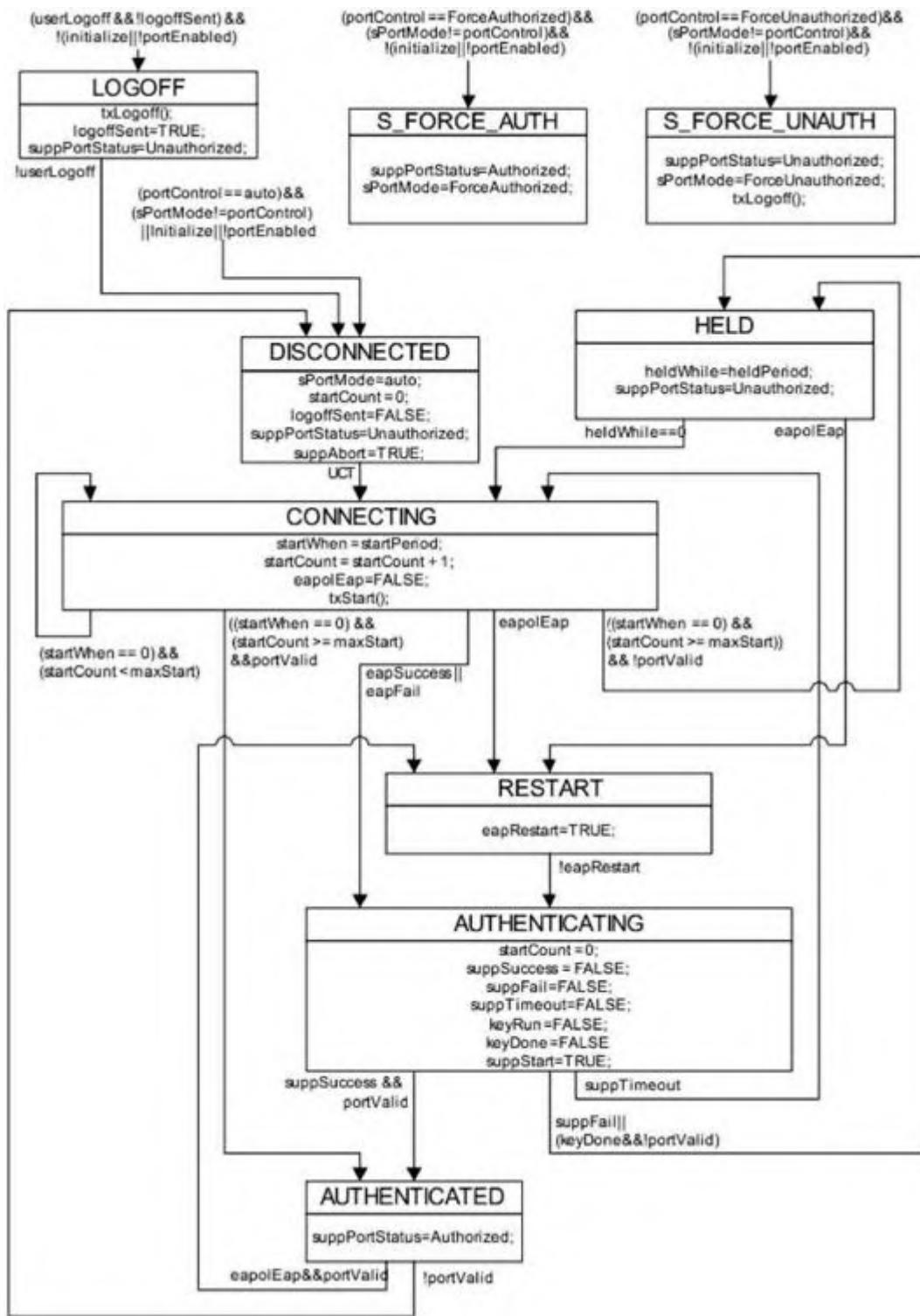


图4-28 PAE SM状态切换

图4-28中涉及的变量定义见表4-8。

表 4-8 PAE SM 变量定义

变 量 名	类 型	说 明
logoffSent	integer	表示 EAPOL-Logoff 包已发送
sPortMode	枚举类型	取值同表 4-6 的 PortControl
startCount	integer	根据规范, SUPP 向 AUTH 发送 EAPOL-Start 消息用于触发整个认证流程。不过, AUTH 有时候并不能及时响应 EAPOL-Start 消息(例如, AUTH 可能就不存在)。该变量用于统计 SUPP 收到 AUTH 响应之前, 发了多少次 EAPOL-Start 消息。如果次数超过 maxStart, 则停止认证
userLogoff	integer	由外部设置, 用于告知 PAE SM 需要 logoff
heldPeriod	integer	和表 4-7 中的 heldWhile 对应, 该值默认为 60s
startPeriod	integer	和表 4-7 中的 startWhen 对应, 该值默认为 30s
maxStart	integer	见 startCount 的解释

图4-28还包括两个函数。

- txStart: 用于发送EAPOL-Start消息给Authenticator。
- txLogoff: 用于发送EAPOL-Logoff消息给Authenticator。

提示 PAE SM中的状态虽然较多, 但笔者觉得它们的划分似乎并无泾渭分明的根据。另外, 规范对它们的描述也仅是说明满足什么条件将进入什么状态。至于为什么划分这么多状态也没有太多可参考的依据。所以, 读者也不必拘泥于求根究底了, 只要把握图4-28即可。

WPAS中, PAE SM相关的代码也比较简单, 此处仅看LOGOFF状态的EA, 如下所示。

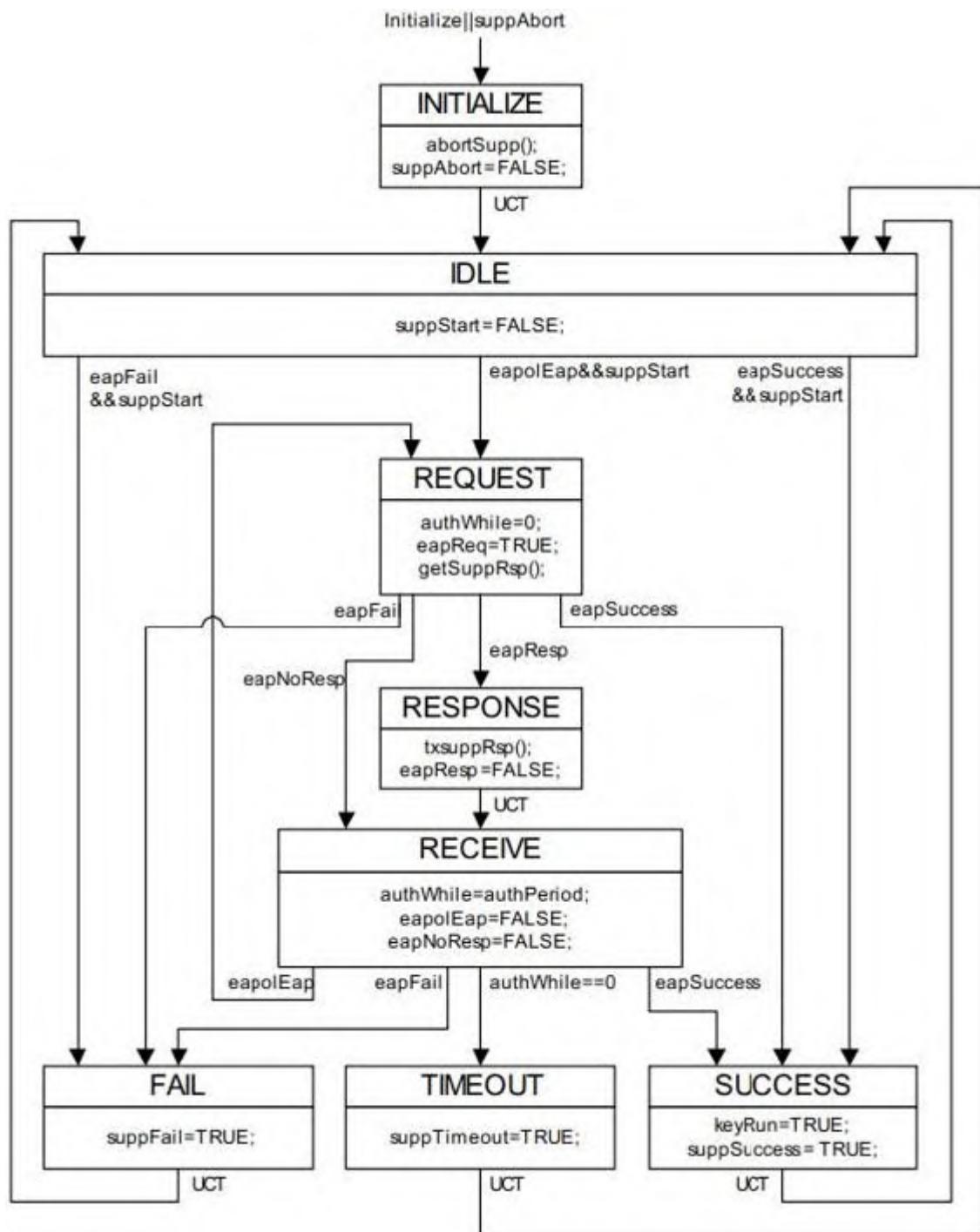
```
[-->eapol_supp_sm.c: : SM_STATE (SUPP_PAE, LOGOFF) ]

SM_STATE (SUPP_PAE, LOGOFF)           // 状态机名为SUPP_PAE, 状态名为
LOGOFF
{
    SM_ENTRY (SUPP_PAE, LOGOFF);
    eapol_sm_txLogoff (sm);           // 对应图4-28中的txLogoff
函数
    sm->logoffSent = TRUE;
    sm->suppPortStatus = Unauthorized;
    // 这个函数内部将通过N180211 API设置WLAN Driver的状态
    // 属于EAPOL模块和WPAS中其他模块的交互处理
    eapol_sm_set_port_unauthorized (sm);
}
```

#### (4) Backend SM

Backend SM (BE SM) 的状态转换如图4-29所示。

需要介绍和BE SM相关的变量authPeriod，它和authWhile（见表4-7）有关，默认值为30秒。



## 图4-29 BE SM状态切换

BE SM包含如下几个重要函数。

- `abortSupp`: 停止认证工作，释放相关的资源。
- `getSuppResp`: 这个函数本意是用来获取EAP Response信息的，然后用`txSuppResp`函数发送出去。但WPAS中，该函数没有包括任何有实质意义的内容。
- `txSuppResp`: 发送EAPOL-Packet包给Authenticator。

BE SM的部分代码如下所示。

```
[-->eapol_supp_sm.c: : SM_STATE (SUPP_BE, REQUEST) ]  
  
SM_STATE(SUPP_BE, REQUEST) // REQUEST状态对应的EA  
{  
    SM_ENTRY(SUPP_BE, REQUEST);  
    sm->authWhile = 0;  
    sm->eapReq = TRUE;  
    eapol_sm_getSuppRsp(sm); // 此函数内部并无任何有实质意义的内  
容, 读者不妨自行阅读它  
}
```

**提示** 前面几节介绍了802.1X中SUPP PACP几个状态机相关的知识。相比EAP SUPP SM而言，虽然PACP状态机的个数增加了不少，但每个状态机包含的状态却少了许多，所以PACP状态机反而容易理解。

有了理论知识后，马上来看EAPOL SUPP模块中的几个重要数据结构和函数。

### 3. EAPOL SUPP代码分析

图4-23和图4-24介绍了EAPOL和EAP模块的关系，那么EAPOL和WPAS其他模块是什么关系呢？相关数据结构如图4-30所示。

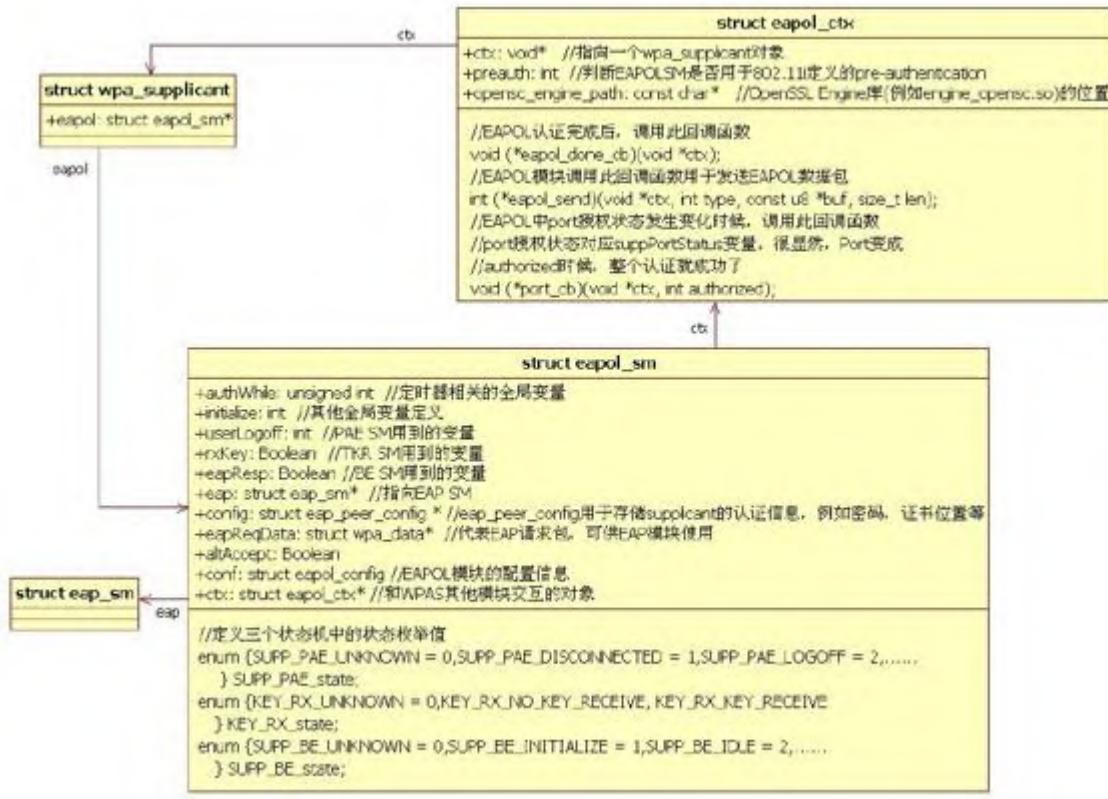


图4-30 WPAS中EAPOL/EAP模块数据结构

由图4-30可知，WPAS定义了一个数据结构eapol\_sm来存储和PACP状态机相关的内容。其内部定义了三个状态机（TKR SM、PAE SM和BE SM）各自的状态信息（由三个状态枚举值表达）、相关变量等。EAPOL模块和WPAS中重要模块wpa\_supplicant的交互接口是通过结构体eapol\_ctx来定义的。EAPOL模块通过eap变量指向EAP模块的代表eap\_sm结构体。

**提示** eapol\_sm和eapol\_ctx实际包含的成员变量非常多，此处仅列举其中一部分。

虽然WPAS包括EAPOL和EAP两个模块，但WPAS其他模块一般只和EAPOL模块交互。至于EAP模块，它的操作（例如EAP的初始化以及EAP SUPP SM的运作）则由EAPOL模块来触发。

### (1) EAPOL模块的初始化

先来看EAPOL和EAP模块的初始化函数，由wpa\_supplicant\_init\_eapol函数完成，代码如下所示。

```

[-->wpas_glue.c: : wpa_supplicant_init_eapol]

int wpa_supplicant_init_eapol(struct wpa_supplicant *wpa_s)
{
#ifdef IEEE8021X_EAPOL
    struct eapol_ctx *ctx;
    ctx = os_zalloc(sizeof(*ctx));
    .....
    ctx->ctx = wpa_s;
    ctx->msg_ctx = wpa_s;
    ctx->eapol_send_ctx = wpa_s;
    ctx->preauth = 0;
    ctx->eapol_done_cb = wpa_supplicant_notify_eapol_done;
    ctx->eapol_send = wpa_supplicant_eapol_send;
    .....// 其他eapol_ctx成员变量的初始化
    ctx->wps = wpa_s->wps;
    ctx->eap_param_needed = wpa_supplicant_eap_param_needed;
    ctx->port_cb = wpa_supplicant_port_cb;
    ctx->cb = wpa_supplicant_eapol_cb;
    ctx->cert_cb = wpa_supplicant_cert_cb;
    ctx->cb_ctx = wpa_s;
    wpa_s->eapol = eapol_sm_init(ctx);           // 初始化EAPOL模
块
    .....
#endif /* IEEE8021X_EAPOL */
    return 0;
}

```

wpa\_supplicant\_init\_eapol首先设置eapol\_ctx对象，然后调用eapol\_sm\_init来完成EAPOL模块的初始化。eapol\_sm\_init的代码如下所示。

```

[-->eapol_supp_sm.c: : eapol_sm_init]

struct eapol_sm *eapol_sm_init(struct eapol_ctx *ctx)
{
    struct eapol_sm *sm;
    struct eap_config conf;
    sm = os_zalloc(sizeof(*sm));           // EAPOL对应的状态机信息
    .....
    sm->ctx = ctx;// eapol_ctx是WPAS中EAPOL模块和其他模块交互的接口
    sm->portControl = Auto;
    sm->heldPeriod = 60;
    sm->startPeriod = 30;

```

```

sm->maxStart = 3;
sm->authPeriod = 30;
os_memset(&conf, 0, sizeof(conf));
conf.opensc_engine_path = ctx->opensc_engine_path;
.....
conf.wps = ctx->wps;
// 初始化EAP Supplicant SM相关资源
sm->eap = eap_peer_sm_init(sm, &eapol_cb, sm->ctx->msg_ctx,
&conf);
.....
// 先设置initialize变量为TRUE，然后初始化相关状态
sm->initialize = TRUE;
eapol_sm_step(sm);
sm->initialize = FALSE; // 设置为FALSE，再初始化相关状态
eapol_sm_step(sm);
sm->timer_tick_enabled = 1;
eloop_register_timeout(1, 0, eapol_port_timers_tick, NULL,
sm);

return sm;
}

```

在eapol\_sm\_init代码中：

- 1) 先通过调用eap\_peer\_sm\_init初始化EAP SUPP SM相关资源。
- 2) 然后完成EAPOL PACP三个状态机的初始化工作。初始化的方法很简单，即先设置initialize为TRUE，然后执行eapol\_sm\_step函数（该函数代码见下文，其主要目的是根据条件以跳转到下一个状态。initialize为TRUE时，将触发一些状态的EA被调用，从而某些变量的初值将被设定）。然后设置initialize为FALSE后，再度执行eapol\_sm\_step函数（这样，对应状态的EA也将被执行，从而剩余变量的初值将被设定）。
- 3) 最后通过注册一个eloop超时任务实现了PT SM。

## (2) 状态机的联动

根据前面的介绍，EAPOL和EAP一共有四个状态机，它们到底是怎么联动的呢？答案就在eapol\_sm\_step中。eapol\_sm\_step的代码如下所示。

[-->eapol\_supp\_sm.c: : eapol\_sm\_step]

```
void eapol_sm_step(struct eapol_sm *sm)
{
```

```
    int i;
```

```
/*
```

笔者一直很好奇EAPOL和EAP中的四个状态机是怎么联动的。通过下面的代码可知，

根据EAPOL和EAP的关系，首先要运行EAPOL中的三个状态机（Port Timers SM由eloop定时任务

来实现），分别是SUPP\_PAЕ、KEY\_RX和SUPP\_BE。然后执行EAP\_SUPP SM。如果changed变量

为TRUE，表示状态发生了切换。由于每个状态对应的EA又有可能改变其中一些变量从而引起其他状

态机状态发生变化，所以，这里有一个for语句来循环处理状态切换，直到四个状态机都没有状态切

换为止。一般情况下，for循环应该是一个无限循环，但此次通过100来控制循

环次数，是为了防止某些情况下状态机陷入死循环而不能退出（这也说明规范中定义的SM在联动时可能有逻辑错误）。

```
*/
```

```
    for (i = 0; i < 100; i++) {
        sm->changed = FALSE;
        SM_STEP_RUN(SUPP_PAЕ);
        SM_STEP_RUN(KEY_RX);
        SM_STEP_RUN(SUPP_BE);
        if (eap_peer_sm_step(sm->eap)) // eap_peer_sm_step返回非零，表示状态有变化
            sm->changed = TRUE;
        if (!sm->changed) break; // 如果没有状态变化，则跳出循环
    }
    /*
```

运行超过100次，需要重新启动EAPOL模块状态机运行，  
eapol\_sm\_step\_timeout将重新调用

```
    eapol_sm_step函数。
*/
    if (sm->changed) {
        eloop_cancel_timeout(eapol_sm_step_timeout, NULL, sm);
        eloop_register_timeout(0, 0, eapol_sm_step_timeout,
NULL, sm);
    }
    /*
```

cb\_status是一个枚举类型的变量，可取值有EAPOL\_CB\_IN\_PROGRESS，  
EAPOL\_CB\_SUCCESS和

```

    EAPOL_CB_FAILURE。
*/
if (sm->ctx->cb && sm->cb_status != EAPOL_CB_IN_PROGRESS) {
    int success = sm->cb_status == EAPOL_CB_SUCCESS ? 1 :
0;
/*
该值在PAE AUTHENTICATED状态中被置为EAPOL_CB_SUCCESS，表示认
证成功。在PAE
    HELD状态被置为EAPOL_CB_FAILURE，表示认证还未成功。
*/
sm->cb_status = EAPOL_CB_IN_PROGRESS;
// 回调通知WPAS，真实的函数是wpa_supplicant_eapol_cb，这个函
数以后介绍
    sm->ctx->cb(sm, success, sm->ctx->cb_ctx);
}
}

```

WPAS中状态机联动代码的实现非常巧妙，它通过循环来处理各个状态机的状态变换，直到四个状态机都稳定为止。

**注意** 初始化结束后，各个状态机的状态为：SUPP\_PAE为DISCONNECTED状态、KEY\_RX为NO\_KEY\_RECEIVE状态、SUPP\_BE为IDLE状态、EAP\_SM为DISABLED状态。

至此，对FRC4137和IEEE 802.1X-2004协议中EAP Supplicant和EAPOL Supplicant涉及的状态机进行了详细介绍。

在具体实现中，WPAS实现的EAPOL和EAP状态机较为严格得遵循了这两个文档。所以，读者只要理解了协议中的状态切换和相关变量，则能轻松理解WPAS的实现。反之，如果仅单纯从代码入手，EAPOL/EAP状态机的代码将会非常难以理解。

关于EAPOL/EAP状态机相关的知识就介绍到这，以后碰到具体代码时，读者根据状态切换图直接进入某个状态中去看其处理函数（即EA）。

**提示** 本章第二条分析路线使用的目标AP采用WPA2-PSK作为认证算法，故后续章节不会涉及太多和EAPOL及EAP相关的代码分析。感兴趣的读者可在本节基础上，自行搭建RAIDUS服务器来研究WPAS中EAPOL/EAP的工作过程。

## 4.5 wpa\_supplicant连接无线网络分析

本节将介绍第二条分析路线，即通过命令行发送命令的方式触发wpa\_supplicant进行相关工作，使手机加入一个利用WPA-PSK进行认证的无线网络。以笔者的Note 2为例，整个过程用到的命令如下所示。

### [命令示例]

```
adb root #获取手机root用户权限。只有root被破解的手机才能成功
adb shell #登录手机shell
#笔者事先已编译wpa_cli并将其放到/system/bin目录中。这个命令用于启动
wpa_cli, -i参数指明unix域控制
#wsocket文件名，它应该和wpa_supplicant启动时设置的控制接口文件名一致
wpa_cli -iwlan0 #该命令执行后，将进入wpa_cli进程，后续操作都在此进程中
开展
#发送ADD_NETWORK命令给wpa_supplicant，它将返回一个新网络配置项的编号
#请参考4.3.3节wpas_ssid结构体介绍
ADD_NETWORK #假设wpa_supplicant返回的新网络配置项编号为0
SET_NETWORK 0 ssid "Test" #设置0号网络的ssid为"Test"
SET_NETWORK 0 key_mgmt WPA-PSK #设置0号网络的key_mgmt为"WPA-PSK"
SET_NETWORK 0 psk "12345Test" #设置0号网络的psk为"12345Test"
ENABLE_NETWORK 0 #使能0号网络，它将触发wpa_supplicant扫描、关联等一系列操作直到加入无线网络"Test"
CTRL+C #退出wpa_cli
dhcpcd wlan0 #启动dhcpcd，wlan0为无线接口设备名。dhcpcd可为手机从AP那
获取一个IP地址
```

dhcpcd成功执行后，手机将从AP那分配到一个IP地址。至此，手机就可以使用"Test"无线网络了。

注意 上述命令执行前有几个注意事项。

1) 先要在Settings中开启无线网络。这个操作完成了wlan驱动及相应固件加载的工作。该工作实际上由netd来完成，而wpa\_cli无法完成它。

2) 开启无线网络后，WifiService和wpa\_supplicant都开始工作了。为了避免WifiService的干扰，可以把Settings中的那些已知的无线网

络信息都清除。

3) 由于wpa\_supplicant支持多个客户端，所以wpa\_cli可以和WifiService共同工作。只要不操作Settings中无线网络相关的选项，WifiService就不会干扰wpa\_cli。

4) 然后按上述步骤执行wpa\_cli。

根据前文所述，所有来自客户端的命令都由wpa\_supplicant\_ctrl\_iface\_receive函数处理（参考4.3.4节）。该函数代码非常简单，就是根据客户端发送的命令进行对应处理。

```
[-->ctrl_iface_unix.c: : wpa_supplicant_ctrl_iface_receive]

static void wpa_supplicant_ctrl_iface_receive(int sock, void
*eloop_ctx,
                               void *sock_ctx)
{
    struct wpa_supplicant *wpa_s = eloop_ctx;
    struct ctrl_iface_priv *priv = sock_ctx;
    char buf[4096]; int res; struct sockaddr_un from;
    socklen_t fromlen = sizeof(from);
    char *reply = NULL; size_t reply_len = 0; int new_attached
= 0;

    res = recvfrom(sock, buf, sizeof(buf) - 1, 0, (struct
sockaddr *) &from, &fromlen);
    .....
    buf[res] = '\0';
    // 客户端第一次和WPAS连接时，需要发送"ATTACH"命令
    if (os_strcmp(buf, "ATTACH") == 0) {
        .....// 略过相关处理
    }.....// "DETACH"和"LEVEL"命令处理
    else {
#ifndef CONFIG_P2P && defined(ANDROID_P2P)
        .....// P2P处理。虽然WPAS编译时打开了CONFIG_P2P和ANDROID_P2P
              // 但本章不讨论P2P相关的内容
#endif
        // 大部分的命令处理都在wpa_supplicant_ctrl_iface_process函数中
        reply = wpa_supplicant_ctrl_iface_process(wpa_s,
buf, &reply_len);
    }
    if (reply) { // 回复客户端
```

```
        sendto(sock, reply, reply_len, 0, (struct sockaddr *)
&from, fromlen);
        os_free(reply);
    } .....
/*
Client成功ATTACH后，将通知EAPOL模块。因为有些认证流程需要用户的参与
(例如输入密码之类的)，
所以当客户端连接上后，EAPOL模块将判断是否需要和客户端交互。读者可阅读
eapol_sm_notify_ctrl_attached函数。
*/
if (new_attached)
    eapol_sm_notify_ctrl_attached(wpa_s->eapol);
}
```

如上述代码所示，绝大部分命令都由  
wpa\_supplicant\_ctrl\_iface\_process函数处理。下面将按顺序来分析  
其处理ADD\_NETWORK、SET\_NETWORK以及ENABLE\_NETWORK的代码。

#### 4.5.1 ADD\_NETWORK命令处理

```
[-->ctrl_iface.c: : wpa_supplicant_ctrl_iface_process]

char * wpa_supplicant_ctrl_iface_process(struct wpa_supplicant
*wpa_s, char *buf,
                                         size_t *resp_len)
{
    char *reply;
    const int reply_size = 4096;
    int ctrl_rsp = 0;
    int reply_len;
    .....
    reply = os_malloc(reply_size);
    .....
    // 开始命令处理
    .....
    else if (os_strcmp(buf, "ADD_NETWORK") == 0) {
        reply_len = wpa_supplicant_ctrl_iface_add_network(
wpa_s, reply, reply_size);
    }else if
        .....// 其他命令处理

    if (reply_len < 0) {// 命令处理出错
        os_memcpy(reply, "FAIL\n", 5);
        reply_len = 5;
    }
    .....
    *resp_len = reply_len;
    return reply;
}
```

ADD\_NETWORK的真正处理在wpa\_supplicant\_ctrl\_iface\_add\_network函数中，其代码如下所示。

```
[-->ctrl_iface.c: : wpa_supplicant_ctrl_iface_add_network]

static int wpa_supplicant_ctrl_iface_add_network(struct
wpa_supplicant *wpa_s,
                                         char *buf, size_t buflen)
{
    struct wpa_ssid *ssid;
    int ret;
```

```

/*
wpa_config_add_network返回一个wpa_ssid对象，读者还记得它吗？
wpa_ssid是无线网络
配置项在WPAS中的反映（请参考4.3.3节wpa_ssid结构体介绍）。
wpa_config_add_network
内部就是分配一个wpa_ssid对象，然后将其保存到一个链表中。注意，
wpa_config是wpa_supplicant。
conf在代码中的代表。所以，此处添加的无线网络信息将会保存到配置文件中，
以备下次使用。
*/
ssid = wpa_config_add_network(wpa_s->conf);
.....
wpas_notify_network_added(wpa_s, ssid);
ssid->disabled = 1; // disabled为1表示该无线网络未启用，需要通过
ENABLE_NETWORK来
启动它
// 设置该无线网络的默认配置项
wpa_config_set_network_defaults(ssid);
// 返回该网络的编号（由wpa_ssid的id变量表示。它在
wpa_config_add_network函数中被赋值）
ret = os_snprintf(buf, buflen, "%d\n", ssid->id);
.....
return ret;
}

```

上述代码比较简单，就是分配一个wpa\_ssid对象，然后设置它的一些默认属性。整个函数返回该wpa\_ssid对象的id，即它在链表中的顺序。

wpa\_ssid的默认属性对后续流程有一些影响，默认属性都是什么呢？来看看wpa\_config\_set\_network\_defaults函数，代码如下所示。

```
[-->config.c: : wpa_config_set_network_defaults]

void wpa_config_set_network_defaults(struct wpa_ssid *ssid)
{
    // 设置proto、pairwise_cipher、group_cipher以及key_mgmt的信息，
    // 读者还记得这些变量的含义吗
    // 请参考4.3.3节安全相关成员变量及背景知识介绍
    ssid->proto = DEFAULT_PROTO;
    ssid->pairwise_cipher = DEFAULT_PAIRWISE;
    ssid->group_cipher = DEFAULT_GROUP;
    ssid->key_mgmt = DEFAULT_KEY_MGMT;
#ifndef IEEE8021X_EAPOL

```

```
ssid->eapol_flags = DEFAULT_EAPOL_FLAGS; // EAP  
相关变量，见下文解释  
ssid->eap_workaround = DEFAULT_EAP_WORKAROUND;  
ssid->eap.fragment_size = DEFAULT_FRAGMENT_SIZE;  
#endif /* IEEE8021X_EAPOL */  
#ifdef CONFIG_HT_OVERRIDES  
.....// 和802.11n有关，本书不涉及  
#endif /* CONFIG_HT_OVERRIDES */  
}
```

上述代码中出现了三个和EAPOL相关的变量，此处简单介绍一下。

### (1) eapol\_flags

它和动态WEP key有关。只适用于非WPA安全环境中，可取值有三个，分别如下。

1：代码中定义为BIT (0)，表示需要为单播数据传输使用动态WEP Key，对应宏为

EAPOL\_FLAG\_REQUIRE\_KEY\_UNICAST。

2：代码中定义为BIT (1)，表示需要为组播数据传输使用动态WEP Key，对应宏为

EAPOL\_FLAG\_REQUIRE\_KEY\_BROADCAST。

3：单播和组播都使用动态WEP Key，对应宏为DEFAULT\_EAPOL\_FLAGS。

### (2) eap\_workaround

身份认证方法多种多样，而有些AS (Authenticator服务器) 并不严格遵守规范。该变量表示碰到这种情况时，WPAS是否可以采取“绕”(workaround本意是“变通”)过去的方式来对待这些AS。由于这种不严格的情况非常普遍，所以该值默认是1。

### (3) fragment\_size

该变量和EAPOL消息分片大小有关。默认的DEFAULT\_FRAGMENT\_SIZE大小为1398，表示EAPOL消息只要不超过这个大小，就不用对其进行分

片。

“ADD\_NETWORK”命令比较简单，它最终将返回给客户端对应的无线网络配置的编号。在本例中，它是0。

下面来看客户端通过“SET\_NETWORK”为该无线网络配置项设置参数的处理过程。

## 4.5.2 SET\_NETWORK命令处理

SET\_NETWORK对应的命令处理函数为  
wpa\_supplicant\_ctrl\_iface\_set\_network，其代码如下所示。

```
[-->ctrl_iface.c: : wpa_supplicant_ctrl_iface_set_network]

static int wpa_supplicant_ctrl_iface_set_network(
    struct wpa_supplicant *wpa_s, char *cmd)
{
    int id;
    struct wpa_ssid *ssid;
    char *name, *value;

    // SET_NETWORK的参数是: "<network id> <variable name>
<value>"
    name = os_strchr(cmd, ' '); *name++ = '\0';           // 获取
name
    value = os_strchr(name, ' '); *value++ = '\0';         // 获取
value
    id = atoi(cmd);                                         // 获取
id
    .....
    // 从wpa_config中的无线网络配置列表中找到对应编号的无线网络配置项
    ssid = wpa_config_get_network(wpa_s->conf, id);
    .....
/*
    为该网络设置对应的配置值。wpa_config_set函数的具体实现与4.3.4
节"wpa_supplicant_
    init_iface分析之一"介绍的wpa_config_process_global函数类似，其
内部也是通过定义
    一些宏和数组来完成配置项的设置，不讨论其细节。就本例而言，当三个
SET_NETWORK命令处理
    完毕时，wpa_ssid的
    ssid="Test"、key_mgmt=WPA_KEY_MGMT_PSK、
    passphrase="12345Test"。
    注意：虽然在命令行中设置的是psk="12345Test"，但实际上密码值将保存在
passphrase变量中。
*/
    if (wpa_config_set(ssid, name, value, 0) < 0) {....}
    // 清空对应的PMKSA缓存信息。wpa_s->wpa指向一个wpa_sm对象
    wpa_sm_pmksa_cache_flush(wpa_s->wpa, ssid);
```

```

    if (wpa_s->current_ssid == ssid || wpa_s->current_ssid ==
NULL)
        eapol_sm_invalidate_cached_session(wpa_s->eapol);

    if ((os_strcmp(name, "psk") == 0 && value[0] == '' &&
ssid->ssid_len) ||
        (os_strcmp(name, "ssid") == 0 && ssid->passphrase))
        wpa_config_update_psk(ssid); // 将字符串形式的passphrase转
成key, 见下文介绍
    else if (os_strcmp(name, "priority") == 0)
        wpa_config_update_prio_list(wpa_s->conf);

    return 0;
}

```

我们在3.3.7节的开头部分曾介绍过Key和Passphrase的区别。一般而言，Passphrase（也叫Password）表现为human-readable的字符串，而Key则一般是二进制或十六进制的数据。STA和AP交互的是Key，而用户设置的是Passphrase。所以上述代码中需要将Passphrase转换成Key，这是通过wpa\_config\_update\_psk函数来完成的。其代码如下所示。

```

[-->config.c: : wpa_config_update_psk]

void wpa_config_update_psk(struct wpa_ssid *ssid)
{
#ifndef CONFIG_NO_PBKDF2 // 本例支持该宏，如果没有它的话，用户只能输入
十六进制的Key
    // 对用户设置的psk和ssid进行hash计算，最终的结果作为真正的Pre-
Shared Key
    pbkdf2_sha1(ssid->passphrase, (char *) ssid->ssid, ssid-
>ssid_len,
                4096, ssid->psk, PMK_LEN);
    ssid->psk_set = 1;
#endif /* CONFIG_NO_PBKDF2 */
}

```

SET\_NETWORK命令处理的介绍到此为止。下面来看最后一个关键命令ENABLE\_NETWORK的处理流程。

#### 4.5.3 ENABLE\_NETWORK命令处理

ENABLE\_NETWORK命令由wpa\_supplicant\_ctrl\_iface\_enable\_network进行处理，其代码如下所示。

```
[-->ctrl_iface.c: : wpa_supplicant_ctrl_iface_enable_network]

static int wpa_supplicant_ctrl_iface_enable_network(struct
wpa_supplicant *wpa_s,
                                         char *cmd)
{
    int id;
    struct wpa_ssid *ssid;

    if (os_strcmp(cmd, "all") == 0) { // 使能所有无线网络
        ssid = NULL;
    } else {
        id = atoi(cmd); // 本例中的id为0
        ssid = wpa_config_get_network(wpa_s->conf, id); // 找到id为0的无线网络配置对象
        .....
        // 在前面ADD_NETWORK中, disabled为1, 表示还没有使能它。
        disable为2的情况和P2P有关
        if (ssid->disabled == 2) {....}
    }
    wpa_supplicant_enable_network(wpa_s, ssid);
    return 0;
}
```

来看wpa\_supplicant\_enable\_network，其代码如下所示。

```
[-->wpa_supplicant.c: : wpa_supplicant_enable_network]

void wpa_supplicant_enable_network(struct wpa_supplicant *wpa_s,
struct wpa_ssid *ssid)
{
    struct wpa_ssid *other_ssid;
    int was_disabled;
    if (ssid == NULL) {
        .....// 处理ENABLE_NETWORK all的情况
    } else if (ssid->disabled && ssid->disabled != 2) {
        if (wpa_s->current_ssid == NULL) // WPAS当前没有活跃的无线
```

```

网络, 所以current_ssid为空
    wpa_s->reassociate = 1; // 注意这个变量的值
    // ADD_NETWORK只是添加了一个无线网络配置项
    // 接下来要发起扫描工作以和对应的无线网络进行交互
    // 下面这个函数将发起scan操作。后面两个0代表时间。详情见下节分析
    wpa_supplicant_req_scan(wpa_s, 0, 0);
}
was_disabled = ssid->disabled;
ssid->disabled = 0; // 设置disabled为0
if (was_disabled != ssid->disabled)
    wpas_notify_network_enabled_changed(wpa_s, ssid);
}
}

```

正如代码中注释所说, ADD\_NETWORK不过是为WPAS添加了一个无线网络配置项罢了。该无线网络是否存在? 通过SET\_NETWORK配置的信息是否正确? 这些问题的解答首先从无线网络扫描开始。

## 1. 无线网络扫描流程分析

ENABLE\_NETWORK将发起无线网络扫描请求, 这是由wpa\_supplicant\_req\_scan完成的, 其代码如下所示。

```

[-->scan.c: : wpa_supplicant_req_scan]

void wpa_supplicant_req_scan(struct wpa_supplicant *wpa_s, int
sec, int usec)
{
#ifndef ANDROID // Android平台
上, 该宏被定义
.....// 不讨论非Android平台上的代码
#endif
    eloop_cancel_timeout(wpa_supplicant_scan, wpa_s, NULL);
    // 在本例中, sec和usec都是0, 所以wpa_supplicant_scan将很快得到执行。该函数是扫描的核心代码
    eloop_register_timeout(sec, usec, wpa_supplicant_scan,
wpa_s, NULL);
}

```

wpa\_supplicant\_scan是无线网络扫描的核心函数, 其代码比较复杂, 我们分段来看。

### (1) wpa\_supplicant\_scan分析之一

这一段代码主要和scan请求的参数准备有关。

[-->scan.c: : wpa\_supplicant\_scan代码段一]

```
static void wpa_supplicant_scan(void *eloop_ctx, void
*timeout_ctx)
{
    struct wpa_supplicant *wpa_s = eloop_ctx;
    struct wpa_ssid *ssid; int scan_req = 0, ret;
    struct wpabuf *extra_ie; // 用于存储
Information Element信息
    struct wpa_driver_scan_params params; // 发给驱动的scan
请求命令
    // 用于记录一个scan请求能包含多少个ssid。请参考4.3.4节关于
capability的介绍
    size_t max_ssids;
    enum wpa_states prev_state;
    // wpa_state取值为WPA_INACTIVE，由4.3.4节
wpa_supplicant_driver_init函数中的代码设置
    if (wpa_s->wpa_state == WPA_INTERFACE_DISABLED) { return; }
    // disconnected为0, scan_req为1, 都是wpa_supplicant构造时的默认
值
    if (wpa_s->disconnected && !wpa_s->scan_req) { .....}
    /*
        搜索wpa_config中所有的无线网络配置项，看其中是否有使能的无线网络。本
例中，在扫描之前，
        已经将目标wpa_ssid的disabled变量置为0，这样，下面这个函数调用将
返回非0值，使得整
        个if判断为假。
    */
    if (!wpa_supplicant_enabled_networks(wpa_s->conf) &&
!wpa_s->scan_req) { .....}

    /*
        ap_scan是一个很有意思的参数，它和AP扫描和选择有关，默认值为1。值为1：
表示WPAS来完成
        AP扫描和选择的绝大部分工作（包括关联、EAPOL认证等工作）。值为0：表示
驱动完成AP扫描和选
        择的工作。这种驱动比较少见，笔者未能找到关于WPA_DRIVER_FLAGS_WIRED
标志的合理解释，有
        知晓的读者不妨和大家分享一下相关知识。值为2：和0类似，不过在
NDIS（Windows上的网络设备
        驱动）中用得较多。
    */
    if (wpa_s->conf->ap_scan != 0 && (wpa_s->drv_flags &
```

```

WPA_DRIVER_FLAGS_WIRED)) {.....}

    if (wpa_s->conf->ap_scan == 0) { // 如果驱动能完成大部分工作的话,
WPAS的工作量将大大减少
        wpa_supplicant_gen_assoc_event(wpa_s);
        return;                                // 无须后面的流程
    }

.....// CONFIG_P2P: P2P相关, 本章不讨论
if (wpa_s->conf->ap_scan == 2)
    max_ssids = 1;
else {
    max_ssids = wpa_s->max_scan_ssids;    // 一个scan请求能包含
多少个ssid
    if (max_ssids > WPAS_MAX_SCAN_SSIDS)
        max_ssids = WPAS_MAX_SCAN_SSIDS;
}
scan_req = wpa_s->scan_req;                  // scan_req
为1
wpa_s->scan_req = 0;                         // scan_req
被置为0
os_memset(&params, 0, sizeof(params));
// 初始化scan请求的参数, 其类型为wpa_driver_scan_params

```

根据第3章关于无线网络扫描的介绍, 一个Probe Request要么指定 wildcard ssid以扫描周围所有的无线网络, 要么指定某个ssid以扫描特定无线网络。为了方便WPAS的使用, wlan driver新增了一个功能, 使得上层可通过一次scan请求来扫描多个不同ssid的无线网络。一个 scan请求在代码中对应的数据结构就是wpa\_driver\_scan\_params。而 wpa\_supplicant\_scan最重要的工作就是准备好这个请求。

## (2) wpa\_supplicant\_scan分析之二

接着来看代码段二。

[-->scan.c: : wpa\_supplicant\_scan代码段二]

```

.....// 接上段代码
prev_state = wpa_s->wpa_state; // 此时的wpa_state是
WPA_INACTIVE
if (wpa_s->wpa_state == WPA_DISCONNECTED || wpa_s-
>wpa_state == WPA_INACTIVE)
    wpa_supplicant_set_state(wpa_s, WPA_SCANNING); // 设置WPAS
状态为WPA_SCANNING

```

```

/*
connect_without_scan指向一个wpa_ssid对象。它对应的应用场景是：WPAS
事先通过某种方
式（例如后续章节将要介绍的WPS）已经知道要连接的无线网络了，所以此处就无
须扫描，仅关联它即可。
*/
if (scan_req != 2 && wpa_s->connect_without_scan) {
    for (ssid = wpa_s->conf->ssid; ssid; ssid = ssid->next)
{
    if (ssid == wpa_s->connect_without_scan) break;
}
wpa_s->connect_without_scan = NULL;
if (ssid) {
    wpa_supplicant_associate(wpa_s, NULL, ssid); // 关联到
目标网络
    return;
}
}

```

// 搜索wpa\_config中的所有无线网络配置项，看看哪些需要包含到这次scan请  
求中

```

ssid = wpa_s->conf->ssid;
/*
```

prev\_scan\_ssid用于记录上一次scan请求的最后一个ssid。它对应了如下的  
应用场景。

假设scan请求一次只能携带2个ssid，如果要扫描wpa\_config中配置的全部网  
络项（假设是4个），

则需要发起两次scan请求。所以，当prev\_scan\_ssid上一次扫描的并非全部  
无线网络的话（由

wildcardssid来判断），则此处要接着扫描之前没有扫描的那些无线网络。

以本例而言，prev\_scan\_ssid初始值是WILDCARD\_SSID\_SCAN（其值为  
1）。

```

*/
if (wpa_s->prev_scan_ssid != WILDCARD_SSID_SCAN) {
    while (ssid) {
        if (ssid == wpa_s->prev_scan_ssid) {
            ssid = ssid->next;
            break;
        }
        ssid = ssid->next;
    }
}

if (scan_req != 2 && wpa_s->conf->ap_scan == 2) {
    .....// 不考虑这种情况
#endif ANDROID
```

```

.....
#endif
} else {
    struct wpa_ssid *start = ssid, *tssid;
    int freqs_set = 0;
    if (ssid == NULL && max_ssids > 1)
        ssid = wpa_s->conf->ssid;
    while (ssid) {
        /*
         * 有一些AP被设置为hidden ssid。即它不响应wildcard ssid扫描
         的Probe Request,
         同时，自己发送的Beacon帧也不携带ssid信息。这样，只有知道ssid
         的STA才能和这
         些AP连接上，其安全性略有提高。scan_ssid就是用来判断此无线网
         络是否需要指明ssid。
         本例中的"Test"无线网络没有隐藏 ssid，所以scan_ssid值为0。
         否则需要通过SET_
             NETWORK 0 scan_ssid 1来设置它。
         */
        if (!ssid->disabled && ssid->scan_ssid) {
            // 把ssid信息加到params的ssids数组中
            params.ssids[params.num_ssids].ssid = ssid-
>ssid;
            params.ssids[params.num_ssids].ssid_len = ssid-
>ssid_len;
            params.num_ssids++;
            // 如果本次scan请求的ssid个数已经达到driver能支持的最
大数，则跳出循环
            if (params.num_ssids + 1 >= max_ssids) break;
        }
        ssid = ssid->next;
        if (ssid == start)
            break;
        if (ssid == NULL && max_ssids > 1 && start != wpa_s-
>conf->ssid)
            ssid = wpa_s->conf->ssid;
    }
}
/* 处理扫描时的频率选择。如果已经知道目标无线网络的工作信道，可以直接
设定频率参数以
优化扫描过程。否则，无线网卡将尝试在各个信道上搜索目标无线网络。本
例没有使用频率参数。
*/
for (tssid = wpa_s->conf->ssid; tssid; tssid = tssid-
>next) {
    if (tssid->disabled) continue;

```

```

        if ((params.freqs || !freqs_set) && tssid-
>scan_freq) {
            int_array_concat(&params.freqs, tssid-
>scan_freq);
        } else {
            os_free(params.freqs);
            params.freqs = NULL;
        }
        freqs_set = 1;
    }
    int_array_sort_unique(params.freqs); // 对所有频率参数进
行升序排序
}

if (ssid && max_ssids == 1) { // 如果scan请求最
多只能包含一个ssid
    if (!wpa_s->prev_scan_wildcard) {
        params.ssids[0].ssid = NULL; // 扫描wildcast ssid
        params.ssids[0].ssid_len = 0;
        wpa_s->prev_scan_wildcard = 1;
    } else {
        wpa_s->prev_scan_ssid = ssid;
        wpa_s->prev_scan_wildcard = 0;
    }
} else if (ssid) {
    wpa_s->prev_scan_ssid = ssid;
    params.num_ssids++;
} else {
    wpa_s->prev_scan_ssid = WILDCARD_SSID_SCAN;
    params.num_ssids++;
}
// 对频率参数进行修改，和P2P以及WPS有关，本章略过它们
wpa_supplicant_optimize_freqs(wpa_s, &params);
// 是否需要携带附件的IE信息。主要用在WPS等情况，本章略过它们
extra_ie = wpa_supplicant_extra_ies(wpa_s, &params);

if (params.freqs == NULL && wpa_s->next_scan_freqs) {
    params.freqs = wpa_s->next_scan_freqs;
} else os_free(wpa_s->next_scan_freqs);
wpa_s->next_scan_freqs = NULL;
/*

```

scan请求可以设置一个过滤条件，扫描完毕后，driver wrapper会过滤掉那些不符合条件的无线

网络。注意，filter\_ssids用来保存那些不能被过滤的无线网络ssid。即，扫描到的无线网络不在

filter\_ssids中时，它将被过滤掉。过滤的代码在driver\_nl80211.c nl80211\_scan\_filtered

函数中，其调用之处在同一文件里的bss\_info\_handler函数中。

```
/*
params.filter_ssids =
wpa_supplicant_build_filter_ssids(wpa_s->conf,
                                    &params.num_filter_ssids);
if (extra_ie) {
    params.extra_ies = wpabuf_head(extra_ie);
    params.extra_ies_len = wpabuf_len(extra_ie);
}

#ifndef CONFIG_P2P
.....
#endif /* CONFIG_P2P */
```

上述wpa\_supplicant\_scan代码段主要展示了如何填写扫描请求参数，复杂之处在于其对细节的处理。下面来看最后一个代码段。

### (3) wpa\_supplicant\_scan分析之三

当scan请求的参数准备好后，wpa\_supplicant\_scan将直接向driver wrapper发起scan请求。

[-->scan.c: : wpa\_supplicant\_scan代码段三]

```
// 调用driver wrapper的scan2函数。下文将直接分析driver_n180211的
scan2函数
ret = wpa_supplicant_trigger_scan(wpa_s, &params);
// 释放分配的资源
wpabuf_free(extra_ie);
os_free(params.freqs);
os_free(params.filter_ssids);

if (ret) {.....// 错误处理}
}
```

对于driver\_n180211来说，对应的函数是wpa\_driver\_n180211\_scan，马上来看其代码。

[-->driver\_n180211.c: : wpa\_driver\_n180211\_scan]

```
static int wpa_driver_n180211_scan(void *priv, struct
wpa_driver_scan_params *params)
{
    struct i802_bss *bss = priv;                                // i802_bss是
```

```

driver wrapper的上下文信息
    struct wpa_driver_nl80211_data *drv = bss->drv;
                                            // 获取

wpa_driver_nl80211_data对象
    int ret = 0, timeout;
    struct nl_msg *msg, *ssids, *freqs, *rates;
    size_t i;
    drv->scan_for_auth = 0;
    msg = nlmsg_alloc(); ssids = nlmsg_alloc(); freqs =
nlmsg_alloc();
    rates = nlmsg_alloc();
    // 这个函数的主要功能是将wpa_driver_scan_params参数转换成对应的
netlink command
    os_free(drv->filter_ssids);
    drv->filter_ssids = params->filter_ssids;
    params->filter_ssids = NULL;
    drv->num_filter_ssids = params->num_filter_ssids;

    nl80211_cmd(drv, msg, 0, NL80211_CMD_TRIGGER_SCAN);
    NLA_PUT_U32(msg, NL80211_ATTR_IFINDEX, drv->ifindex);      //
指定网卡设备编号

    for (i = 0; i < params->num_ssids; i++)
// 填充ssid信息
        NLA_PUT(ssids, i + 1, params->ssids[i].ssid_len, params-
>ssids[i].ssid);

        if (params->num_ssids) nla_put_nested(msg,
NL80211_ATTR_SCAN_SSIDS, ssids);

        if (params->extra_ies)                                //
填充附加IE信息
            NLA_PUT(msg, NL80211_ATTR_IE, params-
>extra_ies_len, params->extra_ies);

        if (params->freqs) {
// 填充频率信息
            for (i = 0; params->freqs[i]; i++) NLA_PUT_U32(freqs, i
+ 1, params->freqs[i]);

            nla_put_nested(msg, NL80211_ATTR_SCAN_FREQUENCIES,
freqs);
        }
        if (params->p2p_probe) {.....// P2P相关}
/*
发送请求给wlan驱动。返回值只是表示该命令是否正确发送给了驱动。扫描结束

```

事件将通过

```
driver event返回给WPAS。下文将分析如何处理扫描结束事件。  
*/  
ret = send_and_recv_msgs(drv, msg, NULL, NULL);  
msg = NULL;  
if (ret) { .....// 错误处理}  
  
timeout = 10;  
/*
```

一般情况下，driver完成扫描后需要通知WPAS一个scan complete事件。如果驱动不通知的话，

WPAS就会自己去查询driver以获取扫描到的无线网络信息。如何知晓driver是否会通知该事件呢？

WPAS中是通过scan\_complete\_events变量来判断的。值得指出的是，该变量的取值是测试出来的。

即scan\_complete\_events初始值为0。如果扫描后收到了scan complete事件，该值将被

修改为1。由于本例中，该变量是第一次碰到，所以其值为0。

```
*/  
if (drv->scan_complete_events) timeout = 30;  
eloop_cancel_timeout(wpa_driver_nl80211_scan_timeout, drv,  
drv->ctx);  
// 本例中，timeout为10秒  
eloop_register_timeout(timeout, 0,  
wpa_driver_nl80211_scan_timeout, drv, drv->ctx);  
.....  
return ret;  
}
```

就本例而言，ENABLE\_WORK命令处理的第一步就是要扫描周围的无线网络。至于目标无线网络是否存在，则属于扫描结果处理流程的工作了。

#### (4) 无线网络扫描流程总结

图4-31所示为触发扫描功能的流程图。

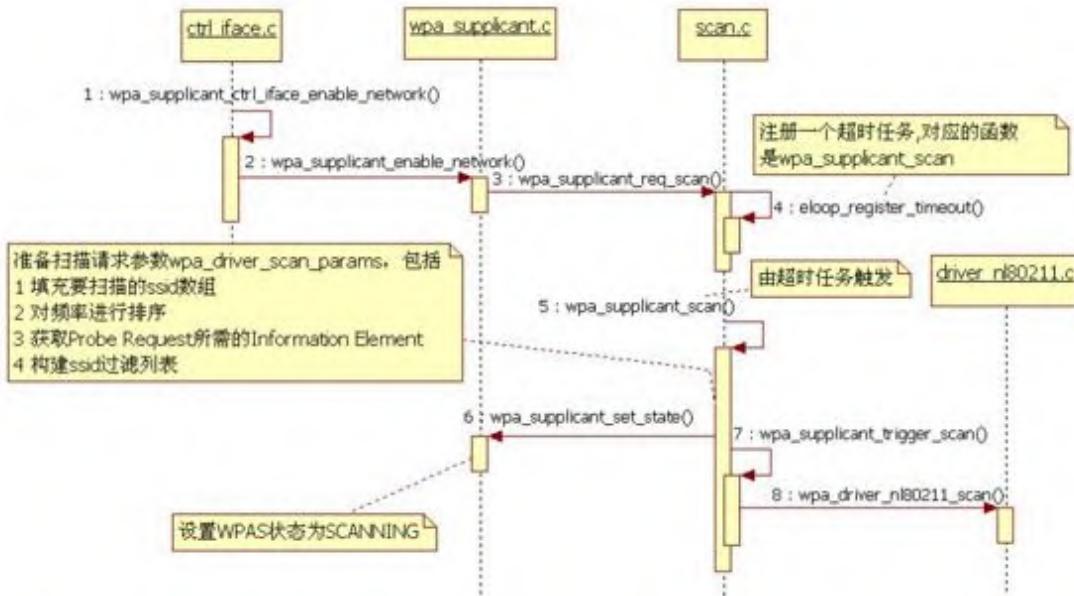


图4-31 触发扫描的流程图

图4-31所示函数中，`wpa_supplicant_scan`的内容较为丰富，其中有很多细节内容。读者初次学习时，可以先不考虑这些细节，只要把握图4-31中的函数调用流程即可。

马上来看扫描结果的处理流程。

## 2. 扫描结果处理流程分析

在上一节的扫描请求中，`driver_nl80211`发送了`NL80211_CMD_TRIGGER_SCAN`命令给wlan driver以通知它开始扫描周围的无线网络。当wlan driver完成此任务后，它将向4.3.4节中`wpa_driver_nl80211_init_nl_global`函数中注册的三个netlink组播之一的“scan”组播地址发送结果。而`driver_nl80211`对应处理来自wlan driver netlink消息处理回调函数为`process_global_event`，而它最终又会调用`do_process_drv_event`进行处理。所有，直接来看`do_process_drv_event`中和扫描结果相关的代码即可。

[-->`driver_nl80211.c`: : `do_process_drv_event`]

```
static void do_process_drv_event(struct wpa_driver_nl80211_data
*drv,
        int cmd, struct nlattr **tb)
{
```

```

/*
    ap_scan_as_station和hostapd有关，该值默认等于
NL80211_IFTYPE_UNSPECIFIED。读者可参考4.3.4节"wpa_supplicant_init_iface分析之三"中对
wpa_driver_nl80211_init
    的分析。
*/
if (drv->ap_scan_as_station != NL80211_IFTYPE_UNSPECIFIED &&
    (cmd == NL80211_CMD_NEW_SCAN_RESULTS || cmd ==
NL80211_CMD_SCAN_ABORTED)) {.....}

switch (cmd) {
    // driver wrapper发送NL80211_CMD_TRIGGER_SCAN命令后，wlan
    driver也会回复同名的一个消息
    case NL80211_CMD_TRIGGER_SCAN:
        wpa_printf(MSG_DEBUG, "nl80211: Scan trigger");
        break;
    .....// 其他消息处理
    case NL80211_CMD_NEW_SCAN_RESULTS:
        // 收到来自driver的scan回复，所以设置scan_complete_events变量
        为1
        drv->scan_complete_events = 1;
        // 取消超时任务
        eloop_cancel_timeout(wpa_driver_nl80211_scan_timeout,
drv,  drv->ctx);
        /*
        图4-1所示的WPAS软件结构中曾提到driver wrapper将通过发送driver
event的方式
        触发WPAS其他模块进行对应处理。下面这个函数即是用来处理和发送与scan
        相关的driver
            event的。其详情见下文。
        */
        send_scan_event(drv, 0, tb);
        break;
    .....
}
return
}

```

上述代码中的send\_scan\_event将解析来自wlan driver的扫描完毕事件，然后再向WPAS发送driver event，其代码如下所示。

[-->driver\_nl80211.c: : send\_scan\_event]

```

static void send_scan_event(struct wpa_driver_nl80211_data *drv,
int aborted,
    struct nlattr *tb[])
{
    // 代表driver event的数据结构
    // 它是一个联合体，包含了assoc_info、auth_info、scan_info等较多内容
    union wpa_event_data event;
    struct nlattr *nl;
    int rem;
    // 扫描信息，包含频率数组（int类型的数组）以及wpa_driver_scan_ssid
    // 数组
    // 请读者特别注意scan_info不是扫描结果
    struct scan_info *info;
#define MAX_REPORT_FREQS 50
    int freqs[MAX_REPORT_FREQS];
    int num_freqs = 0;
    /*
        scan_for_auth变量和wlan driver有关，它描述了如下的应用场景。
        当WPAS发起认证（authentication）操作时，它将向driver发送
NL80211_CMD_
        AUTHENTICATE命令。driver可能因为某种原因（超时等），其内部存储的目
标BSS（代表目标无
线网络）信息失效。这时，它将返回ENOENT给WPAS。正常情况下，WPAS应该
重新扫描。但为了加
快这个流程，WPAS可以单独扫描目标无线网络（因为WPAS还是保存了目标无线
网络的信息，例如
        频道等），然后再发起认证操作。从笔者角度来看，该场景对应的问题是wlan
        driver没有目标
        BSS信息，而WPAS有目标BSS信息。WPAS和wlan driver是两个不同模块，二
者的
        信息并不能
        总是保持一致。既然WPAS有目标BSS信息，那么可以通过更加快捷的方法让
        wlan driver也得
        到这个信息（通过在指定频道到扫描目标BSS），从而加快整个流程（从WPAS
        角度来看，用户加入无
        线网络的流程已经进行到Authentication阶段，此时再退回到重新扫描的阶
段实在有些麻烦）。
        关于scan_for_auth的官方解释，请通过前面介绍的git blame命令获取
        commit message，
        方法是先通过"git blame ./src/drivers/driver_nl80211.c | grep
        scan_for_auth"
        获得和scan_for_auth相关的commit号，然后再用git log commit号查
        看。
        git blame的用法请参考4.2.4节。
*/
    if (drv->scan_for_auth) {.....}

```

```

os_memset(&event, 0, sizeof(event));
info = &event.scan_info;
info->aborted = aborted;

if (tb[NL80211_ATTR_SCAN_SSIDS]) { // 遍历
NL80211_ATTR_SCAN_SSIDS属性
    nla_for_each_nested(nl, tb[NL80211_ATTR_SCAN_SSIDS],
rem) {
        struct wpa_driver_scan_ssid *s = &info-
>ssids[info->num_ssids];
        s->ssid = nla_data(nl); s->ssid_len = nla_len(nl);
        info->num_ssids++;
        if (info->num_ssids == WPAS_MAX_SCAN_SSIDS) break;
    }
}
if (tb[NL80211_ATTR_SCAN_FREQUENCIES]) { // 获取netlink消息中的
频率信息
    nla_for_each_nested(nl,
tb[NL80211_ATTR_SCAN_FREQUENCIES], rem) {
        freqs[num_freqs] = nla_get_u32(nl);
        num_freqs++;
        if (num_freqs == MAX_REPORT_FREQS - 1) break;
    }
    info->freqs = freqs;
    info->num_freqs = num_freqs;
}
// wpa_supplicant_event被driver event模块用来发送driver事件给
WPAS
wpa_supplicant_event(drv->ctx, EVENT_SCAN_RESULTS, &event);
}

```

上述代码中请注意struct scan\_info的作用，它定义于联合体wpa\_event\_data中，包含了本次扫描请求扫描了哪些SSID、对应的频率等。

**提示** scan\_info不是scan后得到的结果信息，而是代表驱动处理scan请求时的一些处理信息。对于那些不支持通知scan complete事件的driver而言，scan\_info就没法获得。例如，超时扫描处理wpa\_driver\_nl80211\_scan\_timeout中调用wpa\_supplicant\_event时就没有scan\_info。scan\_info到底有什么作用呢？后文将详细介绍它。

wpa\_supplicant\_event是driver wrapper向WPAS通知driver event的接口函数，其代码如下所示。

```
[-->events.c: : wpa_supplicant_event]

void wpa_supplicant_event(void *ctx, enum wpa_event_type event,
                           union wpa_event_data *data)
{
    struct wpa_supplicant *wpa_s = ctx;
    u16 reason_code = 0;
    int locally_generated = 0;
    .....
    switch (event) {
        .....
        // driver event总类非常多，现在只分析EVENT_SCAN_RESULT即可
        case EVENT_SCAN_RESULTS:
            wpa_supplicant_event_scan_results(wpa_s, data);
            break;
        .....
    }
}
```

代码很简单，下面直接来分析此处的目标函数  
wpa\_supplicant\_event\_scan\_results。

### (1) wpa\_supplicant\_event\_scan\_results函数分析

函数代码如下。

```
[-->events.c: : wpa_supplicant_event_scan_results]

static void wpa_supplicant_event_scan_results(struct
wpa_supplicant *wpa_s,
                                              union wpa_event_data *data)
{
    const char *rn, *rn2;
    struct wpa_supplicant *ifs;
    // 笔者编译的WPAS实际上打开了ANDROID_P2P宏，但本章不讨论P2P相关内容
#ifdef ANDROID_P2P
    if (_wpa_supplicant_event_scan_results(wpa_s, data, 0) < 0)
#else
    if (_wpa_supplicant_event_scan_results(wpa_s, data) < 0)
#endif
        return;
    /*
        get_radio_name函数和4.3.4节"wpa_supplicant_init_iface分析之
        三"中提到的
        /sys/class/net/wlan0/phy80211/name文件内容有关。driver_nl80211
        实现了这个函数，
```

它将返回上面这个文件的内容。

```
*/
```

```
if (!wpa_s->driver->get_radio_name) return;  
/*
```

下面这段代码的作用主要是看虚拟接口设备（参考4.3.4节中对Virtual Interface的介绍）

是不是使用了同一个物理设备。如果使用了同一个物理设备，则这个虚拟设备获得的扫描结果可以

和其他虚拟设备共享（避免重复扫描）。本章不对其中细节进行介绍。

```
*/
```

```
rn = wpa_s->driver->get_radio_name(wpa_s->drv_priv);  
/*
```

global指向wpa\_global对象，ifaces是wpa\_global的成员变量，指向一个wpa\_supplicant

对象的队列，可参考图4-7和图4-11。在WPAS中，每一个wpa\_supplicant对象会和一个虚拟接口

设备关联。

```
*/
```

```
for (if s = wpa_s->global->ifaces; ifs; ifs = ifs->next) {  
    if (ifs == wpa_s || !ifs->driver->get_radio_name)  
        continue;  
    rn2 = ifs->driver->get_radio_name(ifs->drv_priv);  
    if (rn2 && os_strcmp(rn, rn2) == 0) { // 比较两个虚拟设备对应的物理设备是否为同一个
```

```
#ifdef ANDROID_P2P
```

```
.....
```

```
#else // 和其他虚拟设备分享扫描结果
```

```
    _wpa_supplicant_event_scan_results(ifs, data);
```

```
#endif
```

```
}
```

```
}
```

```
}
```

上述代码中，\_wpa\_supplicant\_event\_scan\_results是核心处理函数并且内容较多，所以下面将分段研究它。

## (2) \_wpa\_supplicant\_event\_scan\_results分析之一

先来看第一个代码段，代码如下所示。

[-->events.c: : \_wpa\_supplicant\_event\_scan\_results代码段一]

```

#define ANDROID_P2P
static int _wpa_supplicant_event_scan_results(struct
wpa_supplicant *wpa_s,
                                              union wpa_event_data *data, int
suppress_event)
#else
static int _wpa_supplicant_event_scan_results(struct
wpa_supplicant *wpa_s,
                                              union wpa_event_data *data)
#endif
{
    struct wpa_bss *selected;  struct wpa_ssid *ssid = NULL;
    struct wpa_scan_results *scan_res;// 这才是真正的扫描结果
    int ap = 0;
    .....
    wpa_supplicant_notify_scanning(wpa_s, 0);
    .....// 和P2P相关, 本章不讨论
    // 获得无线网络扫描结果。就本例而言, 前面得到的scan_info信息将传递到下
面这个函数
    scan_res = wpa_supplicant_get_scan_results(wpa_s,data ?
&data->scan_info : NULL, 1);
    if (scan_res == NULL) {.....// 扫描结果为空, 重新发起扫描}

#ifndef CONFIG_NO_RANDOM_POOL
/*
    把扫描结果中得到的无线网络频率、信号强度等信息取出来, 然后写到图4-1中
的crypto模块。
    这样, 能增加随机数生成的随机性。WPAS中nonce的生成都会利用随机数生成
器。感兴趣的读者
    不妨自行研究相关内容。
*/
.....
#endif /* CONFIG_NO_RANDOM_POOL */

```

wpa\_supplicant\_get\_scan\_results比较重要, 我们通过代码来介绍它。

[-->scan.c: : wpa\_supplicant\_get\_scan\_results]

```

struct wpa_scan_results * wpa_supplicant_get_scan_results(struct
wpa_supplicant *wpa_s,
                                              struct scan_info *info, int new_scan)
{
    struct wpa_scan_results *scan_res;
    size_t i;
    int (*compar)(const void *, const void *) =

```

```

wpa_scan_result_compar;
/*
调用driver interface的get_scan_results2函数，对nl80211来说，其真实函数是wpa_driver_nl80211_get_scan_results，它将向wlan driver发送NL80211_CMD_GET_SCAN命令以获取扫描结果。另外，wpa_driver_nl80211_get_scan_results中还涉及一个比较有意思的函数
    wpa_driver_nl80211_check_bss_status。其作用请读者利用前面介绍的git blame方法来查询。
*/
scan_res = wpa_drv_get_scan_results2(wpa_s);
if (scan_res == NULL) {return NULL;} //对扫描结果进行排序，排序函数是wpa_scan_result_compar，它将根据无线网络的信噪比（SNR）
#define CONFIG_WPS
.....// WPS相关
#endif /* CONFIG_WPS */
/*
对扫描结果进行排序，排序函数是wpa_scan_result_compar，它将根据无线网络的信噪比（SNR）
    进行排序信噪比越高，无线网络信号越强。注意，除了SNR外，wpa_scan_result_compar还有别的比较条件，感兴趣的读者可行阅读该函数。
*/
qsort(scan_res->res, scan_res->num, sizeof(struct wpa_scan_res *), compar);
.....
*/
更新WPAS中保存的那些bss信息（即无线网络信息，每一个无线网络由wpa_bss表示）。我们在4.3.4节中曾经介绍过和wpa_bss相关的知识。WPAS维护了一个wpa_bss链表，每次扫描时都可能更新这个链表
    （添加新扫描得到的wpa_bss、删除某些老旧的wpa_bss）。
*/
wpa_bss_update_start(wpa_s);
for (i = 0; i < scan_res->num; i++)
    wpa_bss_update_scan_res(wpa_s, scan_res->res[i]);
*/
这里用上了scan_info。那些没有包含在本次scan_info中的wpa_bss不用更新。结合4.5.3节
    “wpa_supplicant_scan分析之二”中提到的prev_scan_ssid的作用，能想到为什么不更新那些
        没有包含在scan_info中的wpa_bss吗？
*/
wpa_bss_update_end(wpa_s, info, new_scan);

```

```

        return scan_res;
    }
}

```

图4-32所示为扫描结果对应的数据结构。

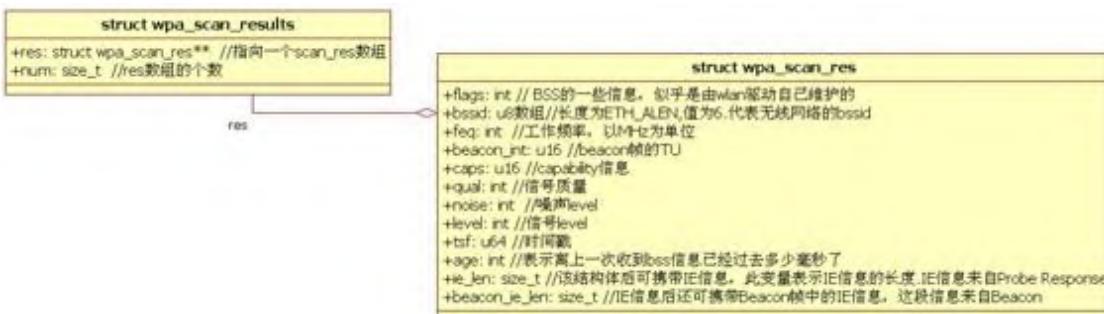


图4-32 wpa\_scan\_res相关数据结构

### (3) \_wpa\_supplicant\_event\_scan\_results分析之二

接着来看\_wpa\_supplicant\_event\_scan\_results下一个代码片段。

[-->events.c: : \_wpa\_supplicant\_event\_scan\_results代码段二]

.....// 接\_wpa\_supplicant\_event\_scan\_results代码片段一  
/\*

如果wpa\_supplicant对扫描结果有特殊处理，则调用scan\_res\_handler对应的函数处理。

目前只有P2P情况下scan\_res\_handler才起作用。

\*/

if (wpa\_s->scan\_res\_handler) { .....return 0; }

.....// hostapd的情况，本书不讨论

```
#ifdef ANDROID_P2P
    if (!suppress_event)
#endif
    {
        .....// 通知客户端，扫描结束
        wpas_notify_scan_done(wpa_s, 1);
    }

```

```
    if ((wpa_s->conf->ap_scan == 2 &&
!wpas_wps_searching(wpa_s))) { }
```

```
    if (wpa_s->disconnected) { .....}
    /*

```

还记得4.5.3节“wpa\_supplicant\_scan分析之一”中的ap\_scan变量吗？下

面这个

wpas\_driver\_bss\_selection函数判断是否由driver来控制无线网络的选择。显然，

本例中它将返回0值。不过，由于本例不支持bgscan（由CONFIG\_BGSCAN宏控制），

bgscan\_notify\_scan返回0。

这样，下面这个if判断失败。

```
/*
if
(!wpas_driver_bss_selection(wpa_s) &&bgscan_notify_scan(wpa_s,
scan_res) == 1) {
    wpa_scan_results_free(scan_res);
    return 0;
}
// 从扫描结果中选择一个合适的无线网络
// 注意，在本例中，下面这个函数将返回目标无线网络对应的wpa_bss对象
selected = wpa_supplicant_pick_network(wpa_s, scan_res,
&ssid);
```

上面这段代码相对简单。注意最后一句代码中调用的wpa\_supplicant\_pick\_network函数。它将根据scan\_res（扫描结果）、wpa\_bss（代表一个真实BSS的信息）和wpa\_ssId（代表用户设置的某个无线网络配置项）的匹配情况来选择合适的无线网络。匹配检查包含很多内容，例如ssId是否匹配、安全设置是否匹配、速率是否匹配等。最终，该函数返回一个被选中的目标无线网络的wpa\_bss对象。

提示 wpa\_supplicant\_pick\_network函数实际上检查的项非常多。由于篇幅问题，本章不能一一道来，感兴趣的读者请仔细研究。

#### (4) \_wpa\_supplicant\_event\_scan\_results分析之三

接着来看\_wpa\_supplicant\_event\_scan\_results的第三段代码。

[-->events.c: : \_wpa\_supplicant\_event\_scan\_results代码段三]

```
.....// 接代码段二
if (selected) { // 本例中返回的selected无线网络wpa_bss对象代表目标"Test"无线网络
    int skip;
/*
判断是否需要漫游。简单来说，漫游表示需要切换无线网络。对本例而言，
```

由于之前没有和AP关联，  
所以此处是需要漫游的（即需要切换无线网络）。

wpa\_supplicant\_need\_to\_roam中还包括对同一个ESS中如何选择更合适的BSS有一些  
简单的判断，主要是根据信号强度来选择。

```

*/
skip = !wpa_supplicant_need_to_roam(wpa_s, selected,
ssid, scan_res);
wpa_scan_results_free(scan_res);
if (skip) { // 无须切换网络，所以没有太多要做的工作
    wpa_supplicant_rsn_preatht_scan_results(wpa_s);
    return 0;
}
// 关联至目标无线网络。我们下一节再分析它
if (wpa_supplicant_connect(wpa_s, selected, ssid) < 0)
{....}
// 预认证(Pre-Authentication)处理，和802.11中的Fast
Transition有关，本书不讨论
    wpa_supplicant_rsn_preatht_scan_results(wpa_s);
} else { .....// 没有匹配的无线网络情况的处理 }
return 0;
}

```

到此为止，扫描结果的处理基本完毕，剩下的工作就是通过  
wpa\_supplicant\_connect向目标AP发起关联等请求以加入“Test”无线  
网络。

在介绍wpa\_supplicant\_connect之前，先总结一下扫描结果处理流程。

### (5) 扫描结果处理流程总结

扫描结果的处理流程相对比较复杂，如图4-33所示。其中有几个函数  
包含一些非常重要的细节，读者在研究过程中要特别注意。

- wpa\_supplicant\_pick\_nework：在扫描结果中找到一个最佳的无线  
网络。
- wpa\_supplicant\_need\_to\_roam：判断是否需要切换网络。
- wpa\_supplicant\_rsn\_preatht\_scan\_results：更新PMKSA缓存信  
息。其工作和Pre-Authentication有关。不过Pre-Authenticaton真正

的实现也需要AP的支持。

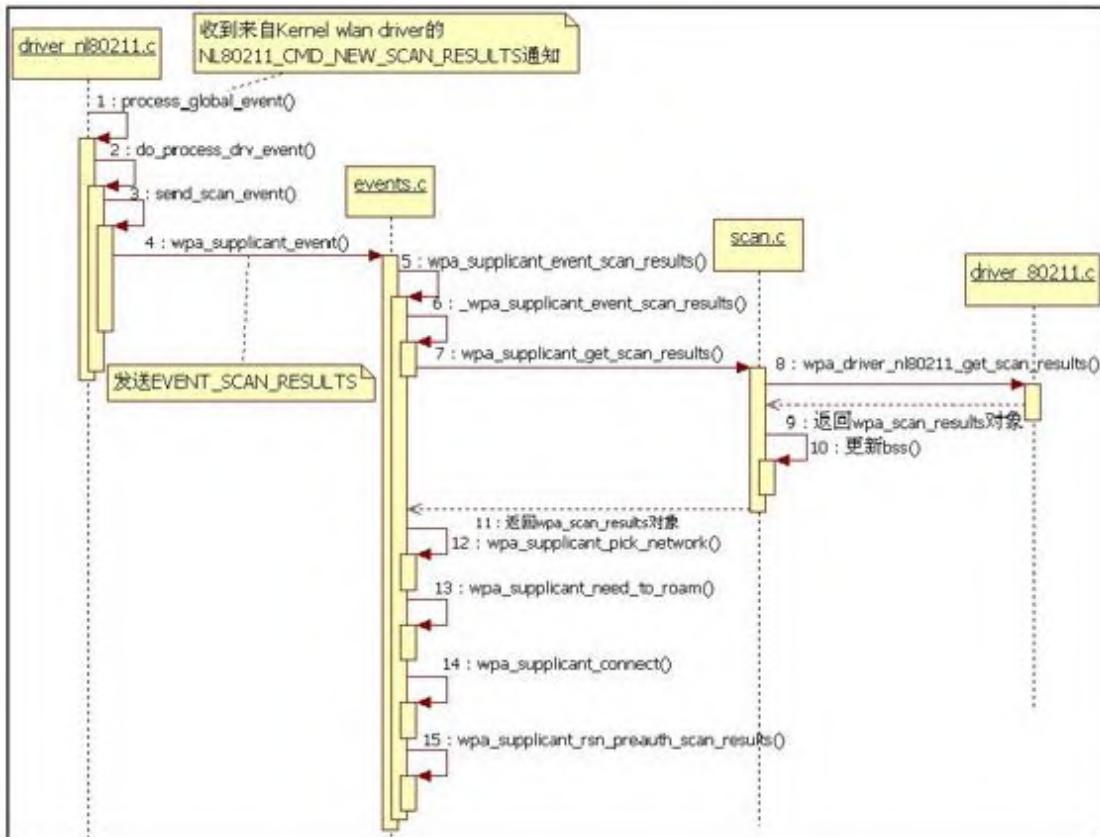


图4-33 扫描处理流程

图4-34所示为笔者通过AirPcap截获的目标AP发送的Probe Response帧。

```

- Tagged parameters (257 bytes)
  # Tag: SSID parameter set: Test
  # Tag: Supported Rates 1(B), 2(B), 5.5(B), 11(B), 6, 9, 12, 18, [Mbit/sec]
  # Tag: Os Parameter set: Current channel: 6
  # Tag: Country Information: country code US, Environment: any
  # Tag: ERP Information
  # Tag: RSN Information
    Tag Number: RSN Information (48)
    Tag Length: 20
    RSN version: 1
  # Group Cipher Suite: 00-0f-ac (Ieee8021) AES (CCM)
    Group Cipher Suite OUI: 00-0f-ac (Ieee8021)
    Group Cipher Suite type: AES (CCM) (4)
    Pairwise Cipher Suite Count: 1
  # Pairwise Cipher Suite List 00-0f-ac (Ieee8021) AES (CCM)
    # Pairwise Cipher Suite: 00-0f-ac (Ieee8021) AES (CCM)
    Auth Key Management (AKM) Suite Count: 1
  # Auth Key Management (AKM) List 00-0f-ac (Ieee8021) PSK
    # Auth Key Management (AKM) suite: 00-0f-ac (Ieee8021) PSK
      Auth Key Management (AKM) OUI: 00-0f-ac (Ieee8021)
      Auth Key Management (AKM) Type: FSK (2)
  # RSN Capabilities: 0x0000
    .... .... .0 = RSN Pre-Auth capabilities: Transmitter does not support pre-authentication
    .... .... ..0. = RSN NO Pairwise capabilities: Transmitter can support WEP default key 0 simultaneously with Pairwise key
    .... .... ..00. = RSN PTKSA Replay Counter capabilities: 1 replay counter per PTKSA/GTKSA/STAKEysSA (0x0000)
    .... .... ..00. .... = RSN GTKSA Replay counter capabilities: 1 replay counter per PTKSA/GTKSA/STAKEysSA (0x0000)
    .... .... .0. .... = Management Frame Protection Required: False
    .... .... ..0. .... = Management Frame Protection Capable: False
    .... ..0. .... = Peerkey Enabled: False
  # Tag: Extended Supported Rates 24, 36, 48, 54, [Mbit/sec]
  # Tag: Vendor Specific: Microsoft: WPA Information Element
    Tag Number: vendor Specific (221)
    Tag Length: 22
    OUI: 00-50-f2 (Microsoft)
    Vendor Specific OUI Type: 1
    Type: WPA Information Element (0x01)
    WPA Version: 1
  # Multicast Cipher Suite: 00-50-f2 (Microsoft) AES (CCM)
    Unicast Cipher Suite Count: 1
  # Unicast Cipher Suite List 00-50-f2 (Microsoft) AES (CCM)
    Auth Key Management (AKM) Suite Count: 1
  # Auth Key Management (AKM) List 00-50-f2 (Microsoft) PSK

```

图4-34 Probe Response帧内容

请读者注意图4-34中所示的RSN信息，它描述了AP所支持的RSN功能。下节介绍WPAS如何关联到目标无线网络时将用到它。

### 3. 关联无线网络处理流程分析

关联无线网络处理的流程从wpa\_supplicant\_connect开始，其代码如下所示。

```

[-->events.c: : wpa_supplicant_connect]

int wpa_supplicant_connect(struct wpa_supplicant *wpa_s, struct
                           wpa_bss *selected,
                           struct wpa_ssid *ssid)
{
    // WPS相关，本章不讨论
    if (wpas_wps_scan_pbc_overlap(wpa_s, selected, ssid))
{.....}
/*
    4.5.3节介绍的wpa_supplicant_enable_network函数中，reassociate值被设置为1。

```

当WPAS处于ASSOCIATING状态时，wpa\_s->pending\_bssid用于存储目标网络的BSSID。

```

        */
        if (wpa_s->reassociate || (os_memcmp(selected->bssid, wpa_s-
>bssid, ETH_ALEN) != 0 &&
        ((wpa_s->wpa_state != WPA_ASSOCIATING && wpa_s->wpa_state
!= WPA_AUTHENTICATING) ||
        os_memcmp(selected->bssid, wpa_s->pending_bssid,
ETH_ALEN) != 0))) {
            // 和EAP-SIM/AKA认证方法有关。在此处初始化相关资源。本章不讨论
            if (wpa_supplicant_scard_init(wpa_s, ssid)) {
                wpa_supplicant_req_new_scan(wpa_s, 10, 0);
                return 0;
            }
            wpa_supplicant_associate(wpa_s, selected, ssid); // 发起关联操作
        } else {.....}
        return 0;
    }
}

```

就本例而言，`wpa_supplicant_connect`最重要的工作就是触发STA发起关联操作。关联操作通过`wpa_supplicant_associate`函数来完成。这部分代码比较复杂，我们分段来看。

### (1) `wpa_supplicant_associate`分析之一

和发起无线扫描请求一样，`wpa_supplicant_associate`函数主要目的就是填充一个用于向wlan driver发起关联请求的`struct wpa_driver_associate_params`类型的对象，然后调用driver interface对应的接口函数。由于关联时考虑的因素非常多，所以对应的处理也比较烦琐。本节先介绍第一段内容。

[-->`wpa_supplicant.c`: : `wpa_supplicant_associate`代码段一]

```

void wpa_supplicant_associate(struct wpa_supplicant *wpa_s,
                               struct wpa_bss *bss, struct wpa_ssid *ssid)
{
    u8 wpa_ie[200]; size_t wpa_ie_len;
    int use_crypt, ret, i, bssid_changed;
    int algs = WPA_AUTH_ALG_OPEN; // 认证方法
    enum wpa_cipher cipher_pairwise, cipher_group; // 单播数据和组播数据加密方法
    struct wpa_driver_associate_params params; // 此函数主要目的是正确填充params的内容
    int wep_keys_set = 0; struct wpa_driver_capa capa;
    int assoc_failed = 0; struct wpa_ssid *old_ssid;
}

```

```

#define CONFIG_HT_OVERRIDES
    .....// 802.11n相关的内容
#endif /* CONFIG_HT_OVERRIDES */
#ifndef CONFIG_IBSS_RSN
    .....// IBSS相关内容
#endif /* CONFIG_IBSS_RSN */
#ifndef ANDROID_P2P
    int freq = 0;
#endif

    if (ssid->mode == WPAS_MODE_AP || ssid->mode ==
WPAS_MODE_P2P_GO ||
        ssid->mode == WPAS_MODE_P2P_GROUPFORMATION) { .....
return; }
#ifndef CONFIG_TDLS
    .....// TDLS是WFA定义的另外一项规范。本书不讨论
#endif /* CONFIG_TDLS */

/*
我们曾在4.3.3节功能相关成员变量及背景知识中曾介绍过CONFIG_SME相关信息。
Galaxy Note 2不支持WPA_DRIVER_FLAGS_SME参数。
*/
    if ((wpa_s->drv_flags & WPA_DRIVER_FLAGS_SME) && ssid->mode
== IEEE80211_MODE_INFRA) {
        sme_authenticate(wpa_s, bss, ssid);
        return;
    }
    os_memset(&params, 0, sizeof(params));
    wpa_s->reassociate = 0;
    // wpas_driver_bss_selection于判断是否由wlan driver来完成无线网络选择，本例它返回FALSE
    if (bss && !wpas_driver_bss_selection(wpa_s)) {
#ifndef CONFIG_IEEE80211R
        .....// 802.11R相关
#endif /* CONFIG_IEEE80211R */
        bssid_changed = !is_zero_ether_addr(wpa_s->bssid);
        os_memset(wpa_s->bssid, 0, ETH_ALEN); // 设置
wpa_supplicant bssid数组成员值为全0
        // 将BSSID复制到wpas->pending_bssid中
        os_memcpy(wpa_s->pending_bssid, bss->bssid, ETH_ALEN);
        if (bssid_changed)
            wpas_notify_bssid_changed(wpa_s);
#ifndef CONFIG_IEEE80211R
        .....// 和802.11R有关
#endif /* CONFIG_IEEE80211R */
#ifndef CONFIG_WPS

```

```

} else if (.....// WPS相关) {
    .....
#endif /* CONFIG_WPS */
} else os_memset(wpa_s->pending_bssid, 0, ETH_ALEN);

// 取消计划扫描和普通扫描任务
wpa_supplicant_cancel_sched_scan(wpa_s);
wpa_supplicant_cancel_scan(wpa_s);
/*
    清空上一次association时使用的WPA/RSN IE信息，这些信息保存在
wpa_supplicant
    对象的assoc_wpa_ie（类型为u8*）对应的buffer中。
*/
wpa_sm_set_assoc_wpa_ie(wpa_s->wpa, NULL, 0);

```

如果不考虑各种编译宏选项（对WPS、802.11R支持），上述代码段还算比较简单。

## (2) wpa\_supplicant\_associate分析之二

接着来看代码段二。

[-->wpa\_supplicant.c: : wpa\_supplicant\_associate代码段二]

```

.....// 接代码段一
#ifndef IEEE8021X_EAPOL
    // 本例中，为目标无线网络配置的key_mgmt为SET_NETWORK 0 key_mgmt
    WPA-PSK
        // 通过"SET_NETWORK 0 key_mgmt WPA-PSK"命令来完成
        if (ssid->key_mgmt & WPA_KEY_MGMT_IEEE8021X_NO_WPA) {
            .......// 处理key_mgmt为非WPA的情况}
#endif /* IEEE8021X_EAPOL */
/*
    auth_alg为认证方法，可取值有WPA_AUTH_ALG_OPEN、
    WPA_AUTH_ALG_SHARED等。根据第3章
    对WPA的介绍，如果要使用WPA的话，在和AP关联时必须使用Open System（即
    WPA_AUTH_ALG_OPEN）。
    如果没有设置该值，其值默认为0。
*/
if (ssid->auth_alg) {.....}
/*
    下面这个if判断很重要，请读者注意。
    wpa_bss_get_vendor_ie函数用于获取wpa_bss中的和vendor相关的IE，
    此处IE对应的标志是

```

WPA\_IE\_VENDOR\_TYPE（该值被定义为：#define WPA\_IE\_VENDOR\_TYPE 0x0050f201）。

读者可参考图4-34中Probe Response帧中最后一个IE项（Microsoft: WPA IE，

因为MS的OUI是00-50-f2）。注意：该OUI也供WFA定义的几种规范使用。

wpa\_bss\_get\_ie(bss, WLAN\_EID\_RSN) 用于获取RSN IE。图4-34中也包含RSN IE。

wpa\_key\_mgmt\_wpa(ssid->key\_mgmt) 用于判断无线网络配置时设置的 key\_mgmt 是否

和WPA相关（本例中，key\_mgmt被设为WPA-PSK，它属于WPA的一种）。

简而言之，下面这个if条件就是判断目标无线网络是否支持WPA/RSN功能，并且无线网络配置项

是否也设置key\_mgmt与WPA相关。很显然，本例满足下面这个if条件。

```
/*
if (bss && (wpa_bss_get_vendor_ie(bss, WPA_IE_VENDOR_TYPE)
|| wpa_bss_get_ie(bss, WLAN_EID_RSN)) &&
wpa_key_mgmt_wpa(ssid->key_mgmt)) {
    int try_opportunistic;
    /*
    ssid->proto默认值为DEFAULT_PROTO，它由4.5.1节中ADD_NETWORK
命令处理的
    wpa_config_set_network_defaults函数完成。
    proactive_key_caching默认值为0。
    */
    try_opportunistic = ssid->proactive_key_caching &&
(ssid->proto & WPA_PROTO_RSN);
    /*
    下面这个函数用于从pmksa缓存中取出current_ssid对应的pmkid
cache项（类型为rsn_
    pmksa_cache），然后将其赋值给wpa_sm中的cur_pmksa变量中。此时
我们还没有pmksa
    缓存信息，故if条件失败。
    */
    if (pmksa_cache_set_current(wpa_s->wpa, NULL, bss-
>bssid, wpa_s->current_ssid,
        try_opportunistic) == 0) {
        // 设置eapol_sm的cached_pmk为1（由该函数第二个参数决
定），表示要使用pmksa
        eapol_sm_notify_pmkid_attempt(wpa_s->eapol, 1);
    }
    wpa_ie_len = sizeof(wpa_ie);
    /*
    wpa_supplicant_set_suites函数比较繁琐，其目的是生成一个用于
关联请求的IE信息。

```

这些信息包括：group\_cipher、pairwise\_cipher、key\_mgmt。信息的选择需要考虑AP的情况，

即图4-34中AP包含的RSN和WPA IE。最终的选择如下：

```
proto=WPA_PROTO_RSN,
group_cipher=pairwise_cipher=WPA_CIPHER_CCMP
    key_mgmt=WPA_KEY_MGMT_PSK
    感兴趣的读者不妨自行阅读wpa_supplicant_set_suites。
/*
    if (wpa_supplicant_set_suites(wpa_s, bss, ssid, wpa_ie,
&wpa_ie_len)) {.....}
    } else if (wpa_key_mgmt_wpa_any(ssid->key_mgmt)) {
        .....
#endif CONFIG_WPS
        .....// WPS处理
#endif /* CONFIG_WPS */
    } else {.....}
.....// P2P和interworking相关的处理
// 清除wlan driver中的key设置
wpa_clear_keys(wpa_s, bss ? bss->bssid : NULL);
use_crypt = 1;
/*
将WPAS中定义的数据类型转换成driver wrapper使用的数据类型。例如，WPAS
使用WPA_CIPHER_CCMP，
而driver wrapper对应的值为CIPHER_CCMP。
*/
cipher_pairwise = cipher_suite2driver(wpa_s-
>pairwise_cipher);
cipher_group = cipher_suite2driver(wpa_s->group_cipher);
if (wpa_s->key_mgmt == WPA_KEY_MGMT_NONE ||
    wpa_s->key_mgmt == WPA_KEY_MGMT_IEEE8021X_NO_WPA)
{.....}
if (wpa_s->key_mgmt == WPA_KEY_MGMT_WPS) use_crypt = 0;

#ifndef IEEE8021X_EAPOL
    if (wpa_s->key_mgmt == WPA_KEY_MGMT_IEEE8021X_NO_WPA) {
        .....// 选择单播和组播数据加密方法}
#endif /* IEEE8021X_EAPOL */

if (wpa_s->key_mgmt == WPA_KEY_MGMT_WPA_NONE) {.....}
// 设置wpa_sm的状态为WPA_ASSOCIATING
wpa_supplicant_set_state(wpa_s, WPA_ASSOCIATING);
```

这段代码主要是根据AP的情况选择合适的加密方法及认证方法。虽然目的很简单，但判断条件以及相关函数却比较烦琐，而且还有相当一

部分代码是针对P2P、WPS等不同情况的处理。建议读者先了解其目的，以后根据工作需要再来研读其中的细节。

### (3) wpa\_supplicant\_associate分析之三

下面来看wpa\_supplicant\_associate最后一段代码。

[-->wpa\_supplicant.c: : wpa\_supplicant\_associate代码段三]

```
.....// 接代码段二
if (bss) { // 填充struct wpa_driver_associate_params中的信息
    params.ssid = bss->ssid;
    params.ssid_len = bss->ssid_len;
    if (!wpas_driver_bss_selection(wpa_s)) {
        params.bssid = bss->bssid;
        params.freq = bss->freq;
    }
}.....// 其他处理
// 填充参数
params.wpa_ie = wpa_ie;    params.wpa_ie_len = wpa_ie_len;
params.pairwise_suite = cipher_pairwise; params.group_suite
= cipher_group;
params.key_mgmt_suite = key_mgmt2driver(wpa_s->key_mgmt);
params.wpa_proto = wpa_s->wpa_proto;params.auth_alg = algs;
params.mode = ssid->mode;
for (i = 0; i < NUM_WEP_KEYS; i++) {.....// 设置wep key相关信息}
params.wep_tx_keyidx = ssid->wep_tx_keyidx;
.....
/*
设置wlan driver是否丢弃未加密数据包。注意，在通过RSN身份验证前，EAPOL 4-Way Handshake等
EAPOL数据是没有加密的，所以这个变量只针对非EAPOL/EAP数据包。本例中use_crypt为1。
*/
params.drop_unencrypted = use_crypt;

#ifndef CONFIG_IEEE80211W
    .... // 802.11w相关
#endif /* CONFIG_IEEE80211W */

params.p2p = ssid->p2p_group;
/*
uapsd为Unscheduled Automatic Power Save Delivery的缩写，也称为
WMM power save,
```

属于WFA定义的一种标准，和节电有关。

```
/*
if (wpa_s->parent->set_sta_uapsd)
    params.uapsd = wpa_s->parent->sta_uapsd;
else
    params.uapsd = -1;

#ifndef ANDROID_P2P
.....// P2P相关
#endif
// 调用driver interface的associate函数以发起关联请求。该函数非常重
要，下一节介绍
ret = wpa_drv_associate(wpa_s, &params);
if (ret < 0) {.....// 错误处理}

if (wpa_s->key_mgmt == WPA_KEY_MGMT_WPA_NONE) {.....// 其他
处理
#endif CONFIG_IBSS_RSN
} else if (.....) {.....// IBSS相关
#endif /* CONFIG_IBSS_RSN */
} else {
    int timeout = 60;// 设置一个超时时间，身份认证必须在60秒内完成
    if (assoc_failed) timeout = ssid->mode ==
WPAS_MODE_IBSS ? 10 : 5;
    else if (wpa_s->conf->ap_scan == 1) // 对本例而言，timeout
最终取值为10秒
    timeout = ssid->mode == WPAS_MODE_IBSS ? 20 : 10;
    // 设置一个超时任务：对应函数为wpa_supplicant_timeout，用于处
理身份认证超时的情况
    wpa_supplicant_req_auth_timeout(wpa_s, timeout, 0);
}
.....
if (wpa_s->current_ssid && wpa_s->current_ssid != ssid)
    eapol_sm_invalidate_cached_session(wpa_s->eapol);// 设置
上一次eapol session无效

old_ssid = wpa_s->current_ssid;
wpa_s->current_ssid = ssid;// 设置wpa_supplicant对象的
current_ssid和current_bss变量
wpa_s->current_bss = bss;
// 下面这个函数实际上是将加密/身份验证信息设置到wpa_sm对应的变量中去
wpa_supplicant_rsn_supp_set_config(wpa_s, wpa_s-
>current_ssid);
// 配置eapol sm和eap sm。其中：portControl被置为AUTO
// eapSuccess=altAccept=eapFail=altReject=FALSE。这个过程没有状
态发生变化
```

```

wpa_supplicant_initiate_eapol(wpa_s);
if (old_ssid != wpa_s->current_ssid)
    wpas_notify_network_changed(wpa_s);
}

```

至此，`wpa_supplicant_associate`函数就分析完毕。读者如果还记得第3章关于STA加入AP的步骤的话，可能会有一些疑惑。STA加入AP的顺序是先发送Authentication请求，然后再发送Association请求。但此处仅调用了`wpa_drv_associate`函数，似乎忽略了Authentication这一步。该问题的回答将放在下一节，直接来看driver nl80211对应的`wpa_driver_nl80211_associate`函数。

#### (4) `wpa_driver_nl80211_associate`函数分析

代码如下所示。

```

[-->driver_nl80211.c: : wpa_driver_nl80211_associate]

static int wpa_driver_nl80211_associate(void *priv,
                                         struct wpa_driver_associate_params
                                         *params)
{
    struct i802_bss *bss = priv;
    struct wpa_driver_nl80211_data *drv = bss->drv;
    int ret = -1; struct nl_msg *msg;

    // 处理AP和IBSS的情况
    if (params->mode == IEEE80211_MODE_AP) return
wpa_driver_nl80211_ap(drv, params);
    if (params->mode == IEEE80211_MODE_IBSS)
        return wpa_driver_nl80211_ibss(drv,
params);

    // 目前大部分手机不支持WPA_DRIVER_FLAGS_SME
    if (!(drv->capa.flags & WPA_DRIVER_FLAGS_SME)) {
        enum nl80211_iftype nlmode = params->p2p ? // 本例对应的是
非p2p情况
            NL80211_IFTYPE_P2P_CLIENT : NL80211_IFTYPE_STATION;
        // 设置设备类型，可参考4.3.4节
wpa_driver_nl80211_finish_drv_init函数分析
        if (wpa_driver_nl80211_set_mode(priv, nlmode) < 0)
return -1;
        return wpa_driver_nl80211_connect(drv, params);
    }
}

```

```

/*
注意：根据第3章对STA加入AP操作的描述，STA需要首先发送
authentication请求，然后再发送
association请求。在“wpa_supplicant_associate分析之一”中，有一个
sme_authenticate
函数，它将发起authentication请求。对于不支持SME的wlan driver来说，WPAS只要向wlan driver
发送connect命令即可完成association和authentication这两步。这种实现方式和wlan driver
以及cfg80211有关。本书不拟对此展开详细讨论。读者可在wpa_supplicant
官方源码中利用
"git log a8c5b43a"命令查看相关信息。
*/
.....// 对于支持SME的wlan driver，则需要发送
NL80211_CMD_ASSOCIATE命令
}

```

来看wpa\_driver\_nl80211\_connect函数，其代码如下所示。

```

[-->driver_nl80211.c: : wpa_driver_nl80211_connect]

static int wpa_driver_nl80211_connect(struct
wpa_driver_nl80211_data *drv,
    struct wpa_driver_associate_params *params)
{
    struct nl_msg *msg;    enum nl80211_auth_type type;
    int ret = 0;    int algs;

    msg = nlmsg_alloc();
    nl80211_cmd(drv, msg, 0, NL80211_CMD_CONNECT); // 构造一个
NL80211_CMD_CONNECT命令

    NLA_PUT_U32(msg, NL80211_ATTR_IFINDEX, drv->ifindex); // 设置
本次命令对应的网络接口
    // 设备索引号填充bssid信息
    if (params->bssid) NLA_PUT(msg, NL80211_ATTR_MAC, ETH_ALEN,
params->bssid);

    if (params->freq) {.....// 填充频率信息}
    if (params->ssid) { // 填充ssid信息
        NLA_PUT(msg, NL80211_ATTR_SSID, params->ssid_len, params-
>ssid);
        os_memcpy(drv->ssid, params->ssid, params->ssid_len); //
将ssid信息保存起来
    }
}

```

```

    drv->ssid_len = params->ssid_len;
}

.....
if (params->pairwise_suite != CIPHER_NONE) {
    .... // 处理pairwise_cipher取值情况
    NLA_PUT_U32(msg, NL80211_ATTR_CIPHER_SUITES_PAIRWISE,
cipher);
}
.... // 其他各项参数处理
// 向wlan driver发送NL80211_CMD_CONNECT命令
ret = send_and_recv_msgs(drv, msg, NULL, NULL);
msg = NULL;
if (ret) {..... // 错误处理}
ret = 0;
.....
return ret;
}

```

至此，WPAS就把CONNECT请求发给了驱动，驱动将完成Authentication帧和Association Request帧的处理。

和scan请求类似，关联无线网络处理流程主要是为了填充一个wpa\_driver\_associate\_params类型的对象。不过该对象中各个参数的设置颇有来头，需要对802.11规范有相当了解才可以做到正确无误。

## (5) 关联无线网络处理流程总结

来看一下关联无线网络处理流程所涉及的几个重要函数调用，如图4-35所示。

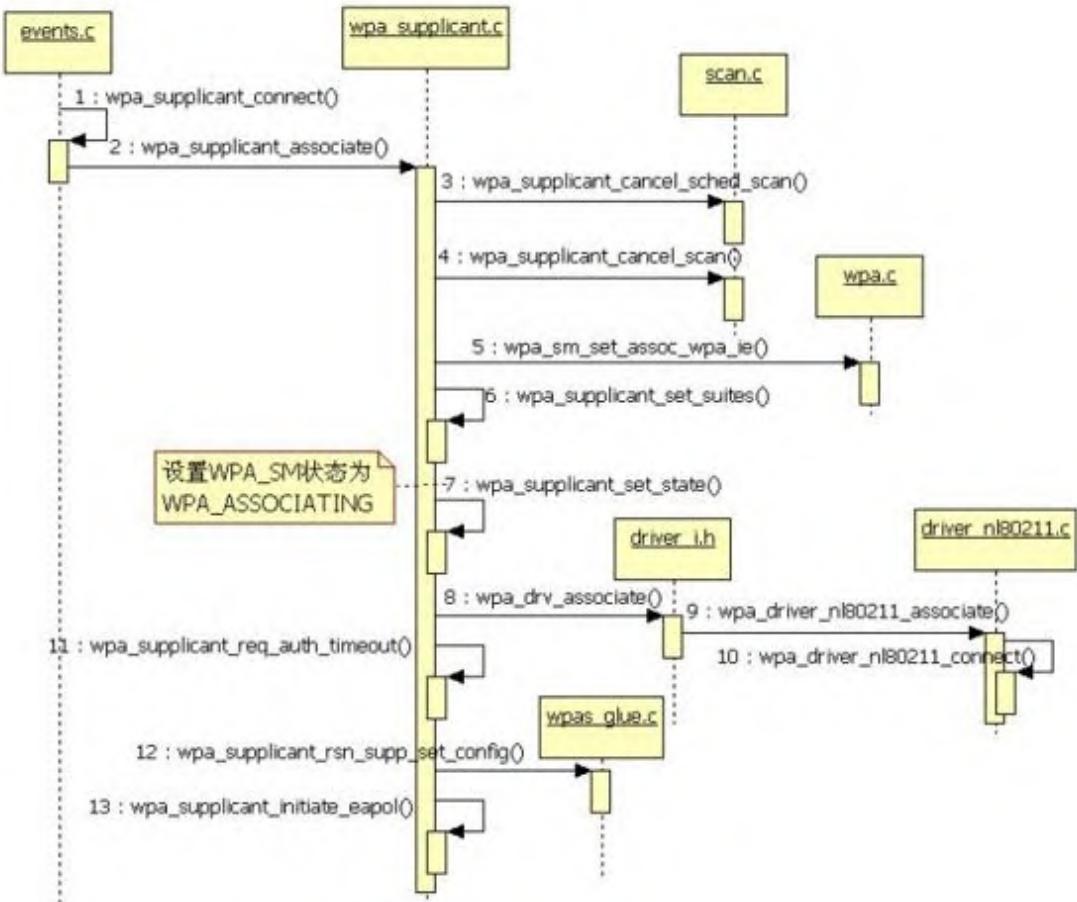


图4-35 wpa\_supplicant\_connect流程

图4-35中最复杂的就是wpa\_supplicant\_associate函数，希望读者能结合代码进行阅读。

WPAS成功调用wpa\_supplicant\_associate后，将等待NL80211\_CMD\_CONNECT命令的处理结果。该结果由wlan driver通过NL80211\_CMD\_CONNECT类型的消息返回给driver wrapper。下面一节将分析WPAS如何处理该消息。

#### 4. 关联事件处理流程分析

结合上节所述内容，WPAS中的driver wrapper将收到来自wlan driver的NL80211\_CMD\_CONNECT命令，其对应的处理代码如下所示。

[-->driver\_nl80211.c: : do\_process\_drv\_event]

```

static void do_process_drv_event(struct wpa_driver_nl80211_data
*drv, int cmd,
                                struct nlattr **tb)
{
    // 曾在4.5.3节“扫描结果处理流程分析”中见过该函数处理
    NL80211_CMD_NEW_SCAN_RESULTS
    .....
    switch (cmd) {
        .....
        case NL80211_CMD_CONNECT:// wlan driver返回该命令的处理结果
        case NL80211_CMD_ROAM:
            /*
                调用mlme_event_connect进行处理，其中NL80211_ATTR_STATUS_CODE
                属性保存了AP的处理结果
                (即Association请求的处理结果)、NL80211_ATTR_MAC属性保存了AP的
                MAC地址(就是bssid)、
                NL80211_ATTR_REQ_IE属性保存了Association Request请求时包含的
                IE (STA发送的IE)、
                NL80211_ATTR_RESP_IE属性保存了Association Response帧包含的IE
                信息 (AP发送的IE)。
            */
            mlme_event_connect(drv, cmd,
                tb[NL80211_ATTR_STATUS_CODE],
                tb[NL80211_ATTR_MAC], tb[NL80211_ATTR_REQ_IE],
                tb[NL80211_ATTR_RESP_IE]);
            .....
    }
}

```

当wlan driver返回NL80211\_CMD\_CONNECT命令时，其真正的处理函数实际上是mlme\_event\_connect，此函数代码如下所示。

```

[-->driver_nl80211.c: : mlme_event_connect]

static void mlme_event_connect(struct wpa_driver_nl80211_data
*drv,
                                enum nl80211_commands cmd, struct nlattr
*status, struct nlattr *addr,
                                struct nlattr *req_ie, struct nlattr *resp_ie)
{
    /*
        driver wrapper通知WPAS其他模块时候使用的事件类型。我们曾在4.5.3节“扫
        描结果处理流程分析”
        中对send_scan_event函数分析时见过。
    */
    union wpa_event_data event;

```

```

    if (drv->capa.flags & WPA_DRIVER_FLAGS_SME) {.....// wlan
driver支持SME时的处理}

    os_memset(&event, 0, sizeof(event));
    if (cmd == NL80211_CMD_CONNECT && nla_get_u16(status) !=
WLAN_STATUS_SUCCESS) {
        .....// 关联AP失败的处理
        wpa_supplicant_event(drv->ctx, EVENT_ASSOC_REJECT,
&event);
        return;
    }

    drv->associated = 1;
    if (addr)// 保存bssid
        os_memcpy(drv->bssid, nla_data(addr), ETH_ALEN);// 保存
bssid信息

    if (req_ie) {// 保存Association Request ie信息
        event.assoc_info.req_ies = nla_data(req_ie);
        event.assoc_info.req_ies_len = nla_len(req_ie);
    }
    if (resp_ie) {// 保存Association Response ie信息
        event.assoc_info.resp_ies = nla_data(resp_ie);
        event.assoc_info.resp_ies_len = nla_len(resp_ie);
    }
    // 通过发送NL80211_GET_SCAN命令获取STA的工作频率
    event.assoc_info.freq = nl80211_get_assoc_freq(drv);
    // 重要函数，见下文分析
    wpa_supplicant_event(drv->ctx, EVENT_ASSOC, &event);
}

```

wpa\_supplicant\_event中处理EVENT\_ASSOC消息的是  
wpa\_supplicant\_event\_assoc，下面将分段介绍它。

### (1) wpa\_supplicant\_event\_assoc分析之一

代码如下所示。

[-->events.c: : wpa\_supplicant\_event\_assoc代码段一]

```

static void wpa_supplicant_event_assoc(struct wpa_supplicant
*wpa_s,
                                         union wpa_event_data *data)
{
    u8 bssid[ETH_ALEN];  int ft_completed;

```

```

int bssid_changed; struct wpa_driver_capa capa;
.....// CONFIG_AP处理
// 判断Fast Transition是否完成，由于本例没有设置CONFIG_80211R宏，所以下面这个函数返回0
ft_completed = wpa_ft_is_completed(wpa_s->wpa);
/*
在4.5.3节“wpa_supplicant_associate分析之一”中，wpa_supplicant对象的assoc_wpa_ie被清空。此处需要保存这些IE信息。wpa_supplicant_event_associnfo比较烦琐，主要是更新RSN/WPA IE信息，请感兴趣的读者自行阅读。
*/
if (data && wpa_supplicant_event_associnfo(wpa_s, data) < 0)
return;

wpa_supplicant_set_state(wpa_s, WPA_ASSOCIATED); // 设置wpa_sm装为WPA_ASSOCIATED

// 从driver wrapper中获得bssid信息
if
(wpa_drv_get_bssid(wpa_s, bssid) >= 0 && os_memcmp(bssid, wpa_s->bssid, ETH_ALEN) != 0) {
    random_add_randomness(bssid, ETH_ALEN); // 和随机数相关
    bssid_changed = os_memcmp(wpa_s->bssid, bssid, ETH_ALEN);
    os_memcpy(wpa_s->bssid, bssid, ETH_ALEN); // 设置bssid
    os_memset(wpa_s->pending_bssid, 0, ETH_ALEN); // 清空pending_bssid
    if (bssid_changed)
        wpas_notify_bssid_changed(wpa_s);

    // 和动态WEP Key有关，本书不讨论
    if
(wpa_supplicant_dynamic_keys(wpa_s) && !ft_completed) wpa_clear_keys(wpa_s, bssid);
    /*
就本例而言（ap_scan为1，并且current_ssid不为空），下面这个函数作用不大。对其他情况而言，该函数负责的工作我们在前面都见识过了。
*/
    if (wpa_supplicant_select_config(wpa_s) < 0) {.....// 错误处理}
    /*
这个if条件对应的代码段让笔者非常困惑，因为在4.5.3节“wpa_supplicant_associate

```

分析之三”的最后已经为wpa\_s->current\_bss赋值了。此处为何又再赋值？通过笔者实测，bss

和current\_bss的值是相同的。添加这段代码的人也没有说明为什么（读者不妨参考“git show

8f770587”命令的结果）。有知晓其来龙去脉的读者不妨和大家分享相关知识。

```
/*
if (wpa_s->current_ssid) {
    struct wpa_bss *bss = NULL;
    bss = wpa_supplicant_get_new_bss(wpa_s, bssid);
    if (!bss) {
        wpa_supplicant_update_scan_results(wpa_s);
        bss = wpa_supplicant_get_new_bss(wpa_s, bssid);
    }
    if (bss)
        wpa_s->current_bss = bss;
}
if (wpa_s->conf->ap_scan==1&&wpa_s-
>drv_flags&WPA_DRIVER_FLAGS_BSS_SELECTION) {
    .....// wlan driver支持BSS SELECTION功能时对应的处理
}
}
```

wpa\_supplicant\_event\_assoc函数的历史比较悠久（根据git commit信息，它从0.6.3版本开始就存在。而实际历史可能还要更久，因为WPAS从0.6.3版本后才开始使用git来管理代码）。在笔者研究的WPAS相关代码中，wpa\_supplicant\_event\_assoc函数算是相当复杂的了。其中一个主要原因是该函数内部有很多细节需要考虑，有些细节甚至是为了解决某个bug。对于初学者而言，笔者建议可只关注该函数涉及的主要流程。

## (2) wpa\_supplicant\_event\_assoc分析之二

接下来分析该函数最后一段代码。

[-->events.c: : wpa\_supplicant\_event\_assoc代码段二]

```
.....// 接代码段一
#ifndef CONFIG_SME
    .....// 支持SME的情况
#endif /* CONFIG_SME */

if (wpa_s->current_ssid) {
/*
```

初始化SIM/USIM卡。根据代码中的注释：在ap\_scan为1的情况下，该函数在此之前就被调用过。

而其他情况下，需要在此处调用它。在4.5.3节关联无线网络处理流程分析中见过此函数。

```
/*
    wpa_supplicant_scard_init(wpa_s, wpa_s->current_ssid);
}
wpa_sm_notify_assoc(wpa_s->wpa, bssid);
// 初始化wpa_sm中和后续EAPOL-Key交换相关联的变量
/*
    wpa_s->l2在4.3.4节wpa_supplicant_driver_init函数分析中通过
l2_packet_init函数调用被赋值。
    所以下面这个if条件为真。对Linux平台来说,
l2_packet_notify_auth_start
    实际上是一个空函数。
*/
if (wpa_s->l2) l2_packet_notify_auth_start(wpa_s->l2);

if (!ft_completed) {// 设置EAPOL相关外部变量
    eapol_sm_notify_portEnabled(wpa_s->eapol, FALSE);
    eapol_sm_notify_portValid(wpa_s->eapol, FALSE);
}
// 本例采用的就是WPA-PSK认证算法
if (wpa_key_mgmt_wpa_psk(wpa_s->key_mgmt) || ft_completed)
    eapol_sm_notify_eap_success(wpa_s->eapol, FALSE);
// 设置EAPOL模块中的portEnabled为TRUE，将状态机一系列动作。详情见下
文
eapol_sm_notify_portEnabled(wpa_s->eapol, TRUE);
wpa_s->eapol_received = 0;
if (.....){.....}
} else if (!ft_completed) {
    wpa_supplicant_req_auth_timeout(wpa_s, 10, 0); // 重新注
册一个认证超时任务
}
wpa_supplicant_cancel_scan(wpa_s); // 取消scan任务

.....// 处理wlan driver支持WPA_DRIVER_FLAGS_4WAY_HANDSHAKE功能
以及ft_complete
为1时的情况
```

```
if (wpa_s->pending_eapol_rx) {
/*
    pending_eapol_rx变量对应如下的应用场景。
    有时候在WPAS收到来自wlan driver的EVENT_ASSOC事件之前（即
wpa_supplicant_event_
    assoc被调用之前），AP就发送EAPOL消息过来（后文会分析相关代码）以触
```

发认证流程。而由于

WPAS还未处理EVENT\_ASSOC事件。所以，EAPOL消息会先保存到pending\_eapol\_rx中，直到

wpa\_supplicant\_event\_assoc函数中再来处理（即收到了EVENT\_ASSOC事件）。笔者测试过

程序中发现有些AP会出现这种情况。

```
* /  
}  
.....// WEP的情况  
.....// IBSS的情况  
}
```

wpa\_supplicant\_event\_assoc函数终于分析完毕。正如上一节最后所提到的那样，该函数实际上非常复杂，细节内容很难在本章篇幅内一一覆盖。同时，对初学者来说，流程比细节更重要，所以可先略过这些细节。

上述代码中，eapol\_sm\_notify\_portEnabled被调用以设置portEnabled为TRUE。EAPOL模块包含众多SM，它们是否会因为这个变量的变化而随之发生状态改变呢？来看下节。

### (3) eapol\_sm\_notify\_portEnabled函数

eapol\_sm\_notify\_portEnabled的代码如下所示。

```
[-->eapol_supp_sm.c: : eapol_sm_notify_portEnabled]  
  
void eapol_sm_notify_portEnabled struct eapol_sm *sm, Boolean  
enabled)  
{  
    ....  
    sm->portEnabled= enabled; // 设置portEnabled变量为TRUE  
    eapol_sm_step(sm); // 状态机联动。这部分代码在4.4.2节介绍“状态机联动”时出现过  
}
```

eapol\_sm\_step将依次更新SUPP\_PAE、KEY\_RX、SUPP\_BE和EAP\_SM状态信息。在该函数执行之前，这四个状态机的状态依次如下。

- SUPP\_PAE为DISCONNECTED状态；
- KEY\_RX为NO\_KEY\_RECEIVE状态；

- SUPP\_BE为IDLE状态；
- EAP\_SM为DISABLED状态。

根据代码（eapol\_supp\_sm.c中SM\_STEP(SUPP\_PAE)）以及图4-28可知，SUPP\_PAE下一个要进入的状态是CONNECTING，其EA(Entry Aciton)代码如下。

```
[-->eapol_supp_sm.c: : SM_STATE (SUPP_PAE, CONNECTING) ]

SM_STATE (SUPP_PAE, CONNECTING)
{
    // SUPP_PAE_state此时的值为SUPP_PAE_DISCONNECTED, 故send_start
    // 为0
    int send_start = sm->SUPP_PAE_state == SUPP_PAE_CONNECTING;
    SM_ENTRY (SUPP_PAE, CONNECTING);
    if (send_start) {
        sm->startWhen = sm->startPeriod;
        sm->startCount++;
    } else {
#define CONFIG_WPS
        sm->startWhen = 1;
#endif /* CONFIG_WPS */
        sm->startWhen = 3; // 设置startWhen值为3
#endif /* CONFIG_WPS */
    }
    eapol_enable_timer_tick(sm); // 使能Port Timers SM
    sm->eapoleap = FALSE;
    if (send_start) eapol_sm_txStart(sm); // 发送EAPOL Start包。
    此时该函数不会被调用
}
```

读者可参考代码以及状态机示意图以了解其他状态机的切换。该函数执行完毕后，各个状态机的变化如下。

- SUPP\_PAE进入CONNECTING状态；
- KEY\_RX不变 (NO\_KEY\_RECEIVE)；
- SUPP\_BE进入IDLE状态；
- EAP\_SM不变 (DISABLED状态)。

注意 根据图4-21关于EAP SUPP\_SM的描述，当portEnabled值为TRUE时，应该从DISABLED状态切换至INITIALIZE状态。不过，在4.5.3节“wpa\_supplicant\_associate分析之三”中曾调用过wpa\_supplicant\_initiate\_eapol函数。在该函数中，由于本例使用的认证算法是WPA-PSK，所以force\_disabled变量为TRUE，导致EAP SUPP SM不能转换至INITIALIZE状态。（参考eap.c::SM\_STEP(EAP)函数中第一个else if判断。）

#### (4) 关联事件处理流程总结

关联事件处理流程中涉及的重要函数调用如图4-36所示。注意，wpa\_supplicant\_event\_assoc中略去了部分细节代码。请读者清楚图中的几个重要函数后，再根据本节所述内容研究这些细节。

至此，WPAS已经和目标AP完成了Association操作，接下来将进入802.1X身份认证流程。根据图3-44所示，接下来的工作将是4-Way Handshake和Group Key Handshake的处理。

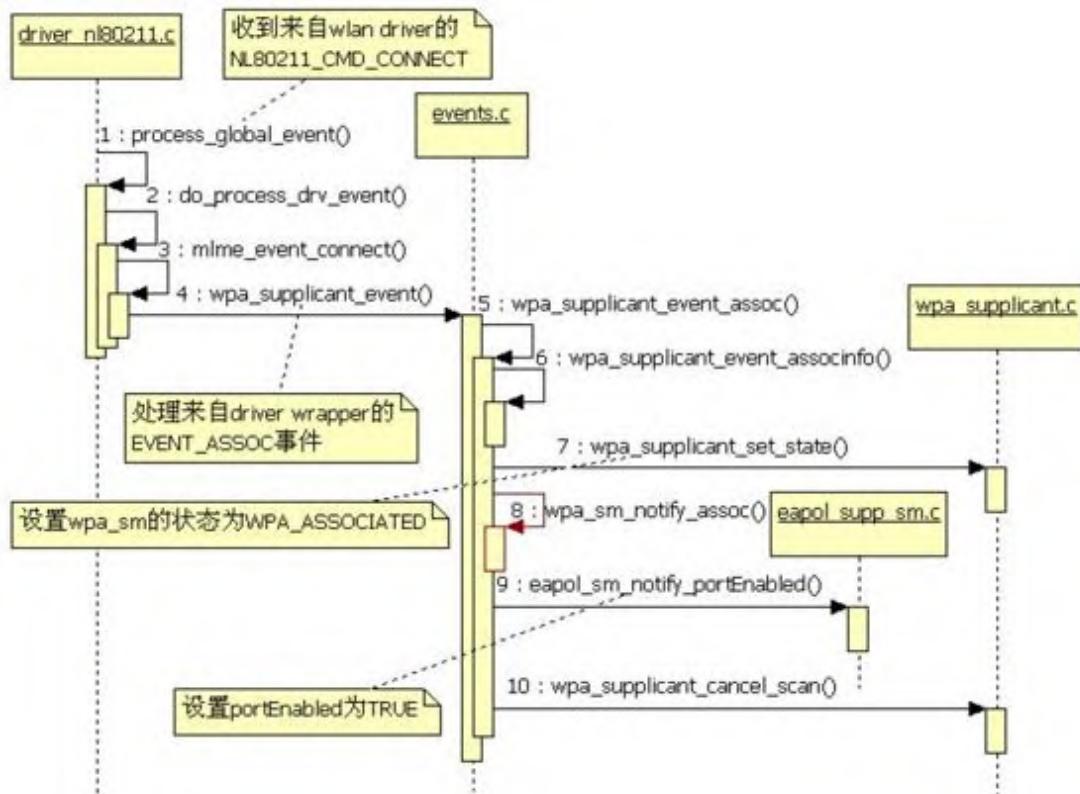


图4-36 NL80211\_CMD\_CONNECT处理流程

## 5. EAPOL-Key交换流程分析

对本例而言，本节所说的EAPOL-Key交换包括4-Way Handshake和Group Key Handshake过程（注意，由于采用的是PSK认证方式，故本节分析的交互流程将不涉及STA和Authenticator Server开展的身份认证的流程）。

首先发起的是4-Way Handshake Key交换。对于WPA-PSK认证方法来说，STA不会发送EAPOL-Start消息给AP。根据笔者的测试，完成关联操作后，Galaxy Note 2会发送一个Null function（没有实际数据）的数据包给AP（如图4-37所示），而AP接收到该消息后发现STA还未通过认证，所以它就会被触发以开始4-Way Handshake流程（参考图3-46的左图）。

75387 109.764545 Tp-LinkT_bd:87:ae	SamsungE_69:88:e2	802.11	96 Association Response, SN=1, FN=0
75389 109.772388 SamsungE_69:88:e2	Tp-LinkT_bd:87:ae	802.11	48 Null function (No data), SN=76,
75390 109.772591	SamsungE_69:88:e2	(802.11)	34 Acknowledgement, Flags=.....C
75421 109.888533 Tp-LinkT_bd:87:ae	SamsungE_69:88:e2	EAPOL	157 Key (Message 1 of 4)
75437 109.909502 SamsungE_69:88:e2	Tp-LinkT_bd:87:ae	EAPOL	179 Key (Message 2 of 4)
75438 109.909883	SamsungE_69:88:e2	(802.11)	34 Acknowledgement, Flags=.....C
75445 109.913002 Tp-LinkT_bd:87:ae	SamsungE_69:88:e2	EAPOL	237 Key (Message 3 of 4)
75450 109.916714 SamsungE_69:88:e2	Tp-LinkT_bd:87:ae	EAPOL	157 Key (Message 4 of 4)

图4-37 Null function数据包

由图4-37可知，Galaxy Note 2向AP发送一个Null function数据包后，AP就开始了4-Way Handshake流程。它首先发送一个EAPOL帧给STA（图4-37中“Message 1 of 4”这一项）。我们在4.3.4节的最后曾提到12\_packet模块（参考图4-1）用来接收PACKET类型socket数据的函数是wpa\_supplicant\_rx\_eapol。实际上，该函数就是WPAS中用来接收EAP/EAPOL数据包的。所以，wpa\_supplicant\_rx\_eapol将处理AP发送过来的EAPOL帧。马上来看此函数，代码如下所示。

```
[-->wpa_supplicant.c: : wpa_supplicant_rx_eapol]  
  
void wpa_supplicant_rx_eapol(void *ctx, const u8 *src_addr,  
                           const u8 *buf, size_t len)  
{  
    struct wpa_supplicant *wpa_s = ctx;  
    /*  
     * 读者还记得4.5.3节“wpa_supplicant_event_assoc分析之二”最后关于  
     * pending_eapol_rx  
     * 的解释吗？当wpa_state状态不为WPA_ASSOCIATED的时候，如果收到AP发来的  
     * 数据包，则先保存
```

起来，然后留待wpa\_supplicant\_event\_assoc中去处理。

```
*/
```

```
if (wpa_s->wpa_state < WPA_ASSOCIATED) {  
    wpabuf_free(wpa_s->pending_eapol_rx);  
    wpa_s->pending_eapol_rx = wpabuf_alloc_copy(buf, len); //
```

复制数据

```
if (wpa_s->pending_eapol_rx) {  
    os_get_time(&wpa_s->pending_eapol_rx_time);  
    os_memcpy(wpa_s->pending_eapol_rx_src,  
src_addr,ETH_ALEN);  
}  
return;  
}  
.....// CONFIG_AP处理
```

```
if (wpa_s->key_mgmt == WPA_KEY_MGMT_NONE) { return; }  
/*
```

eapol\_received用于记录收到的EAPOL/EAP包个数。初次进来，需要设置认证超时任务（这个

任务已经设置过多次了，不过wpa\_supplicant\_req\_auth\_timeout会先取消上一次设置的认证超时

任务，然后再设置新的）。

```
*/
```

```
if (wpa_s->eapol_received == 0 &&  
    (! (wpa_s->drv_flags & WPA_DRIVER_FLAGS_4WAY_HANDSHAKE)  
||  
    !wpa_key_mgmt_wpa_psk(wpa_s->key_mgmt) || wpa_s->wpa_state != WPA_COMPLETED) &&  
    (wpa_s->current_ssid == NULL || wpa_s->current_ssid->mode != IEEE80211_MODE_IBSS)  
) {  
    wpa_supplicant_req_auth_timeout(wpa_s,  
        (wpa_key_mgmt_wpa_ieee8021x(wpa_s->key_mgmt) ||  
        wpa_s->key_mgmt == WPA_KEY_MGMT_IEEE8021X_NO_WPA ||  
        wpa_s->key_mgmt == WPA_KEY_MGMT_WPS) ? 70 : 10, 0);  
}  
wpa_s->eapol_received++;
```

```
if (wpa_s->countermeasures) { return; }
```

```
.....// CONFIG_IBSS_RSN处理
```

```
os_memcpy(wpa_s->last_eapol_src, src_addr, ETH_ALEN);  
/*
```

对于非PSK认证方法（如802.1X认证），EAPOL/EAP帧由eapol\_sm\_rx\_eapol进行处理。

另外，与四次握手和组播密钥交换相关的EAPOL-Key帧也不在eapol\_sm\_rx\_eapol中处理。

```

*/
if (!wpa_key_mgmt_wpa_psk(wpa_s->key_mgmt) &&
    eapol_sm_rx_eapol(wpa_s->eapol, src_addr, buf, len) > 0)
return;
/*
driver_nl80211没有实现poll函数。该函数的目的是为了从驱动中获取
Association过程中涉及
的IE信息对于nl80211来说，关联完成后，我们就已经收到了这些信息，所以不需要再次获取它们了。
*/
wpa_drv_poll(wpa_s);
if (!(wpa_s->drv_flags & WPA_DRIVER_FLAGS_4WAY_HANDSHAKE))
    wpa_sm_rx_eapol(wpa_s->wpa, src_addr, buf, len); // 关键函
数。见下文分析
else if (wpa_key_mgmt_wpa_ieee8021x(wpa_s->key_mgmt)) {
    eapol_sm_notify_portValid(wpa_s->eapol, TRUE);
}
}

```

wpa\_sm\_rx\_eapol是本节的重点。在介绍之前，先来介绍EAPOL-Key交换的背景知识。一方面对3.3.7节RSNA介绍的补充，另一方面该背景知识对我们理解wpa\_sm\_rx\_eapol函数也有重要帮助。

### (1) 背景知识介绍<sup>[22][23]</sup>

由3.3.7节可知，RSNA过程中有两组Key需要交换和派生。

- PTK (Pairwise Transient Key, 成对临时密钥) 用于加解密单播数据。它从PMK (Pairwise Master Key) 派生 (derive) 而来。对于SOHO网络环境来说，PMK就是PSK。它由用户设置的passphrase扩展而来（请读者参考4.5.2节中的wpa\_config\_update\_psk）。AP和STA需要通过4-Way Handshake协议交换和派生PTK。
- GTK (Group Transient Key, 组临时密钥) 用于加解密组播数据。它由GMK (Group Master Key, 其来源由AP或Authenticator Server决定) 派生而来。AP和STA通过Group Key Handshake协议交换和派生GTK。Group Key Handshake必须在完成4-Way Handshake之后才能开展。

**提示** Group Key Handshake对应的场景比较特殊。对组播数据而言，AP和所有和它关联的STA使用同一个GTK。不过，如果中途有STA离开（取消和AP的关联）的话，从安全角度考虑，AP和剩下的STA之间最

好更换一个新的GTK。AP将根据情况来触发Group Key Handshake流程。另外，AP可通过4-Way Handshake告知STA其当前使用的GTK。这种情况下，AP和STA之间无须开展Group Key Handshake流程。

下面，通过实例来介绍4-Way Handshake涉及的四次帧交换以及其中包含的数据信息。

第一个EAPOL-Key帧（以后简称Message A）由AP发送给STA，其内容如图4-38所示。Key Descriptor Type及以下的内容属于Key Descriptor。Key Descriptor用来描述EAPOL-Key帧的Key信息，由多个字段组成。

- 第一个字段是Key Descriptor Type，目前取值有三个，分别是RC4、WPA和RSN（即WPA2），802.11中可使用后两个。
- Key Information字段（2字节）包含多个标志位。下文将一一介绍它们。
- Key Length字段（2字节）表示PTK密钥的长度。对CCMP来说，它是16字节（128位），而对TKIP来说，其长度是32字节（256位）。读者可参考图3-47中的CCMP-TK、TKIP-TK和TKIP-MIC Key。CCMP和TKIP长度不一致的原因是CCMP可用一个Key完成数据加解密和MIC校验，而TKIP使用不同的Key来完成加解密以及MIC校验。
- Replay Counter字段（8字节）和防止重放攻击有关。一个简单的应用场景为STA之前收到过Replay Counter为1的包。假如它又收到一个Replay Counter为0的包，则认为发生了重放攻击，STA将丢弃Replay Counter为0的包。对Message A来说，该值必须为0。
- Key Nonce字段（32字节）用于存储Nonce值。对Message A来说，此Nonce值由AP生成，所以也叫ANonce。
- Key IV字段（16字节）表示初始向量，用于Key生成。Message A中该字段为全0。
- Key RSC字段（8字节）是Key Receive Sequence Counter的缩写，也和重放攻击有关。该字段用于四次握手的第三帧以及组播Key握手的第一帧中。

- Key ID字段（8字节）对WPA/RSN来说，该字段未使用。
- Key MIC字段（可变字节长度）表示存储MIC数据，其长度和具体的算法有关。笔者查询了802.11文档<sup>[22]</sup>，规范中列出的几种算法对应的MIC长度都是16字节。
- Key Data Length字段（2字节）表示Key Data长度。图4-38没有携带Key Data，所以该项为0。如果该项不为0，Key Descriptor还将在Key Data Length后添加一个“Key Data”项。

```

└─ 802.1X Authentication
    Version: 802.1X-2004 (2)
    Type: Key (3)
    Length: 95
    Key Descriptor Type: EAPOL RSN Key (2)
    └─ Key Information: 0x008a
        ..... .010 = Key Descriptor Version: AES Cipher, HMAC-SHA1 MIC (2)
        ..... .... 1... = Key Type: Pairwise Key
        ..... .... .00 ... = Key Index: 0
        ..... .... 0... .... = Install: Not set
        ..... .... 1.... .... = Key ACK: Set
        ..... .... 0 ..... .... = Key MIC: Not set
        ..... .... 0. .... .... = Secure: Not set
        ..... .... 0.. .... .... = Error: Not set
        ..... .... 0.... .... = Request: Not set
        ..... .... 0 ..... .... = Encrypted Key Data: Not set
        Key Length: 16
        Replay Counter: 1
        WPA KeyNonce: c51a0100b8ac79e245c50067ffbb2331696bd2b632542980...
        Key IV: 00000000000000000000000000000000
        WPA Key RSC: 0000000000000000
        WPA Key ID: 0000000000000000
        WPA Key MIC: 00000000000000000000000000000000
        WPA Key Data Length: 0

```

Key Descriptor

图4-38 Message A的内容

图4-38中，Key Information字段的内容如图4-39所示。

B0	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11	B12	B13	B14	B15
Key Descriptor Version	Key Type	Reserved	Install	Key Ack	Key MIC	Se- cu- re	Error	Request	Encrypted Key Data	SMK Message	Reserved			

图4-39 Key Information字段

如图4-39所示：

- Key Descriptor Version: 用于指示Key Descriptor的版本号。当值为2时，表示Key Descriptor的Key Data用AES加密，而Key MIC由HMAC-SHA1算法计算而来。
- Key Type: 值为1，表示该帧用于PTK派生，值为0表示该帧用于GTK派生。
- Reserved: 该字段必须为0（图4-38中，该字段也被称为Key Index）。
- Install: 当Key Type为1时，Install值为1表示STA需要安装PTK（将PTK传给驱动，下文源码分析时将看到相关函数）。Key Type为0时，Install必须为0。
- Key ACK: AP发送EAPOL-Key帧给STA时，如果需要STA发送回复数据，则设置其为1。
- Key MIC: 如果EAPOL-Key帧包含MIC信息，其值被设为1，否则为0。
- Secure: 当STA或AP派生出了PTK或GTK后，STA或AP发出的EAPOL-Key帧将该值设为1。
- Error: 使用TKIP时，如果STA检查到MIC错误，则设置该值为1。对于其他情况下的MIC错误，该值和Request都必须被设为1。
- Request: STA请求AP发起4-Way Handshake（Key Type同时被设为1）或者Group Key Handshake（Key Type同时被设为0）流程时，其值被设为1。或者和Error都被设为1以报告MIC错误。
- Encrypted Key Data: 表示Key Data是否加密。
- SMK: Station-to-station link Master Key的缩写，是另外一种Key交换协议。本书不讨论。

同上述介绍，相信读者对Key Descriptor有了一定的了解。现在马上来介绍四次握手协议中每个EAPOL-Key帧所包含的内容以及接收方的处理逻辑。

Message A由AP发送给STA，其内容如下。

- Key Info: Error位为0，因为这是第一帧数据。Secure位为0表示该帧没有加密的数据。Key MIC为0表示该帧不包含MIC数据。Key ACK位为1表示AP要求STA回复此帧。Key Install为0表示现在还无法安装PTK。Key Type设为1表示当前是Pairwise密钥派生。本例中，Key Description Version值为2。
- Replay Counter: 本例中它被设为1。STA需要保存这个值以检测重放攻击。
- KeyNonce: AP生成的随机数，也叫ANonce。
- 由于本帧不包含Key信息，所以其他字段都设为0。

提醒 802.11规范中指出，Message A的Key Data可以携带PMKID信息（包含在RSN IE或其他厂商自定义的IE中）。不过本例中，Message A没有携带它。

STA收到Message A的处理逻辑如下。

- 1) 生成一个Nonce，也叫SNonce。
- 2) 派生PTK。
- 3) 构造第二个EAPOL-Key帧，称为Message B。

图4-40为Message B的截图。

```

[+] 802.1X Authentication
    version: 802.1X-2001 (1)
    Type: Key (3)
    Length: 117
    Key Descriptor Type: EAPOL RSN Key (2)
    [+] Key Information: 0x010a
        .... .... .010 = Key Descriptor Version: AES Cipher, HMAC-SHA1 MIC (2)
        .... .... 1... = Key Type: Pairwise Key
        .... .... 00 .... = Key Index: 0
        .... .... 0... .... = Install: Not set
        .... .... 0.... .... = Key ACK: Not set
        .... .... 1 .... .... = Key MIC: Set
        .... .... 0. .... .... = Secure: Not set
        .... .... 0.... .... = Error: Not set
        .... .... 0.... .... = Request: Not set
        .... .... .... .... = Encrypted Key Data: Not set
    Key Length: 0
    Replay Counter: 1
    WPA KeyNonce: fdfe9f8acc81387a5d446171ae2fd10d63462f1682d31d6...
    Key IV: 00000000000000000000000000000000
    WPA Key RSC: 0000000000000000
    WPA Key ID: 0000000000000000
    WPA Key MIC: bfec81d8e21e944e1730f761d335e1af
    WPA Key Data Length: 22
    [+] WPA Key Data: 30140100000fac040100000fac040100000fac020000
        [+] Tag: RSN Information
            Tag Number: RSN Information (48)
            Tag length: 20
            RSN Version: 1
            [+] Group Cipher Suite: 00-0f-ac (IEEE8021) AES (CCM)
                Pairwise Cipher Suite Count: 1
            [+] Pairwise Cipher Suite List 00-0f-ac (IEEE8021) AES (CCM)
                Auth Key Management (AKM) Suite Count: 1
            [+] Auth Key Management (AKM) List 00-0f-ac (IEEE8021) PSK
            [+] RSN Capabilities: 0x0000

```

图4-40 Message B的内容

如图4-40所示：

- 由Key Info的设置可知该帧包括MIC数据（Key MIC位为1）。
- Replay Counter的值必须等于Message A中的Replay Counter。
- Key MIC是对整个EAPOL-Key帧进行计算而来，计算方法由Key Descriptor Version指定（图4-40中的HMAC-SHA1 MIC方法）。
- Key Data包含一个RSN Information Element。该RSN IE来自STA之前和AP在关联操作时获得的RSN IE。规范没有说明为何此处要包含RSN IE。不过[23]倒是有一句简单的说法。即为了防止STA中途更改安全参数。

AP收到Message B的处理如下。

- 1) 派生PTK。
- 2) 检查Message B的MIC值是否正确。如果发现MIC错误，则丢弃（而且是silently）Message B。

这种情况下，4-Way Handshake流程已经被中断。如果MIC正确，AP将构造第三个EAPOL-Key帧，此处称为Message C，其内容如图4-41所示。

```
802.1X Authentication
Version: 802.1X-2004 (2)
Type: Key (3)
Length: 175
Key Descriptor Type: EAPOL RSN Key (2)
Key Information: 0x13ca
    .... .... .010 = Key Descriptor Version: AES Cipher, HMAC-SHA1 MIC (2)
    .... .... 1... = Key Type: Pairwise Key
    .... .... .00 .... = Key Index: 0
    .... .... .1.. .... = Install: Set
    .... .... 1.... .... = Key ACK: Set
    .... .... 1. .... .... = Key MIC: Set
    .... .... 1. .... .... = Secure: Set
    .... .... 0.. .... .... = Error: Not set
    .... .... 0.... .... = Request: Not set
    .... .... 1 .... .... .... = Encrypted Key Data: Set
Key Length: 16
Replay Counter: 2
WPA Key Nonce: c51a0100b8ac79e245c50067ffbb2331696bd2b632542980...
Key IV: 00000000000000000000000000000000
WPA Key RSC: 0000000000000000
WPA Key ID: 0000000000000000
WPA Key MIC: 0aea6c0c6f7f1275a76a547eeeca5372a
WPA Key Data Length: 80
WPA Key Data: 4f7f651a1e419f5707f8cd7bd14c02b80a5ba79b91586815...
```

图4-41 Message C的内容

由图4-41可知：

- Install为1，表示STA收到该帧后就可以安装PTK了。Secure为1表示AP已经派生了PTK。Encrpted Key Data为1，表示Key Data被加密了。
- Replay Counter为2，比前面的值增加1。
- Key Nonce值和Message A的一样。

- MIC对EAPOL-Key整个进行计算得来。
- Key IV为0。注意，如果Key Descriptor Version为2时，该字段必须为0。否则可以是其他随机数。
- Key Data由PTK加密后而来。其解密后的内容包括AP在Probe Response或Beacon帧包含RSN IE信息，另外还有可能包括GTK信息。如果有GTK的话，4-Way Handshake完毕后就无须开展Group Key Handshake流程。

STA收到Message C的处理如下。

- 1) 检查Replay Counter，计算MIC以及利用自己的PTK解密Key Data以获取RSN IE。
- 2) 如果Message C包含GTK信息，则取出GTK。注意，GTK以及和Key相关的信息存储在KDE（Key Data Element）的IE中。关于KDE和GTK的格式，请读者继续阅读下文。
- 3) 构造并发送最后一个EAPOL-Key帧，称为Message D。
- 4) 为Driver安装PTK。

Message D的内容如图4-42所示。由图可知，Replay Counter和Message C一样。MIC通过对EAPOL-Key整个进行计算得来。

AP收到Message D的处理如下。

- 1) 再次计算MIC，如果正确则为driver安装PTK。
- 2) 更新Replay Counter，如果以后需要更新Key，则使用一个不同的Replay Counter。

至此，4-Way Handshake成功完成，而AP和STA以后的单播数据将全部通过PTK进行加密。这就使得我们无法通过AirPcap解析Group Key Handshake数据包。关于Group Key Handshake的流程请看参考资料[22][23]。

```

└─ 802.1X Authentication
    version: 802.1X-2001 (1)
    Type: Key (3)
    Length: 95
    Key Descriptor Type: EAPOL RSN Key (2)
    └─ Key Information: 0x030a
        .... .... .010 = Key Descriptor Version: AES Cipher, HMAC-SHA1 MIC (2)
        .... .... 1... = Key Type: Pairwise Key
        .... .... .00 ... = Key Index: 0
        .... .... .0... .... = Install: Not set
        .... .... 0.... .... = Key ACK: Not set
        .... .... 1 .... .... = Key MIC: Set
        .... .... 1. .... .... = Secure: Set
        .... 0... .... .... = Error: Not set
        .... 0.... .... .... = Request: Not set
        .... 0 .... .... .... = Encrypted Key Data: Not set
    Key Length: 0
    Replay Counter: 2
    WPA KeyNonce: 00000000000000000000000000000000000000000000000000000000000000...
    Key IV: 00000000000000000000000000000000
    WPA Key RSC: 0000000000000000
    WPA Key ID: 0000000000000000
    WPA Key MIC: f62f8a6dc46854bbbc41b4beb61959a6
    WPA Key Data Length: 0

```

图4-42 Message D的内容

根据前文所述，AP在Message C中可以携带GTK信息。这样4-Way Handshake完毕后，STA无须开展Group Key Handshake。

**注意** 经过测试，有些AP每次在4-Way Handshake完毕后就发起Group Key Handshake，而有些AP在Message C中直接包含了GTK，这样就避免了Group Key Handshake。接下来的代码分析以后一种情况为目标。

另外，请注意图3-47，该图展示了PTK的组成。以CCMP为例，PTK包含三个部分，分别是KCK、KEK和TK。其中KCK用来处理MIC字段，KEK用来处理Key Data字段，而TK则用于握手协议完毕后的数据加密。

介绍完背景知识后，马上来研究wpa\_sm\_rx\_eapol函数，它是不是遵守了规范所描述的流程呢？

## (2) wpa\_sm\_rx\_eapol函数

在分析代码之前，先介绍两个重要的数据结构，如图4-43所示。

- struct ieee802\_1x\_hdr为EAPOL帧头信息。
- struct wpa\_eapol\_key为EAPOL-Key帧的数据信息。

<pre> <b>struct ieee802_1x_hdr</b> +version: u8 //版本,目前只支持EAPOL_VERSION(定义如下) +type: u8 //EAPOL包的类型,取值为IEEE802_1X_TYPE_EAPOL_XXX +length: be16 //be是Big Endian的缩写  //定义于eapol_common.h文件, 其中version和type取值如下 #define EAPOL_VERSION 2 //定义EAPOL包的类型, 供type变量使用 enum {     IEEE802_1X_TYPE_EAP_PACKET = 0,     IEEE802_1X_TYPE_EAPOL_START = 1,     IEEE802_1X_TYPE_EAPOL_LOGOFF = 2,     IEEE802_1X_TYPE_EAPOL_KEY = 3,     IEEE802_1X_TYPE_EAPOL_ENCAPSULATED_ASF_ALERT = 4 };  enum { //定义具体KEY的类型     EAPOL_KEY_TYPE_RC4 = 1, EAPOL_KEY_TYPE_RSN = 2,     EAPOL_KEY_TYPE_WPA = 254 }; </pre>	<pre> <b>struct wpa_eapol_key</b> +type: u8 //取值为EAPOL_KEY_TYPE_XXX +key_info: u8数组//元素个数为2, BigEndian排列 +key_length: u8数组//元素个数为2, BigEndian排列 +replay_counter: u8数组//元素个数为WPA_REPLAY_COUNTER_LEN(8) +key_nonce: u8数组//元素个数为WPA_NONCE_LEN(32) +key_iv: u8数组//元素个数为16 +key_rsc: u8数组//元素个数为WPA_KEY_RSC_LEN(8) +key_id: u8数组//元素个数为8 +key_mic: u8数组//元素个数为16 +key_data_length: u8数组//元素个数为2, BigEndian排列 </pre>
--	--

图4-43 EAPOL-Key帧对应的数据结构

wpa\_sm\_rx\_eapol代码如下所示。

```

[-->wpa.c: : wpa_sm_rx_eapol]

int wpa_sm_rx_eapol(struct wpa_sm *sm, const u8 *src_addr,
                     const u8 *buf, size_t len)
{
    size_t plen, data_len, extra_len;
    struct ieee802_1x_hdr *hdr; struct wpa_eapol_key *key;
    u16 key_info, ver; u8 *tmp; int ret = -1;
    struct wpa_peerkey *peerkey = NULL;
    .....// 参数检查
    tmp = os_malloc(len); os_memcpy(tmp, buf, len);

    hdr = (struct ieee802_1x_hdr *) tmp;
    key = (struct wpa_eapol_key *) (hdr + 1);
    plen = be_to_host16(hdr->length);
    data_len = plen + sizeof(*hdr);
    // 检查EAPOL-Key帧的类型
    if (hdr->version < EAPOL_VERSION) { }
    if (hdr->type != IEEE802_1X_TYPE_EAPOL_KEY) // 本函数只处理
EAPOL-Key帧
    { ret = 0; goto out; }

    if (key->type != EAPOL_KEY_TYPE_WPA && key->type !=
EAPOL_KEY_TYPE_RSN) {
        ret = 0; goto out; // 只处理key类型为WPA和RSN的情况
    }
    /*

```

通知lowerLayerSuccess。该函数最后一个参数表示此次调用是否来自EAPOL模块内部，

由于wpa\_sm\_rx\_eapol并非EAPOL模块内部，所以该参数为0。

```

*/
eapol_sm_notify_lower_layer_success(sm->eapol, 0);

// 取出Key Information字段，并保存到key_info变量中
key_info = WPA_GET_BE16(key->key_info);

ver = key_info & WPA_KEY_INFO_TYPE_MASK; // 获取key
descriptor version
if (ver != WPA_KEY_INFO_TYPE_HMAC_MD5_RC4 && ver !=
    WPA_KEY_INFO_TYPE_HMAC_SHA1_AES)
    goto out; // 根据规范的要求做相应处理
.....// CONFIG_IEEE80211R和CONFIG_IEEE80211W的处理

// 加密算法兼容性检查
if (sm->pairwise_cipher==WPA_CIPHER_CCMP&&ver!=
    WPA_KEY_INFO_TYPE_HMAC_SHA1_AES) {
    // 下面这个if判断用于检查组播数据加密设置是否正确
    if(sm->group_cipher!=WPA_CIPHER_CCMP&&!
    (key_info&WPA_KEY_INFO_KEY_TYPE)) {
        .....// 打印一句兼容性警告
        } else      goto out;
    }
.....// CONFIG_PEERKEY处理。PeerKey对应peerkey handshake流程,
它和IEEE802.11e DLS有关

// 检查Replay Counter，如果收到的Replay Counter比之前接收的值小，则丢弃该帧
if (!peerkey && sm->rx_replay_counter_set && os_memcmp(key-
>replay_counter,
                sm->rx_replay_counter, WPA_REPLY_COUNTER_LEN) <=
0) goto out;

// STA收到的EAPOL-Key帧必须设置ACK位或SMK位为1
if (!(key_info & (WPA_KEY_INFO_ACK |
WPA_KEY_INFO_SMK_MESSAGE))) ) goto out;

// 只有STA向AP发送的EAPOL-Key帧才能设置Request位
if (key_info & WPA_KEY_INFO_REQUEST) goto out;

/*
MIC位不为0，则需要解析MIC数据。对STA来说，只有Message C会携带MIC
信息。
wpa_supplicant_verify_eapol_key_mic比较简单，请读者自行阅读它。
其功能是：STA
根据KCK计算MIC，然后将其和接收到的MIC进行比较。如果MIC值检查正常，
```

则PTK正确。

```
    */
    if (((key_info & WPA_KEY_INFO_MIC) && !peerkey &&
        wpa_supplicant_verify_eapol_key_mic(sm, key, ver, tmp,
        data_len))  goto out;

    extra_len = data_len - sizeof(*hdr) - sizeof(*key);

    if (WPA_GET_BE16(key->key_data_length) > extra_len)  goto
out;// 参数检查

    extra_len = WPA_GET_BE16(key->key_data_length);
    // Encrpted Key Data位为1, 表示该帧包含加密数据, 需要利用KEK进行
解密
    // 该函数也比较简单, 请读者自行阅读
    if (sm->proto == WPA_PROTO_RSN && (key_info &
WPA_KEY_INFO_ENCR_KEY_DATA)) {
        if (wpa_supplicant_decrypt_key_data(sm, key, ver))  goto
out;
        extra_len = WPA_GET_BE16(key->key_data_length);
    }
    // Key Info Type值为1, 表示为Pairwise Key。值为0表示Group Key
    if (key_info & WPA_KEY_INFO_KEY_TYPE) {
        if (key_info & WPA_KEY_INFO_KEY_INDEX_MASK)      goto
out;
        if (peerkey) {    peerkey_rx_eapol_4way(sm, peerkey, key,
key_info, ver);
            } else if (key_info & WPA_KEY_INFO_MIC) { // 对STA而言, 只
有Message C包含MIC信息
                wpa_supplicant_process_3_of_4(sm, key, ver); // 处理
Message C
            } else { // 处理Message A
                wpa_supplicant_process_1_of_4(sm, src_addr, key,
ver);
            }
        } else if (key_info & WPA_KEY_INFO_SMK_MESSAGE) { // SMK的情况
            peerkey_rx_eapol_smk(sm, src_addr, key, extra_len,
key_info, ver);
        } else { // Group Key Handshake处理
            if (key_info & WPA_KEY_INFO_MIC) { // 组播密钥交换。本章不讨
论此函数
                wpa_supplicant_process_1_of_2(sm, src_addr,
key, extra_len, ver);
            } else .... // 打印一句警告
        }
    }
```

```

    ret = 1;
out:
    os_free(tmp);
    return ret;
}

```

wpa\_sm\_rx\_eapol的流程比较简单，就是先处理EAPOL-Key的基本信息（如计算MIC、解密Key Data），然后根据情况处理Message A、Message C或者Group Key Handshake的Message 1。

由于本例不涉及Group Key Handshake流程，所以下面将介绍Message A及Message C的处理过程。

### (3) wpa\_supplicant\_process\_1\_of\_4函数

wpa\_supplicant\_process\_1\_of\_4用于处理Message A，其代码如下所示。

```

[-->wpa.c: : wpa_supplicant_process_1_of_4]

static void wpa_supplicant_process_1_of_4(struct wpa_sm *sm,
                                         const unsigned char *src_addr,
                                         const struct wpa_eapol_key *key, u16 ver)
{
    struct wpa_eapol_ie_parse ie;    struct wpa_ptk *ptk;
    u8 buf[8];          int res;

    if (wpa_sm_get_network_ctx(sm) == NULL) {.....// 错误处理}

    wpa_sm_set_state(sm, WPA_4WAY_HANDSHAKE); // 设置状态为
WPA_4WAY_HANDSHAKE

    os_memset(&ie, 0, sizeof(ie));

#ifndef CONFIG_NO_WPA2
    if (sm->proto == WPA_PROTO_RSN) {
        const u8 *_buf = (const u8 *) (key + 1); // buf中是已经解密
的Key Data数据
        size_t len = WPA_GET_BE16(key->key_data_length);
        // 解析Message A中包含的RSN信息。对本例而言，Message A中没有
RSN信息
        if (wpa_supplicant_parse_ies(_buf, len, &ie) < 0)
goto failed;
    }

```

```

#endif /* CONFIG_NO_WPA2 */
    // 如果根据RSN IE中的pmkid判断是否有PMKSA缓存项。本例没有RSN IE，所以不存在ie.pmkid
    res = wpa_supplicant_get_pmk(sm, src_addr, ie.pmkid);
    if (res == -2) return;
    if (res)
        goto failed;
    // 结合前面对背景知识的描述，STA需要创建自己的Nonce
    if (sm->renew_snonce) {
        if (random_get_bytes(sm->snonce, WPA_NONCE_LEN)) goto
failed;
        sm->renew_snonce = 0;
    }
/*
tptk变量存储的是临时PTK，因为现在还不确定PTK是否正确。注意，当收到
Message C时，

wpa_sm_rx_eapol的wpa_supplicant_verify_eapol_key_mic函数将把sm-
>tptk的内容复制到
sm->ptk变量中作为正式的ptk存储。
*/
    ptk = &sm->tptk;
    wpa_derive_ptk(sm, src_addr, key, ptk); // 派生PTK并将结果保存到
tptk变量中
    os_memcpy(buf, ptk->u.auth.tx_mic_key, 8);
    os_memcpy(ptk->u.auth.tx_mic_key, ptk->u.auth.rx_mic_key,
8);
    os_memcpy(ptk->u.auth.rx_mic_key, buf, 8);
    sm->tptk_set = 1; // tptk_set为1表示临时PTK存在
    // 构造并发送Message B。请读者结合参考资料[23]来研究此函数
    if (wpa_supplicant_send_2_of_4(sm, sm->bssid, key, ver, sm-
>snonce,
                                sm->assoc_wpa_ie, sm->assoc_wpa_ie_len, ptk))
        goto failed;

    // 包括AP的Nonce信息
    os_memcpy(sm->anonce, key->key_nonce, WPA_NONCE_LEN);
    return;
failed:
    wpa_sm_deauthenticate(sm, WLAN_REASON_UNSPECIFIED);
}

```

STA将Message B发送出去后，AP将接收到它并根据AP的处理逻辑进行处理。假设一切顺利，AP将构造并发送Message C给STA。

STA依然在wpa\_sm\_rx\_eapol中处理Message C，其过程如下。

1) 由于Message C包含MIC以及Key Data数据，故它们将在wpa\_sm\_rx\_eapol中的wpa\_supplicant\_verify\_eapol\_key\_mic及wpa\_supplicant\_decrypt\_key\_data被处理。这两个函数的代码请感兴趣的读者自行阅读。

2) 接下来调用wpa\_supplicant\_process\_3\_of\_4。这是我们分析的重点。

#### (4) wpa\_supplicant\_process\_3\_of\_4函数

wpa\_supplicant\_process\_3\_of\_4的代码如下所示。

```
[-->wpa.c: : wpa_supplicant_process_3_of_4]

static void wpa_supplicant_process_3_of_4(struct wpa_sm *sm,
                                         const struct wpa_eapol_key *key, u16 ver)
{
    u16 key_info, keylen, len;
    const u8 *pos;      struct wpa_eapol_ie_parse ie;

    wpa_sm_set_state(sm, WPA_4WAY_HANDSHAKE); // 还是处于
WPA_4WAY_HANDSHAKE状态

    key_info = WPA_GET_BE16(key->key_info);
    pos = (const u8 *) (key + 1);
    len = WPA_GET_BE16(key->key_data_length);
    /*

解析Message C中包含的IE信息。注意，key data中的数据已经由
wpa_supplicant_decrypt_key_data
解密过了。
*/
    if (wpa_supplicant_parse_ies(pos, len, &ie) < 0) goto
failed;
    if (ie.gtk && !(key_info & WPA_KEY_INFO_ENCR_KEY_DATA))
goto failed;

.....// CONFIG_IEEE80211W
// 校验RSN信息。该校验似乎也是为了防止安全设置项中途发生改变
    if (wpa_supplicant_validate_ie(sm, sm->bssid, &ie) < 0)
goto failed;
    // 比较Message A和Message C中的Nonce值
    if (os_memcmp(sm->anonce, key->key_nonce, WPA_NONCE_LEN) != 0)
        goto failed;
```

```

keylen = WPA_GET_BE16(key->key_length);
.....// 参数检查
// 构造并发送Message D, 请读者自行阅读该函数
if (wpa_supplicant_send_4_of_4(sm, sm->bssid, key, ver,
key_info,
NULL, 0, &sm->ptk)) goto failed;

sm->renew_snonce = 1;
/*
调用driver_nl80211的wpa_driver_nl80211_set_key函数将key发给驱动。以后，所有
无线网络数据都将被加密。另外，如果wpa_supplicant.conf文件配置
wpa_ptk_rekey
（用来控制PTK的生命周期，时间为秒）时，该函数内部还将注册一个超时函数
wpa_sm_rekey_ptk。一旦
wpa_ptk_rekey到期，STA将重新和AP开展4-Way Handshake以重新派生新的PTK。
请读者自行阅读wpa_supplicant_install_ptk函数。
*/
if (key_info & WPA_KEY_INFO_INSTALL) // 安装Key
    if (wpa_supplicant_install_ptk(sm, key)) goto failed;

// Message C必须设置Secure位
if (key_info & WPA_KEY_INFO_SECURE) {
    // 下面这个函数和管理帧加密有关系
    wpa_sm_mlme_setprotection(sm, sm->bssid,
MLME_SETPROTECTION_PROTECT_TYPE_RX,
        MLME_SETPROTECTION_KEY_TYPE_PAIRWISE);
    eapol_sm_notify_portValid(sm->eapol, TRUE); // 设置EAPOL
SM portValid为TRUE
}
wpa_sm_set_state(sm, WPA_GROUP_HANDSHAKE); // 设置状态为
WPA_GROUP_HANDSHAKE
// 对本例而言，Message C中包含了GTK信息
if (ie.gtk && wpa_supplicant_pairwise_gtk(sm, key,
        ie.gtk, ie.gtk_len, key_info) < 0)
goto failed;
// 和IEEE80211w有关。本书不讨论它
if (ieee80211w_set_keys(sm, &ie) < 0) goto failed;
/*

```

下面这个函数将调用driver wrapper的set\_rekey\_info函数。该函数和GTK的更新有关。

它对应以下应用场景。

当某STA因为省电或别的什么原因进入休眠状态时，如果AP在STA休眠过程中更新

了GTK。当该STA

醒来时，其组播消息密钥肯定失效了。这种情况该如何处理呢？通过set\_rekey\_info函数，我们可

将该工作交给wlan芯片来完成。即由wlan芯片来负责处理EAPOL-Key帧交换以更新GTK。注意，

如果wlan driver不支持该功能，它将唤醒STA，并将该工作交给WPAS来完成。

这部分功能属于WoWLAN（无线局域网唤醒）机制的一部分，目前只有很少的系统支持它。读者可阅读

参考资料[24][25]以加深对WoWLAN的认识。

```
 */
wpa_sm_set_rekey_offload(sm);
return;
failed:
    wpa_sm_deauthenticate(sm, WLAN_REASON_UNSPECIFIED);
}
```

wpa\_supplicant\_process\_3\_of\_4代码逻辑和背景知识介绍中关于Message C的处理逻辑大体一致。对于包含GTK信息的Message C来说，wpa\_supplicant\_pairwise\_gtk将被调用以处理GTK对应的KDE信息。马上来看此函数。

### (5) wpa\_supplicant\_pairwise\_gtk函数

介绍wpa\_supplicant\_pairwise\_gtk之前，先来看看KDE和GTK的格式，如图4-44所示。

Type (0xdd)	Length	OUI	Data Type	Data
Octets:	1	1	3	1 (Length - 4)

KeyID (0,1,2, or 3)	Tx	Reserved (0)	Reserved (0)	GTK
bits 0-1	bit 2	bit 3-7	1 octet	(Length - 6) octets

图4-44 KDE和GTK格式

图4-44中，上图为KDE的格式，其中Type取值为0xdd。Data包含具体的信息。当Data中的信息为GTK时，OUT取值为00-0F-AC，Data Type取值为1。下图为GTK的格式。KeyID字段可取值有0~3共4个，它和动态Key有关。Tx字段为1，表示发送和接收组播数据都需要使用该GTK。值为0表示仅接收组播数据需要使用该GTK。

```

[-->wpa.c: : wpa_supplicant_pairwise_gtk]

static int wpa_supplicant_pairwise_gtk(struct wpa_sm *sm,
                                       const struct wpa_eapol_key *key,
                                       const u8 *gtk, size_t gtk_len, int
key_info)
{
#ifndef CONFIG_NO_WPA2
    struct wpa_gtk_data gd;
    os_memset(&gd, 0, sizeof(gd));
    if (gtk_len < 2 || gtk_len - 2 > sizeof(gd.gtk))      return
-1; // 参数检查

    gd.keyidx = gtk[0] & 0x3;
    // 下面这个函数对图4-44中的Tx位进行一些处理
    // 它对某些AP错误设置Tx位的情况采取了“绕过去”的策略
    gd.tx = wpa_supplicant_gtk_tx_bit_workaround(sm, !(gtk[0] &
BIT(2)));
    gtk += 2; gtk_len -= 2;

    os_memcpy(gd.gtk, gtk, gtk_len);
    gd.gtk_len = gtk_len;
    // 检查组播数据加密设置是否合适，然后调用
    wpa_supplicant_install_gtk函数安装GTK
    if (wpa_supplicant_check_group_cipher(sm, sm->group_cipher,
gtk_len, gtk_len,
                                         &gd.key_rsc_len, &gd.alg) ||
        wpa_supplicant_install_gtk(sm, &gd, key->key_rsc))
return -1;
    // 最后一个关键函数
    wpa_supplicant_key_neg_complete(sm, sm->bssid, key_info &
WPA_KEY_INFO_SECURE);
    return 0;
#else
    return -1;
#endif
}

```

上述代码中，请读者自行阅读除wpa\_supplicant\_key\_neg\_complete外的其他几个函数。下面来看最后一个关键函数。

[-->wpa.c: : wpa\_supplicant\_key\_neg\_complete]

```

static void wpa_supplicant_key_neg_complete(struct wpa_sm
*sm, const u8 *addr, int secure)
{

```

```

wpa_sm_cancel_auth_timeout(sm); // 取消超时任务
// 该函数内部将调用wpa_supplicant.c中的
wpa_supplicant_set_state。请读者自行阅读
// 不光是一个简单的设置状态，它还需要调用driver的set_supp_port函数
wpa_sm_set_state(sm, WPA_COMPLETED); // 设置WPAS的状态为
WPA_COMPLETED

if (secure) { // secure为1
    wpa_sm_mlme_setprotection( sm, addr,
MLME_SETPROTECTION_PROTECT_TYPE_RX_TX,
        MLME_SETPROTECTION_KEY_TYPE_PAIRWISE);
    eapol_sm_notify_portValid(sm->eapol, TRUE);
    if (wpa_key_mgmt_wpa_psk(sm->key_mgmt)) // 本例采用了PSK认证，故可以通知eapSuccess
        eapol_sm_notify_eap_success(sm->eapol, TRUE);
    eloop_register_timeout(1, 0, wpa_sm_start_preatht, sm,
NULL);
}
if (sm->cur_pmksa && sm->cur_pmksa->opportunistic)
    sm->cur_pmksa->opportunistic = 0;
.....// CONFIG_IEEE80211R
}

```

至此，4-Way Handshake的处理就全部完成。而802.1X中Port的状态以及WPAS的状态则由此过程中相关的函数触发以发生转换。下一节将简单介绍这部分的内容。

## (6) WPAS状态机变化

上述4-Way Handshake处理流程节中，EAPOL模块、EAP模块以及WPAS的状态都会发生对应的转换。在上述wpa\_supplicant\_key\_neg\_complete中，wpa\_sm\_set\_state将设置WPAS的状态为WPA\_COMPLETED。马上来看wpa\_sm\_set\_state的代码，如下所示。

```
[-->wpa_i.h: : wpa_sm_set_state]

static inline void wpa_sm_set_state(struct wpa_sm *sm, enum
wpa_states state)
{
/*
调用回调函数。由4.3.4节的wpa_supplicant_init_wpa函数中设置，其真实
函数为
    _wpa_supplicant_set_state，而该函数又会调用
wpa_supplicant_set_state。
```

```

    */
    sm->ctx->set_state(sm->ctx->ctx, state);
}

```

wpa\_sm\_set\_state将调用前面设置的回调函数进行处理。

最终的处理函数wpa\_supplicant\_set\_state如下所示。

[-->wpa\_supplicant.c: : wpa\_supplicant\_set\_state]

```

void wpa_supplicant_set_state(struct wpa_supplicant *wpa_s,
                               enum wpa_states state)
{
    enum wpa_states old_state = wpa_s->wpa_state;
    .....
    if (state == WPA_COMPLETED && wpa_s->new_connection) {
#ifndef CONFIG_CTRL_IFACE || !defined(CONFIG_NO_STDOUT_DEBUG)
        struct wpa_ssid *ssid = wpa_s->current_ssid;
#endif
        wpa_s->new_connection = 0;
        wpa_s->reassociated_connection = 1;
        /*
        设置driver的IfOperStatus为IF_OPER_UP, driver nl80211中对应的
        函数是
    
```

```

        wpa_driver_nl80211_set_operstate。
    */
    wpa_drv_set_operstate(wpa_s, 1);
#ifndef IEEE8021X_EAPOL
/*
        以driver nl80211来说, 下面这个函数将调用
wpa_driver_nl80211_set_supp_port
        以设置驱动中STA的标志为NL80211_STA_FLAG_AUTHENTICATED。
*/
    wpa_drv_set_supp_port(wpa_s, 1);
#endif /* IEEE8021X_EAPOL */
    wpa_s->after_wps = 0;
} else if (state == WPA_DISCONNECTED || state ==
WPA_ASSOCIATING ||
            state == WPA_ASSOCIATED) {
    wpa_s->new_connection = 1;// ASSOCIATING或ASSOCIATED状态下, new_connection被置为1
    wpa_drv_set_operstate(wpa_s, 0);// 设置driver的
    IfOperStatus为IF_OPER_DORMANT
#ifndef IEEE8021X_EAPOL
    wpa_drv_set_supp_port(wpa_s, 0);// 取消STA的

```

```

NL80211_STA_FLAG_AUTHENTICATED标志
#endif /* IEEE8021X_EAPOL */
}
wpa_s->wpa_state = state;
.....
}

```

当WPAS的状态变成WPA\_COMPLETED后，还需要设置driver的IfOperStatus以及NL80211\_STA\_FLAG\_AUTHENTICATED标志位。如此这般，WPAS才算和Driver上下同心。

## (7) EAPOL/EAP状态机变化

在4-Way Handshake过程中，EAPOL/EAP状态机也会发生变化。在4.5.3节eapol\_sm\_notify\_portEnabled函数分析的最后，已知EAPOL/EAP状态机的状态分别如下。

- SUPP\_PAE进入CONNECTING状态；
- KEY\_RX不变（NO\_KEY\_RECEIVE）；
- SUPP\_BE进入IDLE状态；
- EAP\_SM不变（DISABLED状态）。

在4-Way Handshake过程中，一共有两个和EAPOL相关的函数被调用：eapol\_sm\_notify\_portValid函数portValid被设为TRUE。eapol\_sm\_notify\_eap\_success函数代码如下所示。

```

[-->eapol_supp_sm.c: : eapol_sm_notify_eap_success]

void eapol_sm_notify_eap_success(struct eapol_sm *sm, Boolean
success)
{
    if (sm == NULL) return;
    sm->eapSuccess = success;
    sm->altAccept = success;
    if (success) eap_notify_success(sm->eap);
    eapol_sm_step(sm); // 状态机联动
}
// 直接来看eap_notify_success函数
void eap_notify_success(struct eap_sm *sm)

```

```

{
    if (sm) {
        sm->decision = DECISION_COND_SUCC;
        sm->EAP_state = EAP_SUCCESS; // EAP_SM状态直接被设置为
EAP_SUCCESS
    }
}

```

根据4.4.2节状态机联动中对eapol\_sm\_step的分析，四个状态机更新的顺序分别是SUPP\_PAE、KER\_RX、SUPP\_BE和EAP\_SM。分别结合四个状态机的切换图，我们可知第一轮循环中：

- eapSuccess将先触发SUPP\_PAE进入AUTHENTICATING状态，该状态对应的EA将设置suppStart为TRUE。
- eapSuccess和suppStart将触发SUPP\_BE进入SUCCESS状态，该状态对应的EA将设置keyRun和suppSuccess为TRUE。
- EAP\_SM状态在eap\_notify\_success中直接被置为EAP\_SUCCESS。但在状态切换中，由于force\_disabled变量为TRUE，导致EAP SUPP SM将直接转回DISABLED状态（和4.4.2节介绍的状态机联动的情况类似）。
- KER\_RX状态不变。

由于上述状态机均有状态变化，下面进入第二轮循环：

- suppSuccess和portValid为TRUE将触发SUPP\_PAE进入AUTENTICATED状态（由于条件变量不会再改变，故SUPP\_PAE将不再发生状态变化）。该状态的EA见下文代码。
- 由于UCT的存在，SUPP\_BE将跳转到IDLE状态。
- KEY\_RX和EAP\_SM状态保持不变。

以下是SUPP\_PAE的EA代码。

```

[-->eapol_supp_sm.c: : SM_STATE (SUPP_PAE, AUTHENTICATED) ]

SM_STATE (SUPP_PAE, AUTHENTICATED)
{
    SM_ENTRY (SUPP_PAE, AUTHENTICATED);

```

```

sm->suppPortStatus = Authorized;
// 也会调用wpa_drv_set_supp_port函数
eapol_sm_set_portAuthorized(sm);
/*
设置cb_status为EAPOL_CB_SUCCESS，将触发eapol_sm_step在退出前调用
wpa_supplicant_eapol_cb，此函数对于WPA/RSN企业认证法有重要作用。请
读者自行阅读。
*/
sm->cb_status = EAPOL_CB_SUCCESS;
}

```

请读者结合SUPP\_PA和SUPP\_BE的状态切换图来理解上述的状态变化过程。

至此，STA就通过了AP的身份验证。下一步的工作就是dhcpcd从AP那获取一个IP，然后手机就可以上网了。

### (8) EAPOL-Key交换流程总结

下面总结4-Way Handshake的流程，如图4-45和图4-46所示。

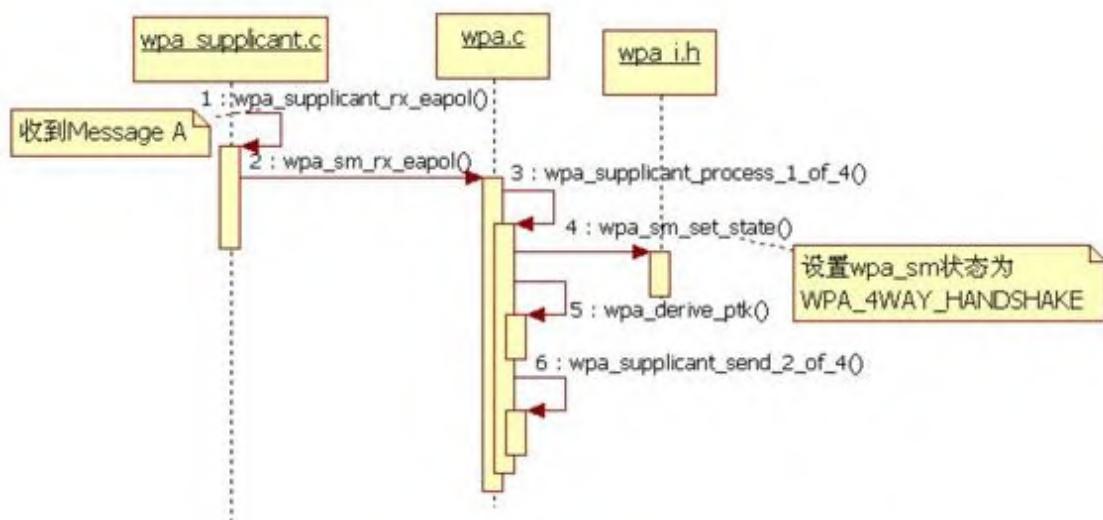


图4-45 Message A处理流程

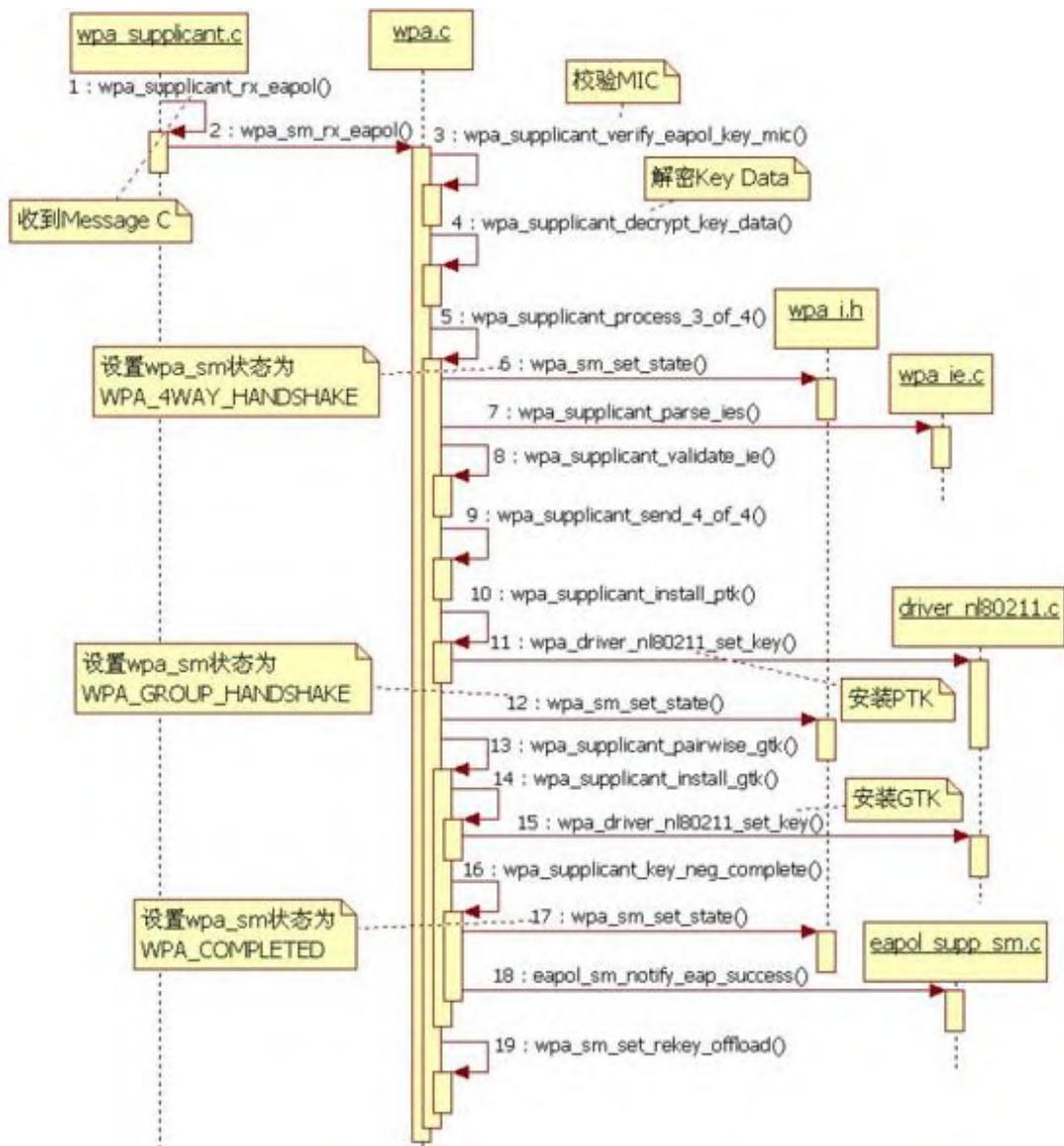


图4-46 Message C处理流程

图中分别为4-Way Handshake中Message A与Message C的处理流程。其中，Message C还携带了GTK信息，这样就无须Group Key Handshake了。

## 4.6 本章总结和参考资料说明

### 4.6.1 本章总结

本章对wpa\_supplicant进行了一番剖析，涉及如下几个重点内容。

- WPAS的启动：在这条分析路线中，读者能感受到WPAS中多种多样的数据结构以及之间较为复杂的关系。同时，对极具背景含义的成员变量也进行了深入介绍。
- EAPOL/EAP模块：先向读者介绍了理论知识，然后结合代码分析了WPAS如何将理论知识转化成代码。从笔者角度来看，状态机的代码比看状态切换图要复杂得多。读者不妨自己亲身体验一下。
- 扫描、关联及4-Way Handshake分析：通过一个实例讲解了这一流程涉及的重要函数和相关知识点。该流程实际上包含的代码远比书中给多，原因是WPAS支持的许多功能（如WPS、P2P、802.11R、802.11W等）都在这一流程中有涉及。根据笔者的学习经验，只有先清楚一个最简单的流程，后面才能循序渐进地把其他功能的分析加进来。

**提示** 本章编撰时，市面上搭载Android 4.2的手机较少，所以此处的分析目标是Android 4.1中的wpa\_supplicant。不过笔者比较了它和Android 4.2版本中的wpa\_supplicant。虽然4.2版本中的WPAS变化较大，主要体现在一些新增功能和bug修改上。读者如果搞清楚了本章内容，再研究4.2版本中的WPAS会很轻松。

## 4.6.2 参考资料说明

### 1. 概述

[1]

[http://en.wikipedia.org/wiki/Extensible\\_Authentication\\_Protocol](http://en.wikipedia.org/wiki/Extensible_Authentication_Protocol)

说明：维基百科关于EAP各种方法的一个简单介绍。

[2] [http://hostap.epitest.fi/wpa\\_supplicant-devel/](http://hostap.epitest.fi/wpa_supplicant-devel/)

说明：wpa\_supplicant官方开发文档。读者可以简单浏览一下。

### 2. wpa\_ssid结构体

[3] 《802.11-2012》附录M.4“Suggested pass-phrase-to-PSK mapping”

说明：该节介绍了passphrase转换成PSK的方法，甚至还有伪代码实现。感兴趣的读者不妨结合WPAS中的代码来研究。

[4] 《802.11-2012》第8.4.2.27.2节“Cipher suites”

[5] 《802.11-2012》第8.4.2.27.3节“AKM suites”

说明：上述两小节分别介绍了Cipher和AKM suites的情况。注意，其中定义的取值定义是指在RSN IE中的取值，和代码中定义的宏不是一回事。

[6] 《802.11-2012》第12章“Fast BSS Transition”

说明：官方文档。难度较大，建议读者阅读《Secure Roaming in 802.11 Networks》一书后再去看它。此书是笔者目前阅读到的关于Wi-Fi Roaming相关知识介绍最完整的一本。

[7] 《Real 802.11 Security: Wi-Fi Protected Access and 802.11i》第6章“How IEEE 802.11 WEP Works and Why It Doesn’t”

说明：关于WEP的介绍。对安全感兴趣的读者请仔细阅读此书。

[8]

[http://www.codealias.info/technotes/opportunistic\\_pmk\\_pre-caching](http://www.codealias.info/technotes/opportunistic_pmk_pre-caching)

说明：关于Opportunistic PMK Caching的简单介绍。

[9] 《Secure Roaming in 802.11 Networks》第8章“Opportunistic Key Caching”一节

说明：相比参考资料[8]而言，这一节对OKC有更为详尽的介绍。

### 3. wpa\_supplicant结构体

[10] 《802.11无线网络权威指南（第2版）》第7章“802.11: RSN、TKIP与CCMP”，P171–P172

[11] 《802.11-2012》第11.4节“TKIP countermeasures procedures”

说明：上述两个参考资料介绍了TKIP countermeasures的处理方式。请先阅读参考资料[10]。

[12]

[http://www.cisco.com/en/US/docs/solutions/Enterprise/Mobility/vowlan/41dg/vowlan\\_ch5.html](http://www.cisco.com/en/US/docs/solutions/Enterprise/Mobility/vowlan/41dg/vowlan_ch5.html)

[13] 《Secure Roaming in 802.11 Networks》第5.2.5节“Background Scanning”

说明：关于Background Scan技术的介绍。

[14] <http://network.chinabyte.com/359/12453859.shtml>

[15] <http://www.docin.com/p-365323002.html>

说明： 和GAS以及802.11u相关的一些介绍。

#### 4. wpa\_supplicant\_init\_iface分析之三

[16] <http://www.mjmwired.net/kernel/Documentation/rfkill.txt>

[17] <http://lwn.net/Articles/335382/>

说明：这两篇资料介绍了rfkill相关的信息。感兴趣的读者不妨仔细阅读。

[18]

<http://wenku.baidu.com/view/c74758d280eb6294dd886c53.html>

说明： RFC2863 3.1.13节”IfAdminStatus and IfOperStatus”描述了IfOperStatus的取值情况及相关说明。

[19]

<http://wireless.kernel.org/en/developers/Documentation/nl80211/kerneldoc>

说明：Linux Wireless Kernel官方网站中关于nl80211内核部分的一些解释。

#### 5. EAP模块分析

[20] <http://tools.ietf.org/pdf/rfc4137.pdf>

说明：RFC4137文档的PDF版。相比TXT版而言，它用图来描述状态机的状态切换。

#### 6. EAPOL模块分析

[21] 802.1X 2004版

说明：WPAS中的802.1X实现是基于802.1X 2004版。相比2010版而言，笔者觉得2004版的内容更具备条理性，尤其是其关于EAPOL各状态机的描述非常清晰。

## 7. EAPOL-Key交换流程分析

[22] 《Real 802.11 Security: Wi-Fi Protected Access and 802.11i》第10章“WPA and RSN Key Hierarchy”

[23] 《802.11-2012》第11.6节“Keys and key distribution”

说明：这两篇参考资料对Pairwise Key、Group Key以及4-Way Handshake、Group Key Handshake都有详细的介绍。

[24] [wireless.kernel.org/en/users/Documentation/WoWLAN](http://wireless.kernel.org/en/users/Documentation/WoWLAN)

[25] [msdn.microsoft.com/en-us/library/windows/hardware/ff571052\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/hardware/ff571052(v=vs.85).aspx)

说明：这两篇文章对WoWLAN有详细介绍。读者可简单阅读。

# 第5章 深入理解WifiService

本章所涉及的源代码文件名及位置

- IWifiManager.aidl

frameworks/base/wifi/java/android/net/wifi/IWifiManager.aidl

- SystemServer. java

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

- WifiService. java

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

- StateMachine. java

frameworks/base/core/java/com/android/internal/util/StateMachine.java

- AsyncChannel. java

frameworks/base/core/java/com/android/internal/util/AsyncChannel.java

- WifiManager. java

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

- WifiStateMachine. java

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

- WifiNative. java

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

- android\_net\_wifi\_Wifi.cpp

frameworks/base/core/jni/android\_net\_wifi\_Wifi.cpp

- wifi.c hardware/libhardware\_legacy/wifi/wifi.c
- WifiMonitor.java  
frameworks/base/wifi/java/android/net/wifi/WifiMonitor.java
- SuplicantStateTracker.java  
frameworks/base/wifi/java/android/net/wifi/SuplicantStateTracker.java
- WifiWatchdogStateMachine.java  
frameworks/base/wifi/java/android/net/wifi/WifiWatchdogStateMachine.java
- NetworkInfo.java  
frameworks/base/core/java/android/net/NetworkInfo.java
- CaptivePortalTracker.java  
frameworks/base/core/java/android/net/CaptivePortalTracker.java

# 读累了记得休息一会儿哦~

公众号：古德猫宁李

- 电子书搜索下载
- 书单分享
- 书友学习交流

网站：[沉金书屋 https://www.chenjin5.com](https://www.chenjin5.com)

- 电子书搜索下载
- 电子书打包资源分享
- 学习资源分享

## 5.1 概述

WifiService是Android Java Framework中负责Wi-Fi功能的核心服务。它主要借助wpa\_supplicant（以后简称WPAS）来管理和控制Android平台中的Wi-Fi功能。虽然WPAS才是Android平台中整个Wi-Fi模块的真正核心，但WifiService作为Java Framework中Wi-Fi功能的总入口，其重要性也不言而喻。

同WPAS的分析类似，本节也将通过两条路线来研究WifiService。

线路一：WifiService的创建及初始化。

线路二：在Settings中打开Wi-Fi功能、扫描无线网络及加入目标无线网络。

最后，我们还将介绍WifiWatchdogStateMachine和Captive Portal Check这两个颇有意思的知识点。

图5-1所示为WifiService及WifiManager的类图结构。

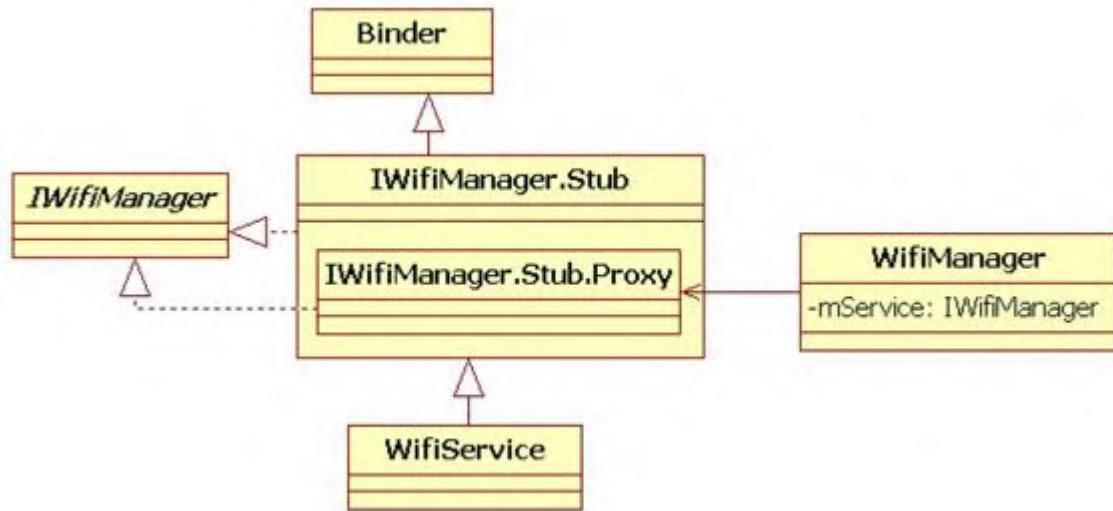


图5-1 WifiService和WifiManager类图结构

图5-1中：

- IWifiManager、 IWifiManager. Stub和IWifiManager. Stub. Proxy类均由IWifiManager. aidl文件在编译时通过aidl工具转换而来。
- WifiService派生自IWifiManager. Stub类，它是Binder服务端。
- WifiManager是WifiService的客户端。它通过成员变量mService和WifiService进行Binder交互。

注意 建议对Binder不熟悉的读者阅读《深入理解Android：卷 I》第6章和《深入理解Android：卷 II》第2章。

下面来分析路线一，即WifiService的创建及初始化分析。

## 5.2 WifiService的创建及初始化

WifiService在SystemServer进程中被创建，代码如下所示。

[-->SystemServer.java: : ServerThread: run]

```
public void run()
.....
try {
    wifi = new WifiService(context); // 创建一个WifiService对
象
    ServiceManager.addService(Context.WIFI_SERVICE, wifi);
} .....
try {
    connectivity = new ConnectivityService(
        context, networkManagement,
networkStats, networkPolicy);

ServiceManager.addService(Context.CONNECTIVITY_SERVICE,
connectivity);
.....
/*
    检查是否需要启动Wi-Fi。如果关机前Wi-Fi是打开的，则重启后Wi-Fi
功能将在此函数中打开
    该函数比较简单，其中有一处涉及wifiWatchdogStateMachine的代
码。留待5.4节介绍。
*/
    wifi.checkAndStartWifi();
    wifiP2p.connectivityServiceReady();
} .....
.....
```

由上述代码可知，SystemServer<sub>①</sub>中和WifiService相关的只有两处函数调用。其中，WifiService的创建是第一条分析路线的起点。

正式介绍WifiService之前，本节先介绍两个基础知识，分别是HSM (Hierarchical State Machine, 结构化状态机) 和 AsyncChannel。

① 对SystemServer感兴趣的读者不妨阅读《深入理解Android：卷II》第3章。

### 5.2.1 HSM和AsyncChannel介绍

HSM（对应的类是StateMachine）和AsyncChannel是Android Java Framework中两个重要的类。不过，它们目前还仅由Framework内部使用，SDK中并没有包含它们。这两个类的作用如下。

- HSM在传统状态机对所有状态都一视同仁的基础上做了一些改变，使得状态和状态之间有了层级关系。HSM中的状态层级关系与Java中父子类的派生和继承关系类似，即在父状态中实现generic的功能，而在子状态中实现一些特定的处理。不过，和Java中类派生不同的是，HSM中父子状态对应的是毫无派生关系的两个类，所以使用时需要创建两个对象。而Java中子类则从其父类派生，实际使用时创建一个子类对象即可，该子类对象就能完成父类的工作。
- AsyncChannel用于两个Handler之间的通信。具体的通信方式为源Handler通过sendMessage向目标Handler发送消息，而目标Handler通过replyToMessage回复源Handler处理结果。注意，这两个Handler可位于同一个进程，也可分属两个不同的进程。

本节先来介绍HSM。

**注意** 由于HSM和AsyncChannel并非本书的主题，故本章仅介绍它们的用法。对实现原理感兴趣的读者不妨在了解它们用法的基础上，自行研究相关代码。

#### 1. HSM使用

HSM对应的类叫StateMachine，下面通过一个例子来介绍其用法。

##### [HSM示例]

```
// 此例来源于StateMachine.java文件中的注释
// StateMachineTest是StateMachine的子类
class StateMachineTest extends StateMachine {
    StateMachineTest(String name) {
        super(name);
        // 为状态机添加一个状态。代码中一般用缩进的方式表达层级关系
        addState(mP0);
```

```

// 添加一个状态mS0，其父状态为mP0
addState(mS0, mP0);
addState(mP1, mP0);

addState(mS1, mP1); // 添加一个状态mS1，其父状态为mP1
addState(mS5, mS1)
addState(mS2, mP1);
addState(mS3, mS2);
addState(mS4, mS2);

// setInitialState函数用于设置状态机的初始状态，本例中该状态是mS5
setInitialState(mS5);
} // StateMachineTest构造函数结束

```

上述代码中StateMachineTest所涉及的状态及层级关系如图5-2所示。

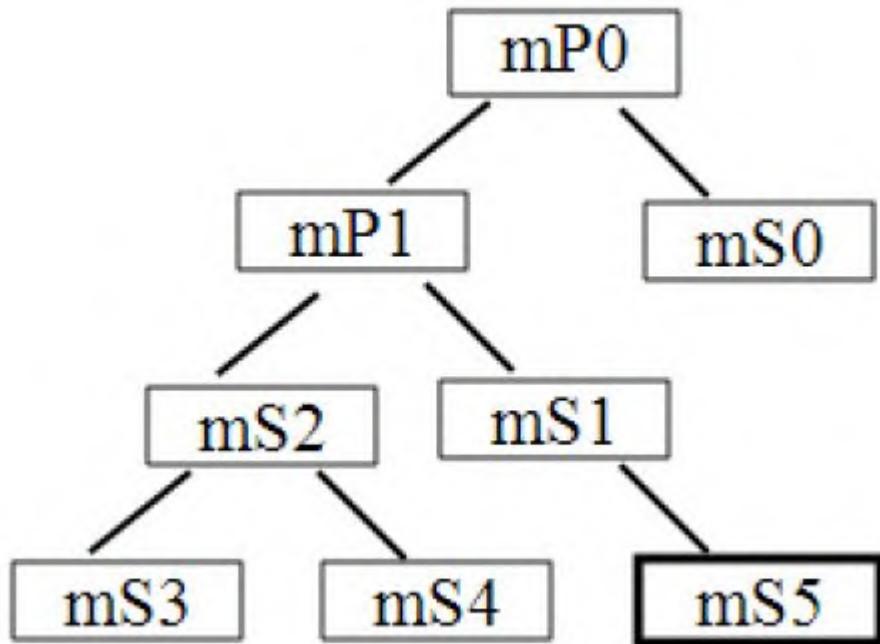


图5-2 HSM示例中状态关系

图5-2中，mS5是初始状态，由代码中的setInitialState函数设置。接着来看StateMachineTest的代码。

### [HSM示例]

```

// 接上面的代码。P0从State类派生。在HSM中，状态由类State来表达
class P0 extends State {

```

```

/*
    enter代表一个状态的Entry Action, SM进入此状态时将调用其EA。
    exit代表一个状态的Exit Action, SM退出某状态时将调用其EXA。
*/
public void enter() {.....// do sth here}
public void exit() {.....// do sth here}
/*

```

除了EA和EXA外，每个State中最重要的函数就是processMessage了。

在HSM中，外界和HSM交互的方式就是向其sendMessage。Message由当前State的processMessage

函数来处理。如果当前State成功处理此message，则返回HANDLED。否则返回NOT\_HANDLED。

在Message处理中，如果子状态返回NOT\_HANDLED，则其父状态的processMessage将被调用。

如果当前状态及祖先状态都不能处理，则HSM的unhandledMessage将被调用。而HSM的派生类

可重载unhandledMessage函数以处理这个不能被当前状态及祖先状态处理的消息。

```

/*
    public boolean processMessage(Message message) {
        return HANDLED; // P0能处理任何Message
    }
}

class P1 extends State {.....// 实现P1的enter,exit和
processMessage函数}

class S0 extends State {.....// 实现S0的enter,exit和
processMessage函数}

.....// S1到S4的定义

class S5 extends State {
    public void enter() {.....// 实现S5的enter函数}
    public void exit() {.....// 实现S5的exit函数}
    public boolean processMessage(Message message) {
        switch(message.what) {
            case: TRANSITION_CMD:
                transitionTo(mS4); // 切换状态时，需要调用此函数
                break;
            case: TRANSITION_CMD_DEFER_MSG:
                // deferMessage用于保留某个消息。而被保留的消息将
                // 留待到下一个状态中去处理
                deferMessage(message);
                transitionTo(mS1);
                break;
            default:
                break;
        }
    }
}
```

```

        }
        return HANDLED;
    }
}

.....// StateMachine其他一些可重载函数。以后碰到它们时再介绍

// 定义各个状态对应的对象
private P0 mP0 = new P0(); private P1 mP1 = new P1();
private S0 mS0 = new S0(); private S1 mS1 = new S1();
private S2 mS2 = new S2(); private S3 mS3 = new S3();
private S4 mS4 = new S4(); private S5 mS5 = new S5();

// 定义消息
final static int TRANSITION_CMD = 0;
final static int TRANSITION_CMD_DEFER_MSG = 1;
}

// 主函数
public void main() throws Exception {
    StateMachineTest smTest = new
StateMachineTest("StateMachineTest");
    smTest.start(); // 启动状态机
    synchronized (sm5) {
        // 外界只能通过obtainMessage以及sendMessage发送消息给SM去
执行
        smTest.sendMessage(obtainMessage(TRANSITION_CMD));
    }
    smTest.sendMessage(obtainMessage(TRANSITION_CMD_DEFER_MSG));
    .....
}
.....
}
}

```

上面代码介绍了HSM中一些重要的API。

- **addState:** 添加一个状态。同时还可指定父状态。
- **transitionTo:** 将状态机切换到某个状态。
- **obtainMessage:** 由于HSM内部是围绕一个Handler来工作的，所以外界只能调用HSM的obtainMessage以获取一个Message<sub>①</sub>。
- **sendMessage:** 发送消息给HSM。HSM中的Handler会处理它。

- `deferMessage`: 保留某个消息。该消息将留待下一个新状态中去处理。其内部实现就是把这些被`deferred`的`message`保存到一个队列中。当HSM切换到新状态后，这些`deferred`消息将被移到HSM内部Handler所对应消息队列的头部，从而新状态能首先处理这些`deferred`消息。
- `start`: 启动状态机。
- 停止状态机可使用`quit`或`quitNow`函数。这两个函数均会发送`SM_QUIT_CMD`消息给HSM内部的Handler，不过效果略有区别。当使用`quit`时，`SM_QUIT_CMD`添加在消息队列尾；而使用`quitNow`时，`SM_QUIT_CMD`被添加到消息队列头。

HSM中状态和状态之间的层级关系体现在哪些方面呢？以上述代码为例：

- SM启动后，初始状态的EA将按派生顺序执行。即其祖先状态的EA先执行，子状态的EA后执行。以示例代码中的初始状态mS5为例。当HSM的`start`调用完毕后，EA调用顺序为mP0、mP1、mS1、mS5。
- 当State发生切换时，旧State的exit先执行， newState的enter后执行，并且新旧State派生树上对应的State也需要执行exit或enter函数。以mS5切换到mS4为例，在此切换过程中，首先执行的是EXA，其顺序是mS5, mS1。注意，EXA执行的终点是离mS4和mS5最近的一个公共（即同时是mS4和mS5的祖先）祖先State（此处是mP1），但公共祖先状态的EXA不会执行。然后执行的是EA，其顺序是mS2, mS4。同理，公共祖先的EA也不会执行。细心的读者可以发现，HSM中EA和EXA执行顺序和C++类构造/析构函数执行顺序类似。EA执行顺序由祖先类开始直至子孙类，而析构函数的执行先从子孙类开始，直到祖先类。
- State处理Message时，如果子状态不能处理（返回`NOT_HANDLED`），则交给父状态去处理。这一点也和C++中类的派生函数类似。

HSM的介绍就到此为止，感兴趣的读者可自行研究HSM的实现。

## 2. AsyncChannel使用

AsyncChannel用于两个Handler之间的通信，其用法包含两种不同的应用模式（usage model）。

- 简单的request/response模式下，Server端无须维护Client的信息，它只要处理来自Client的请求即可。连接时，Client调用connectSync（同步连接）或connect（异步连接，连接成功后Client会收到CMD\_CHANNEL\_HALF\_CONNECTED消息）即可连接到Server。
- 与request/response模式相反，AsyncChannel中另外一种应用模式就是Server端维护Client的信息。这样，Server可以向Client发送自己的状态或者其他一些有意义的信息。与这种模式类似的应用场景就是第4章介绍的wpa\_cli和wpa\_supplicant。wpa\_cli可以发送命令给WPAS去执行。同时，WPAS也会将自己的状态及其他一些消息通知给wpa\_cli。

在WifiService相关模块中，第二种应用模式使用得较多。另外，AsyncChannel中Client和Server端在最开始建立连接关系时，可以采用同步或异步的方式。以异步方式为例介绍第二种应用模式中AsyncChannel的使用步骤。

- 1) Client调用AsyncChannel的connect函数。Client的Handler会收到一个名为CMD\_CHANNEL\_HALF\_CONNECTED消息。
- 2) Client在处理CMD\_CHANNEL\_HALF\_CONNECTED消息时，需通过sendMessage函数向Server端发送一个名为CMD\_CHANNEL\_FULL\_CONNECTION的消息。
- 3) Server端的Handler将收到此CMD\_CHANNEL\_FULL\_CONNECTION消息。成功处理它后，Server端先调用AsyncChannel的connected函数，然后通过sendMessage函数向Client端发送CMD\_CHANNEL\_FULLY\_CONNECTED消息（特别注意，详情见下文）。
- 4) Client端收到CMD\_CHANNEL\_FULLY\_CONNECTED消息。至此，Client和server端成功建立连接。
- 5) Client和Server端的两个Handler可借助sendMessage和replyToMessage来完成请求消息及回复消息的传递。注意，只有针对那些需要回复的情况，Server端才需调用replyToMessage。
- 6) 最后，Client和Server的任意一端都可以调用disconnect函数以结束连接。该函数将导致Client和Server端都会收到

CMD\_CHANNEL\_DISCONNECTED消息。

特别说明 上述步骤的描述来自AsyncChannel.java文件中的注释。但实际上Server端代码在处理CMD\_CHANNEL\_FULL\_CONNECTION消息时并不能按照上面的描述开展工作。因为AsyncChannel对象一般由客户端创建，而CMD\_CHANNEL\_FULL\_CONNECTION消息无法携带AsyncChannel对象（AsyncChannel对象无法通过Binder进行跨进程传递）。所以，Server端并不能获取客户端创建的这个AsyncChannel对象，它也就没办法调用AsyncChannel的connected函数。

那么，Server端的正确处理应该是什么样子呢？接下来将通过代码向读者展示正确的做法。

下面结合WifiManager中的相关代码来介绍AsyncChannel中第二种模式涉及的一些重要函数。

WifiManager的init函数中会创建一个AsyncChannel以和WifiService中的某个Handler建立连接关系，代码如下所示。

[-->WifiManager.java: : init]

```
private void init() {  
    /*  
     * 该函数内部通过Binder机制调用wifiService的  
     * getWifiServiceMessenger函数，返回值  
     * 是一个类型为Messenger的对象。Messenger从Parcelable派生，其内部有  
     * 一个IMessenger  
     * 对象用于支持跨进程的Binder通信  
     */  
}
```

WifiService中，getWifiServiceMessenger的代码如下。

```
public Messenger getWifiServiceMessenger() {  
    .....// 权限检查
```

```
        return new Messenger(mAsyncServiceHandler); // 通过
Messenger封装了目标Handler
    }
}

mWifiServiceMessenger = getWifiServiceMessenger();
.....
// 创建一个HandlerThread。对HandlerThread不熟悉的读者请参考脚注
sHandlerThread = new HandlerThread("WifiManager");
sHandlerThread.start();

// Client中的Handler，它将运行在sHandlerThread线程中
// AsyncChannel对Client Handler运行在什么线程没有要求
mHandler = new ServiceHandler(sHandlerThread.getLooper());
/*
connect是AsyncChannel的重要函数。此处使用的connect函数原型如下：
connect(Context srcContext, Handler srcHandler, Messenger
dstMessenger)
    srcContext: 为Client端的Context对象，AsyncChannel内部将使用
它。
    srcHandler: 为Client端的Handler。
    dstMessenger: 是Server端Handler在Client端的代表。
*/
mAsyncChannel.connect(mContext, mHandler,
mWifiServiceMessenger);
.....
}
```

connect函数将触发Client端Handler收到一个CMD\_CHANNEL\_HALF\_CONNECTED消息。马上来看WifiManager中ServiceHandler。

[-->WifiManager.java: : ServiceHandler]

```
private class ServiceHandler extends Handler {
    .....// 此处只关注和AsyncChannel相关的内容
    public void handleMessage(Message message) {
        .....
        switch (message.what) {
            case AsyncChannel.CMD_CHANNEL_HALF_CONNECTED:// 半连接
成功
                if (message.arg1 ==
                    AsyncChannel.STATUS_SUCCESSFUL) {
```

```

        // 向Server端发送CMD_CHANNEL_FULL_CONNECTION消息
        mAsyncChannel.sendMessage(AsyncChannel
            .CMD_CHANNEL_FULL_CONNECTION);
    }.....
    break;
case AsyncChannel.CMD_CHANNEL_FULLY_CONNECTED: // 连接成功
    break;
case AsyncChannel.CMD_CHANNEL_DISCONNECTED:// 连接关闭
    mAsyncChannel = null;
    getLooper().quit();// 连接关闭, 退出线程
    break;
.....
}
.....
}

```

WifiService中的目标Handler是AsyncServiceHandler，其代码如下所示。

```

[-->WifiService: : AsyncServiceHandler]

private class AsyncServiceHandler extends Handler {
    .....
    // 请读者先看它是如何处理CMD_CHANNEL_FULL_CONNECTION消息的
    public void handleMessage(Message msg) {
        switch (msg.what) {
            case AsyncChannel.CMD_CHANNEL_HALF_CONNECTED: {
                // 处理因ac.connect调用而收到的
                // CMD_CHANNEL_HALF_CONNECTED消息
                // 该消息携带了一个AsyncChannel对象, 即ac
                if (msg.arg1 == AsyncChannel.STATUS_SUCCESSFUL)
                {
                    // 保存这个AsyncChannel对象, 用于向client发送
                    // 消息
                    mClients.add((AsyncChannel) msg.obj);
                    /*
                     注意, Server端可在此处向Client端发送
                     CMD_CHANNEL_FULLY_CONNECTED消息。
                     例如:
                     AsyncChannel sample = (AsyncChannel)
                     msg.obj;

```

```

sample.sendMessage(AsyncChannel.CMD_CHANNEL_FULLY_CONNECTED);
    /*
    }
    break;
}
case AsyncChannel.CMD_CHANNEL_DISCONNECTED: {
    mClients.remove((AsyncChannel) msg.obj);
    break;
}
case AsyncChannel.CMD_CHANNEL_FULL_CONNECTION: { ///
Server端先收到此消息
/*
    新创建一个AsyncChannel对象ac，然后调用它的connect函数。
其中：
    msg.replyTo代表Client端的Handler，也就是WifiManager
    中ServiceHandler。
    connect函数将触发CMD_CHANNEL_HALF_CONNECTED消息被发
送，而且该消息
    会携带对应的AsyncChannel对象，即此处的ac。
    请读者回到handleMessage的前面去看
    CMD_CHANNEL_HALF_CONNECTED的处理。
*/
    AsyncChannel ac = new AsyncChannel(); // 创建一个新的
    的AsyncChannel对象
    ac.connect(mContext, this, msg.replyTo);
    break;
}
.....
}
}

```

根据WifiService的代码并结合上文“特别说明”，由于Server端无法得到Client端的AsyncChannel对象，所以它干脆自己又新创建了一个AsyncChannel，并connect到客户端。这样，Server和Client端实际上有两个不同的AsyncChannel对象，并且都需要调用connect函数。

**提示** 如果AsyncChannel支持跨进程传递，那么Server端只要获取Client端传递过来的AsyncChannel对象，并调用其connected（注意，不是connect）函数即可。

介绍完HSM和AsyncChannel后，马上来看WifiService的创建和相关的初始化工作。

① 关于Android中Handler的实现原理，读者可阅读《深入理解Android：卷 I》5.4节。

② 关于Java Binder的实现机制，读者可参考《深入理解Android：卷 II》第2章。

## 5.2.2 WifiService构造函数分析

WifiService构造函数的代码如下所示。

[-->WifiService.java: : WifiService构造函数]

```
WifiService(Context context) {
    mContext = context;

        // 从系统属性“wifi.interface”中取出无线网络设备接口名。默认值
为“wlan0”
        mInterfaceName =
SystemProperties.get("wifi.interface", "wlan0");
        // 创建一个WifiStateMachine对象，它是WifiService相关模块中的
核心
        mWifiStateMachine = new WifiStateMachine(mContext,
mInterfaceName);

        /*
        RSSI轮询机制。RSSI为Receive Signal Strength Indication（接
收信号强度指示）
            的缩写，它反映了无线网络质量的好坏。WPAS支持的RSSI信息包括：接收
信号强度、连接速度
            （link speed）噪声强度（noise）和频率。在WPAS中，RSSI信息由
wpa_signal_info
            结构体来表达。
        */
        mWifiStateMachine.enableRssiPolling(true);

        // 和BatteryStatsService交互。感兴趣的读者可阅读《深入理解
Android: 卷II》5.5.2节
        mBatteryStats = BatteryStatsService.getService();

        .....// 广播事件注册等处理。由于篇幅问题，本章将略去一些重要程
度较低的代码
        HandlerThread wifiThread = new
HandlerThread("WifiService");
        wifiThread.start();
        // mAsyncServiceHandler用于AsyncChannel，其交互对象来自
WifiManager
        mAsyncServiceHandler = new
AsyncServiceHandler(wifiThread.getLooper());
```

```
// mWifiStateMachineHandler也用于AsyncChannel，其交互对象来自WifiStateMachine
mWifiStateMachineHandler = new
WifiStateMachineHandler(wifiThread.getLooper());
.....// 其他一些工作
}
```

WifiService构造函数中主要工作是创建一些核心对象，其中：

- WifiStateMachine是WifiService中的核心，是本章分析的重点对象。
- mAsyncServiceHandler和mWifiStateMachineHandler都和AsyncChannel相关，与它们交互的Handler分别位于WifiManager和WifiStateMachine中。

马上来看最重要的WifiStateMachine。

### 5.2.3 WifiStateMachine介绍

首先来看WifiStateMachine的构造函数，其内容较多，我们分两段来介绍。

#### 1. WifiStateMachine构造函数分析之一

```
[-->WifiStateMachine.java: : WifiStateMachine构造函数代码段  
一]  
  
public WifiStateMachine(Context context, String wlanInterface)  
{  
    super(TAG);  
    mContext = context;  
    mInterfaceName = wlanInterface;  
  
    // 创建一个NetworkInfo，它实际上代表一个网络设备的状态信息 (status  
    // of a network interface)  
    mNetworkInfo = new  
    NetworkInfo(ConnectivityManager.TYPE_WIFI,  
                0, NETWORKTYPE, "");  
  
    // 和BatteryStatsService交互，BSS注册的服务名叫“batteryinfo”  
    mBatteryStats = IBatteryStats.Stub.asInterface  
                    (ServiceManager.getService("batteryinfo"));  
  
    // 创建和NewtorkManagmentService交互的Binder客户端  
    IBinder b =  
    ServiceManager.getService(Context.NETWORKMANAGEMENT_SERVICE);  
    mNwService = INetworkManagementService.Stub.asInterface(b);  
    /*  
     * 判断系统是否支持Wi-Fi Display功能。本书不讨论WFD，感兴趣的读者可阅读  
     * 笔者的一篇博文  
     * http:// blog.csdn.net/innost/article/details/8474683  
     * " Android Wi-Fi Display (Miracast) 介绍"。  
     */  
    mP2pSupported =  
    mContext.getPackageManager().hasSystemFeature(  
        PackageManager.FEATURE_WIFI_DIRECT);  
  
    /*
```

`WifiNative`: 用于和`wpa_supplicant`交互。它和4.2.3节中控制API知识相关。

`WifiMonitor`: 内部将创建一个线程，并借助`WifiNative`去接收并处理来自WPAS的信息。

`WifiConfigStore`: 它对应一个配置文件，位置为`/data/misc/wifi/ifconfig.txt`。

该文件用于存储每个无线网络的配置项。例如代理地址、静态IP地址等。读者可在`Settings`

中选择某个无线网络，长按以弹出修改对话框，然后选择“高级选项”即可设置这些信息。

```
/*
mWifiNative = new WifiNative(mInterfaceName);
mWifiConfigStore = new WifiConfigStore(context,
mWifiNative);
mWifiMonitor = new WifiMonitor(this, mWifiNative);

// 用于保存DHCP的一些信息
mDhcpInfoInternal = new DhcpInfoInternal();
// WifiInfo用于存储手机当前连接上的无线网络的一些信息，包括IP地址、
ssid等内容
mWifiInfo = new WifiInfo();

// SuplicantStateTracker用于跟踪WPAS的状态，它也是一个
StateMachine
mSupplicantStateTracker = new SupplicantStateTracker
(context, this,
mWifiConfigStore, getHandler());
// LinkProperties用于描述网络链接(network link)的一些属性，如IP
地址、DNS地址和路由设置
mLinkProperties = new LinkProperties();

// WifiApConfigStore和Soft AP模式有关，用于存储Soft AP模式中使用
到的一些配置信息
// WifiApConfigStore是一个StateMachine。配置信息存储于/data//
misc/wifi/softap.conf中
WifiApConfigStore wifiApConfigStore =
WifiApConfigStore.makeWifiApConfigStore(
    context, getHandler());
wifiApConfigStore.loadApConfiguration();

// mWifiApConfigChannel的类型是AsyncChannel，它将和
wifiApConfigStore中的某个Handler通信
mWifiApConfigChannel.connectSync(mContext, getHandler(),
    wifiApConfigStore.getMessenger());
```

```
mNetworkInfo.setIsAvailable(false);
mLinkProperties.clear();
.....
// 设置扫描间隔时间：当驱动不支持Background扫描时，Framework将定时
开展扫描工作
// 默认值为300秒
mDefaultFrameworkScanIntervalMs =
mContext.getResources().getInteger(
    R.integer.config_wifi_framework_scan_interval);
/*
driver stop延迟，默认是120秒。该变量和emergency calls（紧急呼叫）
有关。
处于这种模式下，即使用户选择关闭Wi-Fi，WifiStateMachine也不会立即执行它，而是要
等待一段时间才真正去关闭Wi-Fi。
*/
mDriverStopDelayMs = mContext.getResources().getInteger(
    R.integer.config_wifi_driver_stop_delay);

// 是否支持Background扫描
mBackgroundScanSupported =
mContext.getResources().getBoolean(
    R.bool.config_wifi_background_scan_support);

// 和P2P有关。以后再介绍
mPrimaryDeviceType = mContext.getResources().getString(
    R.string.config_wifi_p2p_device_type);

// WIFI_SUSPEND_OPTIMIZATIONS_ENABLED变量用于控制手机睡眠期间是否保持Wi-Fi开启
mUserWantsSuspendOpt.set(Settings.Global
    .getInt(mContext.getContentResolver(),
Settings.Global.WIFI_SUSPEND_OPTIMIZATIONS_ENABLED, 1) == 1);

.....// 处理ACTION_START_SCAN广播事件
.....// 处理ACTION_SCREEN_ON/OFF广播事件
.....// 处理ACTION_DELAYED_DRIVER_STOP广播事件
.....// 监视ContentProvider中
WIFI_SUSPEND_OPTIMIZATIONS_ENABLED设置的变化
// mScanResultCache用于保存扫描结果
mScanResultCache = new LruCache<String, ScanResult>
(SCAN_RESULT_CACHE_SIZE);
.....// 申请WakeLock
```

重点介绍其中的三个对象，分别是WifiNative、WifiMonitor以及SuplicantStateTracker。

### (1) WifiNative

根据上文描述，WifiNative用于和WPAS通信，其内部定义了较多的native方法（对应的JNI模块是android\_net\_wifi\_Wifi）。本节介绍其中最重要的两个方法。

第一个方法是startSuplicant，用于启动WPAS。startSuplicant是一个native函数，其JNI [①](#) 函数为 android\_net\_wifi\_startSuplicant，代码如下所示。

```
[-->android_net_wifi_Wifi.c: :  
 android_net_wifi_startSuplicant]  
  
static jboolean android_net_wifi_startSuplicant(JNIEnv* env,  
 jobject, jboolean  
 p2pSupported)  
{  
     return (jboolean) (::wifi_start_supplicant(p2pSupported) ==  
 0);  
}
```

wifi\_start\_supplicant代码如下所示。

```
[-->wifi.c::wifi_start_supplicant]  
  
int wifi_start_supplicant(int p2p_supported)  
{  
    char supp_status[PROPERTY_VALUE_MAX] = {'\0'};  
    int count = 200;  
    // 该宏在build/core/combo/include/arch/linux-arm/AndroidConfig.h  
    // 中被定义为1  
    #ifdef HAVE_LIBC_SYSTEM_PROPERTIES  
        const prop_info *pi;  
        unsigned serial = 0, i;  
    #endif  
        // 和P2P有关  
        if (p2p_supported) { // P2P_SUPPLICANT_NAME值  
            for("p2p_supplicant"
```

```

        strcpy(supplicant_name, P2P_SUPPLICANT_NAME);
        // P2P_PROP_NAME值为“init.svc.p2p_supplicant”
        strcpy(supplicant_prop_name, P2P_PROP_NAME);
        /*
            P2P_CONFIG_FILE的值
        为“/data/misc/wifi/p2p_supplicant.conf”。下面这个函数将把
            /system/etc/wifi/wpa_supplicant.conf的内容复制到
        P2P_CONFIG_FILE中。
        */
        if (ensure_config_file_exists(P2P_CONFIG_FILE) < 0)
    return -1;
    } else {
        strcpy(supplicant_name, SUPPLICANT_NAME); //
        SUPPLICANT_NAME值为“wpa_supplicant”
        // SUPP_PROP_NAME值为“init.svc.wpa_supplicant”
        strcpy(supplicant_prop_name, SUPP_PROP_NAME);
    }
    // 如果WPAS已经启动，则直接返回
    if (property_get(supplicant_name, supp_status, NULL)
        && strcmp(supp_status, "running") == 0)
return 0;
    // SUPP_CONFIG_FILE的值
    为“/data/misc/wifi/wpa_supplicant.conf”
    if (ensure_config_file_exists(SUPP_CONFIG_FILE) < 0)
return -1;
    // entropy文件，用于增加随机数生成的随机性
    if (ensure_entropy_file_exists() < 0)
        ALOGE("Wi-Fi entropy file was not created");
    // 关闭之前创建的wpa_ctrl对象
    wifi_wpa_ctrl_cleanup();

    for (i=0; i<MAX_CONNS; i++)
        exit_sockets[i][0] = exit_sockets[i][1] = -1;

```

```

#endif HAVE_LIBC_SYSTEM_PROPERTIES
    // supplicant_prop_name值为“init.svc.wpa_supplicant”
    pi = __system_property_find(supplicant_prop_name);
    .....
#endif

```

```

        property_get("wifi.interface", primary_iface,
WIFI_TEST_INTERFACE);
    /*

```

通过设置“ctrl.start”属性来启动wpa\_supplicant服务。该属性将触发init fork一个子

进程用于运行wpa\_supplicant。同时，init还会添加一个新的属性

```

"init.svc.wpa_supplicant"用于跟踪wpa_supplicant的状态。
*/
property_set("ctl.start", supplicant_name);
sched_yield();
// 下面这个循环用于查询"init.svc.wpa_supplicant"的属性值
// 如果其值变成"running", 表示wpa_supplicant成功运行
while (count-- > 0) { // count初值为200。while循环最多等待20秒
#endif HAVE_LIBC_SYSTEM_PROPERTIES
    if (pi == NULL) {
        pi = __system_property_find(supplicant_prop_name);
    }
    if (pi != NULL) {
        __system_property_read(pi, NULL, supp_status);
        if (strcmp(supp_status, "running") == 0)      return
0;
        else if (pi->serial != serial && // 如果WPAS已经停止,
则直接返回-1
            strcmp(supp_status, "stopped") == 0)
            return -1;
    }
#else
    .....
#endif
    usleep(100000); // 等待wpa_supplicant的状态
}
return -1;
}

```

图5-3显示了wpa\_supplicant运行过程中及退出后"init.svc.wpa\_supplicant"属性值的变化。

```

shell@android:/ # getprop | grep wpa
[init.svc.wpa_supplicant]: [running]
shell@android:/ # getprop | grep wpa
[init.svc.wpa_supplicant]: [stopped]

```

图5-3 init.svc.wpa\_supplicant属性

提示 对Android属性机制和init工作原理感兴趣的读者不妨阅读《深入理解Android：卷 I》第3章。

第二个要介绍的函数是connectToSupplicant，它将通过WPAS控制API和WPAS建立交互关系。

[-->WifiNative.java: : connectToSupplicant]

```
public boolean connectToSupplicant() {
    // mInterface的值为“wlan0”，由属性“wifi.interface”决定
    return connectToSupplicant(mInterface); // 调用native函数
}
private native boolean connectToSupplicant(String iface);
```

与connectToSupplicant对应的JNI函数是android\_net\_wifi\_connectToSupplicant，其代码如下所示。

[-->android\_net\_wifi\_Wifi.cpp: : android\_net\_wifi\_connectToSupplicant]

```
static jboolean android_net_wifi_connectToSupplicant(JNIEnv* env,
    jobject, jstring jIface)
{
    ScopedUtfChars ifname(env, jIface);
    return (jboolean)
        (::wifi_connect_to_supplicant(ifname.c_str()) == 0);
}
```

wifi\_connect\_to\_supplicant的代码如下所示。

[-->wifi.c: : wifi\_connect\_to\_supplicant]

```
int wifi_connect_to_supplicant(const char *ifname)
{
    char path[256];
    /*
        Android 4.2支持STA和P2P设备并发（concurrent）工作，STA用
        PRIMARY（值为0）来标示，
        而P2P设备用SECONDARY（值为1）代表。is_primary_interface用于判断
        ifname是否代表STA。
    */
    if (is_primary_interface(ifname)) {
        // IFACE_DIR的值为“/data/system/wpa_supplicant”。笔者测试
```

的几个手机中都没有该文件夹

```
    if (access(IFACE_DIR, F_OK) == 0) {
        snprintf(path, sizeof(path), "%s/%s", IFACE_DIR,
primary_iface);
    } else {
        strlcpy(path, primary_iface, sizeof(path));
    }
    return wifi_connect_on_socket_path(PRIMARY, path); // PRIMARY值为0
} else {
    sprintf(path, "%s/%s", CONTROL_IFACE_PATH, ifname);
    return wifi_connect_on_socket_path(SECONDARY, path); // SECONDARY值为1
}
}
```

来看wifi\_connect\_on\_socket\_path，其代码如下所示。

[-->wifi.c: : wifi\_connect\_on\_socket\_path]

```
int wifi_connect_on_socket_path(int index, const char *path)
{
    char supp_status[PROPERTY_VALUE_MAX] = {'\0'};

    // 判断wpa_supplicant进程是否已经启动
    if (!property_get(supplicant_prop_name, supp_status, NULL)
        || strcmp(supp_status, "running") != 0)
        return -1;
    // 创建第一个wpa_ctrl对象，用于发送命令
    ctrl_conn[index] = wpa_ctrl_open(path);
    .....
    // 创建第二个wpa_ctrl对象，用于接收unsolicited event
    monitor_conn[index] = wpa_ctrl_open(path);
    .....
    // 必须调用wpa_ctrl_attach函数以启用unsolicited event接收功能
    if (wpa_ctrl_attach(monitor_conn[index]) != 0) {.....}
    // 创建一个socketpair，它用于触发WifiNative关闭和WPAS的连接
    if (socketpair(AF_UNIX, SOCK_STREAM, 0,
exit_sockets[index]) == -1) {.....}
    return 0;
}
```

由于Android 4.2支持两个并发设备，所以每个并发设备各有两个wpa\_ctrl对象。

- `ctrl_conn[PRIMARY]`、`monitor_conn[PRIMARY]`：用于STA设备。  
`ctrl_conn`用于向WPAS发送命令并接收对应命令的回复，而  
`monitor_conn`用于接收来自WPAS的unsolicited event。
- `ctrl_conn[SECONDARY]`、`monitor_conn[SECONDARY]`：这两个  
`wpa_ctrl`对象用于P2P设备。

另外，`exit_sockets`保存了`socketpair`创建的socket句柄，这些句柄  
 用于WifiService通知WifiNative去关闭它和WPAS的连接。

**提示** `wifi.c`中，`wifi_send_command`会使用`ctrl_conn`中的`wpa_ctrl`  
 对象向WPAS发送命令并接收到回复，而`wifi_recv`函数将使用  
`monitor_conn`中的`wpa_ctrl`对象接收来自WPAS的消息。这两个函数比  
 较简单，请读者可自行阅读它。

下面来看WifiMonitor，它将使用`monitor_conn`中的`wpa_ctrl`对象。

## (2) WifiMonitor

WifiMonitor最重要的内容是其内部的WifiMonitor线程，该线程专门  
 用于接收来自WPAS的消息。代码如下所示。

```
[-->WifiMonitor.java: : MonitorThread]

class MonitorThread extends Thread {
    public MonitorThread() {
        super("WifiMonitor");
    }
    public void run() {
        if (connectToSupplicant()) { // 连接WPAS,
mStateMachine指向WifiStateMachine
            // 连接成功后，将向WifiStateMachine发送
SUP_CONNECTION_EVENT消息

mStateMachine.sendMessage(SUP_CONNECTION_EVENT);
        } else {

mStateMachine.sendMessage(SUP_DISCONNECTION_EVENT);
        return;
    }
    for (;;) {
        // waitForEvent内部会调用wifi.c中的
```

```

wifi_wait_on_socket函数
    String eventStr = mWifiNative.waitForEvent();
    // 解析WPAS的消息格式。EVENT_PREFIX_STR的值为“CTRL-
EVENT-”
    if (!eventStr.startsWith(EVENT_PREFIX_STR)) {
        .....// 非“CTRL-EVENT-”消息
        continue;
    }
    // 处理“CTRL-EVENT-”消息
    String eventName =
eventStr.substring(EVENT_PREFIX_LEN_STR);
    int nameEnd = eventName.indexOf(' ');
    if (nameEnd != -1)
        eventName = eventName.substring(0,
nameEnd);
    .....
    int event;
    if (eventName.equals(CONNECTED_STR)) // 对应
为“CONNECTED”消息
        event = CONNECTED;
    .....
    else if (eventName.equals(STATE_CHANGE_STR)) // 对应为“STATE-CHANGED”
        event = STATE_CHANGE;
    else if (eventName.equals(SCAN_RESULTS_STR)) // 对应为“SCAN-RESULTS”
        event = SCAN_RESULTS;
    .....
    else if (eventName.equals(DRIVER_STATE_STR)) // 对应为“DRIVER-STATE”
        event = DRIVER_STATE;
    else if (eventName.equals(EAP_FAILURE_STR)) // 对应为“EAP-FAILURE”
        event = EAP_FAILURE;
    else
        event = UNKNOWN;
    /*
        提取消息中的其他信息，以CONNECTED消息为例，其消息全内
容为：
        CTRL-EVENT-CONNECTED - Connection to
        xx:xx:xx:xx:xx:xx completed
        其中，xx:xx:xx:xx:xx:xx代表目标AP的BSSID。
    */
    String eventData = eventStr;
    if (event == DRIVER_STATE || event ==
LINK_SPEED)

```

```

        eventData = eventData.split(" ")[1];
        else if (event == STATE_CHANGE || event ==
EAP_FAILURE) {
            .....
        } .....
        if (event == STATE_CHANGE) { // WPAS状态发生变化
            handleSupplicantStateChange(eventData);
        } else if (event == DRIVER_STATE) {
            handleDriverEvent(eventData);
        } .....
        else {// 其他事件处理
            handleEvent(event, eventData);
        }
        mRecvErrors = 0;
    }
}
.....
}

```

上述代码中：

- handleSupplicantStateChange用于处理WPAS的状态变化（见下文解释），它将先把这些变化信息交给WifiStateMachine去处理。而WifiStateMachine将根据处理情况再决定是否需要由下一节介绍的SupplicantStateTracker来处理。handleSupplicant StateChange代码比较简单，读者可自行阅读它。
- handleDriverEvent用于处理来Driver的信息<sup>②</sup>。
- handleEvent用于处理其他消息事件。详情见下文。

注意 WPAS的状态指的是wpa\_sm状态机中的状态，包括WPA\_DISCONNECTED、WPA\_INTERFACE\_DISABLED、WPA\_INACTIVE、WPA\_SCANNING、WPA\_AUTHENTICATING、WPA\_ASSOCIATING、WPA\_ASSOCIATED、WPA\_4WAY\_HANDSHAKE、WPA\_GROUP\_HANDSHAKE、WPA\_COMPLETED共10个状态。WifiService定义了SupplicantState类来描述WPAS的状态，包括DISCONNECTED、INTERFACE\_DISABLED、INACTIVE、SCANNING、AUTHENTICATING、ASSOCIATING、ASSOCIATED、FOUR\_WAY\_HANDSHAKE、GROUP\_HANDSHAKE、COMPLETED、DORMANT、UNINITIALIZED、INVALID共13个状态。其中最后三个状态是WifiService定义的，但笔者在代码中没有找到使用它们的地方。

下面简单介绍handleEvent，其代码如下所示。

```
[-->WifiMonitor.java: : handleEvent]

void handleEvent(int event, String remainder) {
    switch (event) {
        case DISCONNECTED:
            handleNetworkStateChange(NetworkInfo.DetailedState
                .DISCONNECTED,
            remainder);
            break;
        case CONNECTED:// 该事件表示WPAS成功加入一个无线网络
            handleNetworkStateChange(NetworkInfo.DetailedState.CONNECTED,
            remainder);
            break;
        case SCAN_RESULTS:// 该事件表示WPAS已经完成扫描，客户端可以
            // 来查询扫描结果
            mStateMachine.sendMessage(SCAN_RESULTS_EVENT); //
            // 处理扫描结果消息
            break;
        case UNKNOWN:
            break;
    }
}
```

先介绍SuplicantStateTracker。后文再分析CONNECTED和SCAN\_RESULTS消息的处理流程。

### (3) SuplicantStateTracker

SuplicantStateTracker用于跟踪和处理WPAS的状态变化。根据前面对WPAS中的状态以及WifiService中的状态介绍可知。在WifiService中，WPAS的状态由SuplicantState来表示，而和它相关得状态管理模块就是此处的SuplicantStateTracker。SuplicantStateTracker也从StateMachine派生，并且它还定义了8个状态对象。相关代码如下所示。

```
[-->SuplicantStateTracker.java: : SuplicantStateTracker]
```

```
public SuplicantStateTracker(Context c, WifiStateMachine wsm,
                           WifiConfigStore wcs, Handler t) {
    super(TAG, t.getLooper());
```

```

mContext = c;
mWifiStateMachine = wsm;
mWifiConfigStore = wcs;
addState(mDefaultState);
addState(mUninitializedState, mDefaultState);
addState(mInactiveState, mDefaultState);
addState(mDisconnectState, mDefaultState);
addState(mScanState, mDefaultState);
addState(mHandshakeState, mDefaultState);
addState(mCompletedState, mDefaultState);
addState(mDormantState, mDefaultState);

setInitialState(mUninitializedState); // 初始状态为
mUninitializedState
start(); // 启动状态机
}

```

- SuplicantState中的AUTHENTICATING、ASSOCIATING、ASSOCIATED、FOUR\_WAY\_HANDSHAKE和GROUP\_HANDSHAKE均对应于此处的mHandshakeState。
- SuplicantState中的UNINITIALIZED和INVALID对应于此处的mUninitializedState。

SuplicantStateTracker比较简单，而且它也不影响本章的分析流程。读者可在阅读完本章的基础上，自行对其开展研究。

下面来看WifiStateMachine构造函数的最后一部分。

## 2. WifiStateMachine构造函数分析之二

[-->WifiStateMachine.java: : WifiStateMachine构造函数代码段  
二]

.....// WifiStateMachine中的状态。说实话，笔者还没见过如此复杂的状态机

```

addState(mDefaultState);
addState(mInitialState, mDefaultState);
addState(mDriverUnloadingState, mDefaultState);
addState(mDriverUnloadedState, mDefaultState);
addState(mDriverFailedState, mDriverUnloadedState);
.....// WifiStateMachine一共定义了30个状态
addState(mSoftApStoppingState, mDefaultState);

```

```

        setInitialState(mInitialState); // 设置初始状态为mInitialState
        .....// 和StateMachine日志记录相关设置
        start(); // 启动状态机
    }
}

```

WifiStateMachine共定义30个状态，其种类和层级关系如图5-4所示。

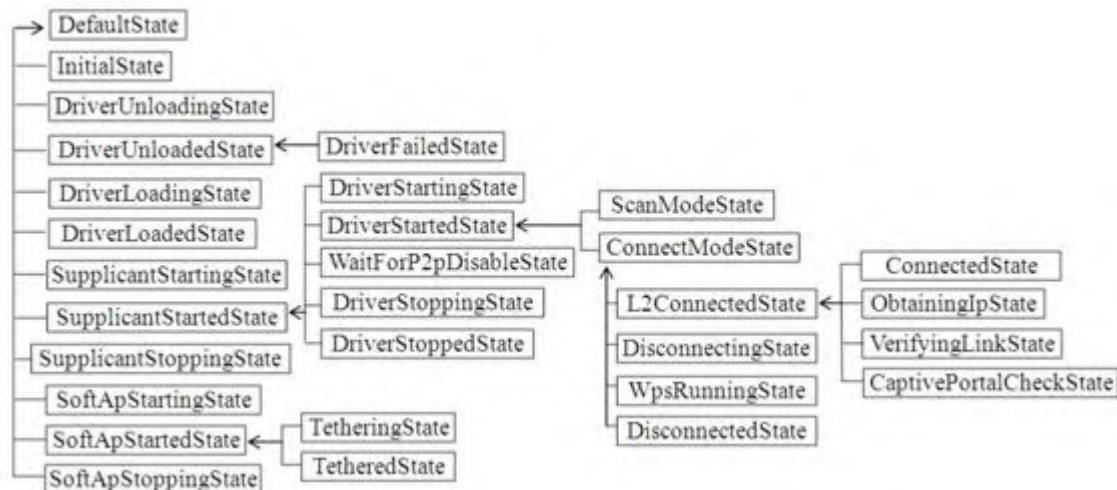


图5-4 WifiStateMachine中的状态及层级关系

图5-4中，箭头所指的状态为父状态。本节先介绍和初始状态的相关代码，其他状态的功能等碰到它们时再来分析。

**提示** 如果算上SupplicantStateTracker中的8个状态以及后续章节将要介绍的P2pStateMachine中的15个状态，Java层中Wi-Fi相关的状态机竟然多达63个状态（还没有计算Wi-Fi模块其他代码中定义的好些个状态机所包含的状态）。笔者很难理解为什么WifiService相关模块会定义如此多的状态。这些状态使得WifiService的分析难度陡增。而且，在整个Wi-Fi模块中，wpa\_supplicant作为核心已经完成了绝大部分的工作，为什么WifiService还会如此复杂呢？欢迎读者对此问题和笔者展开讨论。

WifiStateMachine的初始状态是mInitialState，其类型是InitialState。根据前文对HSM的介绍，其enter函数将被调用（由于InitialState的父状态DefaultState并未实现enter函数，故此处略去）。

[-->WifiStateMachine.java: : InitialState: enter]

```
class InitialState extends State {
    public void enter() {
        // 判断Wlan Driver是否已经加载，其内部实现通过"wlan.driver.status"属性的值来判断
        if (mWifiNative.isDriverLoaded())
            transitionTo(mDriverLoadedState);
        else transitionTo(mDriverUnloadedState); // 假设此时驱动还没有加载，故我们将转入此状态
        // 获取和WifiP2pService交互的对象
        mWifiP2pManager = (WifiP2pManager)
mContext.getSystemService(
        Context.WIFI_P2P_SERVICE);
        // mWifiP2pChannel用于和WifiP2pService中的某个Handler交互
        mWifiP2pChannel.connect(mContext, getHandler(),
                mWifiP2pManager.getMessenger());
        try {
            mNwService.disableIpv6(mInterfaceName);
            // 禁止Ipv6，NWService将和Netd交互
        } .....
    }
}
```

结合上述代码，当WifiStateMachine开始运行后，其最终将进入DriverUnloadedState。由于DriverUnloadedState的enter函数没有做什么有意义的工作，所以此处不再讨论它。

至此，WifiService第一条分析路线就算结束。虽然WifiService创建工作涉及的流程并不长，但相信读者也会感觉WifiService的代码难度其实并不算小。从下一节开始，读者还将进一步体会到这一点。

① 可参考《深入理解Android：卷 I》第2章JNI相关的重要知识。

② 笔者搜索了相关代码，在wlan芯片厂商提供的一些供WPAS使用的动态库中会发送DRIVER-EVENT。相关代码可参考  
hardware/broadcom/wlan/bcmdhd/wpa\_supplicant\_8\_lib/driver\_cmd\_n180211.c

## 5.3 加入无线网络分析

本节介绍WifiService的分析路线二，即在Settings中打开无线功能并加入一个无线网络。这条线路分为两个部分。

- 首先是Settings中的处理。它将利用WifiManager来操作无线网络。本节先介绍这一部分的内容，然后提取Settings中使用WifiManager的关键函数。
- 分析前面部分所涉及的WifiManager关键函数，这些函数将促使WifiService完成一系列复杂的操作。

先来看Settings中的Wi-Fi相关处理。

### 5.3.1 Settings操作Wi-Fi分析

Settings中设置Wi-Fi的页面如图5-5所示。



图5-5 Wi-Fi设置页面

左图所示为当前搜索到的无线网络信息。当选择图中“Test”无线网络时，进入右图。

右图所示为“Test”无线网络设置对话框。用户在“密码”一栏中输入密码后，点击“连接”按钮即可加入目标无线网络“Test”。

在Settings的代码中，和图5-5所示UI相关的类如图5-6所示。

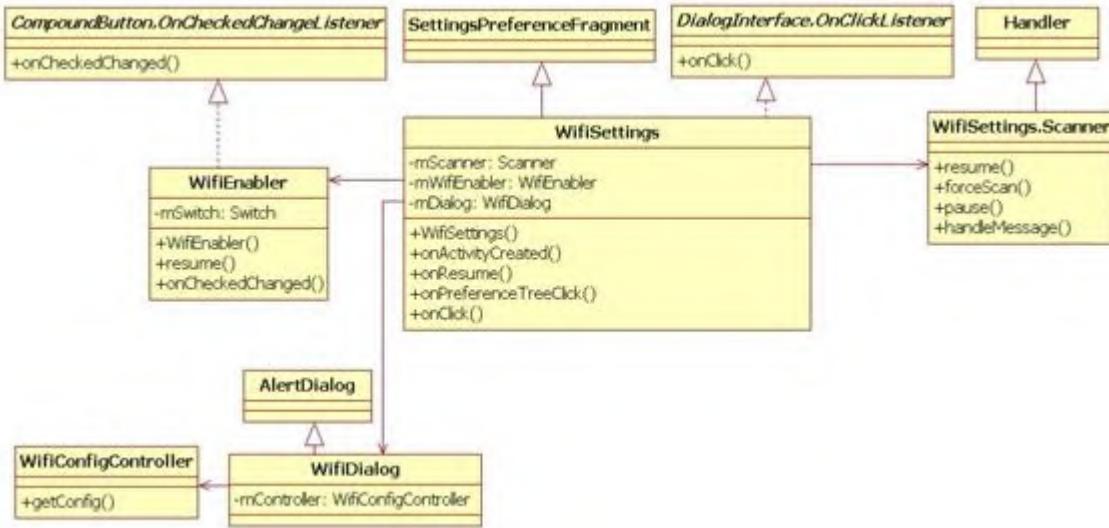


图5-6 Settings中Wi-Fi设置相关类

图5-6中，`WiFiSettings`对应于图5-5的左图。另外，`WifiEnabler`类中有一个`mSwitch`对象，该对象就是图5-5左图中右上角对应的Wi-Fi开关按钮。

`WiFiDialog`类对应于图5-5中的右图。`WiFiDialog`显示的无线网络配置信息由`WiFiConfigController`来控制和管理。

`WiFiSettings`的内部类`Scanner`用于处理和无线网络扫描相关的工作。

下面将按调用顺序来分析这些类的功能。首先是`WiFiSettings`相关类的创建。

## 1. WiFiSettings相关类初始化分析

先来看`WiFiSetting`的构造函数，代码如下所示。

[-->`WiFiSettings.java`]

```

public WiFiSettings() {
    mFilter = new IntentFilter();
    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.CONFIGURED_NETWORKS_CHANGED_ACTION);

mFilter.addAction(WifiManager.LINK_CONFIGURATION_CHANGED_ACTION);

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

    mReceiver = new BroadcastReceiver() {
        public void onReceive(Context context, Intent intent)
{
            handleEvent(context, intent); // 处理广播事件
        }
    };

    mScanner = new Scanner(); // 创建一个
Scanner对象
}

```

WifiSettings创建了一个广播接收对象，该对象关注的广播事件较多，其中四个重要的广播事件如下。

- WIFI\_STATE\_CHANGED\_ACTION：该广播事件反映了Wi-Fi功能所对应的状态，这些状态是WIFI\_STATE\_DISABLED（Wi-Fi功能已被关闭）、WIFI\_STATE\_DISABLING（Wi-Fi功能正在关闭中）、WIFI\_STATE\_ENABLED（Wi-Fi功能已被打开）、WIFI\_STATE\_ENABLING（Wi-Fi功能正在打开中）和WIFI\_STATE\_UNKNOWN（Wi-Fi功能状态未知）。
- SCAN\_RESULTS\_AVAILABLE\_ACTION：该广播事件表示无线网络扫描完毕，可以从WPAS中获取扫描结果。
- SUPPLICANT\_STATE\_CHANGED\_ACTION：该广播事件用于表示WPAS的状态发生了变化。它和5.2.3节中SupplicantStateTracker介绍的SupplicantState相关。
- NETWORK\_STATE\_CHANGED\_ACTION：该广播事件用于表示Wi-Fi连接状态发生变化。其携带的信息一般是一个NetworkInfo对象。

NetworkInfo类在前文代码注释中曾介绍过，它用于表达一个网络接口的状态（Describes the status of a network interface）。

提示 以后碰到实际代码时再来分析上述广播事件的具体处理方法。

在图5-5中左图所示的UI初始化过程中，WifiSettings的onActivityCreated函数将被调用，代码如下所示。

[-->WifiSettings.java: : onActivityCreated]

```
public void onActivityCreated(Bundle savedInstanceState) {  
    super.onActivityCreated(savedInstanceState);  
    mP2pSupported =  
  
    PackageManager().hasSystemFeature(PackageManager.FEATURE_WIFI_DIRECT);  
    mWifiManager = (WifiManager)  
    getSystemService(Context.WIFI_SERVICE);  
    ....  
    Switch actionBarSwitch = new Switch(activity); // 创建Wi-Fi  
    // 开关按钮对应的Switch对象  
    mWifiEnabler = new WifiEnabler(activity,  
    actionBarSwitch); // 创建WifiEnabler对象  
    ....  
}
```

在onActivityCreated函数中，Switch和WifiEnabler对象被创建。马上来看WifiEnabler的构造函数，代码如下所示。

[-->WifiEnabler.java: : WifiEnabler]

```
public WifiEnabler(Context context, Switch switch_) {  
    mContext = context; mSwitch = switch_;  
    mWifiManager = (WifiManager)  
    context.getSystemService(Context.WIFI_SERVICE);  
    // WifiEnabler关注如下三个广播事件  
    mIntentFilter = new  
    IntentFilter(WifiManager.WIFI_STATE_CHANGED_ACTION);  
  
    mIntentFilter.addAction(WifiManager.SUPPLICANT_STATE_CHANGED_ACTION);  
  
    mIntentFilter.addAction(WifiManager.NETWORK_STATE_CHANGED_ACTION)
```

```
N) ;  
}
```

WifiEnabler和WifiSettings设置的广播接收对象在WifiSettings中的onResume函数中被注册，相关代码如下所示。

[-->WifiSettings.java: : onResume]

```
public void onResume() {  
    super.onResume();  
    if (mWifiEnabler != null)    mWifiEnabler.resume(); // 先调用  
    WifiEnabler的resume函数  
    getActivity().registerReceiver(mReceiver, mFilter); // 注册广  
    播接收对象  
    ....  
}
```

[-->WifiEnabler.java: : resume]

```
public void resume() {  
    mContext.registerReceiver(mReceiver, mIntentFilter);  
    mSwitch.setOnCheckedChangeListener(this);  
}
```

由于WifiEnabler先注册广播接收对象，所以对于WIFI\_STATE\_CHANGED\_ACTION、SUPPLICANT\_STATE\_CHANGED\_ACTION和NETWORK\_STATE\_CHANGED\_ACTION广播来说，WifiEnabler的广播接收对象将先被触发以处理这些消息，然后才是WifiSettings的广播接收对象①。

**提示** 研究WifiEnabler广播接收对象的代码后，会发现实际上它真正处理的只有WIFI\_STATE\_CHANGED\_ACTION广播。WifiEnabler将根据该广播的信息以更新Switch的界面。

广播虽然是Android平台中特有的信息发布机制，但广播派发的过程却并不轻松。从运行效率角度考虑，一个进程中，同样的广播最好用一个广播接收对象来处理。所以，笔者觉得WifiEnabler中的这个广播接收对象有些浪费。

假设用户通过图5-5左图右上角的按钮打开了Wi-Fi功能，这个点击事件会触发什么动作呢？来看下文。

## 2. 启用Wi-Fi功能

由图5-6可知，WifiEnabler实现了CompoundButton的onCheckedChangeListener接口，故用户点击事件将触发WifiEnabler的onCheckedChanged函数。

[-->WifiEnabler.java: : onCheckedChanged]

```
public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {  
    .....  
    // 调用WifiManager的setWifiEnabled函数  
    if (mWifiManager.setWifiEnabled(isChecked))  
        mSwitch.setEnabled(false);  
    .....  
}
```

WifiManager的setWifiEnabled函数将触发WifiService开展一系列的动作。这些动作的细节内容我们留待下文分析。在WifiService开展这一系列的动作的过程中，它会通过发送广播的方式向外界发布一些信息。所以，只要关注WifiSettings和WifiEnabler如何处理这些广播事件即可。

根据前文所述，WifiEnabler注册的广播事件对象没有做什么有意义的事情，所以下面直接来看WifiSettings的广播接收对象。

[-->WifiSettings.java: : handleEvent]

```
private void handleEvent(Context context, Intent intent) {  
    String action = intent.getAction();  
    if (WifiManager.WIFI_STATE_CHANGED_ACTION.equals(action))  
    {  
        updateWifiState(intent.getIntExtra(WifiManager.EXTRA_WIFI_STATE  
        ,  
        WifiManager.WIFI_STATE_UNKNOWN));  
    } else if  
(WifiManager.SCAN_RESULTS_AVAILABLE_ACTION.equals(action) ||  
WifiManager.CONFIGURED_NETWORKS_CHANGED_ACTION.equals(action)  
||  
WifiManager.LINK_CONFIGURATION_CHANGED_ACTION.equals(action)) {
```

```

        updateAccessPoints(); // 更新图5-5左图中的无线网络列表
    } else if
(WifiManager.SUPPLICANT_STATE_CHANGED_ACTION.equals(action)) {
    SuplicantState state = (SuplicantState)
intent.getParcelableExtra(
        WifiManager.EXTRA_NEW_STATE);
    // 只在SuplicantState处于握手阶段才调用
updateConnectionState
    if (!mConnected.get() &&
SuplicantState.isHandshakeState(state))

updateConnectionState(WifiInfo.getDetailedStateOf(state));
} else if
(WifiManager.NETWORK_STATE_CHANGED_ACTION.equals(action)) {
    NetworkInfo info = (NetworkInfo)
intent.getParcelableExtra(
        WifiManager.EXTRA_NETWORK_INFO);
    mConnected.set(info.isConnected());
    changeNextButtonState(info.isConnected());
    updateAccessPoints();
    updateConnectionState(info.getDetailedState());
    .....
} else if (WifiManager.RSSI_CHANGED_ACTION.equals(action))
{
    updateConnectionState(null);
}
}

```

### (1) 触发扫描

根据前面对几个广播信息的描述，当Wi-Fi功能被启用时，将收到WIFI\_STATE\_CHANGED\_ACTION广播，而该广播的处理函数是updateWifiState。代码如下所示。

[-->WifiSettings.java: : updateWifiState]

```

private void updateWifiState(int state) {
    .....
    switch (state) {
        case WifiManager.WIFI_STATE_ENABLED:
            mScanner.resume(); // 启动扫描
            return;
    .....
}

```

```

    .....
}

[-->WifiSettings.java::Scanner]

private class Scanner extends Handler {
    private int mRetry = 0;
    void resume() {
        if (!hasMessages(0))      sendEmptyMessage(0);
    }
    .....
    public void handleMessage(Message message) {
        if (mWifiManager.startScanActive()) mRetry = 0;// 发起扫描
        else if (++mRetry >= 3) .....// 扫描失败

        sendEmptyMessageDelayed(0, WIFI_RESCAN_INTERVAL_MS);//
每1秒发起一次扫描
    }
}

```

请读者暂时记住WifiManager的startScanActive函数，下文再详细分析。

## (2) 更新AP列表

假设WPAS扫描完毕，则WifiSettings将收到  
SCAN\_RESULTS\_AVAILABLE\_ACTION广播，该广播的处理函数为  
updateAccessPoints。

[-->WifiSettings.java: : updateAccessPoints]

```

private void updateAccessPoints() {
    if (getActivity() == null) return;
    final int wifiState = mWifiManager.getWifiState(); // 获取Wi-Fi的状态

    switch (wifiState) {
        case WifiManager.WIFI_STATE_ENABLED:
            // 创建AP列表
            final Collection<AccessPoint> accessPoints =
constructAccessPoints();
            getPreferenceScreen().removeAll();
            if (accessPoints.size() == 0)

```

```

        addMessagePreference(R.string.wifi_empty_list_wifi_on);
        for (AccessPoint accessPoint : accessPoints) { // 添加到UI中显示
            getPreferenceScreen().addPreference(accessPoint);
        }
        break;
        .....
    }
}

```

来看上述代码中的constructAccessPoints函数，代码如下所示。

[-->WifiSettings.java: : constructAccessPoints]

```

private List<AccessPoint> constructAccessPoints() {
    ArrayList<AccessPoint> accessPoints = new
    ArrayList<AccessPoint>();
    Multimap<String, AccessPoint> apMap = new Multimap<String,
    AccessPoint>();

    // getConfiguredNetworks将从WPAS中读取wpa_supplicant.conf中保
    存的那些无线网络信息
    final List<WifiConfiguration> configs =
    mWifiManager.getConfiguredNetworks();
    if (configs != null) {
        for (WifiConfiguration config : configs) {
            // 为每一个已经保存的无线网络创建一个新AccessPoint对象
            AccessPoint accessPoint = new
            AccessPoint(getApplicationContext(), config);
            accessPoint.update(mLastInfo, mLastState);
            accessPoints.add(accessPoint);
            apMap.put(accessPoint.ssid, accessPoint);
        }
    }
    // 获取扫描结果。本章不讨论getScanResult函数
    final List<ScanResult> results =
    mWifiManager.getScanResults();
    if (results != null) {
        for (ScanResult result : results) { // 略过那些没有
            SSID的AP或者IBSS网络
            if (result.SSID == null || result.SSID.length()
            == 0 ||
                result.capabilities.contains("[IBSS]"))
            continue;
        }
    }
}

```

```

        boolean found = false;
        // 如果之前保存的无线网络包含在此次扫描结果中，则更新该
        // 无线网络的一些信息
        for (AccessPoint accessPoint :
apMap.getAll(result.SSID))
            if (accessPoint.update(result))    found =
true;

        if (!found) {
            // 比较扫描结果和之前保存的无线网络信息，如果是新发现的
            AP，则创建一个AP对象
            AccessPoint accessPoint = new
AccessPoint(getActivity(), result);
            accessPoints.add(accessPoint);
            apMap.put(accessPoint.ssid, accessPoint);
        }
    }
}
Collections.sort(accessPoints);
return accessPoints;
}

```

### (3) 加入目标无线网络

当WifiSettings界面显示出周围的无线网络后，用户下一步要做的就是从列表中选择加入某个无线网络。其中，处理用户选择AP事件的函数为onPreferenceTreeClick，代码如下所示。

```

[-->WifiSettings.java: : onPreferenceTreeClick]

public boolean onPreferenceTreeClick(PreferenceScreen screen,
Preference preference) {
    if (preference instanceof AccessPoint) {
        mSelectedAccessPoint = (AccessPoint) preference;
        if (.....// 对于没有安全设置的无线网络，直接连接它即可) {
            mSelectedAccessPoint.generateOpenNetworkConfig();

mWifiManager.connect(mSelectedAccessPoint.getConfig(),
mConnectListener);
        } else
            showDialog(mSelectedAccessPoint, false); // 弹出
图5-5右图所示的对话框
    }.....
    return true;
}

```

```
// showDialog代码
private void showDialog(AccessPoint accessPoint, boolean edit)
{
    .....
    mDlgAccessPoint = accessPoint;
    mDlgEdit = edit;
    showDialog(WIFI_DIALOG_ID); // showDialog将创建一个
    wifiDialog对象
}
```

由于WifiDialog及配置控制类WifiConfigController比较简单，故此处不讨论它们。

当用户设置完目标无线网络的信息（例如输入密码）后，点击图5-5右图所示对话框的“连接”按钮。此动作将触发WifiSettings的submit函数被调用，代码如下所示。

[-->WifiSettings.java: : submit]

```
void submit(WifiConfigController configController) {
    final WifiConfiguration config =
configController.getConfig();
    if (config == null) {
        .....
    } .....// 其他处理
    } else {
        if (configController.isEdit() ||
requireKeyStore(config))
            mWifiManager.save(config, mSaveListener);
        else    mWifiManager.connect(config,
mConnectListener); // 连接目标无线网络
    }
    .....
}
```

至此，WifiSettings的工作就告一段落，它的后续工作就是等待并处理广播事件。如果一切顺利，它将接收一个NETWORK\_STATE\_CHANGED\_ACTION广播事件以告知手机成功已经加入目标无线网络。

### 3. Settings操作Wi-Fi知识总结

从本质上来说，WifiSettings的内容其实并不复杂。但根据笔者的经验，初学者并不能很快把握WifiSettings的工作流程。原因有如下几点。

自从Settings UI中引入Fragment以来，整个Settings的代码比之前要复杂得多。对UI不熟悉的读者可先阅读SDK文档中关于Fragment的介绍。

WifiSettings和WifiEnabler中的广播接收对象互相干扰。这两个广播接收对象会处理一些相同的广播。并且这些广播将先由WifiEnabler处理，然后再由WifiSettings处理。不论是调试还是分析源码，初学者需要在两个类之间来回切换以研究它们是如何处理广播消息的，使得精力很容易被分散。不过，结合前文的介绍，读者完全可以忽略WifiEnabler中的广播接收对象。

在本节所述的工作流程中，WifiSettings会接收到很多次广播。这些广播由WifiService及相关模块发送。虽然发送广播以及反馈信息是一种正确的做法，但也应该控制广播发送的次数。以WifiSettings中处理SUPPLICANT\_STATE\_CHANGED\_ACTION广播为例，只有在网络未连接及SupplicantState处于握手阶段时它才做一些有意义的事情。结合前文对SupplicantState状态的介绍，它一共有13个状态，减去其中没有地方使用的3个状态（DORMANT、UNINITIALIZED、INVALID，参考5.2.3节关于WifiMonitor的介绍），WifiSetting将接收到最多10次（实际过程中还要减去一些没有被触发的状态）  
SUPPLICANT\_STATE\_CHANGED\_ACTION广播。这10次广播中，又有多少需要真正被处理呢（即处于未连接状态并且SupplicantState处于握手阶段）？

另外，笔者提炼了上述代码中WifiSettings和WifiManager交互的几个重要函数以作为下一节的分析重点，这些函数如下。

- setWifiEnabled: 启用Wi-Fi功能。
- startScanActive: 启动AP扫描。
- connect: 连接至目标AP。

① 对Android广播机制实现原理感兴趣的读者不妨阅读《深入理解Android：卷II》6.4节。

### 5.3.2 WifiService操作Wi-Fi分析

本节将围绕setWifiEnabled、startScanActive和connect函数来介绍WifiService的工作流程。先来看setWifiEnabled函数。

#### 1. setWifiEnabled函数分析

WifiService的setWifiEnabled函数将会调用WifiStateMachine的setWifiEnabled，故此处直接来看。

[-->WifiStateMachine.java: : setWifiEnabled]

```
public void setWifiEnabled(boolean enable) {
    mLastEnableUid.set(Binder.getCallingUid());
    if (enable) { // 发送两条消息
        sendMessage(obtainMessage(CMD_LOAD_DRIVER,
WIFI_STATE_ENABLING, 0));
        sendMessage(CMD_START_SUPPLICANT);
    } else {
        sendMessage(CMD_STOP_SUPPLICANT);
        sendMessage(obtainMessage(CMD_UNLOAD_DRIVER,
WIFI_STATE_DISABLED, 0));
    }
}
```

其中，CMD\_LOAD\_DRIVER和CMD\_START\_SUPPLICANT消息将交由WifiStateMachine来处理。由于WifiStateMachine此时还处于DriverUnloaded状态，DriverUnloaded的函数processMessage将被调用。

##### (1) CMD\_LOAD\_DRIVER处理流程

先来看它对CMD\_LOAD\_DRIVER的处理，相关代码如下所示。

[-->WifiStateMachine.java: : DriverUnloaded: processMessage]

```
public boolean processMessage(Message message) {
    switch (message.what) {
        case CMD_LOAD_DRIVER:
            transitionTo(mDriverLoadingState); // 转到
```

```

        DriverLoadingState
            break;
        default:  return NOT_HANDLED;
    }
    return HANDLED;
}

```

提示 由于篇幅原因，本章不讨论状态切换过程中所涉及的各状态的exit函数。

先执行DriverLoadingState的enter函数，代码如下所示。

[-->WifiStateMachine.java: : DriverLoadingState: enter]

```

class DriverLoadingState extends State {
    public void enter() {
        final Message message = new Message();
        message.copyFrom(getCurrentMessage());
        // 复制当前消息，即上面的CMD_LOAD_DRIVER消息
        new Thread(new Runnable() { // 单独启动一个线程来加载wlan驱动
            public void run() {
                mWakeLock.acquire();
                switch(message.arg1) {
                    case WIFI_STATE_ENABLING://
                        // CMD_LOAD_DRIVER携带了此信息
                        // 该函数内部将发送
                        // WIFI_STATE_CHANGED_ACTION广播
                        setWifiState(WIFI_STATE_ENABLING);
                        break;
                        .....
                }
                // 加载wlan驱动，如果成功则发送
                // CMD_LOAD_DRIVER_SUCCESS消息
                if(mWifiNative.loadDriver())
                    sendMessage(CMD_LOAD_DRIVER_SUCCESS);
                else .....// 失败的处理
                mWakeLock.release();
            }
        }).start();
    }
}

```

由上述代码可知CMD\_LOAD\_DRIVER消息的处理流程如下。

- DriverUnloaded状态直接切换到DriverLoading状态。
- DriverLoading的enter函数中将创建一个工作线程来加载wlan driver。如果成功，它将发送CMD\_LOAD\_DRIVER\_SUCCESS消息。

WifiNative的loadDriver将借助JNI调用以触发wifi.c中的wifi\_load\_driver函数被调用，其代码如下所示。

[-->Wifi.c: : wifi\_load\_driver]

```
int wifi_load_driver()
{
/*
该宏定义了wlan driver的文件路径名。在AOSP代码中，没有地方定义该宏。不过
Galaxy Note2
对应的driver文件路径是“/lib/modules/dhd.ko”。
*/
#ifndef WIFI_DRIVER_MODULE_PATH
    char driver_status[PROPERTY_VALUE_MAX];
    int count = 100;
    if (is_wifi_driver_loaded())           return 0;

    /*
    DRIVER_MODULE_PATH变量保存了WIFI_DRIVER_MODULE_PATH宏定义的文件
    路径名。
    如果上面那个宏定义了，此处将通过insmod向内核添加wlan driver。
    */
    if (insmod(DRIVER_MODULE_PATH, DRIVER_MODULE_ARG) < 0)
return -1;

    /*
    FIRMWARE_LOADER变量指向WIFI_FIRMWARE_LOADER宏定义的wlan固件加载
    程序文件路径名
    DRIVER_PROP_NAME的值为“wlan.driver.status”。如果没有指定wlan固
    件加载程序，
    则直接设置“wlan.driver.status”属性值为“ok”，否则通
    过“ctrl.start”方式来启动wlan
    固件加载程序。
    */
    if (strcmp(FIRMWARE_LOADER, "") == 0)
property_set(DRIVER_PROP_NAME, "ok");
    else property_set("ctrl.start", FIRMWARE_LOADER);

    sched_yield();
}
```

```

        while (count-- > 0) { // 判断wlan driver是否加载成功
            if (property_get(DRIVER_PROP_NAME, driver_status,
NULL)) {
                if (strcmp(driver_status, "ok") == 0) return 0;
                else if (strcmp(DRIVER_PROP_NAME, "failed") == 0) {
                    wifi_unload_driver();
                    return -1;
                }
            }
            usleep(200000);
        }
        property_set(DRIVER_PROP_NAME, "timeout");
        wifi_unload_driver();
        return -1;
    } else // 如果没有定义WIFI_DRIVER_MODULE_PATH宏，则直接设置“wlan.driver.status”属性值为“ok”
    {
        property_set(DRIVER_PROP_NAME, "ok");
        return 0;
    }
#endif
}

```

## (2) CMD\_LOAD\_DRIVER\_SUCCESS处理流程

下面来看DriverLoadingState是如何处理CMD\_LOAD\_DRIVER\_SUCCESS消息的。

[-->WifiStateMachine.java: : DriverLoadingState:  
processMessage]

```

public boolean processMessage(Message message) {
    switch (message.what) {
        case CMD_LOAD_DRIVER_SUCCESS:
            transitionTo(mDriverLoadedState); // 转到
DriverLoadedState
            break;
        case CMD_LOAD_DRIVER_FAILURE:
            transitionTo(mDriverFailedState);
            break;
        .....
        case CMD_START_SUPPLICANT:// DriverLoadingState不处理
此消息
        case .....// 其他消息
            deferMessage(message);
            // CMD_START_SUPPLICANT消息将放到下一个状态中再去处
理
    }
}

```

```

        break;
    default:
        return NOT_HANDLED;
    }
    return HANDLED;
}
}

```

由上述代码可知，DriverLoadingState不处理CMD\_START\_SUPPLICANT消息，而是将其推迟到下一个状态中再去处理。对于CMD\_LOAD\_DRIVER\_SUCCESS，直接转到DriverLoadedState。DriverLoadedState的enter函数仅仅打印一句简单的日志输出，而它对CMD\_START\_SUPPLICANT的处理却比较复杂。

### (3) CMD\_START\_SUPPLICANT处理流程

CMD\_START\_SUPPLICANT消息将在DriverLoaded状态中得到处理，相关代码如下所示。

```
[-->WifiStateMachine.java: : DriverLoadedState:
processMessage]

public boolean processMessage(Message message) {
    switch(message.what) {
        .....
        case CMD_START_SUPPLICANT:
            try { // 加载wlan固件。使用了netd的SoftAp命令，可参考
2.3.8节
                mNwService.wifiFirmwareReload(mInterfaceName, "STA");
            } .....
            try { // 下面这两个函数对应netd的InterfaceCmd命令。可
参考2.3.3节
                mNwService.setInterfaceDown(mInterfaceName);
                mNwService.setInterfaceIpv6PrivacyExtensions
                    (mInterfaceName, true);
            } .....
            // 启动wpa_supplicant进程。请回顾5.2.3节中
WifiNative介绍
            if (mWifiNative.startSupplicant(mP2pSupported)) {
                mWifiMonitor.startMonitoring(); // 启动
            }
    }
}
```

```

WifiMonitor的Monitor线程
    transitionTo(mSupplicantStartingState);
    // 转到SupplicantStartingState
}.....
break;
case CMD_START_AP:
    .....
    default:
        return NOT_HANDLED;
}
return HANDLED;
}

```

由上述代码可知DriverLoadedState处理CMD\_START\_SUPPLICANT消息的结果。

- wlan固件被加载。
- wpa\_supplicant进程被创建，并且WifiService通过WifiMonitor和它建立了交互关系。
- WifiStateMachine状态切换至SupplicantStartingState。该状态的enter函数没有开展有意义的工作。

当WifiMonitor成功连接至WPAS进程后，它将发送SUP\_CONNECTION\_EVENT消息给WifiStateMachine（参考5.2.3节中关于WifiMonitor的介绍）。下面就来看该消息的处理流程。

#### (4) SUP\_CONNECTION\_EVENT处理流程

SUP\_CONNECTION\_EVENT在SupplicantStartingState状态中得到处理，相关代码如下所示。

[-->WifiStateMachine.java: : SupplicantStartingState:  
processMessage]

```

public boolean processMessage(Message message) {
    switch(message.what) {
        case WifiMonitor.SUP_CONNECTION_EVENT:
            setWifiState(WIFI_STATE_ENABLED); // 发送
WIFI_STATE_CHANGED_ACTION广播
            mSupplicantRestartCount = 0;
    }
}

```

```

        // 发送消息给SupplicantStateTracker状态机。请读者自行研究
mSupplicantStateTracker.sendMessage(CMD_RESET_SUPPLICANT_STATE)
;
    mLastBssid = null;  mLastNetworkId =
WifiConfiguration.INVALID_NETWORK_ID;
    mLastSignalLevel = -1;
    // 设置本机IP地址

mWifiInfo.setMacAddress(mWifiNative.getMacAddress());
mWifiConfigStore.initialize();

initializeWpsDetails();
// 初始化和WPS相关的一些内容。本章将略过和WPS/P2P相关的内容

        // 发送SUPPLICANT_CONNECTION_CHANGE_ACTION广播
sendSupplicantConnectionChangedBroadcast(true);

        transitionTo(mDriverStartedState); // 转到
DriverStartedState
    break;
    .....
}
return HANDLED;
}

```

结合HSM知识以及图5-4中WifiStateMachine中各个状态的层级关系，DriverStarted的父状态是SupplicantStarted，所以上述代码中transitionTo (mDriverStartedState) 这一句函数调用将导致SupplicantStarted和DriverStarted的enter函数依次被调用。

首先调用的是SupplicantStarted的enter函数，相关代码如下所示。

```
[-->WifiStateMachine.java: : SupplicantStartedState: enter]

public void enter() {
    mIsScanMode = false; // 该变量的作用见下文
    mNetworkInfo.setIsAvailable(true);
    // config_wifi_supplicant_scan_interval用于控制扫描间隔，默认是
    15000毫秒
    int defaultInterval = mContext.getResources().getInteger(

```

```

R.integer.config_wifi_supplicant_scan_interval);
    mSupplicantScanIntervalMs =
Settings.Global.getLong(mContext.getContentResolver(),
Settings.Global.WIFI_SUPPLICANT_SCAN_INTERVAL_MS,defaultInterva
l);

// 向WPAS发送“SCAN_INTERVAL 扫描间隔时间”命令
mWifiNative.setScanInterval((int)mSupplicantScanIntervalMs
/ 1000);
}

```

接着来看DriverStartedState的enter函数。

[-->WifiStateMachine.java: : DriverStartedState: enter]

```

public void enter() {
    mIsRunning = true; mInDelayedStop = false;
    updateBatteryWorkSource(null);
    /*
    由于蓝牙运行在2.4GHz频率上，所以为了避免wlan和蓝牙互相干扰，下面这个
    函数将告知
    wlan driver蓝牙是否启用。如果是，wlan芯片会做适当调整。
    */

```

mWifiNative.setBluetoothCoexistenceScanMode(mBluetoothConnectio
nActive);

```

/*
下面这两个函数用设置国家码和频段。其内部是通过发送消息的方式来触发
WifiNative
    setCountryCode和setBand函数被调用。在WifiNative中，这两个函数都会
发送形如
    “DRIVER XXX”命令给WPAS。DRIVER命令是Android平台特有的，用于给wlan
driver发送一些
    定制的命令。
    AOSP源码中，
hardware/broadcom/wlan/bcmdhd/wpa_supplicant_8_lib/driver_cmd_n
180211.c

```

中的wpa\_driver\_n180211\_driver\_cmd函数可用于处理针对博通wlan
driver的“DRIVER XXX”命令。

我们在第4章中没有介绍相关的命令，不过它们难度并不大。请读者以上述
driver\_cmd\_n180211.c

为参考文件，自行分析相关的DRIVER命令。

```

        */
        setCountryCode(); setFrequencyBand();

        setNetworkDetailedState(DetailedState.DISCONNECTED);

        // 下面三个函数都和WPAS中的“DRIVER XXX”命令有关
        mWifiNative.stopFilteringMulticastV6Packets();
        if (mFilteringMulticastV4Packets.get())
            mWifiNative.startFilteringMulticastV4Packets();
        } else mWifiNative.stopFilteringMulticastV4Packets();

        /*
        mIsScanMode默认为FALSE。该变量只能通过CMD_SET_SCAN_TYPE消息来修改。mIsScanMode
        和4.5.3节“wpa_supplicant_scan分析之一”中提到的ap_scan变量有关,
        该变量可取值如下。
        值为1: 表示WPAS来完成AP扫描和选择的绝大部分工作(包括关联、EAPOL认证等工作)。
        值为0: 表示驱动完成AP扫描和选择的工作。
        值为2: 和0类似, 不过在NDIS(Windows上的网络设备驱动)中用得较多。
        下面代码中的SCAN_ONLY_MODE对应值为2, 而CONNECT_MODE对应值为1。
        */
        if (mIsScanMode) {
            mWifiNative.setScanResultHandling(SCAN_ONLY_MODE);
            mWifiNative.disconnect();
            transitionTo(mScanModeState);
        } else {
            mWifiNative.setScanResultHandling(CONNECT_MODE);
            mWifiNative.reconnect(); // 发送“RECONNECT”命令给WPAS
            mWifiNative.status(); // 发送“STATUS”命令给WPAS
            transitionTo(mDisconnectedState); // 进入
            DisconnectedState
        }

        if (mScreenBroadcastReceived.get() == false) {
            PowerManager powerManager =
            (PowerManager) mContext.getSystemService(
                Context.POWER_SERVICE);
            handleScreenStateChanged(powerManager.isScreenOn());
        } else {
            // 发送“DRIVER SETSUSPENDMODE”命令。该命令由Driver厂商提供的库来实现
            mWifiNative.setSuspendOptimizations(mSuspendOptNeedsDisabled ==
            0
                && mUserWantsSuspendOpt.get());
        }
    }
}

```

```

    }
    mWifiNative.setPowerSave(true); // 和P2P PowerSave有关。本章不
讨论
    // 如果支持P2P，则通过mWifiP2pChannel向WifiP2p模块发送消息
    if (mP2pSupported)
mWifiP2pChannel.sendMessage(WifiStateMachine.CMD_ENABLE_P2P);
}

```

上述代码执行完后，WifiStateMachine将转入DisconnectedState。由于DisconnectedState的父状态是ConnectModeState，它的enter函数没有做任何有意义的工作，所以此处只介绍DisconnectedState的enter函数。

[-->WifiStateMachine.java: : DisconnectedState: enter]

```

public void enter() {
    // 下面这段代码和P2P有关
    if (mTemporarilyDisconnectWifi) {

mWifiP2pChannel.sendMessage(WifiP2pService.DISCONNECT_WIFI_RESP
ONSE);
        return;
    }

    mFrameworkScanIntervalMs =
Settings.Global.getLong(mContext.getContentResolver(), 

Settings.Global.WIFI_FRAMEWORK_SCAN_INTERVAL_MS,
        mDefaultFrameworkScanIntervalMs);

/*
当系统支持后台扫描时，如果手机屏幕关闭，则设置mEnableBackgroundScan为
true以启动后台扫描。
mScanResultIsPending用于表示WifiService是否在等待扫描请求的结果。
由于启动后台扫描的时候
会先取消上一次的扫描请求，所以如果mScanResultIsPending为true的话，
则先不启用后台扫描。
*/
    if (mEnableBackgroundScan) {
        if (!mScanResultIsPending)
mWifiNative.enableBackgroundScan(true);
        else { // 设置定时扫描任务。到时间后，AlarmManager将发送一
个"ACTION_START_SCAN"Intent
            // 而WifiStateMachine对该Intent的处理就是调用startScan函数

```

```

        setScanAlarm(true);
    }
/*
如果当前没有P2P连接，并且没有之前保存的AP信息，则发送
CMD_NO_NETWORKS_PERIODIC_SCAN消息
以触发扫描。
*/
if (!mP2pConnected.get() &&
mWifiConfigStore.getConfiguredNetworks().size() == 0)

sendMessageDelayed(obtainMessage(CMD_NO_NETWORKS_PERIODIC_SCAN,
++mPeriodicScanToken, 0),
mSupplicantScanIntervalMs);
}

```

### (5) setWifiEnabled流程总结

笔者初次接触setWifiEnabled函数的流程时，心中的感觉是“一个函数引发的一连串血案”。确实，WifiStateMachine的设计自有独到之处，但是否有些过于复杂了呢？

没找到一种合适的方法用图来描述整个流程，只能用以下文字来描述。

- 1) WifiService的setWifiEnabled函数将调用WifiStateMachine中的同名函数。在WifiStateMachine中，CMD\_LOAD\_DRIVER和CMD\_START\_SUPPLICANT两个消息将发送给状态机去执行。WifiStateMachine最初的状态是DriverUnloadedState。
- 2) DriverUnloadedState接收到CMD\_LOAD\_DRIVER消息后将转入DriverLoadingState。而DriverLoadingState的enter函数将创建一个工作线程来执行WifiNative的loadDriver以加载Wlan驱动。如果driver加载成功，该线程会发送CMD\_LOAD\_DRIVER\_SUCCESS消息给状态机。
- 3) DriverLoadingState将在其processMessage中处理CMD\_START\_SUPPLICANT和CMD\_LOAD\_DRIVER\_SUCCESS消息。其中，DriverLoadingState会延迟对CMD\_START\_SUPPLICANT的处理。而对于CMD\_LOAD\_DRIVER\_SUCCESS，DriverLoadingState将直接转入DriverLoadedState。

4) DriverLoadedState将继续处理CMD\_START\_SUPPLICANT。在其processMessage中，wpa\_supplicant进程将被启动，并且WifiMonitor将建立WifiService和WPAS的关系。同时，状态机将转入SupplicantStartingState。另外，当WifiMonitor成功连接上WPAS后，它将发送一个SUP\_CONNECTION\_EVENT消息。

5) SupplicantStartingState将处理SUP\_CONNECTION\_EVENT消息。这些处理包括设置初始化WPS相关的信息、设置WifiState、发送消息给SupplicantStateTracker状态机、初始化WifiConfigStore等。最后，SupplicantStartingState将转入DriverStartedState。DriverStartedState的父状态是SupplicantStartedState。所以这两个状态的enter函数均会被调用。

6) SupplicantStartedState在其enter函数中将设置扫描间隔。而DriverStartedState在其enter函数中将完成诸如Country Code、Frequency Band、Bluetooth共存模式等设置工作。有些工作需要发送形如"DRIVER XXX"的命令给WPAS。最后，SupplicantStartedState将转入DisconnectedState。

7) DisconnectedState的enter函数将被执行（其父状态ConnectModeState的enter函数没有完成什么实质性的工作）。该函数主要完成了后台扫描及定时扫描工作的一些设置。

接着来看第二个关键函数startScanActive。

## 2. startScanActive函数分析

startScanActive定义在WifiManager中，它将调用WifiService的startScan函数，而WifiService又会调用WifiStateMachine的startScan，所以本节直接从WifiStateMachine开始。

[-->WifiStateMachine.java: : startScan]

```
public void startScan(boolean forceActive) {
    // 对于startScanActive来说，forceActive的值为true
    sendMessage(obtainMessage(CMD_START_SCAN, forceActive ?
        SCAN_ACTIVE : SCAN_PASSIVE, 0));
}
```

## (1) CMD\_START\_SCAN处理流程

WifiStateMachine当前处于DisconnectedState，故其processMessage函数将被调用以处理CMD\_START\_SCAN消息。相关代码如下所示。

```
[-->WifiStateMachine.java: : DisconnectedState:  
processMessage]  
  
public boolean processMessage(Message message) {  
    boolean ret = HANDLED;  
    switch (message.what) {  
        ....  
        case CMD_START_SCAN:  
            // 取消后台扫描  
            if (mEnableBackgroundScan)  
                mWifiNative.enableBackgroundScan(false);  
            ret = NOT_HANDLED; // 注意返回值  
            break;  
        case WifiMonitor.SCAN_RESULTS_EVENT:// 扫描完毕后将收到此消息  
            if (mEnableBackgroundScan && mScanResultIsPending)  
                mWifiNative.enableBackgroundScan(true);  
            ret = NOT_HANDLED;// 注意返回值  
            break;  
        ....  
    }  
    return ret;  
}
```

上述代码重点展示了CMD\_START\_SCAN和SCAN\_RESULTS\_EVENT消息的处理情况。可知DisconnectedState都将返回NOT\_HANDLED。如此，其父状态的processMessage将被调用。沿着图5-4 WifiStateMachine各状态层次关系图并结合代码，最终DisconnectedState的祖父DriverStartedState将被触发，相关代码如下所示。

```
[-->WifiStateMachine.java: : DriverStartedState:  
processMessage]  
  
public boolean processMessage(Message message) {  
    switch (message.what) {  
        ....  
        case CMD_START_SCAN:  
            boolean forceActive = (message.arg1 ==
```

```

SCAN_ACTIVE);
        // 对主动扫描 (Active Scan) 来说, setScanMode将发送“DRIVER SCAN-ACTIVE”命令
        if (forceActive && !mSetScanActive)
            mWifiNative.setScanMode(forceActive);
            mWifiNative.scan(); // 发送“SCAN”命令给WPAS以触发扫描

        if (forceActive && !mSetScanActive)mWifiNative
            .setScanMode(mSetScanActive);

        mScanResultIsPending = true;// 设置
mScanResultIsPending为true
        break;
        .....
    }
    return HANDLED;
}

```

当WPAS扫描完毕后，它将通知WifiMonitor。而WifiMonitor的handleEvent函数将向WifiStateMachine发送SCAN\_RESULTS\_EVENT消息（请参考5.2.3节介绍的WifiMonitor中的handleEvent函数）。下面来看该消息的处理流程。

## (2) SCAN\_RESULTS\_EVENT处理流程

WifiStateMachine的状态是DisconnectedState，由上一节对该状态processMessage函数的介绍可知，对于SCAN\_RESULTS\_EVENT消息，DisconnectedState将返回NOT\_HANDLED。所以其父状态ConnectModeState将接着处理此消息。

[-->WifiStateMachine.java: : ConnectModeState:  
processMessage]

```

public boolean processMessage(Message message) {
    .....
    switch(message.what) {
        .....
        case WifiMonitor.SCAN_RESULTS_EVENT:
            mWifiNative.setScanResultHandling(CONNECT_MODE); // 设置
ap_scan
            return NOT_HANDLED; // 仍然返回NOT_HANDLED
        .....
    }
}

```

```
    return HANDLED;  
}
```

返回值NOT\_HANDLED将导致ConnectModeState的父状态DriverStartedState process Message被调用，不过可惜的是DriverStartedState压根就不处理该消息。所以还得来看DriverStartedState的父状态SupplicantStartedState。相关代码如下所示。

[-->WifiStateMachine.java: : SupplicantStartedState: processMessage]

```
public boolean processMessage(Message message) {  
    .....  
    switch(message.what) {  
        .....  
        case WifiMonitor.SCAN_RESULTS_EVENT:  
            // 从WPAS中获取扫描结果，并保存到mScanResults变量中  
            setScanResults(mWifiNative.scanResults());  
            // 发送SCAN_RESULTS_AVAILABLE_ACTION广播  
            sendScanResultsAvailableBroadcast();  
            mScanResultIsPending = false;  
            break;  
        .....  
    }  
    return HANDLED;  
}
```

和setWifiEnabled函数相比，startScanActive涉及的代码比较简单。而且，与该流程相关状态主要是DisconnectedState以及其祖先状态。

下面来看最后一个函数即connect的处理流程。

### 3. connect函数分析

从WifiManager开始，相关代码如下所示。

[-->WifiManager.java: : connect]

```
public void connect(WifiConfiguration config, ActionListener  
listener) {  
    .....// 参数检查
```

```

// 通过AsyncChannel向WifiService发送消息
message的第二个参数为INVALID_NETWORK_ID，其值为-1
mAsyncChannel.sendMessage(CONNECT_NETWORK,
WifiConfiguration.INVALID_NETWORK_ID,
putListener(listener), config);
}

```

WifiService接收到CONNECT\_NETWORK消息后，将直接把它转发给WifiStateMachine。下面来看WifiStateMachine是如何处理CONNECT\_NETWORK的。

### (1) CONNECT\_NETWORK处理流程

DisconnectedState的父状态ConnectModeState将处理CONNECT\_NETWORK消息，相关代码如下所示。

[-->WifiStateMachine.java: : ConnectModeState:  
processMessage]

```

public boolean processMessage(Message message) {
    switch(message.what) {
        .....
        case WifiManager.CONNECT_NETWORK:
            int netId = message.arg1;// netId为
INVALID_NETWORK_ID
            WifiConfiguration config = (WifiConfiguration)
message.obj;

            if (config != null) { // saveNetwork是关键函数，见下文分
析
                NetworkUpdateResult result =
mWifiConfigStore.saveNetwork(config);
                netId = result.getNetworkId();
            }
        /*

```

下面这段代码中：

selectNetwork：选择netId对应的无线网络。该函数的工作和上面的saveNetwork有些类似。

reconnect将发送“RECONNECT”命令给WPAS，而WPAS的处理就是调用

wpa\_supplicant\_req\_scan，读者可参考4.5.3节。

sendMessage：发送CONNECT\_NETWORK消息给SupplicantStateTracker。

replyToMessage: WifiStateMachine中也有一个  
AsyncChannel, 不过它没有  
连接到任何Handler。该函数将把CONNECT\_NETWORK\_SUCCEEDED  
发给  
CONNECT\_NETWORK消息的发送者（即WifiManager）。  
请读者自行研究WifiManager对CONNECT\_NETWORK\_SUCCEEDED消  
息的处理流程。

```
    */
    if (mWifiConfigStore.selectNetwork(netId) &&
mWifiNative.reconnect()) {

    mSupplicantStateTracker.sendMessage(WifiManager.CONNECT_NETWORK
);
        replyToMessage(message,
WifiManager.CONNECT_NETWORK_SUCCEEDED);
        // 切换到DisconnectingState
        // 考虑到手机之前可能连接到其他AP, 所以此处要先进入
DisconnectingState
        transitionTo(mDisconnectingState);
    } else {.....// 失败处理}
    break;
    .....
}
return HANDLED;
}
```

上述代码中, saveNetwork比较关键, 其代码如下所示。

[-->WifiConfigStore.java: : saveNetwork]

```
NetworkUpdateResult saveNetwork(WifiConfiguration config) {
/*
    如果networkId为-1, 并且该无线网络的SSID为空, 则不能添加无线网络。
    用户在wifiSettings选择的目标无线网络如果之前已经在
wpa_supplicant.conf文件中有信息,
    则它的networkid不为-1。
*/
    if (config == null || (config.networkId ==
INVALID_NETWORK_ID && config.SSID == null))
        return new NetworkUpdateResult(INVALID_NETWORK_ID);

    // 假设本例中该无线网络是新搜索到的, 则newNetwork为true
    boolean newNetwork = (config.networkId ==
INVALID_NETWORK_ID);
```

```

/*
    addOrUpdateNetworkNative将触发"ADD_NETWORK"、"SET_NEWTORK
id param value"等一系列
命令。这些命令和4.5节开头介绍的一样。请读者自行研究
addOrUpdateNetworkNative函数。
*/
NetworkUpdateResult result =
addOrUpdateNetworkNative(config);
int netId = result.getNetworkId();

if (newNetwork && netId != INVALID_NETWORK_ID) {
    mWifiNative.enableNetwork(netId, false); // 发
送"ENABLE_NETWORK id"给WPAS
    mConfiguredNetworks.get(netId).status =
Status.ENABLED;
}
mWifiNative.saveConfig();
// 发送"SAVE_CONFIG"命令，WPAS将保存wpa_config信息到配置文件

// 发送广播
sendConfiguredNetworksChangedBroadcast(config,
result.isNewNetwork() ?
    WifiManager.CHANGE_REASON_ADDED :
WifiManager.CHANGE_REASON_CONFIG_CHANGE);
return result;
}

```

仔细研究selectNetwork和saveNetwork的代码，感觉selectNetwork重复做了一些saveNetwork已经做过的工作，例如selectNetwork也会调用addOrUpdateNetworkNative函数，笔者觉得这段代码应该有优化余地。

另外，WPAS在"ENABLE\_NETWORK"过程中，会历经一系列复杂的过程直到加入目标无线网络（读者可回顾4.5.3节ENABLE\_NETWORK命令处理）。在这个过程中，WPAS的状态（即wpa\_sm状态机的状态）也会跟着发生变化。这些变化将触发WifiMonitor向WifiStateMachine发送SUPPLICANT\_STATE\_CHANGE\_EVENT消息。由于篇幅问题，本章不拟讨论这些消息的处理过程。感兴趣的读者不妨学完本章后再来研究它们。

当WPAS成功加入目标无线网络后，它将发送信息给WifiMonitor例如：

```
CTRL-EVENT-CONNECTED-Connection to 00: 1e: 58: ec: d5: 6d
completed (reauth) [id=1 id_str=]
```

而这个信息将触发WifiMonitor发送NETWORK\_CONNECTION\_EVENT消息给WifiStateMachine。所以此处直接来分析NETWORK\_CONNECTION\_EVENT消息的处理流程即可。

## (2) NETWORK\_CONNECTION\_EVENT消息处理流程

此时WifiStateMachine处于DisconnectingState，不过它并不处理NETWORK\_CONNECTION\_EVENT消息，所以该消息最终将由其父状态ConnectModeState处理。相关代码如下所示。

```
[-->WifiStateMachine.java: : ConnectModeState:  
processMessage]  
  
public boolean processMessage(Message message) {  
    ....  
    switch (message.what) {  
        ....  
        case WifiMonitor.NETWORK_CONNECTION_EVENT:  
            mLastNetworkId = message.arg1; // arg1指向目标AP的  
ID  
            mLastBssid = (String) message.obj; // obj指向目标  
AP的bssid  
            mWifiInfo.setBSSID(mLastBssid);  
            mWifiInfo.setNetworkId(mLastNetworkId);  
  
            setNetworkDetailedState(DetailedState.OBTAINING_IPADDR);  
  
            // 发送NETWORK_STATE_CHANGED_ACTION广播  
            sendNetworkStateChangeBroadcast(mLastBssid);  
            transitionTo(mObtainingIpState); // 进入  
ObtainingIpstate状态  
            break;  
        ....  
    }  
    return HANDLED;  
}
```

来看ObtaingIpState的enter函数（注意，ObtaingIpState的父状态是L2ConnectedState，故父状态的enter函数先执行。L2ConnectedState的enter函数比较简单，此处略），代码如下所示。

```
[-->WifiStateMachine.java: : ObtaingIpState: enter]
```

```

public void enter() {
    // 判断目标AP是否使用静态IP配置
    if (!mWifiConfigStore.isUsingStaticIp(mLastNetworkId)) {
        // 本例中的目标AP用得是动态IP配置。所以下面还要创建一个
        DhcpStateMachine对象
        if (mDhcpStateMachine == null)
            mDhcpStateMachine =
        DhcpStateMachine.makeDhcpStateMachine(
                    mContext, WifiStateMachine.this,
                    mInterfaceName);

    mDhcpStateMachine.registerForPreDhcpNotification();

        // 向DhcpStateMachine发送CMD_START_DHCP消息
    mDhcpStateMachine.sendMessage(DhcpStateMachine.CMD_START_DHCP);
    } else {.....// 静态IP的处理流程}
}

```

DhcpStateMachine是和DHCP相关的一个状态机对象，由于其内容比较简单，本章不详述。在DhcpStateMachine运行过程中，它将向WifiStateMachine发送两个消息CMD\_PRE\_DHCP\_ACTION和CMD\_POST\_DHCP\_ACTION，它们均由ObtainingIpState的父状态L2ConnectedState处理。

### (3) CMD\_PRE/POST\_DHCP\_ACTION处理流程

处理流程如下。

[-->WifiStateMachine.java: : L2ConnectedState:  
processMessage]

```

public boolean processMessage(Message message) {
    .....
    switch (message.what) {
        case DhcpStateMachine.CMD_PRE_DHCP_ACTION:
            // 处理dhcp交互之前的一些工作，例如设置蓝牙共存模式，关闭p2p
            powersave等功能等
            handlePreDhcpSetup();

            // 向DhcpStateMachine发送CMD_PRE_DHCP_ACTION_COMPLETE
            // 消息。DhcpStateMachine将
            // 启动dhcpcd进程以从AP那获取一个IP地址。如果一切顺利，它将
}

```

```

发送CMD_POST_DHCP_ACTION
    // 消息给WifiStateMachine

mDhcpStateMachine.sendMessage(DhcpStateMachine.CMD_PRE_DHCP_ACTION_COMPLETE);
    break;
case DhcpStateMachine.CMD_POST_DHCP_ACTION:
    // 和handlePreDhcpSetup相对应，恢复蓝牙共存模式及打开P2P
PowerSave功能
    handlePostDhcpSetup();
    if (message.arg1 == DhcpStateMachine.DHCP_SUCCESS)
{
    // 下面这个函数将发送
LINK_CONFIGURATION_CHANGED_ACTION广播
    // 本章不讨论和该广播相关的处理流程

handleSuccessfulIpConfiguration((DhcpInfoInternal)
message.obj);
    transitionTo(mVerifyingLinkState); // 转入
VerifyingLinkState
    } .....
    break;
.....
}
return HANDLED;
}

```

在L2ConnectedState中，WPAS其实已经连接上了AP。当收到CMD\_POST\_DHCP\_ACTION消息时，手机也从AP那得到了一个IP地址。不过，WifiService还没有完成其最终的工作，它将转入VerifyingLinkState。该状态将会和一个名为WifiWatchdogStateMachine的对象交互。

**提示** WifiWatchdogStateMachine用于监控无线网络的信号质量，5.4节将详细介绍。

先来看VeryfingLinkState的代码，如下所示。

[-->WifiStateMachine.java: : VeryfingLinkState]

```

class VerifyingLinkState extends State {
    public void enter() {

setNetworkDetailedState(DetailedState.VERIFYING_POOR_LINK);

```

```

        mWifiConfigStore.updateStatus(mLastNetworkId,
DetailedState.VERIFYING_POOR_LINK);
        sendNetworkStateChangeBroadcast(mLastBssid);
    }
    public boolean processMessage(Message message) {
        switch (message.what) {
            case WifiWatchdogStateMachine.POOR_LINK_DETECTED:
                break;
            case WifiWatchdogStateMachine.GOOD_LINK_DETECTED:
                // 如果WifiWatchdogStateMachine判断此时无线网
络的信号良好
                // 它将发送GOOD_LINK_DETECTED消息给
WifiStateMachine
                transitionTo(mCaptivePortalCheckState); ///
转入CaptivePortalCheckState
                break;
            default:
                return NOT_HANDLED;
        }
        return HANDLED;
    }
}

```

CaptivePortalCheckState是Android 4.2新引入的一个状态。CaptivePortalCheckState和一种名为Captive Portal（强制网络门户）认证方法有关。它对应如下一种应用场景：当未认证用户初次上网时，系统将强制用户打开某个特定页面，例如运营商指定的页面。在该页面中，用户必须点击“同意”按钮后才能真正使用网络。该认证方法也叫Portal认证。目前在一些公共场所（如机场、酒店）中被大量使用。关于Capitve Portal的详细信息，读者可阅读参考资料[1]。

**提示** 后面将详细介绍Android 4.2代码中对Captive Portal Check的处理流程。

下面来看CaptivePortalCheckState的代码，如下所示。

[-->WifiStateMachine.java: : CaptivePortalCheckState]

```

class CaptivePortalCheckState extends State {
    public void enter() {
        setNetworkDetailedState(DetailedState.CAPTIVE_PORTAL_CHECK);
    }
}

```

```

    // 设置DetailedState为CAPTIVE_PORTAL_CHECK
    mWifiConfigStore.updateStatus(mLastNetworkId,
DetailedState.CAPTIVE_PORTAL_CHECK);
    // 发送NETWORK_STATE_CHANGED_ACTION广播。请读者记住此处的调用
    sendNetworkStateChangeBroadcast(mLastBssid);
}

public boolean processMessage(Message message) {
    switch (message.what) {
        case CMD_CAPTIVE_CHECK_COMPLETE:
            // 检查完毕。详情见“Captive Portal Check介绍”
            try {
                mNwService.enableIpv6(mInterfaceName);
            } .....

            setNetworkDetailedState(DetailedState.CONNECTED);
            mWifiConfigStore.updateStatus(mLastNetworkId,
DetailedState.CONNECTED);
            sendNetworkStateChangeBroadcast(mLastBssid);
            transitionTo(mConnectedState); // 终于进入
ConnectedState
            break;
        default:
            return NOT_HANDLED;
    }
    return HANDLED;
}
}

```

由上述代码可知，当Captive Portal检查完毕后，  
CaptivePortalCheckState将收到CMD\_CAPTIVE\_CHECK\_COMPLETE消息。  
在该消息的处理过程中，WifiStateMachine终于转入  
ConnectedState，也就是本次旅程的终点。

ConnectedState仅处理POOR\_LINK\_DETECTE消息，相关代码比较简单，  
不赘述。

下面来总结connect的流程。

#### (4) connect流程总结

connect的流程包括如下几个关键点。

- 整个流程起源于WifiManager向WifiStateMachine发送的CONNECT\_NETWORK消息。
  - ConnectModeState将处理此消息。在其处理过程中，它将发送一系列命令给WPAS，而WPAS将完成802.11规范中所定义的身份认证、关联、四次握手等工作。ConnectModeState随之转入DisconnectingState。
  - 当WPAS加入目标无线AP后，它将发送NETWORK\_CONNECTION\_EVENT给WifiStateMachine。DisconnectingState的父状态ConnectModeState将处理此消息，具体处理过程比较简单。最终，WifiStateMachine将转入ObtaingIpState。
  - 在ObtaingIpState的enter函数中，DhcpStateMachine对象将被创建。DhcpStateMachine和dpcpcd有关。相关代码比较简单，请读者自行阅读。在WifiStateMachine和DhcpStateMachine的交互过程中，DhcpStateMachine将向WifiStateMachine发送CMD\_PRE\_DHCP\_ACTION和CMD\_POST\_DHCP\_ACTION消息。在CMD\_POST\_DHCP\_ACTION消息的处理过程中，WifiStateMachine将转入VerifyingLinkState。
  - VerifyingLinkState将和WifiWatchdogStateMachine交互。WifiWatchdogStateMachine用于监控无线网络信号的好坏。如果一切正确，它将转入CaptivePortalCheckState。
  - CaptivePortalCheckState用于检查目标无线网络提供商是否需要Captive Portal Check。如果一切正常，WifiStateMachine最终将转入ConnectedState。该状态就是本章第二条分析路线的终点。

下面介绍本章最后两个重要知识点，WifiWatchdogStateMachine和Captive Portal Check。

## 5.4 WifiWatchdogStateMachine介绍

WifiWatchdogStateMachine用于监控无线网络的信号质量，在WifiService的checkAndStartWifi函数中被创建，其创建函数是makeWifiWatchdogStateMachine，代码如下所示。

```
[-->WifiWatchdogStateMachine.java: :  
makeWifiWatchdogStateMachine]  
  
public static WifiWatchdogStateMachine  
makeWifiWatchdogStateMachine(Context context) {  
    ContentResolver contentResolver =  
    context.getContentResolver();  
  
    ConnectivityManager cm = (ConnectivityManager)  
    context.getSystemService(  
  
    Context.CONNECTIVITY_SERVICE);  
    // 判断手机是否只支持Wi-Fi。对于大部分手机来说，sWifiOnly为false  
    sWifiOnly =  
    (cm.isNetworkSupported(ConnectivityManager.TYPE_MOBILE) ==  
    false);  
    // WIFI_WATCHDOG_ON功能默认是打开的  
    putSettingsGlobalBoolean(contentResolver,  
    Settings.Global.WIFI_WATCHDOG_ON, true);  
  
    // 创建一个WifiWatchdogStateMachine对象，它也是一个HSM  
    WifiWatchdogStateMachine wwsd = new  
    WifiWatchdogStateMachine(context);  
    wwsd.start(); // 启动HSM  
    return wwsd;  
}
```

### 1. WifiWatchdogStateMachine构造函数分析

先来看WifiWatchdogStateMachine的初始化流程。

```
[-->WifiWatchdogStateMachine.java: :  
WifiWatchdogStateMachine]
```

```

private WifiWatchdogStateMachine(Context context) {
    super(TAG);
    mContext = context; mContentResolver =
    context.getContentResolver();
    mWifiManager = (WifiManager)
    context.getSystemService(Context.WIFI_SERVICE);
    // mWsmChannel用于和WifiStateMachine交互
    mWsmChannel.connectSync(mContext, getHandler(),
        mWifiManager.getWifiStateMachineMessenger());
    /*
     * 关键函数: setupNetworkReceiver将创建一个广播接收对象，用于接收
     * NETWORK_STATE_CHANGED_ACTION、
     * WIFI_STATE_CHANGED_ACTION、RSSI_CHANGED_ACTION、
     * SUPPLICANT_STATE_CHANGED_
     * ACTION等广播。
     */
    setupNetworkReceiver();

    // 监控Wifi Watchdog设置的变化情况
    registerForSettingsChanges();
    registerForWatchdogToggle();

    addState(mDefaultState);
    .....// 添加状态，一共有9个状态。如图5-7所示
    // Wifi Watchdog默认是开启的，故状态机转入NotConnectedState状态
    if (isWatchdogEnabled())
        setInitialState(mNotConnectedState);
    else    setInitialState(mWatchdogDisabledState);

    updateSettings();
}

```

上面代码中，WifiWatchdogStateMachine的初始状态是NotConnectedState。不过这个状态仅实现了enter函数，而且该函数中仅实现了一句打印输出的代码，所以NotConnectedState是一个象征意义远大于实际作用的类。

图5-7所示为WifiWatchdogStateMachine中的各个状态及层级关系。

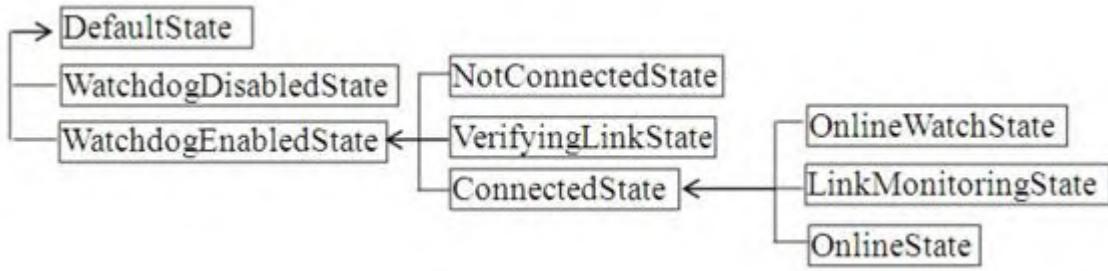


图5-7 WifiWatchdogStateMachine状态及层级关系

WifiWatchdogStateMachine完全靠广播事件来驱动，相关代码在setupNetworkReceiver函数中。

```

[-->WifiWatchdogStateMachine.java: : setupNetworkReceiver]

private void setupNetworkReceiver() {
    mBroadcastReceiver = new BroadcastReceiver() {
        public void onReceive(Context context, Intent intent) {
            String action = intent.getAction();
            if (action.equals(WifiManager.RSSI_CHANGED_ACTION)) {
                obtainMessage(EVENT_RSSI_CHANGE,
                    intent.getIntExtra(WifiManager.EXTRA_NEW_RSSI,
-200), 0).sendToTarget();
            } else if
(action.equals(WifiManager.SUPPLICANT_STATE_CHANGED_ACTION)) {
                sendMessage(EVENT_SUPPLICANT_STATE_CHANGE,
                intent);
            } else if
(action.equals(WifiManager.NETWORK_STATE_CHANGED_ACTION)) {
                sendMessage(EVENT_NETWORK_STATE_CHANGE,
                intent);
            } .....
            else if
(action.equals(WifiManager.WIFI_STATE_CHANGED_ACTION))
                sendMessage(EVENT_WIFI_RADIO_STATE_CHANGE,intent.getStringExtra(
                    WifiManager.EXTRA_WIFI_STATE,
                    WifiManager.WIFI_STATE_UNKNOWN));
        }
    };
    mIntentFilter = new IntentFilter();
    mIntentFilter.addAction(WifiManager.NETWORK_STATE_CHANGED_ACTIO

```

```
N) ;  
.....// 添加感兴趣的广播事件类型  
mContext.registerReceiver(mBroadcastReceiver,  
mIntentFilter);  
}
```

在前面介绍的WifiService工作流程中，SUPPLICANT\_STATE\_CHANGED\_ACTION和NETWORK\_STATE\_CHANGED\_ACTION广播发送的次数非常频繁。所以，WifiWatchdogStateMachine也不会太清闲。

下面，我们直接从WifiStateMachine在VerifyingLinkState的enter函数中发送的NETWORK\_STATE\_CHANGED\_ACTION广播开始分析WifiWatchdogStateMachine的处理流程。

## 2. EVENT\_NETWORK\_STATE\_CHANGE处理流程

该消息被NotConnectedState的父状态WatchdogEnabledState处理，相关代码如下所示。

[-->WifiWatchdogStateMachine.java: : WatchdogEnabledState:  
processMessage]

```
public boolean processMessage(Message msg) {  
    Intent intent;  
    switch (msg.what) {  
        ....  
        case EVENT_NETWORK_STATE_CHANGE:  
            intent = (Intent) msg.obj;  
            NetworkInfo networkInfo = (NetworkInfo)  
intent.getParcelableExtra(  
WifiManager.EXTRA_NETWORK_INFO);  
            mWifiInfo = (WifiInfo)  
intent.getParcelableExtra(WifiManager.EXTRA_WIFI_INFO);  
            // 更新bssid信息  
            updateCurrentBssid(mWifiInfo != null ?  
mWifiInfo.getBSSID() : null);  
  
            switch (networkInfo.getDetailedState()) {  
                case VERIFYING_POOR_LINK:  
                    // WifiStateMachine在VerifyingLinkState中设置的状态
```

态

```

        mLinkProperties = (LinkProperties)
intent.getParcelableExtra(
    WifiManager.EXTRA_LINK_PROPERTIES);
    // mPoorNetworkDetectionEnabled用于判断是否需要监控AP的信号质量
    if (mPoorNetworkDetectionEnabled) {
        if (mWifiInfo == null ||
mCurrentBssid == null) {
            // 下面这个函数将通过mWsmChannel向WifiStateMachine发送
            // GOOD_LINK_DETECTED消息
            sendLinkStatusNotification(true);
        }
    } else
transitionTo(mVerifyingLinkState);
    // 进入VerifyingLinkState
} else sendLinkStatusNotification(true);
break;
case CONNECTED:// WifiStateMachine在ConnectedState中设置的状态
    // 请读者自行分析OnlineWatchState的处理流程
    transitionTo(mOnlineWatchState);
}
.....
}
return HANDLED;
}

```

如果WifiWatchdogStateMachine开启无线网络信号质量监控，将转入VerifyingLinkState，其enter函数如下所示。

[-->WifiWatchdogStateMachine.java: : VerifyingLinkState:  
enter]

```

public void enter() {
    mSampleCount = 0;
    mCurrentBssid.newLinkDetected();
    // 向自己发送CMD_RSSI_FETCH消息
    sendMessage(obtainMessage(CMD_RSSI_FETCH,
++mRssiFetchToken, 0));
}

```

### 3. CMD\_RSSI\_FETCH处理流程

来看CMD\_RSSI\_FETCH消息的处理。

```
[-->WifiWatchdogStateMachine.java: : VerifyingLinkState:  
processMessage]  
  
public boolean processMessage(Message msg) {  
    switch (msg.what) {  
        .....  
        case CMD_RSSI_FETCH:  
            if (msg.arg1 == mRssiFetchToken) {  
                /*  
                 * 向WifiStateMachine发送RSSI_PKTCNT_FETCH消息，  
                 * WifiStateMachine  
                 * 的处理过程就是调用WifiNative的signalPoll和  
                 * pktcntPoll以获取RSSI、  
                 * LinkSpeed、发送Packet的总个数、发送失败的Packet总  
                 * 个数。注意，4.2中  
                 * 的WPAS才支持pktcntPoll。  
                 * WifiStateMachine处理完RSSI_PKTCNT_FETCH后将回  
                 * 复RSSI_PKTCNT_FETCH  
                 * SUCCEEDED消息给WifiWatchdogStateMachine。  
                */  
  
            mWsmChannel.sendMessage(WifiManager.RSSI_PKTCNT_FETCH);  
            // LINK_SAMPLING_INTERVAL_MS值为1000ms  
  
            sendMessageDelayed(obtainMessage(CMD_RSSI_FETCH,  
                                              ++mRssiFetchToken, 0),  
                               LINK_SAMPLING_INTERVAL_MS);  
            }  
            break;  
        case WifiManager.RSSI_PKTCNT_FETCH_SUCCEEDED:  
            // WifiStateMachine回复的消息中携带一个  
            RssiPacketCountInfo对象  
            RssiPacketCountInfo info = (RssiPacketCountInfo)  
            msg.obj;  
            int rssi = info.rssi;  
            /*  
             * WifiWatchdog用了一个名为指数加权移动平均算法 (Volume-  
             * weighted Exponential  
             * Moving Average) 的方法来辨别网络信号质量的好坏。本书不对  
             * 它进行讨论，感兴趣的  
             * 读者不妨自行研究。  
            */  
            long time = mCurrentBssid.mBssidAvoidTimeMax -
```

```

        SystemClock.elapsedRealtime();
        // 假设网络质量很好，则调用sendLinkStateNotification
        // 以发送GOOD_LINK_DETECT消息给WifiStateMachine
        if (time <= 0) sendLinkStatusNotification(true);
        else {
            // 此时的rssl等某个阈值。mGoodLinkTargetRssi由算法
计算得来
            if (rssl >= mCurrentBssid.mGoodLinkTargetRssi)
{
                // 当采样次数大于一定值时，才认为网络状态变好
                // mGoodLinkTargetCount也是通过相关方法计算得
来
                if (++mSampleCount >=
mCurrentBssid.mGoodLinkTargetCount) {
                    mCurrentBssid.mBssidAvoidTimeMax =
0;
                    sendLinkStatusNotification(true);
                }
            } else mSampleCount = 0;
        }
        break;
        .....
    }
    return HANDLED;
}

```

当WifiWatchdogStateMachine检测到Pool link时，它将发送  
POOL\_LINK\_DETECT消息给WifiStateMachine去处理。相关流程请感兴趣的读者自行研究。

WifiWatchdogStateMachine是一个比较有趣的模块。其目的很简单，  
就是监测无线网络的信号质量，然后做相应动作。另外，  
WifiWatchdogStateMachine还使用了一些比较高级的算法来判断网络  
信号质量的好坏，感兴趣的读者不妨进行深入研究。

# 读累了记得休息一会儿哦~

公众号：古德猫宁李

- 电子书搜索下载
- 书单分享
- 书友学习交流

网站：[沉金书屋 https://www.chenjin5.com](https://www.chenjin5.com)

- 电子书搜索下载
- 电子书打包资源分享
- 学习资源分享

## 5.5 Captive Portal Check介绍

Android 4.2中，Captive Portal Check功能集中在CaptivePortalTracker类中，而且它也是一个HSM。

CaptivePortalTracker的创建位于ConnectivityService构造函数中。在那里，它的makeCaptivePortalTracker函数被调用。相关代码如下所示。

[-->CaptivePortalTracker.java: : makeCaptivePortalTracker]

```
public static CaptivePortalTracker  
makeCaptivePortalTracker(Context context,  
                         IConnectivityManager cs) {  
    CaptivePortalTracker captivePortal = new  
    CaptivePortalTracker(context, cs);  
    captivePortal.start(); // 启动HSM  
    return captivePortal;  
}
```

### 1. CaptivePortalTracker构造函数分析

CaptivePortalTracker的构造函数如下所示。

[-->CaptivePortalTracker.java: : CaptivePortalTracker]

```
private CaptivePortalTracker(Context context,  
                           IConnectivityManager cs) {  
    super(TAG);  
  
    mContext = context; mConnService = cs;  
    mTelephonyManager = (TelephonyManager)  
    context.getSystemService(  
  
    Context.TELEPHONY_SERVICE);  
    // 注册一个广播接收对象，用于处理CONNECTIVITY_ACTION消息  
    IntentFilter filter = new IntentFilter();  
    filter.addAction(ConnectivityManager.CONNECTIVITY_ACTION);  
    mContext.registerReceiver(mReceiver, filter);  
    // CAPTIVE_PORTAL_SERVER用于设置进行Captive Portal Check测试的  
    // 服务器地址  
    mServer =
```

```

Settings.Global.getString(mContext.getContentResolver(),
Settings.Global.CAPTIVE_PORTAL_SERVER);
    // 如果没有指明服务器地址的，则采用DEFAULT_SERVER，其地址
是“clients3.google.com”
    if (mServer == null) mServer = DEFAULT_SERVER;
    // 是否开启Captive Portal Check功能，默认是开始
    mIsCaptivePortalCheckEnabled = Settings.Global.getInt
        (mContext.getContentResolver(),
Settings.Global.CAPTIVE_PORTAL_DETECTION_ENABLED, 1) == 1;

addState(mDefaultState); // CaptivePortalTracker只有4个状态
addState(mNoActiveNetworkState, mDefaultState);
addState(mActiveNetworkState, mDefaultState);
addState(mDelayedCaptiveCheckState,
mActiveNetworkState);
    setInitialState(mNoActiveNetworkState);
}

```

CaptivePortalTracker只监听CONNECTIVITY\_ACTION广播，而  
WifiService相关模块并不会发送这个广播。那么，在前面介绍的流程中，哪一步会触发CaptivePortalTracker进行工作呢？来看下节。

## 2. CMD\_CONNECTIVITY\_CHANGE处理流程

当Wi-Fi网络连接成功时，ConnectivityService的handleConnect将被触发，该函数内部将发送一个CONNECTIVITY\_ACTION消息，这个消息将被CaptivePortalTracker注册的广播接收对象处理。相关代码如下所示。

[-->CaptivePortalTracker.java: : onReceive]

```

private final BroadcastReceiver mReceiver = new
BroadcastReceiver() {
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();
        if
(action.equals(ConnectivityManager.CONNECTIVITY_ACTION)) {
            NetworkInfo info = intent.getParcelableExtra(
ConnectivityManager.EXTRA_NETWORK_INFO);
            // 向状态机发送CMD_CONNECTIVITY_CHANGE消息

```

```
sendMessage(obtainMessage(CMD_CONNECTIVITY_CHANGE, info));  
    }  
}  
};
```

CaptivePortalTracker的NoActiveNetworkState将处理该消息。相关代码如下所示。

[-->CaptivePortalTracker.java: : NoActiveNetworkState:  
processMessage]

```
public boolean processMessage(Message message) {
    InetAddress server;           NetworkInfo info;
    switch (message.what) {
        case CMD_CONNECTIVITY_CHANGE:
            info = (NetworkInfo) message.obj;
            // 无线网络已经连接成功，并且手机当前使用的就是Wi-Fi
            // isActiveNetwork将查询ConnectivityService以获取
当前活跃的数据链接类型
            if (info.isConnected() && isActiveNetwork(info))
{
                mNetworkInfo = info;
                transitionTo(mDelayedCaptiveCheckState);
                // 转移到DelayedCaptiveCheckState
}
        .....
    }
    return HANDLED;
}
```

### 3. CMD\_DELAYED\_CAPTIVE\_CHECK处理流程

DelayedCaptiveCheckState代码如下所示。

[-->CaptivePortalTracker.java: : DelayedCaptiveCheckState]

```
private class DelayedCaptiveCheckState extends State {  
    public void enter() {  
        // 发送一个延迟消息，延迟时间为10秒  
  
        sendMessageDelayed(obtainMessage(CMD_DELAYED_CAPTIVE_CHECK,  
                                         ++mDelayedCheckToken, 0),  
                           DELAYED_CHECK_INTERVAL_MS);  
    }  
}
```

```

public boolean processMessage(Message message) {
    switch (message.what) {
        case CMD_DELAYED_CAPTIVE_CHECK:
            if (message.arg1 == mDelayedCheckToken) {
                InetAddress server =
lookupHost(mServer); // 获取Server的IP地址
                if (server != null) {
                    // AP是否需要Captive Portal Check。如果需要
                    // setNotificationVisible将在状态栏中添加一个提醒信息
                    if (isCaptivePortal(server))
setNotificationVisible(true);
                }
                transitionTo(mActiveNetworkState);
                // 转到ActiveNetworkState
            }
            break;
        default:
            return NOT_HANDLED;
    }
    return HANDLED;
}
}

```

上述代码中，`isCaptivePortal`用于判断server是否需要Captive Portal Check。其代码如下所示。

[-->`CaptivePortalTracker.java`: : `isCaptivePortal`]

```

private boolean isCaptivePortal(InetAddress server) {
    HttpURLConnection urlConnection = null;
    if (!mIsCaptivePortalCheckEnabled) return false;
    // mUrl实际访问的地址是: http://clients3.google.com/generate_204
    mUrl = "http://" + server.getHostAddress() +
"/generate_204";
/*

```

Captive Portal的测试非常简单，就是向`mUrl`发送一个HTTP GET请求。如果无线网络提供商没

有设置`PortalCheck`，则HTTP GET请求将返回204。204表示请求处理成功，但没有数据返回。

如果无线网络提供商设置了`Portal Check`，则它一定会重定向到某个特定网页。这样，HTTP GET的

返回值就不是204。

```

        */
    try {
        URL url = new URL(mUrl);
        urlConnection = (HttpURLConnection)
url.openConnection();
        urlConnection.setInstanceFollowRedirects(false);
        urlConnection.setConnectTimeout(SOCKET_TIMEOUT_MS);
        urlConnection.setReadTimeout(SOCKET_TIMEOUT_MS);
        urlConnection.setUseCaches(false);
        urlConnection.getInputStream();
        return urlConnection.getResponseCode() != 204;
    }.....
}

```

处理完毕后，CaptivePortalTracker将转入ActiveNetworkState状态。该状态的内容非常简单，读者可自行阅读它。由于笔者家中所在小区宽带提供商使用了Capive Portal Check，所以笔者利用AirPcap截获了相关网络交换数据，如图5-8所示。

Frame	Source IP	Destination IP	Protocol	Details
615	14.198.0.960	192.168.0.101	HTTP	277 GET /generate_204 HTTP/1.1
617	14.198.97.10	74.125.235.197	HTTP	251 HTTP/1.1 302 Moved Temporarily
658	14.6042210	192.168.0.101	HTTP	471 POST /2/push/log_get HTTP/1.1 [Malformed Packet]
742	16.2483460	192.168.0.101	HTTP	294 GET /mobilesafe/batterysaver/andr/update.md5?mid
744	16.2492220	119.188.64.75	HTTP	251 HTTP/1.1 302 Moved Temporarily
753	16.2794690	192.168.0.101	HTTP	216 GET / HTTP/1.1
782	16.3034720	192.168.10.2	HTTP	1082 HTTP/1.1 200 OK (text/html)
789	16.31278480	192.168.0.101	HTTP	294 GET /mobilesafe/batterysaver/andr/update.zip?mid
791	16.3137270	119.188.64.75	HTTP	251 HTTP/1.1 302 Moved Temporarily
795	16.3176080	192.168.0.101	HTTP	216 GET / HTTP/1.1
818	16.3349720	192.168.10.2	HTTP	1082 HTTP/1.1 200 OK (text/html)
1187	24.6978470	192.168.0.101	HTTP	580 POST /sdk.php HTTP/1.1 (application/x-www-form-

```

# Frame 615: 277 bytes on wire (2216 bits), 277 bytes captured (2216 bits) on interface 0
# Radiotap Header v0, Length 20
# IEEE 802.11 QoS Data, Flags: .....TC
# Logical-Link Control
# Internet Protocol Version 4, Src: 192.168.0.101 (192.168.0.101), Dst: 74.125.235.197 (74.125.235.197)
# Transmission Control Protocol, Src Port: 40685 (40685), Dst Port: http (80), Seq: 1, Ack: 4294967144, Len: 179
# Hypertext Transfer Protocol
# GET /generate_204 HTTP/1.1\r\n
User-Agent: Dalvik/1.6.0 (Linux; U; Android 4.1.2; GT-N7100 Build/J2054K)\r\n
Host: clients3.google.com\r\n
Connection: Keep-Alive\r\n
Accept-Encoding: gzip\r\n
\r\n
[full request URI: http://clients3.google.com/generate_204]

```

图5-8 Captive Portal Check示意图

测试时禁用了无线网络安全，所以AirPcap可以解析这些没有加密的数据包。由图5-8可知，当Note 2发起HTTP GET请求后，小区宽带服务器回复了HTTP 302，所以笔者手机状态栏才会显然如图5-9所示的提示项以提醒该无线网络需要登录。



图5-9 Captive Portal Check提示

#### 4. CaptivePortalTracker总结

和WifiWatchdogStateMachine一样，CaptivePortalTracker也比较有意思。CaptivePortalTracker类出现于4.2版本中，而4.1版本中的Captive Portal Check功能则是由WifiWatchdogStateMachine来完成的。在4.1版本中，WifiWatchdogStateMachine定义了一个WalledGardenCheckState类用于处理Captive Portal Check。

不过，笔者在研究4.1版本和4.2版本相关模块的代码后，发现4.2版本中的处理似乎有一些问题。此处先记下此问题，希望有兴趣的读者参与讨论。问题如下。

4.2版本中，WifiStateMachine在ConnectedState前增加了一个CaptivePortalCheckState。很明显，CaptivePortalCheckState的目的是在WifiStateMachine转入ConnectedState之前完成Captive Portal Check。但根据本节对CaptivePortalTracker的介绍，只有WifiStateMachine进入ConnectedState后，ConnectivityService才会发送CONNECTIVITY\_ACTION广播。而在4.1版本中，WifiWatchdogStateMachine也是在WifiStateMachine转入ConnectedState后进入WalledGardenCheckState的。

提示 WifiStateMachine如何从CaptivePortalCheckState转为ConnectedState呢？

答案在ConnectivityService的handleCaptivePortalTrackerCheck函数中。在那里，WifiStateMachine的captivePortalCheckComplete函数最终会被调用。在该函数中，CMD\_CAPTIVE\_CHECK\_COMPLETE将被发送，CaptivePortalCheckState才会转入ConnectedState状态。但是，

在这个流程中，CaptivePortalTracker并未被真正触发以进行Captive Portal Check。

## 5.6 本章总结和参考资料说明

### 5.6.1 本章总结

本章对WifiService相关模块、WifiWatchdogStateMachine以及CaptivePortalTracker进行了介绍。

- HSM和AsyncChannel是本章的重要知识，希望读者认真学习它们的用法。
- WifiService中的WifiStateMachine是本章的核心。其定义的状态之多、处理之曲折在Android系统中算是非常突出的。如果条件允许的话，读者不妨通过Eclipse来调试它以加深认识。
- WifiWatchdogStateMachine和CaptivePortalTracker内容比较简单，但其功能却比较有意思。对于这部分内容，简单了解即可。

**提示** 笔者一直觉得WifiService的实现过于复杂，而且其运行效率较低。不知道读者看完本章后是否有同感。

## 5.6.2 参考资料说明

[1] [https://en.wikipedia.org/wiki/Captive\\_portal](https://en.wikipedia.org/wiki/Captive_portal)

说明：维基百科对Captive Portal的介绍。读者简单了解即可。

# 第6章 深入理解Wi-Fi Simple Configuration

本章所涉及的源代码文件名及位置

- WpsDialog.java  
packages/apps/Settings/src/com/android/settings/wifi/WpsDialog.java
- WifiStateMachine.java  
frameworks/base/wifi/java/android/net/wifi/WifiStateMachine.java
- WifiConfigStore.java  
frameworks/base/wifi/java/android/net/wifi/WifiConfigStore.java
- wpa\_supplicant.c  
external/wpa\_supplicant\_8/wpa\_supplicant/wpa\_supplicant.c
- ctrl\_iface.c  
external/wpa\_supplicant\_8/wpa\_supplicant/ctrl\_iface.c
- scan.c      external/wpa\_supplicant\_8/wpa\_supplicant/scan.c
- wps.c      external/wpa\_supplicant\_8/src/wps/wps.c
- events.c  
external/wpa\_supplicant\_8/wpa\_supplicant/events.c
- eapol\_supp\_sm.c  
external/wpa\_supplicant\_8/src/eapol\_supp/eapol\_supp\_sm.c
- eap.c      external/wpa\_supplicant\_8/src/eap\_peer/eap.c

- wps\_enrollee.c  
external/wpa\_supplicant\_8/src/wps/wps\_enrollee.c
- wps\_supplicant.c  
external/wpa\_supplicant\_8/wpa\_supplicant/wps\_supplicant.c

## 6.1 概述

在Wi-Fi相关技术体系中，除了802.11定义的标准规范外，Wi-Fi联盟（Wi-Fi Alliance）也推出了两项比较重要的技术规范，分别是WSC和P2P。

- WSC（Wi-Fi Simple Configuration）：该项技术用于简化SOHO环境中无线网络的配置和使用。举一个简单的例子，配置无线网络环境时，网管需要首先为AP设置SSID、安全属性（如身份认证方法、加密方法等）。然后还得把SSID、密码告诉给该无线网络的使用者。可是这些安全设置信息对普通大众而言还是有些复杂。而有了WSC之后，用户只需输入PIN码（Personal Identification Number），或者单击按钮（WSC中，该按钮称为Push Button）甚至用户只要拿着支持NFC的手机到目标AP（它必须也支持NFC）旁刷一下，这些安全设置就能被自动配置好。有了这些信息，手机就能连接上目标无线网络了。显然，相比让用户记住SSID、密码等信息，WSC要简单多了。
- P2P（Wi-Fi Peer-to-Peer）：P2P的商品名（brand name）为Wi-Fi Direct。它支持多个Wi-Fi设备在没有AP的情况下相互连接。笔者认为P2P是Wi-Fi中最具应用前景的一项技术。例如，在家庭中，用户可直接把手机上的内容通过P2P技术传输到电视机上和家人分享。

注意 P2P和第3章无线网络结构中提到的IBSS（Independent BSS）完全不同。IBSS中，各个STA属于完全对等的关系，而P2P则不然。关于P2P的细节，我们留待下章再来分析。

在wpa\_supplicant（以后简称WPAS）中，WSC的功能点分散在第4章介绍的几条分析路线中，为了避免赘述，本章的分析采用如下方法。

- 首先介绍WSC所涉及的基础知识。这是本章的核心。
- 然后分析WSC相关的代码。这部分代码包括Settings、WifiService相关模块（主要是WifiStateMachine）以及WPAS。

下面，先来介绍WSC的理论知识。

## 6.2 WSC基础知识

WSC规范早期的名字叫WPS（Wi-Fi Protected Setup）。WFA推出WPA后不久，WPS规范便被推出。随着WPA2的出现，WFA又制订了WPS的升级版，即WSC。WSC的规范（以2.0.2版为例）全文只有150页。

WSC的目的很简单，就是简化无线网络配置（这也是其英文名为Simple Configuration的原因）。本节将以WSC的技术规范为主，向读者介绍相关的理论知识。

**提示** WSC规范中，安全设置信息可以借助Wi-Fi作为传输手段，也可以借助其他传输方式，如NFC传输等。如果这些信息使用Wi-Fi作为传输手段，称为In-Band交换，否则称为Out-of-Band交换。

## 6.2.1 WSC应用场景<sup>[1]</sup>

WSC定义两个应用场景（usage model），分别是Primary UM和Secondary UM。

- Primary UM包括设置一个新的安全的WLAN，并为该WLAN添加无线设备。该场景和前文介绍的WSC应用场景一样。日常生活中，Primary UM对应的情况更为普遍。
- Secondary UM包括从WLAN中移除某个无线设备、通过添加新的AP或路由器来扩充WLAN的覆盖范围、密钥信息更换（Re-keying credentials）等。

Primary UM常见的两种案例包括PIN和PBC。其中，PIN的使用案例如图6-1所示。

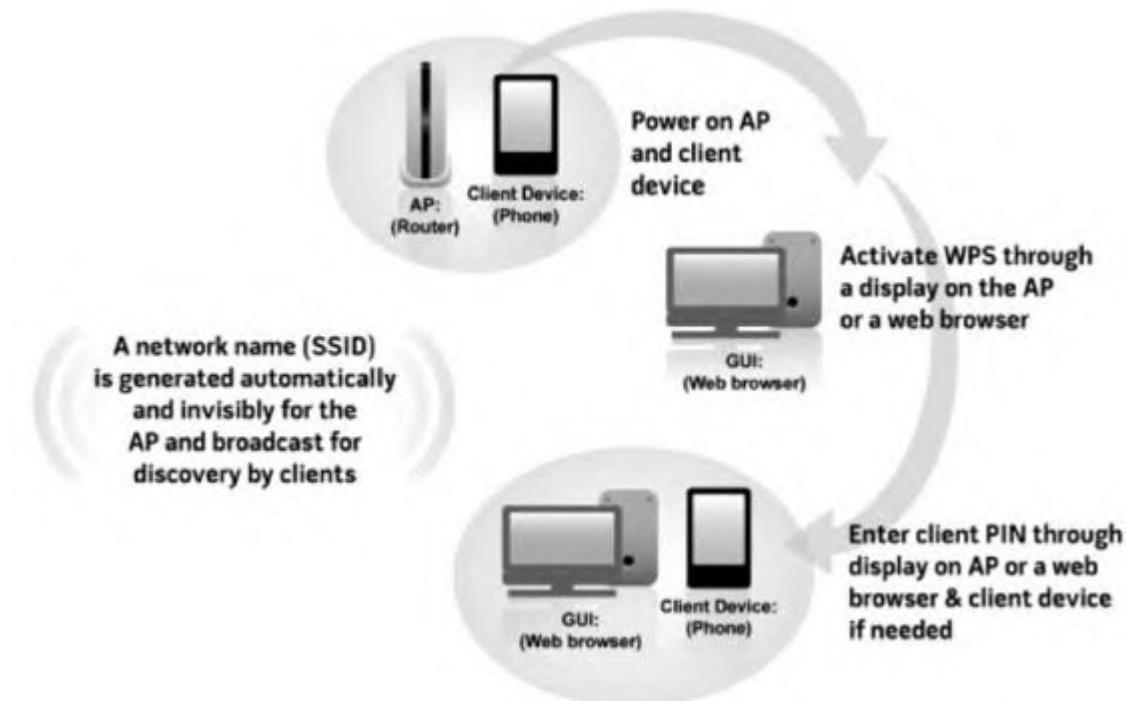


图6-1 WSC PIN案例

图6-1所示为WSC定义的PIN码配置方法，其工作流程如下。

- 1) 打开AP和STA。用户首先从STA相关的设置选项中获取一个PIN码。
- 2) 用户将STA的PIN码通过AP的设置页面传递给AP。
- 3) AP和STA将基于这个PIN码完成安全设置协商。然后STA将完成扫描、关联、四次握手等工作以加入目标AP。

PIN码是长度为8的字符串，图6-2所示为笔者用Galaxy Note 2测试WSC PIN方法时获取的PIN信息。左图所示的页面位于Settings的无线网络设置选项中，有条件的读者不妨一试。



图6-2 Galaxy Note 2 WSC PIN设置

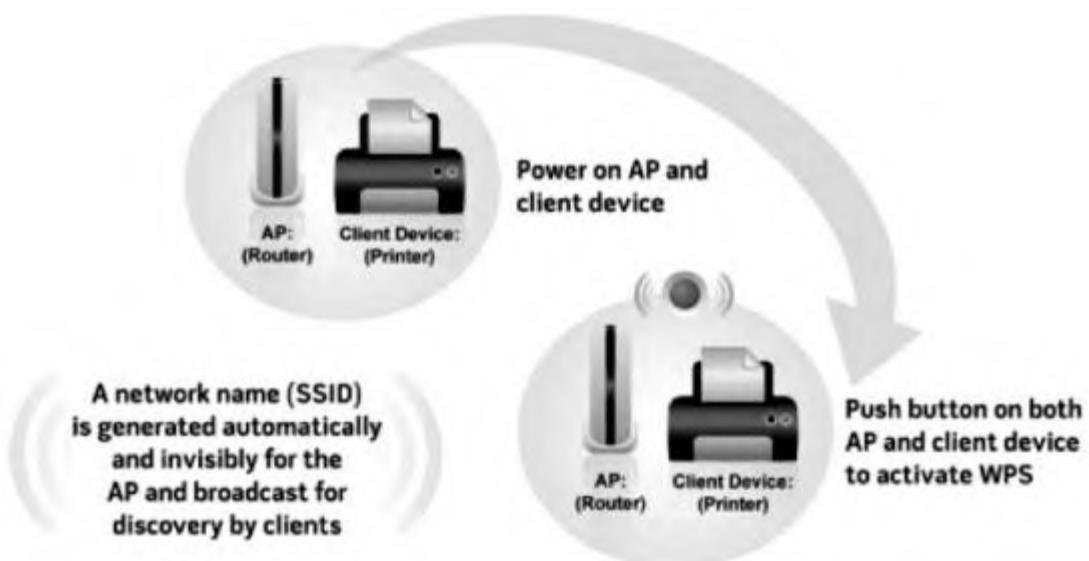
STA中的PIN码需要输入AP中，图6-3所示为笔者家中无线路由器WSC PIN设置页面，注意右下角的黑框中是手机的PIN码，笔者测试时从Galaxy Note 2中获取的PIN码是33871042。



图6-3 AP设置页面

提示 图6-2和图6-3所示的PIN码并不一致。目的是表示系统每次生成的PIN码不是固定的。

相比PIN而言，PB配置方法（Push Button Configuration, PBC）的使用更加简单。PBC案例如图6-4所示。



#### 图6-4 PBC案例

PBC的工作流程如下。

- 1) 用户打开AP和打印机（支持Wi-Fi）。打印机和AP上都有一个小按钮（注意，规范要求该按钮必须标记上WPS以表示它对WSC的支持）。
- 2) 用户只要在AP和打印机上单击该按钮，将触发打印机和AP完成安全设置协商。如此，打印机获取AP的安全设置信息后将顺利加入目标AP。

图6-5所示为笔者家中无线路由器上的WSC按钮。对于Android智能手机，可通过软件中的按钮来模拟真实的Push Button（参考图6-2中左图的“WPS推送按钮”项）。

## 6.2.2 WSC核心组件及接口<sup>[2]</sup>

图6-6所示为WSC定义的三个核心组件，其中：

- Enrollee的角色类似于supplicant，它向Registrar发起注册请求。
- Registrar用于检查Enrollee的合法性。另外，Registrar还能对AP进行配置。
- AP也需要注册到Registrar中。所以，从Registrar角度来看，AP也是Enrollee。

AP、Registrar以及Enrollee三者交互，Enrollee从Registrar那获取AP的安全配置信息，然后Enrollee利用该信息加入AP提供的无线网络。



图6-5 PBC实物

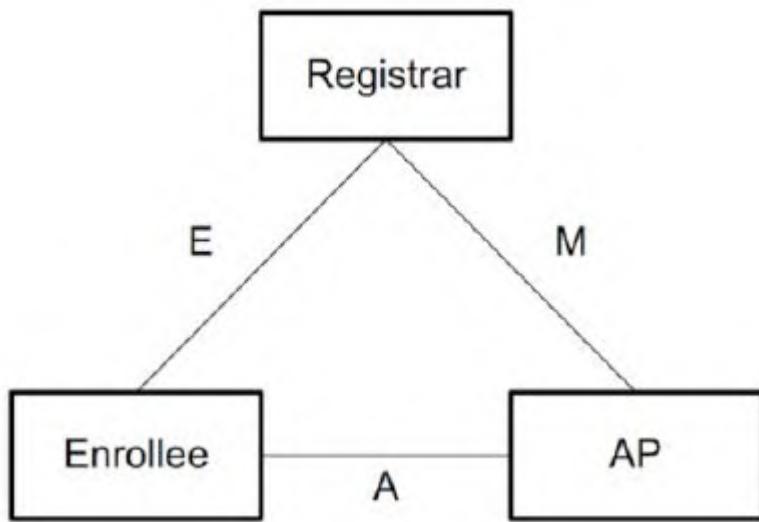


图6-6 WSC核心组件

注意 这三个组件只是逻辑上的概念。在具体实现时，AP和Registrar可以由同一个实体实现，也可分别由不同实体来实现。日常生活中，支持WSC的无线路由器兼具AP和Registrar的功能。这种AP在规范中称为Standalone AP。Android智能手机扮演Enrollee的角色。如果AP和Registrar分别由不同实体来实现，这种Registrar也称为External Registrar。

除了三大组件之外，规范还定义了组件之间的交互接口。例如图6-6中的E、M、A代表三个核心组件之间交互的接口，这些接口定义了交互双方需要实现的一些功能。

规范中关于E、M、A的介绍非常复杂，笔者不拟照搬规范的内容，而是试图通过一种普适的case来介绍E、M、A的功能。这个case就是AP和Registrar组件实现于一个无线路由器中，即Standalone AP，而Enrollee由STA实现。STA和Standalone AP通过Wi-Fi传输数据，即它们将采用In-Band交互手段。

在上述情况下，STA中的Interface E包括的功能如下。

- STA首先要寻找周围支持WSC功能的Standalone AP。此步骤将通过发送携带WSC IE的Probe Request帧来实现。另外，STA必须能生成动态PIN码，该PIN码将用于检验后续安全配置信息的正确性。

- STA关联到Standalone AP后（注意，仅仅是关联成功。由于缺乏安全配置信息，STA无法和AP开展RSNA流程，即四次握手等工作），双方需要借助Registration Protocol（以下简称RP协议）来协商安全配置信息。所以，STA必须实现RP协议的Enrollee的功能。

Standalone AP中Interface E包括的功能如下。

- 回复携带WSC IE的Probe Response帧以表明自己支持WSC功能。
- 实现RP协议定义的Registrar的功能。

**提示** STA和AP可选择实现某种Out-of-Band交互手段，规范中提到的两种手段包括NFC和USB。

对于Interface A来说，STA必须实现802.1X supplicant功能，并支持EAP-WSC算法。Standalone AP需发送携带WSC IE的Beacon帧来表示自己支持WSC功能。同时，AP还必须支持802.1X authenticator功能，并实现EAP-WSC算法。

由于Standalone AP已经集成了三大组件中的AP和Registrar，所以Interface M的功能几乎简化为0。由于本书不讨论AP的实现，所以此处不介绍和它相关的内容。

**提示** WSC规范关于E、M、A的介绍比较复杂，其中还涉及UPnP的使用。本书不讨论UPnP方面的内容，感兴趣的读者不妨参考笔者的一篇博文（<http://blog.csdn.net/innost/article/details/7078539>）。

由上文所述内容可知，WSC的核心知识集中在WSC IE以及RP中，下一节详细介绍。

## 6.3 Registration Protocol详解<sup>[3]</sup>

以前面提到的普适case为例，当STA和Standalone AP采用In-Band交互方法时，RP协议的完整交互流程如图6-7所示。包括两部分，由“Enter password of Enrollee”行隔开，其中，上部分所对应的交互部分被称为Discovery Phase。在此阶段中，STA借助Beacon帧或Probe Request帧搜索周围的AP。对开启了WSC功能的STA来说，这些帧中都必须携带WSC IE。而没有携带WSC IE的帧则表明发送者不支持或者未开启WSC功能。Discovery Phase结束后，STA将确定一个目标AP。

此时用户需要将STA显示的PIN码（如图6-2所示）输入到目标AP的设置页面（如图6-3所示）。接着，STA将关联到目标AP。和非WSC流程不一样的是，STA和AP不会开展四次握手协议，而是先开展EAP-WSC流程。

EAP-WSC流程从EAPOL-Start开始，结束于EAP-Fail帧，一共涉及14次EAPOL/EAP帧交换。在这14次帧交互过程中，STA和AP双方将协商安全配置信息（例如采用何种身份验证方法、何种加密方法，以及PSK等）。另外，这14次帧中，M1～M8属于EAP-WSC算法的内容，它们用于STA和AP双方确认身份以及传输安全配置信息。

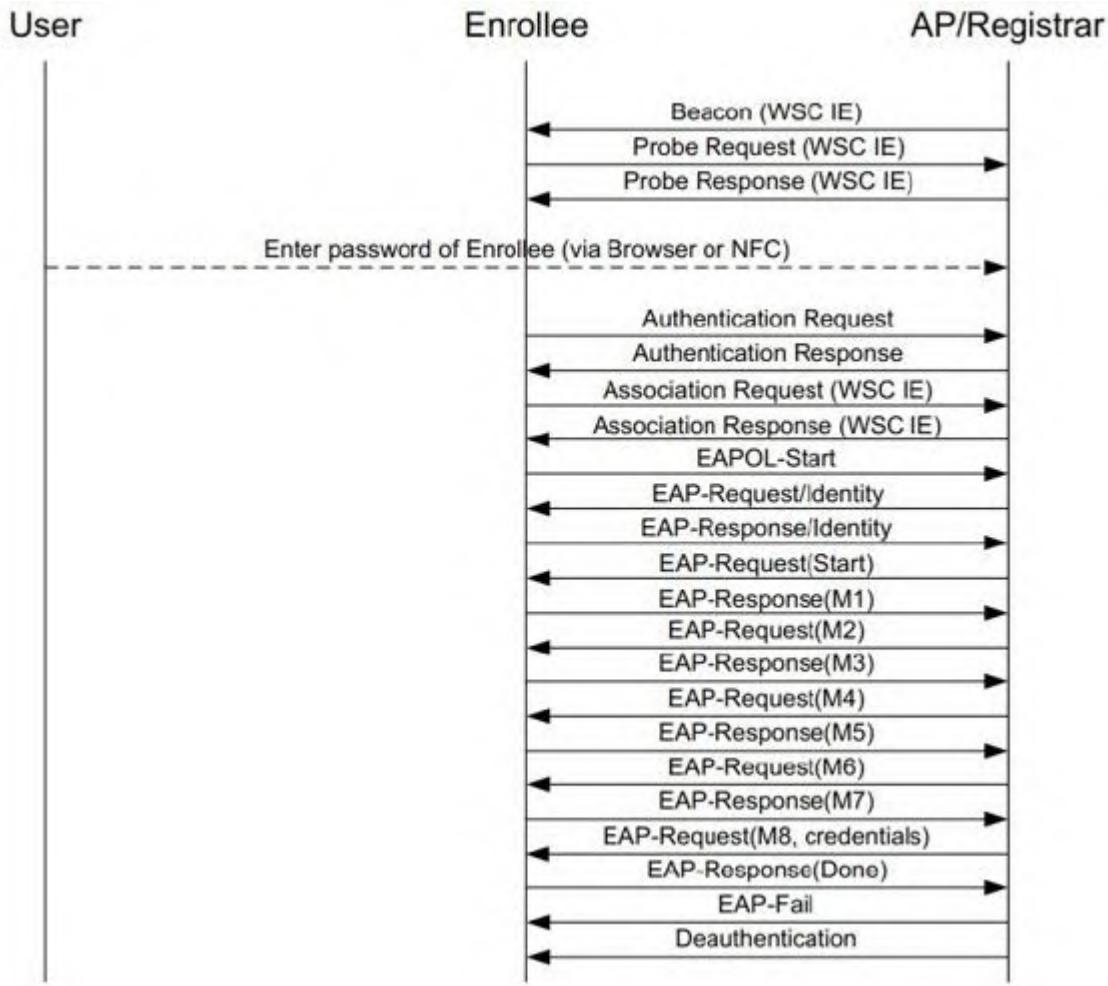


图6-7 完整RP协议交互

虽然EAP-WSC最终以EAP-Fail帧结束，但STA已经和AP借助M1到M8成功完成了安全信息协商，所以STA已经获得了AP的安全配置情况。另外，由于STA收到的是EAP-Fail帧，所以它会断开和AP的连接（AP会发送Deauthentication帧给STA）。

STA将利用协商好的安全配置信息重新和AP进行关联，后续流程和非WSC的无线网络关联一样，即STA关联到AP后，将开展RSNA工作（如四次握手协议、Group Handshake流程）。

关于图6-7所涉及的流程需要注意。

- 如果不使用WSC，用户需要为AP设置安全配置信息，然后STA搜索并关联到目标AP。接着，STA和AP将利用四次握手协议和Group

Handshake协议完成RSNA工作。RSNA工作属于WPA和WPA2规范所指定的，STA和AP必须完成RSNA流程。

- 如果使用WSC，STA和AP共享的只有PIN码（或者用户单击双方的按钮），为了完成RSNA流程，STA需要从AP那获取安全配置信息。而这个安全配置信息的传递则由图6-7所示的EAPOL/EAP帧交互来完成。有了安全配置信息后，STA后续流程和没有使用WSC的情况一样。

根据上面的描述可知，WSC的核心工作就是帮助STA和AP完成安全配置信息协商。由于在这个流程中，用户只需输入PIN码或单击按钮，所以用户的工作量极小。WSC工作完成后，STA和AP的工作和第4章介绍的一样（STA首先关联到AP，然后完成四次握手协议和Group Handshake协议）。

下面分别介绍WSC IE以及EAP-WSC相关知识。首先登场的是WSC IE。

# 读累了记得休息一会儿哦~

公众号：古德猫宁李

- 电子书搜索下载
- 书单分享
- 书友学习交流

网站：[沉金书屋 https://www.chenjin5.com](https://www.chenjin5.com)

- 电子书搜索下载
- 电子书打包资源分享
- 学习资源分享

### 6.3.1 WSC IE和Attribute介绍<sup>[4]</sup>

WSC IE并不属于802.11规范所定义的IE，而是属于Vendor定义的IE。根据802.11规范，Vendor定义的IE组成结构如图6-8所示。

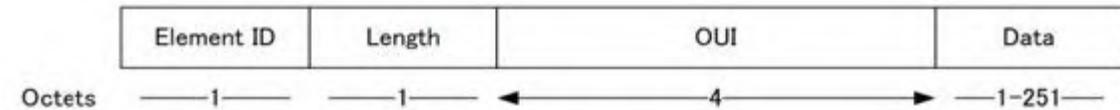


图6-8 Vendor IE的结构

根据图6-8所示的结构，WSC IE对应的设置如下。

- Element ID取值为221。802.11规范中，该值意为“Vendor Specific”。
- Length为OUI及Data的长度。
- OUI取值为0x00-50-F2-04。其中00-50-F2代表Microsoft公司的OUI，04代表WPS。
- WSC IE中，Data域的组织结构为一个或多个Attribute（属性）。Attribute的格式为TLV，即Type（长度为2字节）、Length（长度为2字节，代表后面Value的长度），Value（最大长度为0xFFFF字节）。

图6-9所示为笔者截获的WSC IE。由上文可知，WSC IE的核心是其携带的Attribute。WSC规范定义了多个Attribute，而了解这些Attribute的内容及作用是学习WSC的必经之路。下面将介绍WSC中一些重要的Attribute。

**提示** Attribute不仅被WSC IE使用，还被后文介绍的EAP-WSC包使用。

#### 1. Version和Vendor Extension属性

Version属性表达了发送端使用的WSC版本信息。Version属性对应的内容如图6-10所示。

```
└ Tag: vendor specific: Microsoft: WPS
    Tag Number: vendor specific (221)
    Tag length: 14
    OUI: 00-50-f2 (Microsoft)
    vendor specific OUI Type: 4
    Type: WPS (0x04)
```

图6-9 WSC IE实例

```
└ Version: 0x10
    Data Element Type: Version (0x104a)
    Data Element Length: 1
    Version: 0x10
```

图6-10 Version属性的内容

由图6-10可知，Version属性的Type字段取值为0x104a，Length字段长度为1字节。Version属性已经作废（Deprecated），但为了保持兼容性，规范要求WSC IE必须包含该属性，并且其Value字段需设为0x10。

取代Version属性的是Version2属性，Version2属性并不能单独存在，而是作为Vendor Extension的子属性被包含在WSC IE中。Vendor Extension的示例如图6-11所示。

```

    ☐ Vendor Extension
      Data Element Type: Vendor Extension (0x1049)
      Data Element Length: 9
      Vendor Extension: 00372a000120030101
      Vendor ID: 14122
    ☐ version2: 2.0
      WFA Extension Subelement ID: Version2 (0)
      WFA Extension Subelement Length: 1
      Version2: 0x20
    ☐ Request to Enroll: TRUE
      WFA Extension Subelement ID: Request to Enroll (3)
      WFA Extension Subelement Length: 1
      Request to Enroll: 0x01

```

图6-11 Vendor Extension属性

Vendor Extension头部包含三个部分，分别是Type（值为0x1049）、Length（此处的值为9）、Vendor ID<sub>①</sub>（十进制值为14122，十六进制值为0x00372a）。

Vendor Extension可包含多个子属性，图6-11的Vendor Extension包含了Version2和Request to Enroll两个子属性。

“Vendor Extension: 00372a000120030101”一行代表的是Vendor Extension的Value。它是后面Vendor ID、Version2和Request to Enroll的所有内容。

Request to Enroll子属性代表Enrollee希望开展后续的EAP-WSC流程。

## 2. Request Type和Response Type属性

图6-12所示为Request Type和Response Type两个属性的内容。

☐ Request Type: Enrollee, open 802.1X (0x01)	☐ Response Type: AP (0x03)
Data Element Type: Request Type (0x103a)	Data Element Type: Response Type (0x103b)
Data Element Length: 1	Data Element Length: 1
Request Type: Enrollee, open 802.1X (0x01)	Response Type: AP (0x03)

图6-12 Request Type和Response Type属性

图6-12左图所示为Request Type，右图所示为Response Type。

- Request Type属性必须包含于Probe/Association Request帧中，代表STA作为Enrollee想要发起的动作。该属性一般取值0x01（含义为Enrollee, open 802.1X），代表该设备是Enrollee，并且想要开展WSC后续流程。它还有一个取值为0x00（含义为Enrollee, Info only），代表STA只是想搜索周围支持WSC的AP，而暂时还不想加入某个网络。
- Response Type属性代表发送者扮演的角色。对于AP来说，其取值为0x03（含义为AP），对于Registrar来说，其取值为0x02，对于Enrollee来说，其取值可为0x00（Enrollee, Info only）和0x01（Enrollee, open 802.1X）。Standalone AP也属于AP，故图6-12右图的Response Type取值为0x03。

### 3. Configuration Methods和Primary Device Type属性

Configuration Methods属性用于表达Enrollee或Registrar支持的WSC配置方法。前文提到的PIN和PBC就属于WSC配置方法。考虑到支持Wi-Fi的设备类型很多，例如打印机、相机等，故WSC规范定义的WSC配置方法较多。图6-13所示为Configuration Methods属性。

```
▣ Config Methods: 0x4388
  Data Element Type: Config Methods (0x1008)
  Data Element Length: 2
  Configuration Methods: 0x4388
    .... .... .... 0 = USB: 0x0000
    .... .... ...0. = Ethernet: 0x0000
    .... .... .0.. = Label: 0x0000
    .... .... 1... = Display: 0x0001
    ..0. .... .... = Virtual Display: 0x0000
    .1.. .... .... = Physical Display: 0x0001
    .... .... 0 .... = External NFC: 0x0000
    .... .... 0. .... = Internal NFC: 0x0000
    .... .... .0... .... = NFC Interface: 0x0000
    .... .... 1.... .... = Push Button: 0x0001
    .... ..1. .... .... = Virtual Push Button: 0x0001
    .... .0.. .... .... = Physical Push Button: 0x0000
    .... ..1 .... .... = Keypad: 0x0001
```

### 图6-13 Config Methods属性

- Type字段取值为0x1008，Length字段取值为2，代表Value的内容长度为2字节。
- Configuration Method的Value长度为2字节，共16位。每一位都代表Enrollee或Registrar支持的WSC配置方法。

图6-13所示为Galaxy Note 2所支持的Method，它支持动态PIN码（即STA能动态生成随机PIN码，由Display位表达。静态PIN码由Label位表示）。

注意 Display还需细分为Physical Display或Virtual Display。二者区别是Physical Display表示PIN码能直接显示在设备自带的屏幕上，而Virtual Display只能通过其他方式来查看（由于绝大多数无线路由器都没有屏幕，所以用一般情况下，用户只能在浏览器中通过设备页面来查看）PIN码。Keypad表示可在设备中输入PIN码。另外，是否支持Push Button由Push Button位表达。同PIN码一样，它也分Physical Push Button和Virtual Push Button。由于Galaxy Note 2硬件上并没有专门的按钮（它只不过是在软件中实现了一个按钮用来触发Push Button，读者可参考图6-2中左图的“WPS推送按钮”项），所以这里的设置为支持Virtual Push Button。

Primary Device Type属性代表设备的主类型。在Discovery Phase阶段，交互的一方可指定要搜索的设备类型（需设置Requested Device Type属性，该属性的结构和取值与Primary Device Type一样）。这样，只有那些Primary Device Type和目标设备类型匹配的一方才进行响应。图6-14为Galaxy Note 2的Primary Device Type属性。

WSC规范中，Primary Device Type的结构如图6-15所示属性。

□ Primary Device Type
Data Element Type: Primary Device Type (0x1054)
Data Element Length: 8
Primary Device Type: 000a0050f2040005
Category: Telephone (0x000a)
Subcategory: Smartphone - dual mode (0x0005)

图6-14 Primary Device Type属性

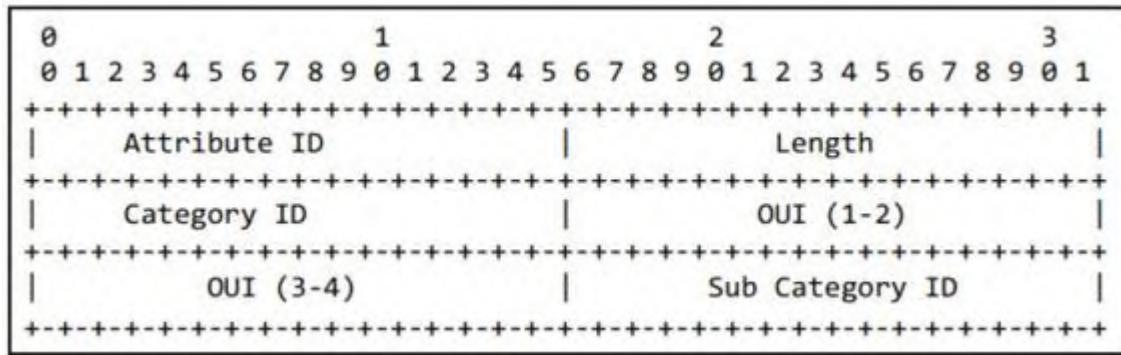


图6-15 Primary Device Type的组成

结合图6-14和图6-15可知。

- Type（即图6-15中的Attribute ID）字段的值为0x1054，Length字段的值为4字节。
- Category ID为WSC规范中定义的设备类型。Galaxy Note 2属于Telephone设备，取值为0x000a。
- OUI默认为WFA的OUI编号，取值为0x00-50-f2-04。注意，图6-14并未单独列举出OUI字段的取值。
- WSC还划分了Sub Category，Galaxy Note 2取值为0x0005，代表Smartphone-dual mode类设备。dual mode表示设备支持两个Wi-Fi频段（注意，规范并未说明dual mode的具体含义。此处的解释为笔者根据规范内容推断而来）。

**提示** 对AP来说，其Category取值为“Network Infrastructure”，Sub Category取值为“AP”。规范还定义了一个名为Secondary Device Type List的属性，该属性包含了设备支持的除主设备类型外的设备类型。具体的设备类型取值和Primary Device Type一样。Discovery Phase阶段中，Secondary Device Type List也可作为搜索匹配条件。

#### 4. Device Password ID和RF Bands属性

Device Password ID属性用于标示设备Password的类型，默认值是PIN（值为0x0000），代表Enrollee使用PIN码（静态或动态PIN码都可以）。如图6-16所示。

提示 Device Password ID其他可取值包括0x0001（User-Specified）、0x0002（Machine-Specified）、0x0003（Rekey）、0x0004（PushButton）等。

另一个比较重要的属性是RF Bands，如图6-17所示。

```
Device Password ID: PIN (default) (0x0000)
Data Element Type: Device Password ID (0x1012)
Data Element Length: 2
Device Password ID: PIN (default) (0x0000)
```

图6-16 Device Password ID属性

```
RF Bands: 2.4 and 5 GHz (0x03)
Data Element Type: RF Bands (0x103c)
Data Element Length: 1
RF Bands: 2.4 and 5 GHz (0x03)
```

图6-17 RF Bands属性

RF Bands代表设备所支持的无线频率。图6-17所示为Galaxy Note 2的RF Bands取值，其Wi-Fi芯片支持2.4GHz和5GHz两个频率。

提示 WSC规范定义的Attribute非常多，由于篇幅原因，本书不一一介绍。如有需要，可根据参考资料[4]来了解相关Attribute的信息。

根据前文所述，支持WSC的设备必须在一些802.11管理帧中设置WSC IE，而不同的管理帧中WSC IE的内容也不尽相同。

① 该值由WFA注册，可在IANA官方网站查询  
(<http://www.iana.org/assignments/enterprise-numbers/enterprise-numbers>)。

## 6.3.2 802.11管理帧WSC IE设置<sup>[4]</sup>

WSC规范规定，不同的管理帧中WSC IE必须包含一些特定属性，下面将简单介绍这些管理帧中WSC IE包含的属性及它们的作用。

### 1. Probe Request和Response帧设置

Probe Request帧中WSC IE包含的属性如表6-1所示。

表 6-1 Probe Request WSC IE 属性设置

属性名	必选 / 可选 / 条件	说 明
Version	必选项	见前文关于该属性的介绍
Request Type	必选项	见前文关于该属性的介绍
Configuration Method	必选项	见前文关于该属性的介绍
UUID-E	必选项	STA 的通用唯一标识符 ( Universally Unique Identifier, UUID )。WSC 规范中，Enrollee 和 Registrar 都有 UUID。为了区分，WSC 分别定义了 UUID-E 和 UUID-R 两个属性
Primary Device Type	必选项	见前文关于该属性的介绍
RF Bands	必选项	见前文关于该属性的介绍
Association State	必选项	表示 STA 和 AP 的关联状态。值 0 表示 “Not Associated”
Configuration Error	必选项	表示设备的配置和关联结果。值 0 表示 “No Error”
Device Password ID	必选项	见前文关于该属性的介绍
其他条件选项	采用 WPS 2.0 及以上版本	如果设备打算采用 WSC 及更高版本 ( 即 WPS 2.0 或 2.1 )，则必须包含属性：Manufacturer、Model Name、Model Number、Device Name、Version2 子属性 ( 取值为 0x20 表示版本 2.0，取值为 0x21 表示版本 2.1 )
其他可选项		Request to Enroll 子属性、Requested Device Type( 可用于设备搜索匹配 )

图6-18所示为Galaxy Note 2的Probe Request WSC IE的内容。

```
Tag: vendor specific: Microsoft: WPS
  Tag Number: vendor specific (221)
  Tag length: 133
  OUI: 00-50-f2 (Microsoft)
  Vendor Specific OUI Type: 4
  Type: WPS (0x04)
+ Version: 0x10
+ Request Type: Enrollee, open 802.1X (0x01)
+ Config Methods: 0x4388
+ UUID E
+ Primary Device Type
+ RF Bands: 2.4 and 5 GHz (0x03)
+ Association State: Not associated (0x0000)
+ Configuration Error: No Error (0x0000)
+ Device Password ID: PIN (default) (0x0000)
+ Manufacturer: samsung
+ Model Name: GT-N7100
+ Model Number: GT-N7100
+ Device Name: t03gzc
- Vendor Extension
  Data Element Type: vendor extension (0x1049)
  Data Element Length: 9
  Vendor Extension: 00372a000120030101
  Vendor ID: 14122
+ version2: 2.0
+ Request to Enroll: TRUE
```

图6-18 Probe Request WSC IE示例

表6-2为Probe Response帧WSC IE属性设置说明。

表 6-2 Probe Response WSC IE 属性设置

属性名	必选 / 可选 / 条件	说 明
Version	必选项	见前文关于该属性的介绍
WSC Configuration State (WPS 中, 该属性又名 “Wi-Fi Protected Setup State”)	必选项	该属性用于描述 AP 或 STA 的 WSC 配置情况。这些配置包括 SSID、加解密方法等。值 0x01 表示“Not Configured”，值 0x02 表示“Configured”。很显然，未完成 WSC 配置的 STA 中，该属性取值 0x01。而 AP 一般早就配置好了，所以 AP 取值为 0x02
Response Type	必选项	见前文关于该属性的介绍
UUID-E	必选项	AP 的 UUID。注意，对于 Registrar 来说，AP 也属于 Enrollee
Manufacturer	必选项	设备厂商名
Model Name	必选项	设备型号名称，用 ASCII 字符串表示
Model Number	必选项	设备编号，可作为 Model Name 的补充信息
Serial Number	必选项	设备序列号
Primary Device Type	必选项	见前文关于该属性的介绍
Device Name	必选项	设备名
Configuration Methods	必选项	见前文关于该属性的介绍
RF Bands	条件选项	支持双模的 AP 必须设置它
Version2 子属性	必选项	见前文关于该属性的介绍

图6-19所示为笔者家中无线路由器开启WSC功能后发送的Probe Response帧WSC IE信息内容。

```
☒ Tag: Vendor Specific: Microsoft: WPS
    Tag Number: vendor specific (221)
    Tag length: 123
    OUI: 00-50-f2 (Microsoft)
    vendor specific OUI Type: 4
    Type: WPS (0x04)
    ☒ Version: 0x10
    ☒ Wifi Protected Setup State: Configured (0x02)
    ☒ Selected Registrar: 0x00
    ☒ Response Type: AP (0x03)
    ☒ UUID E
    ☒ Manufacturer: Tenda
    ☒ Model Name: Tenda
    ☒ Model Number: 123456
    ☒ Serial Number: 198
    ☒ Primary Device Type
        Data Element Type: Primary Device Type (0x1054)
        Data Element Length: 8
        Primary Device Type: 00060050f2040001
        Category: Network Infrastructure (0x0006)
        Subcategory: AP (0x0001)
    ☒ Device Name: Tenda Wireless AP
    ☒ Config Methods: 0x0084
    ☒ RF Bands: 2.4 GHz (0x01)
```

图6-19 Probe Response WSC IE示例

图6-19中还包含了一个名为Selected Registrar的属性，该属性的作用是，当AP已选择合适的Registrar后，该属性值为0x01，否则为0x00。对于Standalone AP来说，如果其内部的Registrar组件启动，则设置该值为0x01。

**提示** 由图6-19的示例可知，笔者使用的AP仅支持WPS，而不支持WSC。

## 2. Association Request/Response和Beacon帧设置

Association Request/Response帧的WSC IE设置比较简单，如表6-3所示。

表 6-3 Association Request/Response WSC IE 属性设置

属性名	必选 / 可选 / 条件	说 明
Version	必选项	见前文关于该属性的介绍
Request/Response Type	必选项	见前文关于该属性的介绍
Version 2	条件项	见前文关于该属性的介绍
其他可选项	可选项	

Association Request帧示例如图6-20所示。

```
☒ Tag: vendor specific: Microsoft: WPS
    Tag Number: vendor specific (221)
    Tag length: 24
    OUI: 00-50-f2 (Microsoft)
    vendor specific OUI Type: 4
    Type: WPS (0x04)
☒ Version: 0x10
☒ Request Type: Enrollee, open 802.1X (0x01)
☒ Vendor Extension
    Data Element Type: vendor Extension (0x1049)
    Data Element Length: 6
    Vendor Extension: 00372a000120
    Vendor ID: 14122
☒ Version2: 2.0
```

图6-20 Association RequestWSC IE属性

Beacon帧WSC IE属性比较多，如表6-4所示。

表 6-4 Beacon WSC IE 属性设置

属性名	必选 / 可选 / 条件	说 明
Version	必选项	见前文关于该属性的介绍
WSC Configuration State	必选项	见表 6-2
AP Setup Locked	条件项	该属性值为 TRUE 表示 AP 已经锁定，即不能开展 RP 协议。以 PIN 方法为例，如果三次 PIN 方法失败后，AP 将锁定 60 秒。如果 AP 被锁定，则 Beacon 帧必须包含该属性
Selected Registrar	条件项	见前文关于该属性的介绍。如果该属性的值为 TRUE，则必须包含该属性
Device Password ID	条件项	见前文关于该属性的介绍。如果包含了 Selected Registrar 属性，则必须包含本属性
Selected Registrar Configuration Methods	条件项	代表 Registrar 支持的配置方法。如果包含了 Selected Registrar 属性，则必须包含本属性
UUID	条件项	如果 AP 支持 dual mode，并且当前使用的是 Push Button 方法，则必须包含本属性

(续)

属性名	必选 / 可选 / 条件	说 明
RF Bands	条件项	见前文关于该属性的介绍
Version 2	条件项	见前文关于该属性的介绍
AuthorizedMACs 子属性	条件项	通过 Vendor Extension 属性包含。该子属性列出了当前已经注册（表示 Enrollee 已经注册到 Registrar 中）的 STA 的 MAC 地址。只有那些在 MAC 地址列表中的 STA 才可开展 EAP-WSC 流程。注意，已注册的 STA MAC 地址由 Registrar 设置给 AP。如果 Registrar 不支持该功能话，则不能使用该属性

笔者家中的无线路由器功能比较简单，其Beacon帧WSC IE仅包含了必选项属性，如图6-21所示。

```

☒ Tag: Vendor Specific: Microsoft: WPS
    Tag Number: Vendor Specific (221)
    Tag length: 14
    OUI: 00-50-f2 (Microsoft)
    Vendor specific OUI Type: 4
    Type: WPS (0x04)
☒ Version: 0x10
☒ wifi Protected Setup State: Configured (0x02)

```

图6-21 Beacon WSC IE属性

介绍完WSC IE，接着来看EAP-WSC流程。

### 6.3.3 EAP-WSC介绍<sup>[4][5]</sup>

由图6-7所示的RP协议交互流程可知，Discovery Phase阶段之后，STA和AP将通过EAP包交换来完成安全信息协商。WSC规范利用EAP的扩展功能新定义了一种EAP算法，即EAP-WSC。EAP-WSC的包格式如图6-22所示。

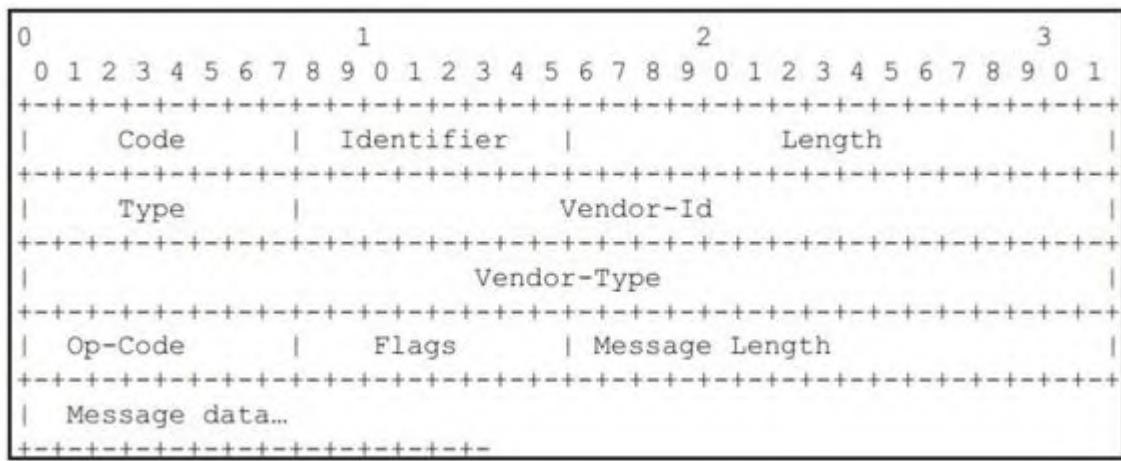


图6-22 EAP-WSC包格式

对于EAP-WSC来说，图6-22中各字段取值情况如下。

- Type取值为254，代表EAP包中的内容由Vendor定义。
- 对于WSC来说，Vendor-Id取值为0x00372a，Vendor Type取值为0x0000-0001（WFA中，该值表示Simple-Config）。
- Op-Code及以后的内容由EAP-WSC定义，其取值有六种情况（见表6-5）。
- Flags包含两个标志位，一个是MF标志位（More Fragments，取值为0x01，代表EAP分片包），另一个是LF标志位（Length Field，值为0x02）。
- 如果LF标志位被设置，则Message Length字段存在。该字段表示Message data的长度。

- Message data为WSC定义的Attribute。和WSC IE类似，WSC规范对不同EAP-WSC包携带的Attribute有严格要求。

表6-5所示为EAP-WSC Op-Code的取值。

表 6-5 EAP-WSC Op-Code 取值说明

Op-Code 值	名 称	说 明
0x01	WSC_Start	该 WSC_Start 包由 AP 发送。当 AP 收到来自 STA 的 EAP-Response/Identity 包且该包 Identity 取值为“WFA-SimpleConfig-Enrollee-1-0”时（该 Identity 表明 STA 希望使用 EAP-WSC 算法），它将发送 WSC_Start 以开始 EAP-WSC 流程
0x02	WSC_ACK	当 AP 或 STA 收到并处理来自对端的 EAP-WSC 包，但没有要回复的信息时，它将发送 WSC_ACK 包给对方
0x03	WSC_NACK	当 STA 或 AP 在 EAP-WSC 时发现错误时，它们将向对方发送 WSC_NACK 包
0x04	WSC_MSG	安全信息协商的内容存储在 WSC_MSG 包中。图 6-7 中，M1 ~ M8 都属于 WSC_MSG 包。下文将详细介绍 M1 ~ M8 相关的知识
0x05	WSC_DONE	由 STA 发送，当它处理完 RF 流程最后一个消息（即 M8）时发送。它表明 STA 已经成功获取安全配置信息
0x06	WSC_FRAG_ACK	当 STA 或 AP 收到并处理分片消息成功后发送此消息

EAP-WSC和第4章介绍的4-Way Handshake类似，STA和AP双方要派生一些Key用于加密所传输的信息。

在STA和AP双方开展EAP-WSC流程前，AP需要确定STA的Identity以及使用的身份验证算法。该过程涉及三次EAP包交换（参考图3-38）。这三次包交换的内容分别如下。

- AP发送EAP-Request/Identity以确定STA的ID。
- 对于打算使用WSC认证方法的STA来说，它需要在回复的EAP-Response/Identity包中设置Identity为“WFA-SimpleConfig-Enrollee-1-0”。
- AP确定STA的Identity为“WFA-SimpleConfig-Enrollee-1-0”后，将发送EAP-Request/WSC\_Start包以启动EAP-WSC认证流程。这个流程讲涉及M1~M8相关的知识。下面我们将结合实例来介绍。

注意 这些知识也是后续分析代码时的理论依据，请读者务必认真体会。

## 1. M1和M2

M1消息由STA发送给AP。图6-23所示为Galaxy Note 2发送的M1消息。

如前文所述，EAP-WSC消息的组成结构也是一个一个Attribute。图6-23所示的大部分Attribute在前文都已见过了，此处仅介绍黑框中所列的几个Attribute。

```
[-] Extensible Authentication Protocol
    Code: Response (2)
    Id: 1
    Length: 416
    Type: Expanded Type (254)
[-] Expanded Type (wifi Alliance, WiFiProtectedSetup)
    EAP-EXT Vendor Id: WFA (0x372a)
    EAP-EXT Vendor Type: simpleConfig (0x01)
    Opcode: WSC Msg (4)
    [-] Flags: 0x00
    [-] Version: 0x10
    [-] Message Type: M1 (0x04)
    [-] UUID E
    [-] MAC Address
[-] Enrollee Nonce
    Data Element Type: Enrollee Nonce (0x101a)
    Data Element Length: 16
    Enrollee Nonce: b4fc972f845183265ef7c39b72b5ee68
[-] Public Key
    Data Element Type: Public Key (0x1032)
    Data Element Length: 192
    Public Key: 6666665c9b474740e9df7248b2158e84be8e833da8904c6e...
[-] Authentication Type Flags: 0x003b
[-] Encryption Type Flags: 0x000d
[-] Connection Type Flags: ESS (0x01)
    [-] Config Methods: 0x4388
    [-] wifi Protected Setup State: Not configured (0x01)
    [-] Manufacturer: samsung
    [-] Model Name: GT-N7100
    [-] Model Number: GT-N7100
    [-] Serial Number: 4df13b8d0429af5b
    [-] Primary Device Type
    [-] Device Name: t03gzc
    [-] RF Bands: 2.4 and 5 GHz (0x03)
    [-] Association state: Not associated (0x0000)
    [-] Device Password ID: PIN (default) (0x0000)
    [-] Configuration Error: No Error (0x0000)
    [-] OS Version: 0x80000000
    [-] Vendor Extension
```

图6-23 M1消息示例

- Message Type: 代表Enrollee和Registrar发送的消息类型，其可取值从0x01（代表Beacon帧）到0x0F（代表WSC\_DONE）。该属性一般只

在EAP-WSC帧中用到。对于M1消息而言，其Message Type取值为0x04。

- **UUID-E**: 代表STA的UUID。MAC Address代表STA的MAC地址。
- **Enrollee Nonce**: 代表STA产生的一串随机数，它用于后续的密钥派生等工作。
- **Public Key**: STA和AP的密钥派生源头也是PMK。不过和第4章介绍的PSK不同的是，在WSC PIN法中并没有使用PSK（PIN码的作用不是PSK）。双方采用了Diffie-Hellman<sup>[6]</sup>（D-H）密钥交换算法。该算法使得通信的双方可以用这个方法确定对称密钥。注意，D-H算法只能用于密钥的交换，而不能进行消息的加密和解密。通信双方确定要用的密钥后，要使用其他对称密钥操作加密算法以加密和解密消息。Public Key属性包含了Enrollee的D-H Key值。
- **Authentication Type Flags**和**Encryption Type Flags**: 表示Enrollee支持的身份验证算法以及加密算法类型。
- **Connection Type Flags**: 代表设备支持的802.11网络类型，值0x01代表ESS，值0x02代表IBSS。

图6-24所示为Galaxy Note 2中Authentication Type Flags和Encryption Type Flags两个属性的取值情况。

Authentication Type Flags: 0x003b Data Element Type: Authentication Type Flags (0x1004) Data Element Length: 2 Authentication Type Flags: 0x003b .... .... .1 = Open: 0x0001 .... .... .1. = WPA_PSK: 0x0001 .... .... ..0.. = Shared: 0x0000 .... .... .1... = WPA: 0x0001 .... .... .1.... = WPA2: 0x0001 .... .... .1.... = WPA2PSK: 0x0001	Encryption Type Flags: 0x000d Data Element Type: Encryption Type Flags (0x1010) Data Element Length: 2 Encryption Type Flags: 0x000d .... .... .1 = None: 0x0001 .... .... ..0.. = WEP: 0x0000 .... .... .1.. = TKIP: 0x0001 .... .... .1... = AES: 0x0001
---	---

图6-24 Authentication/Encryption Type Flags取值示例

图6-24左图所示为Authentication Type Flags的取值情况。其中，“WPA”和“WPA2”标志位是“WPA-Enterprise”以及“WPA2-Enterprise”之意。

AP收到并处理M1后，将回复M2。M2的内容如图6-25所示。

```
■ Extensible Authentication Protocol
  Code: Request (1)
  Id: 2
  Length: 414
  Type: Expanded Type (254)
  └─ Expanded Type (wifi Alliance, wifiProtectedSetup)
    EAP-EXT Vendor Id: WFA (0x372a)
    EAP-EXT Vendor Type: SimpleConfig (0x01)
    Opcode: WSC Msg (4)
    └─ Flags: 0x00
    └─ Version: 0x10
    └─ Message Type: M2 (0x05)
    └─ Enrollee Nonce
      └─ Registrar Nonce
        Data Element Type: Registrar Nonce (0x1039)
        Data Element Length: 16
        Registrar Nonce: 7e65cc79719e54a106aac4d98af71373
    └─ UUID R
    └─ Public Key
      Data Element Type: Public Key (0x1032)
      Data Element Length: 192
      Public Key: 562c4b17fd68a1ab00d352164f52f72d17ee7d2c81c2e3a8..
    └─ Authentication Type Flags: 0x0027
    └─ Encryption Type Flags: 0x000f
    └─ Connection Type Flags: ESS (0x01)
    └─ Config Methods: 0x0084
    └─ Manufacturer: Tenda
    └─ Model Name: Tenda
    └─ Model Number: 123456
    └─ Serial Number: 198
    └─ Primary Device Type
    └─ Device Name: Tenda wireless AP
    └─ RF Bands: 2.4 GHz (0x01)
    └─ Association State: Not associated (0x0000)
    └─ Configuration Error: No Error (0x0000)
    └─ Device Password ID: PIN (default) (0x0000)
    └─ OS version: 0x80000000
  └─ Authenticator
    Data Element Type: Authenticator (0x1005)
    Data Element Length: 8
    Authenticator: f87402ff3109781f (1st 64 bits of HMAC)
```

图6-25 M2消息示例

图6-25中，AP的M2将携带以下信息。

- Registrar Nonce: Registrar生成的随机数。
- Public Key: D-H算法中，Registrar一方的D-H Key值。

- Authenticator: 由HMAC-SHA-256及AuthKey（详情见下文）算法得来一个256位的二进制串。注意，Authenticator属性只包含其中的前64位二进制内容。

提示 AP发送M2之前，会根据Enrollee Nonce、Enrollee MAC以及Registrar Nonce并通过D-H算法计算一个KDK（Key Derivation Key），KDK密钥用于其他三种Key派生，这三种Key分别用于加密RP协议中的一些属性的AuthKey（256位）、加密Nonce和ConfigData（即一些安全配置信息）的KeyWrapKey（128位）以及派生其他用途Key的EMSK（Extended Master Session Key）。

## 2. M3和M4

STA处理完M2消息后，将回复M3消息，其内容如图6-26所示。

```

[+] Extensible Authentication Protocol
  Code: Response (2)
  Id: 2
  Length: 138
  Type: Expanded Type (254)
[+] Expanded Type (wifi Alliance, wifiProtectedSetup)
  EAP-EXT Vendor Id: WFA (0x372a)
  EAP-EXT Vendor Type: SimpleConfig (0x01)
  Opcode: WSC Msg (4)
  [+] Flags: 0x00
  [+] Version: 0x10
  [+] Message Type: M3 (0x07)
  [-] Registrar Nonce
    Data Element Type: Registrar Nonce (0x1039)
    Data Element Length: 16
    Registrar Nonce: 7e65cc79719e54a106aac4d98af71373
  [-] E Hash1
    Data Element Type: E Hash1 (0x1014)
    Data Element Length: 32
    Enrollee Hash 1: b032496740f15fd40dbfc34cbdf3eb4204f3dbfc3357c51...
  [-] E Hash2
    Data Element Type: E Hash2 (0x1015)
    Data Element Length: 32
    Enrollee Hash 2: 3610b09bd61894e79d0dfd19a19b38a38723da23f92a0e83...
  [+] Vendor Extension
  [-] Authenticator
    Data Element Type: Authenticator (0x1005)
    Data Element Length: 8
    Authenticator: db609cd6b23e91d8 (1st 64 bits of HMAC)

```

图6-26 M3消息示例

图6-26中：

- Registrar Nonce值来源于M2的Registrar Nonce属性。
- E Hash1和E Hash2属性的计算比较复杂，详情见下文。
- Authenticator是STA利用AuthKey（STA收到M2的Registrar Nonce后也将计算一个AuthKey）计算出来的一串二进制位。

根据WSC规范，E Hash1和E Hash2的计算过程如下。

- 1) 利用AuthKey和PIN码利用HMAC算法分别生成两个PSK。其中，PSK1由PIN码前半部分生成，PSK2由PIN码后半部分生成。
- 2) 利用AuthKey对两个新随机数128 Nonce进行加密，以得到E-S1和E-S2。
- 3) 利用HMAC算法及AuthenKey分别对(E-S1、PSK1、STA的D-H Key和AP的D-H Key)计算得到E Hash1。E Hash2则由E-S2、PSK2、STA的D-H Key和AP的D-H Key计算而来。

AP收到并处理完M3后将回复M4，其内容如图6-27所示。

```

Extensible Authentication Protocol
  Code: Request (1)
  Id: 3
  Length: 196
  Type: Expanded Type (254)
Expanded Type (Wifi Alliance, wifiProtectedSetup)
  EAP-EXT Vendor Id: WFA (0x372a)
  EAP-EXT Vendor Type: simpleconfig (0x01)
  Opcode: WSC Msg (4)
  Flags: 0x00
  Version: 0x10
  Message Type: M4 (0x08)
Enrollee Nonce
  Data Element Type: Enrollee Nonce (0x101a)
  Data Element Length: 16
  Enrollee Nonce: b4fc972f845183265ef7c39b72b5ee68
  R Hash1
  R Hash2
Encrypted Settings
  Data Element Type: Encrypted Settings (0x1018)
  Data Element Length: 64
  Encrypted Settings: e9d7f034bba51a69cd1f2762599110046a70ffcd09fd17d8...
Authenticator
  Data Element Type: Authenticator (0x1005)
  Data Element Length: 8
  Authenticator: fb2d517068569e96 (1st 64 bits of HMAC)

```

图6-27 M4消息示例

由图6-27可知：

- AP将计算R Hash1和R Hash2。其使用的PIN码为用户通过AP设置界面输入的PIN码。很显然，如果AP设置了错误PIN码的话，STA在比较R Hash 1/2和E Hash 1/2时就会发现二者不一致，从而可终止EAP-WSC流程。
- Encrypted Settings为AP利用KeyWrapKey加密R-S1得到的数据。

### 3. M5和M6

M5消息和M4消息类似，如图6-28所示。

```
Extensible Authentication Protocol
  Code: Response (2)
  Id: 3
  Length: 134
  Type: Expanded Type (254)
    Expanded Type (wifi Alliance, wifiProtectedsetup)
      EAP-EXT Vendor Id: WFA (0x372a)
      EAP-EXT Vendor Type: simpleConfig (0x01)
      Opcode: wsc Msg (4)
      Flags: 0x00
      Version: 0x10
      Message Type: M5 (0x09)
      Registrar Nonce
        Data Element Type: Registrar Nonce (0x1039)
        Data Element Length: 16
        Registrar Nonce: 7e65cc79719e54a106aac4d98af71373
      Encrypted Settings
        Data Element Type: Encrypted Settings (0x1018)
        Data Element Length: 64
        Encrypted Settings: 64d80549bea5494d40bcab02414379a2959668375e9768c6...
      Vendor Extension
      Authenticator
```

图6-28 M5消息示例

图6-28所示的Encrypted Settings为STA利用KeyWrapKey加密E-S1得来。

M6消息如图6-29所示。

```

[+] Extensible Authentication Protocol
  Code: Request (1)
  Id: 4
  Length: 124
  Type: Expanded Type (254)
[-] Expanded Type (wifi Alliance, wifiProtectedsetup)
  EAP-EXT Vendor Id: WFA (0x372a)
  EAP-EXT Vendor Type: simpleConfig (0x01)
  Opcode: WSC Msg (4)
  [+]
    Flags: 0x00
    [+]
      Version: 0x10
    [+]
      Message Type: M6 (0x0a)
    [+]
      Enrollee Nonce
    [-] Encrypted Settings
      Data Element Type: Encrypted Settings (0x1018)
      Data Element Length: 64
      Encrypted Settings: c35d33ced3cca8cccd2acf586fbf2d41d3a95a2c27f796ff4...
    [+]
      Authenticator

```

图6-29 M6消息示例

图6-29所示的M6消息中，Encryped Settings为AP利用KeyWrapKey加密R-S2而来。

#### 4. M7和M8

由M5、M6的内容可知，STA的M7将发送利用KeyWrapKey加密E-S2的信息给AP以进行验证，如图6-30所示。

```

[+] Extensible Authentication Protocol
  Code: Response (2)
  Id: 4
  Length: 134
  Type: Expanded Type (254)
[-] Expanded Type (wifi Alliance, wifiProtectedsetup)
  EAP-EXT Vendor Id: WFA (0x372a)
  EAP-EXT Vendor Type: simpleConfig (0x01)
  Opcode: WSC Msg (4)
  [+]
    Flags: 0x00
    [+]
      Version: 0x10
    [+]
      Message Type: M7 (0x0b)
    [-] Registrar Nonce
      Data Element Type: Registrar Nonce (0x1039)
      Data Element Length: 16
      Registrar Nonce: 7e65cc79719e54a106aac4d98af71373
    [-] Encrypted Settings
      Data Element Type: Encrypted Settings (0x1018)
      Data Element Length: 64
      Encrypted Settings: 605ab4afe8c1a51ce912596d64642eab2662b1ce34d3df47...
    [+]
      Vendor Extension
    [+]
      Authenticator

```

### 图6-30 M7消息示例

当AP确定M7消息正确无误后，它将发送M8消息，而M8将携带至关重要的安全配置信息，如图6-31所示。

```
② Expanded Type (wifi Alliance, wifiProtectedsetup)
  EAP-EXT Vendor Id: wFA (0x372a)
  EAP-EXT Vendor Type: simpleConfig (0x01)
  Opcode: WSC Msg (4)
  ④ Flags: 0x00
  ④ Version: 0x10
  ④ Message Type: M8 (0x0c)
  ④ Enrollee Nonce
  ④ Encrypted Settings
    Data Element Type: Encrypted settings (0x1018)
    Data Element Length: 112
    Encrypted Settings: 29940630e408a6ef095ccbbae41f15f67c5493bf857c6865...
  ④ Authenticator
```

### 图6-31 M8消息示例

图6-31中，安全配置信息保存在Encrypted Settings中，它由KeyWrapKey加密。WSC规范规定，当Enrollee为STA时（对Registrar来说，AP也是Enrollee），Encrypted Settings将包含若干属性，其中最重要的就是Credential属性集合，该属性集的内容如表6-6所示。

表 6-6 Credential 属性集合的内容

属性名	必选 / 可选 / 条件	说明
Network Index	必选项	废弃。为保持兼容性，该属性值必须设置为 1
SSID	必选项	AP 的 SSID
Authentication Type	必选项	AP 选择的认证类型
Encryption Type	必选项	AP 选择的加密类型
Network Key	必选项	对 WPA/WPA2-PSK 来说，该属性保存了 PSK
MAC Address	必选项	AP 的 MAC 地址
其他条件选项	条件选项	EAP Type 等

由表6-6可知，当STA收到M8并解密其中的Credential属性集合后，将得到AP的安全设置信息。很显然，如果不使用WSC，用户需要手动设置这些信息。使用了WSC后，这些信息将在M8中由AP发送给STA。

接下来，STA就可以利用这些信息加入AP对应的目标无线网络了。

## 5. EAP-WSC总结

EAP-WSC M1~M8一共涉及8次EAP包交换，每次帧交换的内容如图6-32所示。

Enrollee → Registrar: $M_1 = \text{Version} \parallel N_1 \parallel \text{Description} \parallel PK_E$
Enrollee ← Registrar: $M_2 = \text{Version} \parallel N_1 \parallel N_2 \parallel \text{Description} \parallel PK_R$ [    ConfigData ]    $\text{HMAC}_{\text{AuthKey}}(M_1 \parallel M_2)$
Enrollee → Registrar: $M_3 = \text{Version} \parallel N_2 \parallel E\text{-Hash1} \parallel E\text{-Hash2} \parallel$ $\text{HMAC}_{\text{AuthKey}}(M_2 \parallel M_3)$
Enrollee ← Registrar: $M_4 = \text{Version} \parallel N_1 \parallel R\text{-Hash1} \parallel R\text{-Hash2} \parallel$ $\text{ENC}_{\text{KeyWrapKey}}(\text{R-S1}) \parallel \text{HMAC}_{\text{AuthKey}}(M_3 \parallel M_4)$
Enrollee → Registrar: $M_5 = \text{Version} \parallel N_2 \parallel \text{ENC}_{\text{KeyWrapKey}}(\text{E-S1}) \parallel$ $\text{HMAC}_{\text{AuthKey}}(M_4 \parallel M_5)$
Enrollee ← Registrar: $M_6 = \text{Version} \parallel N_1 \parallel \text{ENC}_{\text{KeyWrapKey}}(\text{R-S2}) \parallel$ $\text{HMAC}_{\text{AuthKey}}(M_5 \parallel M_6)$
Enrollee → Registrar: $M_7 = \text{Version} \parallel N_2 \parallel \text{ENC}_{\text{KeyWrapKey}}(\text{E-S2}$ [    ConfigData ]    $\text{HMAC}_{\text{AuthKey}}(M_6 \parallel M_7)$
Enrollee ← Registrar: $M_8 = \text{Version} \parallel N_1 \parallel [\text{ENC}_{\text{KeyWrapKey}}(\text{ConfigData})] \parallel$ $\text{HMAC}_{\text{AuthKey}}(M_7 \parallel M_8)$

图6-32 EAP-WSC帧交换内容

图6-32对前面几小节所提到的EAP-WSC帧内容进行了简化，其中：

- Description代表UUID、Manufacturer、MAC地址等信息。
- $PK_E$  和  $PK_R$  代表D-H算法的Enrollee方的Key以及Registrar方的Key。
- $M_x^*$  代表没有包含HMAC-SHA-256结果的第X次消息内容。
- $\text{HMAC}_{\text{AuthenKey}}$  代表利用AuthenKey和HMAC-SHA-256算法进行计算。
- $\text{ENC}_{\text{KeyWrapKey}}$  代表利用KeyWrapKey进行加密。

- “[...]” 中的内容为可选信息。
- N1和N2分别代表Enrollee和Registrar的Nonce。

STA处理完M8消息后，将回复WSC\_DONE消息给AP，表示自己已经成功处理M8消息。接下来的工作就如图6-7所示一样。

- AP发送EAP-FAIL以及Deauthentication帧给STA。STA收到该帧后将取消和AP的关联。
- STA将重新扫描周围的无线网络。由于STA以及获取了AP的配置信息，所以它可以利用这些信息加入AP所在的无线网络。

以上对WSC理论知识进行了一番介绍。其中有一些知识点请读者注意。

- WSC的组成结构。规范中定义了AP、Enrollee和Registrar三大组件。日常生活中比较常见的实体是作为Enrollee的智能手机，以及集成了AP和Registrar功能的无线路由器（Standalone AP）。
- WSC拓展了802.11 IE的内容，而WSC IE包含了由WSC定义的不同Attribute。了解这些Attribute的作用对于理解WSC非常重要。另外，规范还对管理帧包含什么样的Attribute有严格规定。
- STA和Standalone AP使用RP协议交互的流程如图6-7所示。另外，请读者掌握EAP-WSC M1到M8帧包含的属性及作用。

注意 完整的WSC规范所包含的知识点比本节阐述得要多。在此，建议读者先学完本章内容后再去研读WSC规范。

下面来看Android中WSC相关的实现代码。如果读者真正掌握本节所示知识点的话，下面一节的学习过程将非常轻松。

## 6.4 WSC代码分析

本节将介绍Android平台中WSC的实现。和第4章、第5章介绍的Wi-Fi一样，WSC的相关代码将贯穿App层（主要是Settings应用）、Framework层（主要是WifiStateMachine）和wpa\_supplicant。本节将按如下流程介绍WSC。

- 先介绍App层和Framework层的处理，这部分代码相对简单。
- 重点介绍WPAS中的WSC实现。这一部分的难点在于EAP状态机以及EAP-WSC算法。

## 6.4.1 Settings中的WSC处理

本节内容从Settings开始，当选择图6-2左图所示的“WPS PIN条目”后，将弹出图6-2右图所示的WPS设置对话框。该对话框对应的类是WpsDialog中，其代码如下所示。

[-->WpsDialog.java: : WpsDialog]

```
public WpsDialog(Context context, int wpsSetup) {
    super(context);
    mContext = context;
    mWpsSetup = wpsSetup;
    // PIN和PBC方法都在这个WpsDialog中实现
    class WpsListener implements WifiManager.WpsListener {
        public void onStartSuccess(String pin) {
            if (pin != null) {
                // WSC PIN启动成功，WPAS将动态创建一个PIN码，此
                处由WpsDialog显示出来
                updateDialog(DialogState.WPS_START,
                String.format(
                    mContext.getString(R.string.wifi_wps_onstart_pin), pin));
            } else { // PBC处理
                updateDialog(DialogState.WPS_START,
                mContext.getString(
                    R.string.wifi_wps_onstart_pbc));
            }
        }
        public void onCompletion() {
            updateDialog(DialogState.WPS_COMPLETE,
            mContext.getString(R.string.wifi_wps_complete));
        }

        public void onFailure(int reason) {
            .....
        }
    }
    mWpsListener = new WpsListener(); // 监听WPS操作的结果
    mFilter = new IntentFilter();
    // 监听NETWORK_STATE_CHANGED_ACTION广播
```

```

mFilter.addAction(WifiManager.NETWORK_STATE_CHANGED_ACTION);
    mReceiver = new BroadcastReceiver() {
        public void onReceive(Context context, Intent
intent) {
            handleEvent(context, intent); // handleEvent的代
码比较简单，读者不妨自行阅读。
        }
    };
}

```

我们跳过WpsDialog的onCreate函数直接进入其onStart函数，在该函数中，WpsDialog将借助WifiManager和WifiStateMachine交互，相关代码如下所示。

[-->WpsDialog.java: : onStart]

```

protected void onStart() {
    mTimer = new Timer(false);
    mTimer.schedule(new TimerTask() {
        .....// 更新进度条
    } , 1000, 1000);

    mContext.registerReceiver(mReceiver, mFilter);

    WpsInfo wpsConfig = new WpsInfo(); // 构造一个WpsInfo对象
    wpsConfig.setup = mWpsSetup; // 对PIN来说，mWpsSetup的值为
    WpsInfo.DISPLAY
    // 调用WifiManager的startWps函数，该函数内部将向WifiStateMachine
    发送START_WPS消息
    mWifiManager.startWps(wpsConfig, mWpsListener);
}

```

当手机通过WPS成功加入无线网络后，NETWORK\_STATE\_CHANGED\_ACTION广播将在handleEvent中被处理。这部分代码非常简单，请读者自行阅读。

根据代码中的注释可知，startWps将向WifiStateMachine发送START\_WPS消息。WifiStateMachine该如何处理它呢？

## 6.4.2 WifiStateMachine的处理

结合第5章对WifiService相关内容的介绍，假设当前STA没有连接上目标AP，则WifiStateMachine处于DisconnectedState。不过，DisconnectedState并不处理START\_WPS消息，但其父状态ConnectedModeState会处理该消息，故直接来看ConnectedModeState的处理流程。

### 1. START\_WPS处理流程分析

代码如下。

```
[-->WifiStateMachine.java: : ConnectedModeState:  
processMessage]  
  
public boolean processMessage(Message message) {  
    StateChangeResult stateChangeResult;  
    switch(message.what) {  
        ....  
        case WifiManager.START_WPS:  
            WpsInfo wpsInfo = (WpsInfo) message.obj;  
            WpsResult result;  
            switch (wpsInfo.setup) {  
                case WpsInfo.PBC:  
                    result = mWifiConfigStore.startWpsPbc(wpsInfo);  
                    break;  
                .... // WpsInfo.Keypad的处理  
                case WpsInfo.DISPLAY:// 对PIN来说, setup的值为DISPLAY  
                    // 调用WifiConfigStore的startWpsWithPinFromDevice  
                    函数  
                    result =  
mWifiConfigStore.startWpsWithPinFromDevice(wpsInfo);  
                    break;  
                default:  
                    result = new WpsResult(Status.FAILURE);  
                    break;  
            }  
            if (result.status == Status.SUCCESS) {  
                // 回复wifiManager, 界面框中将显示动态PIN码  
                replyToMessage(message,  
WifiManager.START_WPS_SUCCEEDED, result);  
    }
```

```

        transitionTo(mWpsRunningState); // 转入WpsRunningState
    }
    .....// 错误处理
    break;
    .....
}
return HANDLED;
}

```

来看startWpsWithPinFromDevice函数，代码如下所示。

[-->WifiConfigStore.java: : startWpsWithPinFromDevice]

```

WpsResult startWpsWithPinFromDevice(WpsInfo config) {
    WpsResult result = new WpsResult();

    /*
        config.BSSID代表目标AP的MAC地址，此处为空。下面的
        startWpsPinDisplay函数将发送
        "WPS_PIN any"命令给WPAS。WPAS将计算一个动态PIN码返回给用户。这个
        PIN码也就是
        图6-2右图所示的PIN码。
    */
    result.pin = mWifiNative.startWpsPinDisplay(config.BSSID);
    if (!TextUtils.isEmpty(result.pin)) { // WPAS必须返回一
        // 个PIN码
        markAllNetworksDisabled(); // 停止使用其他网
        // 络
        result.status = WpsResult.Status.SUCCESS;
    } else
        result.status = WpsResult.Status.FAILURE;
    return result;
}

```

当WPAS成功返回PIN码后，WifiStateMachine将从DisconnectedState状态进入WpsRunningState。该状态的enter函数没有做什么有意义的工作。

## 2. WPS\_SUCCESS\_EVENT处理流程分析

当WPAS完成WSC流程后，它将发送WPS-SUCCESS给WifiMonitor，而WifiMonitor将发送WPS\_SUCCESS\_EVENT给WifiStateMachine。该消息

将由WpsRunningState状态处理，相关代码如下所示。

```
[-->WifiStateMachine.java: : WpsRunningState: processMessage]

public boolean processMessage(Message message) {
    switch (message.what) {
        case WifiMonitor.WPS_SUCCESS_EVENT: // 收到来自WPAS的
            WPS成功消息
                // 回复WifiManager。如此，WpsDialog中WpsListener对
                // 象的onCompleted函数将被调用
                replyToMessage(mSourceMessage,
                    WifiManager.WPS_COMPLETED);
                mSourceMessage.recycle();
                mSourceMessage = null;
                transitionTo(mDisconnectedState); // 转入
                DisconnectedState
                break;
        .....
    }
}
```

如果WSC流程一切顺利，WifiStateMachine将从WpsRunningState重新进入DisconnectedState。WifiStateMachine以后的流程就和5.3.2节所述的流程完全一样了。

**提示** WifiStateMachine内部也会发起扫描请求，这和第5章分析的流程略有不同。第5章中，扫描请求由WifiSettings发起。

由上文介绍可知，Android App层以及Framework WifiService相关模块对WSC的处理非常简单。它们将通过发送“WPS\_PIN any”命令以触发WPAS开始WSC的处理流程。下面分析WPAS中WSC的处理。

### 6.4.3 wpa\_supplicant中的WSC处理

WPAS中，WSC的处理就不像在App及Framework WifiService中那么简单。先来看WSC的初始化流程。

#### 1. WSC模块初始化

WSC模块的初始化工作位于wpa\_supplicant\_init\_iface（不熟悉的读者请参考4.3.4节“wpa\_supplicant\_init\_iface分析之五”）函数的最后几行中，相关代码如下所示。

```
[-->wpa_supplicant.c: : wpa_supplicant_init_iface]

static int wpa_supplicant_init_iface(struct wpa_supplicant
*wpa_s,
                                      struct wpa_interface *iface)
{
    .....// 其他代码。可参考4.3.4节“wpa_supplicant_init_iface分析之五”
    // 调用wpas_wps_init函数初始化WSC相关模块
    if (wpas_wps_init(wpa_s))  return -1;

    if (wpa_supplicant_init_eapol(wpa_s) < 0)  return -1;
    wpa_sm_set_eapol(wpa_s->wpa, wpa_s->eapol);
    .....
}
```

注意，在WPAS代码中WSC称为WPS2。为了行文方便，以后将不再区分WPS和WSC。

wpas\_wps\_init的代码如下所示。

```
[-->wps_supplicant.c: : wpas_wps_init]

int wpas_wps_init(struct wpa_supplicant *wpa_s)
{
    struct wps_context *wps;      // wps_context是WPS模块的核心数据
结构
    // wps_registrar_config代表Registrar的配置信息
    struct wps_registrar_config rcfg;
```

```
    struct hostapd_hw_modes *modes;
    u16 m;
    wps = os_zalloc(sizeof(*wps));
    .....
/*
```

设置两个重要的回调函数。其中，`cred_cb`在EAP-WSC模块解析`credential`属性集时使用。后面将见到其用法。

`event_cb`用于通知WSC模块发生的一些事件。例如“WSC-SUCCESS”就在`wpa_supplicant_wps_event`中处理。

```
/*
    wps->cred_cb = wpa_supplicant_wps_cred;
    wps->event_cb = wpa_supplicant_wps_event;
    wps->cb_ctx = wpa_s;

    // 初始化设备信息，这些信息来自图6-33中的配置
    wps->dev.device_name = wpa_s->conf->device_name;
    wps->dev.manufacturer = wpa_s->conf->manufacturer;
    wps->dev.model_name = wpa_s->conf->model_name;
    wps->dev.model_number = wpa_s->conf->model_number;
    wps->dev.serial_number = wpa_s->conf->serial_number;
    wps->config_methods = // 将字符串描述的WSC方法转换成对应的标志位
        wps_config_methods_str2bin(wpa_s->conf-
>config_methods);
.......
```

// 配置参数检查，Label和Display不能同时配置。即设备不能同时使用静态PIN码和动态PIN码

```
/*
    WSC规范新增了Virtual Push Button和Virtual Display两种方法,
    wps_fix_config_methods
    函数将判断设备是否支持Push Button或者Display。如果二者支持，需要为
    WSC添加对应的virtual
```

方法。下面这个函数仅在`CONFIG_WPS2`宏被定义的情况下有实际作用。

```
/*
    wps->config_methods = wps_fix_config_methods(wps-
>config_methods);
    wps->dev.config_methods = wps->config_methods;
```

```
    os_memcpy(wps->dev.pri_dev_type, wpa_s->conf->device_type,
WPS_DEV_TYPE_LEN);
```

```
    wps->dev.num_sec_dev_types = wpa_s->conf-
>num_sec_device_types;
    os_memcpy(wps->dev.sec_dev_type, wpa_s->conf-
>sec_device_type,
WPS_DEV_TYPE_LEN * wps->dev.num_sec_dev_types);
```

```

wps->dev.os_version = WPA_GET_BE32(wpa_s->conf-
>os_version);
modes = wpa_s->hw.modes;
// 设置RF Bands相关信息
if (modes) {
    for (m = 0; m < wpa_s->hw.num_modes; m++) {
        if (modes[m].mode == HOSTAPD_MODE_IEEE80211B ||
            modes[m].mode == HOSTAPD_MODE_IEEE80211G)
            wps->dev.rf_bands |= WPS_RF_24GHZ;
        else if (modes[m].mode == HOSTAPD_MODE_IEEE80211A)
            wps->dev.rf_bands |= WPS_RF_50GHZ;
    }
}
if (wps->dev.rf_bands == 0) wps->dev.rf_bands =
WPS_RF_24GHZ | WPS_RF_50GHZ;

os_memcpy(wps->dev.mac_addr, wpa_s->own_addr, ETH_ALEN);
wpas_wps_set_uuid(wpa_s, wps); // 设置uuid, 如果没有配置的话,
则利用MAC地址生成UUID
// STA默认支持的认证算法和加密算法
// 结合前面介绍的理论知识, 读者能想起来对应的Attribute是什么吗
wps->auth_types = WPS_AUTH_WPA2PSK | WPS_AUTH_WPAPSK;
wps->encr_types = WPS_ENCR_AES | WPS_ENCR_TKIP;

os_memset(&rcfg, 0, sizeof(rcfg));
rcfg.new_psk_cb = wpas_wps_new_psk_cb;
rcfg.pin_needed_cb = wpas_wps_pin_needed_cb;
rcfg.set_sel_reg_cb = wpas_wps_set_sel_reg_cb;
rcfg.cb_ctx = wpa_s;
/*
创建一个wps_registrar对象, 该对象代表Registrar。不过Enrollee中用
不到它, 故此处不开展
分析。感兴趣的读者请在学完本章后再自行研究相关内容。
*/
wps->registrar = wps_registrar_init(wps, &rcfg);
.....
wpa_s->wps = wps;

return 0;
}

```

图6-33为Galaxy Note 2中wpa\_supplicant.conf的配置文件。

```
ctrl_interface=wlan0
update_config=1
device_name=t03gzc
manufacturer=samsung
model_name=GT-N7100
model_number=GT-N7100
serial_number=4df13b8d0429af5b
device_type=10-0050F204-5
config_methods=physical display virtual push button keypad
```

图6-33 wpa\_supplicant.conf示例

下面来看WPS\_PIN命令的处理流程。

## 2. WPS\_PIN命令处理

根据前文介绍，WifiStateMachine将发送“WPS\_PIN any”命令给WPAS以触发WSC的工作流程。该命令的处理代码如下所示。

```
[-->ctrl_iface.c: : wpa_supplicant_ctrl_iface_wps_pin]

static int wpa_supplicant_ctrl_iface_wps_pin
    (struct wpa_supplicant *wpa_s,
char *cmd,
            char *buf, size_t buflen)
{
    u8 bssid[ETH_ALEN], *_bssid = bssid;
    char *pin; int ret;
    /*
        cmd传入此函数时已经将“WPS_PIN any”中的“WPS_PIN ”（注意右边引号前的空格）子串忽略了，
        所以cmd的取值是“any”。
    */
    pin = os strchr(cmd, ' ');
    if (pin) *pin++ = '\0';
    if (os strcmp(cmd, "any") == 0)
        _bssid = NULL;
    .....
    if (pin) {
        // 上层没有传入PIN码，故略去此段代码
    }
    // 启动WPS流程。详情见下文
    ret = wpas_wps_start_pin(wpa_s, _bssid, NULL, 0,
```

```

    DEV_PW_DEFAULT);
    .....
done:
    ret = os_snprintf(buf, buflen, "%08d", ret); // 将PIN码转成字符串返回给WifiStateMachine
    return ret;
}

```

上述代码中，wpas\_wps\_start\_pin函数将被调用以开始WPS流程。调用该函数时，相关的参数取值情况是：\_bssid为NULL，DEV\_PW\_DEFAULT值为0。

来看wpas\_wps\_start\_pin的代码，如下所示。

```
[-->wps_supplicant.c: : wpas_wps_start_pin]

int wpas_wps_start_pin(struct wpa_supplicant *wpa_s, const u8
*bssid,
                        const char *pin, int p2p_group, u16 dev_pw_id)
{
    struct wpa_ssid *ssid;
    char val[128];
    unsigned int rpin = 0;
    wpas_clear_wps(wpa_s); // 清空之前的WPS信息

    /*
     wpas_wps_add_network将创建一个wpa_ssid对象，它用于保存一个无线网络的配置信息。
     在wpas_wps_add_network中，该网络的key_mgmt将被为
     WPA_KEY_MGMT_WPS。另外，每一个wpa_ssid对象
     都有一个类型为eap_peer_config的成员，该成员用于保存EAP Supplicant
     的配置信息。对于WPS来说，
     该配置信息的Method被设置为EAP-WSC，identity被设为“WFA-
     SimpleConfig-Enrollee-1-0”。
     wpas_wps_add_network函数比较简单，请读者自行研究。
    */
    ssid = wpas_wps_add_network(wpa_s, 0, bssid);
    .....
    ssid->temporary = 1;
    ssid->p2p_group = p2p_group;
    .....// CONFIG_P2P的处理
    if (pin)
        os_snprintf(val, sizeof(val), "\"pin=%s
dev_pw_id=%u\"", pin, dev_pw_id);

```

```

else {
/*
    生成一个随机PIN码。WSC对PIN码格式有所规定。PIN码一共包含8个数
字，最后一个数字（即最右边的
    一个数字是前7个数字的校验和）。
*/
    rpin = wps_generate_pin();
    // 以图6-2为例，PIN码为“01308204”，所以下面val的值
为“pin=01308204 dev_pw_id=0”
    os_snprintf(val, sizeof(val), "\"pin=%08d
dev_pw_id=%u\"", rpin, dev_pw_id);
}
// 将value值保存到wpa_ssid中eap成员变量的phase1中
wpa_config_set(ssid, "phase1", val, 0);
if (wpa_s->wps_fragment_size)
    ssid->eap.fragment_size = wpa_s->wps_fragment_size;
// 注册一个WSC超时任务，超时时间是120秒。该时间也是由WSC规范规定的
eloop_register_timeout(WPS_PBC_WALK_TIME, 0,
wpas_wps_timeout, wpa_s, NULL);
// 重新关联并发起扫描。该函数比较简单，请读者自行研究。该函数内部将发
起扫描请求
wpas_wps_reassoc(wpa_s, ssid, bssid);
return rpin;
}

```

由上述代码可知，wpas\_wps\_start\_pin添加了一个潜在的和WPS相关的无线网络配置项。接下来的工作自然是需要扫描周围的无线网络以搜索那些支持WSC功能的AP。这一工作正是属于前文介绍的WSC Discovery Phase。

### 3. 发起扫描请求

根据前文对WSC基础知识的介绍，STA发起扫描请求时需要在Probe Request帧中添加WSC IE。WPAS中，扫描工作的代码在wpa\_supplicant\_scan函数中，我们重点关注其中和WSC相关的一部分，如下所示。

```

[-->scan.c: : wpa_supplicant_scan]

static void wpa_supplicant_scan(void *eloop_ctx, void
*timeout_ctx)
{
    .....// 该函数的详情请参考4.5.3节“无线网络扫描流程分析”
}
```

```
wpa_supplicant_optimize_freqs(wpa_s, &params);
// 下面这个函数将处理WSC IE
extra_ie = wpa_supplicant_extra_ies(wpa_s, &params);
}
```

[-->scan.c: : wpa\_supplicant\_extra\_ies]

```
static struct wpabuf *
wpa_supplicant_extra_ies(struct wpa_supplicant *wpa_s, struct
wpa_driver_scan_params *params)
{
    struct wpabuf *extra_ie = NULL;
#ifdef CONFIG_WPS // 处理WPS
    int wps = 0;
    enum wps_request_type req_type = WPS_REQ_ENROLLEE_INFO;
#endif /* CONFIG_WPS */

#ifdef CONFIG_WPS
    /*
        wpas_wps_in_use判断是否需要在Probe Request中添加WSC IE。WPAS的
        判断标准比较简单,
        就是查询所有的wpa_ssid对象, 判断它们的key_mgmt是否设置了
        WPA_KEY_MGMT_WPS。如果有,
        表明搜索的时候需要支持WSC IE。我们在介绍“WPS_PIN命令处理”时曾说过,
        WPAS将添加一个
        wpa_ssid对象, 并设置key_mgmt为WPA_KEY_MGMT_WPS。
        wpas_wps_in_use的返回值也有含义, 返回1表明使用PIN方法, 返回2表明使
        用PBC方法。
        wpas_wps_in_use函数比较简单, 请读者自行阅读。
    */
    wps = wpas_wps_in_use(wpa_s, &req_type);
```

```
    if (wps) {
        struct wpabuf *wps_ie;
        // 构造Probe Request中的WSC IE
        wps_ie = wps_build_probe_req_ie(wps == 2, &wpa_s->wps-
>dev,
                                         wpa_s->wps->uuid,
                                         req_type, 0, NULL);
        if (wps_ie) {
            if (wpabuf_resize(&extra_ie, wpabuf_len(wps_ie)) ==
0)
                wpabuf_put_buf(extra_ie, wps_ie);
            wpabuf_free(wps_ie);
        }
    }
}
```

```

.....// CONFIG_P2P处理
#endif /* CONFIG_WPS */
    return extra_ie;
}

```

wps\_build\_probe\_req\_ie用于构造WSC IE，读者可简单了解一下该函数，相关代码如下所示。

```

[-->wps.c : wps_build_probe_req_ie]

struct wpabuf * wps_build_probe_req_ie(int pbc, struct
wps_device_data *dev,
    const u8 *uuid, enum wps_request_type
req_type,unsigned int num_req_dev_types,
    const u8 *req_dev_types)
{
    struct wpabuf *ie;
    ie = wpabuf_alloc(500);
    .....
    if (wps_build_version(ie) ||wps_build_req_type(ie,
req_type) ||
        wps_build_config_methods(ie, dev->config_methods)
||wps_build_uuid_e(ie, uuid) ||
        wps_build_primary_dev_type(dev, ie) ||
wps_build_rf_bands(dev, ie) ||
        wps_build_assoc_state(NULL, ie) ||
wps_build_config_error(ie, WPS_CFG_NO_ERROR) ||
        wps_build_dev_password_id(ie, pbc ? DEV_PW_PUSHBUTTON :
DEV_PW_DEFAULT) ||
#define CONFIG_WPS2 // WPS2即WSC
        wps_build_manufacturer(dev, ie) ||
wps_build_model_name(dev, ie) ||
        wps_build_model_number(dev, ie) ||
wps_build_dev_name(dev, ie) ||
        wps_build_wfa_ext(ie, req_type == WPS_REQ_ENROLLEE,
NULL, 0) ||
#define CONFIG_WPS2 */
        wps_build_req_dev_type(dev, ie, num_req_dev_types,
req_dev_types)
        ||wps_build_secondary_dev_type(dev, ie)) {.....// 错误
处理}

    .....
    return wps_ie_encapsulate(ie);
}

```

最后，WPAS将发送携带了WSC IE的Probe Request帧。对于Galaxy Note 2来说，其发送的WSC IE信息可参考图6-18。

#### 4. 处理扫描结果

发送扫描请求后，WPAS下一步的工作就是处理搜索到的扫描结果。这部分的流程在4.5.3节扫描结果处理流程中有详细分析。在该流程中，和WSC相关的工作如下。

- `wpa_supplicant_get_scan_results`: 获取扫描结果。该函数内部将对搜索到的AP进行排序，它对WSC有特殊处理。
- `wpa_supplicant_pick_network`: 选择合适的AP作为目标AP。如果使用WSC的话，该函数将优先选择支持WSC的AP。

下面将分别介绍上述两个函数中和WSC处理相关的流程。

##### (1) `wpa_supplicant_get_scan_results`处理

代码如下。

```
[-->scan.c: : wpa_supplicant_get_scan_results]

struct wpa_scan_results *
wpa_supplicant_get_scan_results(struct wpa_supplicant *wpa_s,
                                struct scan_info *info, int new_scan)
{
    .....
    // 获取扫描结果
    scan_res = wpa_drv_get_scan_results2(wpa_s);
#ifdef CONFIG_WPS
    // WPAS当前处于WPS处理过程中，设置排序函数为
    wpa_scan_result_wps_compar
    if (wpas_wps_in_progress(wpa_s)) compar =
    wpa_scan_result_wps_compar;
#endif /* CONFIG_WPS */
    // 利用qsort函数对扫描结果进行升序排序。排序时将使用compar函数将较两个元素A、B的大小
    // compar返回负数，表示A < B，compar返回0，表示A=B，compar返回正数，表示A > B
    qsort(scan_res->res, scan_res->num, sizeof(struct
wpa_scan_res *), compar);
```

```

.....// 更新BSS
return scan_res;
}

```

由上述代码可知，当WPAS正处于WPS处理流程中，搜索到的AP将通过qsort以及wpa\_scan\_result\_wps\_compar进行排序比较。  
wpa\_scan\_result\_wps\_compar的代码如下所示。

[-->scan.c: : wpa\_scan\_result\_wps\_compar]

```

static int wpa_scan_result_wps_compar(const void *a, const void
*b)
{
    struct wpa_scan_res **_wa = (void *) a;
    struct wpa_scan_res **_wb = (void *) b;
    struct wpa_scan_res *wa = *_wa;
    struct wpa_scan_res *wb = *_wb;
    int uses_wps_a, uses_wps_b;
    struct wpabuf *wps_a, *wps_b;
    int res;
    /*
        wpa_scan_get_vendor_ie返回值的类型为u8*, 它指向wpa_scan_res中指
        定IE(此处是
            WSC IE)所在的内存位置。如果wpa_scan_res中没有WSC IE，则返回为空。
        */
    uses_wps_a = wpa_scan_get_vendor_ie(wa, WPS_IE_VENDOR_TYPE)
!= NULL;
    uses_wps_b = wpa_scan_get_vendor_ie(wb, WPS_IE_VENDOR_TYPE)
!= NULL;
    // 无线网络A支持WPS，而B不支持，则返回-1。这样，在“排座位”的时候，A
    将排在前面(A < B)
    if (uses_wps_a && !uses_wps_b)          return -1;
    // 无线网络A不支持WPS，而B支持，则B排在前面(B < A)
    if (!uses_wps_a && uses_wps_b)         return 1;

    // 如果无线网络A和B均支持WPS，则还需要进一步判断
    if (uses_wps_a && uses_wps_b) {
        /*
            wpa_scan_get_vendor_ie_multi将从拷贝指定IE的内容复制到一块新
            的内存中，该内存地址即
                wpa_scan_get_vendor_ie_multi的返回值。
        */
        wps_a = wpa_scan_get_vendor_ie_multi(wa,
WPS_IE_VENDOR_TYPE);
        wps_b = wpa_scan_get_vendor_ie_multi(wb,

```

```

WPS_IE_VENDOR_TYPE);
/*
    如果周围有多个支持WPS的无线网络，则设置Selected Registrar属性
(而且值为1)的AP
    将位于前排。
*/
res = wps_ap_priority_compar(wps_a, wps_b);
wpabuf_free(wps_a); // 释放wpa_scan_get_vendor_ie_multi
创建的新内存
wpabuf_free(wps_b);
if (res) return res;
}
// 对于没有WSC支持的AP，其排座顺序仅考虑它们的信号强度和质量
if (wb->level == wa->level)
    return wb->qual - wa->qual;
return wb->level - wa->level;
}

```

根据上面的代码可知，WSC的AP扫描结果“排座”规则如下。

- 支持WSC功能的AP排在不支持WSC的AP之前。
- 对于两个同时支持WSC功能的AP来说，Selected Registrar值为1的AP排在前面。
- 对于不支持WSC的AP来说，信号强度和质量好的AP排在前面。

当扫描结果排完序后，WPAS的下一步工作就是从众多搜索到的AP中挑选一个作为目标AP以发起关联请求。该工作由wpa\_supplicant\_pick\_network完成。

## (2) wpa\_supplicant\_pick\_network处理

wpa\_supplicant\_pick\_nework的代码比较简单，如下所示。

```

[-->events.c: : wpa_supplicant_pick_nework]

static struct wpa_bss * wpa_supplicant_pick_network(struct
wpa_supplicant *wpa_s,
            struct wpa_scan_results *scan_res, struct
wpa_ssid **selected_ssid)
{
    struct wpa_bss *selected = NULL;  int prio;

```

```

    while (selected == NULL) { // 按照优先级搜索扫描结果
        for (prio = 0; prio < wpa_s->conf->num_prio; prio++) {
            selected = wpa_supplicant_select_bss(wpa_s,
scan_res, wpa_s->conf->pssid[prio],
                selected_ssid);
            if (selected) break;
        }
        .....// 其他处理
    }
    return selected;
}

```

wpa\_supplicant\_select\_bss内部将通过调用wpa\_scan\_res\_match的函数来选取一个合适的无线网络。该函数的代码如下所示。

[-->events.c: : wpa\_scan\_res\_match]

```

static struct wpa_ssid * wpa_scan_res_match(struct
wpa_supplicant *wpa_s,
                                         int i, struct wpa_scan_res *bss, struct
wpa_ssid *group)
{
    const u8 *ssid_; u8 wpa_ie_len, rsn_ie_len, ssid_len;
    int wpa; struct wpa_blacklist *e;
    const u8 *ie; struct wpa_ssid *ssid;

    ie = wpa_scan_get_ie(bss, WLAN_EID_SSID);
    ssid_ = ie ? ie + 2 : (u8 *) "";
    ssid_len = ie ? ie[1] : 0;
    .....
    for (ssid = group; ssid; ssid = ssid->pnnext) {
        int check_ssid = wpa ? 1 : (ssid->ssid_len != 0);
        .....
#define CONFIG_WPS
        if ((ssid->key_mgmt & WPA_KEY_MGMT_WPS) && e && e-
>count > 0) continue;
        /*
         * 通过“WPS_PIN any”命令创建的wpa_ssid还没有设置ssid。
         */
        wpas_wps_ssidWildcardOk
            用于判断是否需要进行ssid检查。该函数内部将利用下文代码中提到的
            wps_is_addrAuthorized函数。
            由于笔者测试用的AP仅支持WPS，所以wpas_wps_ssidWildcardOk返
            回非零值。这样，if条件生效，
            check_ssid被设置为0。
        */

```

```

        if (wpa && ssid->ssid_len == 0 &&
            wpas_wps_ssid_wildcard_ok(wpa_s, ssid, bss))
    check_ssid = 0;
    .....
#endif /* CONFIG_WPS */
.....// 其他判断
// 该函数内部先调用wpas_wps_ssid_bss_match函数以判断是否有合适的AP，如果有则选中它
    if (!wpa_supplicant_ssid_bss_match(wpa_s, ssid, bss))
continue;
.....// 其他判断。例如检查wpa_ssid中的ssid是否匹配搜索结果中的ssid。如果不匹配，则不能选择该AP
    return ssid;
}
return NULL;
}

```

直接来看wpas\_wps\_ssid\_bss\_match函数，代码如下所示。

```
[-->wpa_supplicant.c: : wpas_wps_ssid_bss_match]

int wpas_wps_ssid_bss_match(struct wpa_supplicant *wpa_s, struct
wpa_ssid *ssid,
                                struct wpa_scan_res *bss)
{
    struct wpabuf *wps_ie;

    if (!(ssid->key_mgmt & WPA_KEY_MGMT_WPS))  return -1;
    // 获取该WSC IE信息
    wps_ie = wpa_scan_get_vendor_ie_multi(bss,
WPS_IE_VENDOR_TYPE);
    if (eap_is_wps_pbc_enrollee(&ssid->eap)) { .....// PBC处理}

    // 判断eap_peer_config设置的identity是否为“WFA-SimpleConfig-Enrollee-1-0”
    if (eap_is_wps_pin_enrollee(&ssid->eap)) {
        .....
    /*
        笔者使用的AP没有包含AuthorizedMACs子属性，并且该AP也不支持
WSC。所以下面这个函数将返回1，
如此，if判断失败。读者可参考图6-19。
*/
        // AP返回的Probe Response帧信息
        if (!wps_is_addr_authorized(wps_ie, wpa_s->own_addr,
1)) .....
    }
}
```

```
        else {
            wpa_printf(MSG_DEBUG, "    selected based on WPS IE
"
                       "(Authorized MAC or Active PIN)");
        }
        wpabuf_free(wps_ie);
        return 1; // 选中
    }
    .....
    return -1;
}
```

总之，在周围空间有很多无线网络的情况下，笔者测试WSC时使用的AP将会被选中作为目标AP。接下来的流程就和4.5.3节关联无线网络处理流程分析的内容一样，STA将关联到目标AP。对于非WSC来说，AP和STA将开展4-Way Handshake流程，而对于WSC来说，AP和STA将开展EAP-WSC流程。

马上来看EAP-WSC处理流程，它也是整个WSC流程的核心内容。

## 6.4.4 EAP-WSC处理流程分析

EAP-WSC流程涉及EAPOL中的四个状态机（SUPP\_PAЕ、KEY\_RX、SUPP\_BE、Port Timers）以及EAP SM之间的联动。当STA成功关联到AP后，EAPOL及EAP状态机情况如下（详情请参考4.5.3节eapol\_sm\_notify\_portEnabled分析）。

- SUPP\_PAЕ为DISCONNECTED状态；
- KEY\_RX为NO\_KEY\_RECEIVE状态；
- SUPP\_BE为IDLE状态；
- EAP\_SM为DISABLED状态。

根据图6-7所示，EAP-WSC流程的开始于STA向AP发送的EAPOL-Start帧。是什么原因导致STA发送EAPOL-Start帧呢？来看下文。

### 1. 发送EAPOL-Start

在STA关联到AP流程的最后，eapol\_sm\_notify\_portEnabled将设置portEnabled为1，根据代码（eapol\_supp\_sm.c中的SM\_STEP(SUPP\_PAЕ)）以及图4-28可知，SUPP\_PAЕ要进入的下一个状态是CONNECTING，其EA（Entry Aciton）代码如下。

```
[-->eapol_supp_sm.c: : SM_STATE (SUPP_PAЕ, CONNECTING) ]  
  
SM_STATE (SUPP_PAЕ, CONNECTING)  
{  
    // SUPP_PAЕ_state此时的值为SUPP_PAЕ_DISCONNECTED，故  
    send_start为0  
    // 注意下面这个判断很重要，待会还会回到此处  
    int send_start = sm->SUPP_PAЕ_state == SUPP_PAЕ_CONNECTING;  
    SM_ENTRY (SUPP_PAЕ, CONNECTING);  
    if (send_start) {  
        sm->startWhen = sm->startPeriod;  
        sm->startCount++;  
    } else {  
        #ifdef CONFIG_WPS //
```

```

        sm->startWhen = 1;                                // 注意，如果WPAS支持WPS,
则startWhen值为1
#else /* CONFIG_WPS */
        sm->startWhen = 3;
#endif /* CONFIG_WPS */
}
// 启动Port Timers SM, Port Timers SM将递减startWhen，并调用
eapol_sm_step以重新遍历状态机
eapol_enable_timer_tick(sm);
sm->eapolEap = FALSE; ..
// 由于send_start为0，所以此时还不会发送EAPOL-Start包
if (send_start) eapol_sm_txStart(sm);
}

```

根据代码中的注释，当Port Timers SM运行时，它将递减startWhen变量（结果是startWhen的值变为0），然后通过eapol\_sm\_step重新遍历状态机。在该函数中，PAE的SM\_STEP将被调用以检查是否需要进行状态切换，相关代码如下所示。

```

[-->eapol_supp_sm.c: : SM_STEP (SUPP_PAЕ) ]

SM_STEP(SUPP_PAЕ)
{
    .....// 略去不相关的内容
    else switch (sm->SUPP_PAЕ_state) {           // SUPP_PAЕ_state
还处于CONNECTING状态
    .....
        case SUPP_PAЕ_CONNECTING:
            if (sm->startWhen == 0 && sm->startCount < sm-
>maxStart)
                SM_ENTER(SUPP_PAЕ, CONNECTING);
            // 由于startWhen为0，PAE将重新进入CONNECTING状态
    .....
        break;
    case SUPP_PAЕ_AUTHENTICATING:
    .....
}

```

根据上面代码可知，PAE将再次从CONNECTING状态进入CONNECTING状态。请读者回顾SM\_STATE (SUPP\_PAЕ, CONNECTING) 函数。这一次 sendStart将取值1，所以eapol\_sm\_txStart会被调用，该函数的代码如下所示。

```
[-->eapol_supp_sm.c: : eapol_sm_txStart]

static void eapol_sm_txStart(struct eapol_sm *sm)
{
    // eapol_send函数指针指向wpa_supplicant_eapol_send
    // 相关代码在wpas_glue.c中, 请读者自行阅读
    sm->ctx->eapol_send(sm->ctx->eapol_send_ctx,
    IEEE802_1X_TYPE_EAPOL_START, (u8 *) "", 0);
    sm->dot1xSuppEapolStartFramesTx++;
    sm->dot1xSuppEapolFramesTx++;
}
```

由上述代码可知, eapol\_send的实例wpa\_supplicant\_eapol\_send将最终发送EAPOL-Start帧。

## 2. 状态机切换处理

STA发出EAPOL-Start后, AP将发送EAP-Request/Identity包。STA处理EAP-Request/Identity后将回复EAP-Response/Identity包。上述流程将触发EAPOL中的PAE、BE和EAP状态机联动。此联动过程相当复杂。故本节将以EAP-Request/Identity为入口, 分析WPAS中状态机的切换处理。

**注意** 此处的状态机联动实际上反映的是WPAS中EAP包处理的通用流程。学习过程中, 请读者务必结合4.4节EAP和EAPOL模块的理论知识。

先来看EAP-Request的处理。WPAS中, EAP包接收的函数是wpa\_supplicant\_rx\_eapol(相关分析请参考4.5.3节分析EAPOL-Key交换流程时对wpa\_supplicant\_rx\_eapol的介绍), 我们说过非PSK认证方法将由eapol\_sm\_rx\_eapol处理, 故直接来看eapol\_sm\_rx\_eapol函数, 代码如下所示。

```
[-->eapol_supp_sm.c: : eapol_sm_rx_eapol]

int eapol_sm_rx_eapol(struct eapol_sm *sm, const u8 *src, const
u8 *buf, size_t len)
{
    const struct ieee802_1x_hdr *hdr;  const struct
ieee802_1x_eapol_key *key;
    int data_len;  int res = 1; size_t plen;

    sm->dot1xSuppEapolFramesRx++;
```

```

    hdr = (const struct ieee802_1x_hdr *) buf;
    sm->dot1xSuppLastEapolFrameVersion = hdr->version;
    os_memcpy(sm->dot1xSuppLastEapolFrameSource, src,
    ETH_ALEN);

    plen = be_to_host16(hdr->length);
    .....
#endif CONFIG_WPS
/*
workaround意思为“变通方案”。在WPAS中，它表示为了兼容某些AP的错误行为（例如发送的EAP包格式不符合要求），而采用绕过去的方法来处理。
*/
if (sm->conf.workaround && plen < len - sizeof(*hdr) &&
    hdr->type == IEEE802_1X_TYPE_EAP_PACKET &&
    len - sizeof(*hdr) > sizeof(struct eap_hdr)) {
    .....
}
#endif
data_len = plen + sizeof(*hdr);
switch (hdr->type) {
case IEEE802_1X_TYPE_EAP_PACKET: // 本例中收到的是EAP-Request包，满足此case条件
    .....
    wpabuf_free(sm->eapReqData);
    sm->eapReqData = wpabuf_alloc_copy(hdr + 1, plen);
    if (sm->eapReqData) {
        sm->eapolEap = TRUE; // 设置条件变量
        eapol_sm_step(sm); // 触发状态机运行
    }
    break;
    .....
}
return res;
}

```

WPAS每收到一个EAP包都会触发上述代码中的流程。回顾eapol\_sm\_step中和状态机运转相关的代码。

[-->eapol\_supp\_sm.c: : eapol\_sm\_step]

```
{
    int i;
    for (i = 0; i < 100; i++) {
        sm->changed = FALSE;

```

```

        SM_STEP_RUN(SUPP_PAE);           // 先执行SUPP_PAE状态机
        SM_STEP_RUN(KEY_RX);            // 再运转KEY_RX状态机
        SM_STEP_RUN(SUPP_BE);           // 最后运转SUPP_BE状态机
        if (eap_peer_sm_step(sm->eap))      // 执行EAP_SM状态机
            sm->changed = TRUE;
        if (!sm->changed)
            break;
    }
    .....
}

```

其中，`eap_peer_sm_step`的代码如下所示。

```

[-->eap.c: : eap_peer_sm_step]

int eap_peer_sm_step(struct eap_sm *sm)
{
    int res = 0;
    do {
        循环，直到EAP SM稳定后才退出
        sm->changed = FALSE;
        SM_STEP_RUN(EAP);
        if (sm->changed)
            res = 1;
    } while (sm->changed);
    return res;
}

```

通过上述代码可知，EAPOL和EAP的状态机联动过程如下。

EAPOL先按顺序遍历PAE、KEY\_RX、BE状态机，然后执行EAP状态机。只有EAP SM稳定后（即`eap_peer_sm_step`函数中的`sm->changed`为`FALSE`时）才退出`eap_peer_sm_step`。

如果上述四个状态机有任何一个状态机的状态不稳定（即`sm->changed`为`TRUE`），则继续遍历所有状态机。

特别需要指出的是，状态机A运行时可能会修改一些条件变量从而导致状态机B发生状态切换。虽然第4章对每个状态机的状态切换图都有详细介绍，但读者很难理清楚状态机之间是如何互相影响的。在此，笔者整理了WPAS从发送EAPOL-Start包到接收EAP-Request/Identity以及

回复EAP-Response/Identity这一过程中四个状态机的切换过程，如图6-34所示。

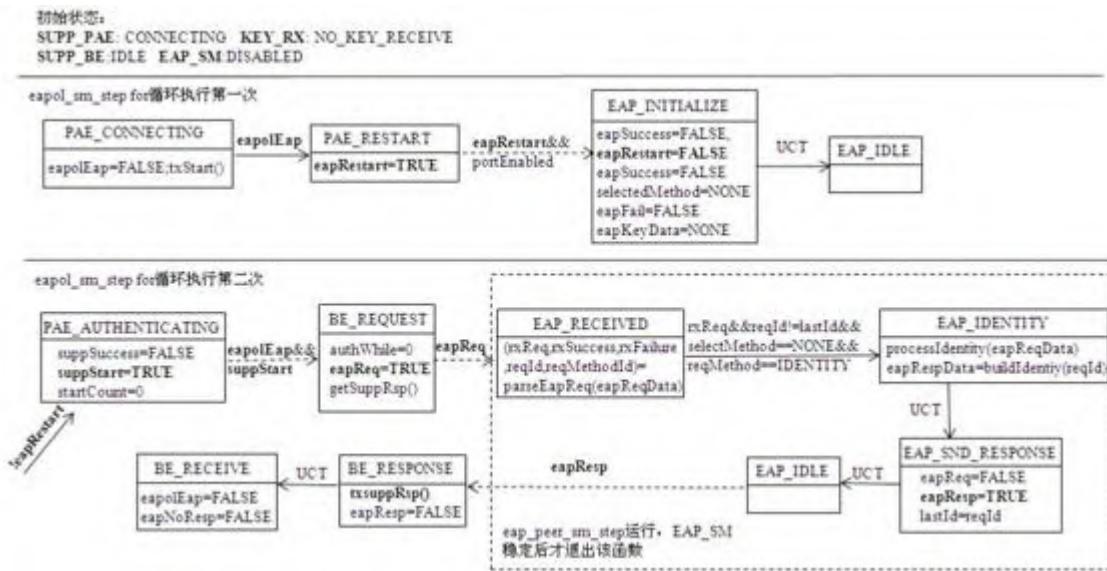


图6-34 EAP-Request/Response Identity流程中的状态机联动

图6-34中第一行显示了PAE、KEY\_RX、BE和EAP\_SM的初始状态。由于EAP-WSC不会收发EAPOL-Key帧，所以KEY\_RX将不参与联动过程。

图中的方框上部所示为状态机以及当前的状态，格式为“状态机名\_状态名”，如PAE\_CONNECTING等。方框下部所示为该状态机对应状态的EA处理（由于篇幅原因，图中EA仅列出了一些重要的处理逻辑）。

当状态机A从一个状态切换到另一个状态时，切换过程用实箭头表示（例如第二行中，PAE\_CONNECTING切换到PAE\_RESTART，切换条件是“eapolEap为TRUE”。当WPAS收到一个EAP帧时，该变量将在上文介绍的eapol\_sm\_rx\_eapol函数中被设置为TRUE）。

当状态机A在其EA处理中修改了某些条件变量（或者外界设置了某个条件变量）导致状态机B发生状态切换时，其切换过程用虚箭头表示。例如第二行中的PAE\_RESTART状态，其EA将设置eapRestart为TRUE，而该条件和portEnabled将共同促使EAP\_SM进入INITIAZE状态。

第二行表示eapol\_sm\_step第一次循环过程中的状态机切换以处理接收到的EAP-Request/Identity包。但这一轮还不会真正处理EAP包。

第三行表示eapol\_sm\_step的第二次循环。在这次循环过程中，EAP状态机将处理EAP-Request/Identity包。在解析该包时，发现它包含了Identity信息，所以EAP SM将进入IDENTITY状态去处理它。处理完毕后，EAP SM将构造一个EAP-Response/Identity包，并设置eapResp变量为TRUE。

第三行中，eapResp变量将使得BE进入RESPONSE状态，该状态的EA将调用txsuppResp发送这个EAP-Response/Identity包。

当图6-34执行完毕后，EAPOL和EAP状态机将进入稳定状态，这样，eapol\_sm\_step得以返回。根据EAP-WSC的流程，WPAS下一步将继续接收并处理EAP包。在这以后的过程中（M1~M8），PAE保持Authenticating状态不变。

当EAPOL收到一个EAP包后，BE将从RECEIVE状态切换至REQUEST状态。EAP将根据EAP包的信息从IDLE状态转移到其他状态（首先是RECEIVED状态，在该状态中将解析EAP包的内容，根据内容以进入GET\_METHOD或METHOD状态以处理EAP包）。

EAP状态机处理完EAP包，BE将进入RESPONSE状态并发送EAP回复包。整个流程将反复执行，直到EAP-WSC流程终结。

所以，对EAP-WSC流程来说，EAPOL状态机的执行过程比较固定。而对EAP SM来说，它将根据EAP包内容的不同而转移到不同的状态。下面我们将直接进入EAP对应的状态以分析不同EAP包的处理过程。

**注意** 根据图4-21关于EAP SM的描述，当portEnabled值为TRUE时，应该从DISABLED状态切换至INITIALIZE状态。不过，4.5.3节“wpa\_supplicant\_associate分析之三”中曾提到，由于force\_disabled变量为TRUE，EAP\_SM是无法转入INITIALIZED状态的。为什么此处它可以呢？原来。由于本例使用的key\_mgmt是WPA\_KEY\_MGMT\_WPS，所以force\_disabled变量将被设置为FALSE，这样EAP SM就可以转换至INITIALIZE状态了。其间的细节内容请读者参考wpa\_supplicant\_initiate\_eapol及内部所调用的eapol\_sm\_notify\_config函数。

### 3. EAP-Request/Identity处理

EAP状态机的RECEIVED状态将对收到的EAP包进行解析，相关代码如下所示。

```
[-->eap.c: : SM_STATE (EAP, RECEIVED) ]  
  
SM_STATE (EAP, RECEIVED)  
{  
    const struct wpabuf *eapReqData;  
    SM_ENTRY (EAP, RECEIVED);  
    eapReqData = eapol_get_eapReqData (sm);  
    eap_sm_parseEapReq (sm, eapReqData); // 解析收到的EAP包  
    sm->num_rounds++;  
}
```

eap\_sm\_parseEapReq的代码如下所示。

```
[-->eap.c: : eap_sm_parseEapReq]  
  
static void eap_sm_parseEapReq (struct eap_sm *sm, const struct  
wpabuf *req)  
{  
    const struct eap_hdr *hdr; size_t plen; const u8 *pos;  
    .....  
    hdr = wpabuf_head (req);  
    plen = be_to_host16 (hdr->length);  
    .....  
    sm->reqId = hdr->identifier;  
    if (sm->workaround) { ..... }  
  
    switch (hdr->code) {  
        case EAP_CODE_REQUEST:  
            .....  
            sm->rxReq = TRUE;  
            pos = (const u8 *) (hdr + 1);  
            sm->reqMethod = *pos++; // 对于EAP-Request Identity  
            包而言, reqMethod=1  
            // 处理EAP-Request/WSC_Start, WSC_Start属于扩展EAP协议  
            // 此处收到的是Identity包, 所以下面这个if条件并不满足  
            if (sm->reqMethod == EAP_TYPE_EXPANDED) {  
                .....//  
                // 对于EAP-Request/WSC_Start而言, reqVendor取值为  
0x0372a  
                sm->reqVendor = WPA_GET_BE24 (pos);  
                pos += 3;  
                // 获取Vendor Type, 值为0x1, 表示SimpleConfig。请参考图
```

6-22

```
        sm->reqVendorMethod = WPA_GET_BE32(pos);
    }
    break;
case EAP_CODE_RESPONSE:
    .....
    break;
case EAP_CODE_SUCCESS:
    sm->rxSuccess = TRUE;
    break;
case EAP_CODE_FAILURE:
    sm->rxFailure = TRUE;           // 在EAP-WSC流程的最后，AP将
发送EAP-Failure包
    break;
.....
}
}
```

根据图6-34所示，EAP SM接着将进入IDENTITY状态（读者可参考eap.c中的eap\_peer\_sm\_step\_local函数），代码如下所示。

[-->eap.c: : SM\_STATE (EAP, IDENTITY) ]

```
SM_STATE(EAP, IDENTITY)
{
    const struct wpabuf *eapReqData;
    SM_ENTRY(EAP, IDENTITY);
    eapReqData = eapol_get_eapReqData(sm); // 获取EAP-Request包内
容
    eap_sm_processIdentity(sm, eapReqData); // 处理Identity，请读
者自行阅读此函数
    wpabuf_free(sm->eapRespData);
    sm->eapRespData = NULL;
    // 构造EAP-Response Identity包
    sm->eapRespData = eap_sm_buildIdentity(sm, sm->reqId, 0);
}
```

来看看eap\_sm\_buildIdentity函数，代码如下所示。

[-->eap.c: : eap\_sm\_buildIdentity]

```
struct wpabuf * eap_sm_buildIdentity(struct eap_sm *sm, int id,
int encrypted)
{
    /*

```

获取wpa\_ssid中的eap\_peer\_config对象，它代表EAP配置信息。该eap\_peer\_config的来历请

读者自行阅读wpa\_supplicant\_initiate\_eapol及内部所调用的eapol\_sm\_notify\_config函数。

总之，下面这个config对象将指向“WPS\_PIN any”命令处理时创建的wpa\_ssid中eap变量，它指向

一个eap\_peer\_config实例。

eap\_get\_config内部将通过函数指针调用eap\_supp\_sm.c中的eapol\_sm\_get\_config函数。

```
 */
struct eap_peer_config *config = eap_get_config(sm);
struct wpabuf *resp;    const u8 *identity; size_t
identity_len;

if (config == NULL) { .....
if (sm->m && sm->m->get_identity && .......) .....
else if (!encrypted && config->anonymous_identity) .....
else {
    // 对WSC来说，identity的值为“WFA-SimpleConfig-Enrollee-1-
0”
    identity = config->identity;
    identity_len = config->identity_len;
}

if (identity == NULL) {
    ....// 没有配置identity
} else if (config->pcsc) { .....
// 构造EAP-Response/Identity回复包
resp = eap_msg_alloc(EAP_VENDOR_IETF, EAP_TYPE_IDENTITY,
identity_len,
                           EAP_CODE_RESPONSE, id);
.....
wpabuf_put_data(resp, identity, identity_len);

return resp;
}
```

EAP-Request/Identity的处理流程比较简单，此处就不再详述。当AP收到来自STA的EAP-Response/Identity后，它将发送EAP-Request/WSC\_Start帧。该帧将导致EAP SM进入GET\_METHOD状态，马上来看相关代码。

#### 4. EAP-Request/WSC\_Start处理

图6-35所示为EAP-Request/WSC\_Start帧的内容。

```
└ Extensible Authentication Protocol
  Code: Request (1)
  Id: 1
  Length: 14
  Type: Expanded Type (254)
  └ Expanded Type (wifi Alliance, wifiProtectedsetup)
    EAP-EXT Vendor Id: WFA (0x372a)
    EAP-EXT Vendor Type: simpleConfig (0x01)
    Opcode: WSC Start (1)
  ┌ Flags: 0x00
```

图6-35 EAP-Request/WSC\_Start示例

首先处理该帧的是EAP\_SM\_GET\_METHOD状态，相关代码如下所示。

```
[-->eap.c: : SM_STATE (EAP, GET_METHOD) ]  
  
SM_STATE (EAP, GET_METHOD)  
{  
    int reinit;  
    EapType method;  
  
    SM_ENTRY (EAP, GET_METHOD);  
  
    if (sm->reqMethod == EAP_TYPE_EXPANDED)    method = sm->reqVendorMethod;  
    else    method = sm->reqMethod;  
    /*  
     * 判断WPAS是否支持此vendor对应的方法。reqVendor的值为0x372a。在  
     * 4.3.2节  
     * “eap_register_methods分析”中，WPAS支持的各种EAP方法将通过在  
     * eap_register_methods  
     * 函数中被注册，其中就有EAP-WSC方法。  
     */  
    if (!eap_sm_allowMethod (sm, sm->reqVendor, method)) goto nak;  
    ....  
    sm->selectedMethod = sm->reqMethod;    // selectedMethod值为  
    EAP_TYPE_EXPANDED (值为254)  
    if (sm->m == NULL) // sm->m指向一个eap_method对象，它代表一种特定  
    的EAP算法
```

```

    sm->m = eap_peer_get_eap_method(sm->reqVendor, method);
    // 获取EAP-WSC算法模块对应的对象
    if (!sm->m) goto nak;

    sm->ClientTimeout = EAP_CLIENT_TIMEOUT_DEFAULT;
    if (reinit) .....
    else
        sm->eap_method_priv = sm->m->init(sm); // 初始化EAP-WSC
算法

    if (sm->eap_method_priv == NULL) {..... // 错误处理}
    sm->methodState = METHOD_INIT;
    return;
    .....
}

```

EAP-WSC算法的注册代码位于eap\_peer\_wsc\_register函数（位于eap\_wsc.c，请读者自行阅读）中，它为EAP-WSC模块定制了三个函数。

- eap\_wsc\_init和eap\_wsc\_deinit：用于EAP-WSC算法模块资源的初始化和释放。
- eap\_wsc\_process：处理EAP-WSC包（即类型为WSC\_MSG的包）。

**提示** EAP-WSC的初始化函数比较简单，请读者自行研读eap\_wsc\_init函数。

GET\_METHOD之后，EAP SM下一个进入的状态是METHOD，其代码如下所示。

```

[-->eap.c: : SM_STATE (EAP, METHOD) ]

SM_STATE (EAP, METHOD)
{
    struct wpabuf *eapReqData; struct eap_method_ret ret;
    SM_ENTRY (EAP, METHOD);
    .....
    eapReqData = eapol_get_eapReqData (sm);           // 先获得请求信息

    os_memset (&ret, 0, sizeof (ret));
    ret.ignore = sm->ignore;   ret.methodState = sm-
>methodState;

```

```

    ret.decision = sm->decision;  ret.allowNotifications = sm-
>allowNotifications;
    wpabuf_free(sm->eapRespData);
    sm->eapRespData = NULL;
    // 对WSC来说，process函数为eap_wsc_process
    sm->eapRespData = sm->m->process(sm, sm->eap_method_priv,
&ret, eapReqData);
    .....// 其他处理
}
}

```

由上面的代码可知，对于EAP-WSC算法来说，process真正的实现是eap\_wsc\_process，下面将详细介绍。

## 5. eap\_wsc\_process介绍

eap\_wsc\_process的代码如下所示。

```
[-->eap_wsc.c: : eap_wsc_process]

static struct wpabuf * eap_wsc_process(struct eap_sm *sm, void
*priv,
                                         struct eap_method_ret *ret,const struct
wpabuf *reqData)
{
    struct eap_wsc_data *data = priv;  const u8 *start, *pos,
*end;
    size_t len; u8 op_code, flags, id; u16 message_length = 0;
    enum wps_process_res res; struct wpabuf tmpbuf;
    struct wpabuf *r;
    // 校验EAP-WSC包的头部信息
    pos = eap_hdr_validate(EAP_VENDOR_WFA, EAP_VENDOR_TYPE_WSC,
reqData, &len);
    .....
    op_code = *pos++;                                // 获取OpCode
    flags = *pos++;                                 // 获取标志位
    if (flags & WSC_FLAGS_LF) { .....// LF标志位处理}
    if (data->state == WAIT_FRAG_ACK) { .....// MF标志位处理}
    // 消息类型检查。当前EAP-WSC模块的状态是WAIT_START
    if (data->state == WAIT_START) {
        .....// 检查类型
        eap_wsc_state(data, MESSG);                // 设置EAP-WSC的
状态
        goto send_msg;                            // 直接跳转到
    }
}
```

```

send_msg
} else if (op_code == WSC_Start) {
    ret->ignore = TRUE;
    return NULL;
}
.....
if (flags & WSC_FLAGS_MF) {.....// 分片处理}
.....
// 关键函数①wps_process_msg
res = wps_process_msg(data->wps, op_code, data->in_buf);
switch (res) {
case WPS_DONE:
    eap_wsc_state(data, FAIL);
    break;
.....// WPS_CONTINUE, WPS_FAILURE和WPS_PENDING的处理
}
.....
send_msg:
if (data->out_buf == NULL) {
    data->out_buf = wps_get_msg(data->wps, &data-
>out_op_code); // 关键函数②
}
.....
eap_wsc_state(data, MMSG); //
r = eap_wsc_build_msg(data, ret, id); // 构造用于回复的EAP-WSC
消息包
.....
return r;
}

```

eap\_wsc\_process中有两个关键函数。

- **wps\_process\_msg:** 对于Enrollee来说，其内部将调用wps\_enrollee\_process\_msg以处理接收到的EAP-WSC\_MSG消息，例如M2、M4、M6、M8等消息。
- **wps\_get\_msg:** 对于Enrollee来说，其内部将调用wps\_enrollee\_get\_msg以构造M1、M3、M5、M7、WSC\_Done等消息。

我们在前面已经介绍过M1~M8的内容。在WAPS的代码中，这部分内容也比较简单，所以本节不展开详细讨论。下面将介绍WPAS如何处理M8消息中的Credentials属性集。毕竟，EAP-WSC算法的目的就是为了得到这个Credentials属性集。

## 6. M8消息处理

M8消息的处理函数是wps\_process\_m8，相关代码如下所示。

```
[-->wps_enrollee.c: : wps_process_m8]

static enum wps_process_res wps_process_m8(struct wps_data
*wps, const struct wpabuf *msg,
                                             struct wps_parse_attr *attr)
{
    struct wpabuf *decrypted;
    struct wps_parse_attr eatr;
    .....
    // 比较接收到的Enrollee Nonce和Authenticator的内容，防止被中间人
    篡改
    if (wps_process_enrollee_nonce(wps, attr->enrollee_nonce)
    ||
        wps_process_authenticator(wps, attr->authenticator,
    msg)) {
        wps->state = SEND_WSC_NACK;
        return WPS_CONTINUE;
    }

    .....
    // 解密Encrypted Settings属性集合
    decrypted = wps_decrypt_encr_settings(wps, attr-
    >encr_settings, attr->encr_settings_len);

    // 校验
    if (wps_validate_m8_encr(decrypted, wps->wps->ap, attr-
    >version2 != NULL) < 0) {
        .....// 错误处理
    }
    // 解析Encrypted Settings属性集合中携带的属性信息
    if (wps_parse_msg(decrypted, &eatr) < 0 ||
        wps_process_key_wrap_auth(wps, decrypted,
    eatr.key_wrap_auth) ||
        wps_process_creds(wps, eatr.cred,
    eatr.cred_len, eatr.num_cred,
        attr->version2 != NULL) ||
    wps_process_ap_settings_e(wps, &eatr, decrypted,
                                attr->version2 != NULL)) {.....// 错误处
理}
    wpabuf_free(decrypted);
    wps->state = WPS_MSG_DONE;
```

```
        return WPS_CONTINUE;
}
```

上面代码中：

- wps\_decrypt\_encr\_settings先解密Encrpyted Settings属性，解密后的内容保存在decrypted变量中， decrypted是一块内存缓冲。
- 然后调用wps\_parse\_msg来解析decrypted缓冲。根据6.2.3节对M7和M8的介绍， Encrypted Settings包含一系列属性。
- 调用wps\_process\_creds处理Encyrpted Settings中的Credentials 属性集。该属性集的内容可参考表6-6。

wps\_process\_creds的代码如下所示。

```
[-->wps_enrollee.c: : wps_process_creds]

static int wps_process_creds(struct wps_data *wps, const u8
*cred[],
                                size_t cred_len[], size_t num_cred, int wps2)
{
    size_t i;
    int ok = 0;
    .....
    for (i = 0; i < num_cred; i++) {
        int res;
        res = wps_process_cred_e(wps, cred[i], cred_len[i],
wps2);
        // 处理属性集中的每一项属性
        if (res == 0) ok++;
        .....
    }
    .....
    return 0;
}
```

wps\_process\_cred\_e函数的最后将通过cred\_cb回调函数将属性传递给wpa\_supplicant。该回调函数在6.4.3节WSC模块初始化分析时介绍的wpas\_wps\_init函数中被设置为wpa\_supplicant\_wps\_cred，而此函数的代码如下所示。

```
[-->wps_supplicant.c: : wpa_supplicant_wps_cred]
```

```

static int wpa_supplicant_wps_cred(void *ctx, const struct
wps_credential *cred)
{
    struct wpa_supplicant *wpa_s = ctx;
    struct wpa_ssid *ssid = wpa_s->current_ssid;
    u8 key_idx = 0; u16 auth_type;
    .....
/*
wps_cred_processing默认为0，表示WPAS内部处理credentials信息。值
为1表示将credentials
等信息将通过ctrl_iface发送给客户端去处理。值为2表示credentials信息
由WPAS内部处理，但也会发送
给客户端。
*/
    if (wpa_s->conf->wps_cred_processing == 1) return 0;

    auth_type = cred->auth_type; // 获取AP的认证算法
    if (auth_type == (WPS_AUTH_WPAPSK | WPS_AUTH_WPA2PSK))
auth_type = WPS_AUTH_WPA2PSK;

    // 检查认证算法设置是否正确
    if (auth_type != WPS_AUTH_OPEN && auth_type != WPS_AUTH_SHARED &&
        auth_type != WPS_AUTH_WPAPSK && auth_type != WPS_AUTH_WPA2PSK) return 0;

    // EAP-WSC工作基本完成，此时需要更新wpa_ssid对象的信息
    if (ssid && (ssid->key_mgmt & WPA_KEY_MGMT_WPS)) {
        os_free(ssid->eap.identity);
        ssid->eap.identity = NULL ssid->eap.identity_len = 0;
        os_free(ssid->eap.phase1); ssid->eap.phase1 = NULL;
        os_free(ssid->eap.eap_methods); ssid->eap.eap_methods =
NULL;
        if (!ssid->p2p_group)
            ssid->temporary = 0;
    }
    .....
    // 先恢复wpa_ssid的默认设置
    wpa_config_set_network_defaults(ssid);
    os_free(ssid->ssid);
    ssid->ssid = os_malloc(cred->ssid_len);
    if (ssid->ssid) {
        os_memcpy(ssid->ssid, cred->ssid, cred->ssid_len);
        ssid->ssid_len = cred->ssid_len;
    }
    // 结合属性信息，更新wpa_ssid中的各个项。首先更新加密算法设置

```

```

switch (cred->encr_type) {
    .....
    case WPS_ENCR_TKIP:
        ssid->pairwise_cipher = WPA_CIPHER_TKIP;
        break;
    case WPS_ENCR_AES:
        ssid->pairwise_cipher = WPA_CIPHER_CCMP;
        break;
    }
....// 更新认证算法设置
switch (auth_type) {
case WPS_AUTH_OPEN:
    ssid->auth_alg = WPA_AUTH_ALG_OPEN;
    ssid->key_mgmt = WPA_KEY_MGMT_NONE;
    ssid->proto = 0;
    break;
.....
case WPS_AUTH_WPA2PSK:
    ssid->auth_alg = WPA_AUTH_ALG_OPEN;
    ssid->key_mgmt = WPA_KEY_MGMT_PSK;
    ssid->proto = WPA_PROTO_RSN;
    break;
}

if (ssid->key_mgmt == WPA_KEY_MGMT_PSK) {
    // 更新PSK
    if (cred->key_len == 2 * PMK_LEN) {
        if (hexstr2bin((const char *) cred->key, ssid-
>psk, PMK_LEN)) return -1;
        ssid->psk_set = 1;
        ssid->export_keys = 1;
    } .....
}
// 处理某些使用混合加密模式的AP对WPS支持不够完善的情况
wpa_wps_security_workaround(wpa_s, ssid, cred);

#ifndef CONFIG_NO_CONFIG_WRITE
    if (wpa_s->conf->update_config && // 将配置信息写到配置文件中对
    应的无线网络项中
        wpa_config_write(wpa_s->confname, wpa_s->conf)) {
....}
#endif /* CONFIG_NO_CONFIG_WRITE */
    return 0;
}

```

图6-36所示为Galaxy Note 2最终所设置的无线网络配置信息。有了网络信息，当STA和AP断开后，它将使用新的网络信息向AP发起关联请求。这部分内容我们已经在第4章中重点介绍过了。

```
network={  
    ssid="Tenda_487A08"  
    psk="5un556j2aiapti"  
    proto=RSN  
    key_mgmt=WPA-PSK  
    pairwise=CCMP  
    auth_alg=OPEN  
}
```

图6-36 WSC最终获取的无线网络配置信息

**提示** 请读者自行研究MSG\_Done消息的构造，在那里WPAS将发送WPS-SUCCESS信息给上层的WifiMonitor。这样，WifiStateMachine才能收到WPS\_SUCCESS\_EVENT消息。

回顾整个EAP-WSC流程，从代码角度来说，该流程较难的部分不在EAP-WSC本身，而是在于状态机联动。读者不妨仔细阅读关于状态机切换处理部分，然后研究EAP-WSC的内容。

**提示** 建议读者自行研究WPAS代码中M1~M7的处理流程以加深对EAP-WSC的理解。

# 6.5 本章总结和参考资料说明

## 6.5.1 本章总结

本章对Wi-Fi Simple Configuration和其中的PIN方法进行了深入介绍。主要内容包括以下。

- WSC的理论知识，它是本章的核心。WSC中最重要的是RP协议以及WSC IE及各种Attribute的作用。希望读者能结合本章给的实例图来认真学习它们。
- 掌握理论知识后，本章介绍了Android平台中WSC的代码实现。它包括Settings、WifiService相关模块以及wpa\_supplicant相关模块。仅就EAP-WSC算法本身而言，其难度并不大。
- 由于WSC使用了EAP-WSC算法，所以本章还介绍了EAPOL和EAP状态机之间的联动过程。这部分代码的难度比较大，需要结合4.4节中EAP和EAPOL模块介绍的状态机切换图来学习。

注意 WSC规范中还定义了另外一种比较常用的PBC方法。PBC和PIN类似，它也会用到一个PIN码，只不过这个PIN码为“00000000”。读者可阅读参考资料[7]来了解PBC。

# 读累了记得休息一会儿哦~

公众号：古德猫宁李

- 电子书搜索下载
- 书单分享
- 书友学习交流

网站：[沉金书屋 https://www.chenjin5.com](https://www.chenjin5.com)

- 电子书搜索下载
- 电子书打包资源分享
- 学习资源分享

## 6.5.2 参考资料说明

本章参考资料其实只有一个，即WSC规范2.0.2版。读者可在百度文库上搜索到该文档，其地址为  
<http://wenku.baidu.com/view/aa2e8a20cf789eb172dc83d.html>。

### 1. WSC应用场景介绍

[1] “WSC-2.0.2”第1节“Introduction”

### 2. WSC核心组件及接口介绍

[2] “WSC-2.0.2”第4节“Core Architecture”

### 3. Registration Protocol介绍

[3] “WSC-2.0.2”第6.1节“In-band Setup Using aStandalone AP/Registrar”

[4] “WSC-2.0.2”第8节“Message Encoding”

[5] “WSC-2.0.2”第7节“Registration Protocol Definition”

[6] [http://en.wikipedia.org/wiki/Diffie-Hellman\\_key\\_exchange](http://en.wikipedia.org/wiki/Diffie-Hellman_key_exchange)

说明：维基百科关于D-H算法的描述，读者可通过它了解D-H的相关知识。

### 4. Push Button Configuration (PBC)

[7] “WSC-2.0.2”第11节“Push Button Configuration”

说明：规范中关于PBC的介绍只有7页，建议读者学完本章后再来看它。

# 第7章 深入理解Wi-Fi P2P

本章所涉及的源代码文件名及位置

- WifiP2pSettings.java  
packages/apps/Settings/src/com/android/settings/wifi/p2p/WifiP2pSettings.java
- WifiP2pService.java  
frameworks/base/wifi/java/android/net/wifi/p2p/WifiP2pService.java
- p2p\_supplicant.c  
external/wpa\_supplicant\_8/wpa\_supplicant/p2p\_supplicant.c
- driver.h      external/wpa\_supplicant\_8/src/drivers/driver.h
- p2p.c      external/wpa\_supplicant\_8/src/p2p/p2p.c
- driver\_nl80211.c  
external/wpa\_supplicant\_8/src/drivers/driver\_nl80211.c
- ctrl\_iface.c  
external/wpa\_supplicant\_8/wpa\_supplicant/ctrl\_iface.c
- p2p\_pd.c      external/wpa\_supplicant\_8/src/p2p/p2p\_pd.c
- p2p\_go\_neg.c  
external/wpa\_supplicant\_8/src/p2p/p2p\_go\_neg.c

## 7.1 概述

承接第6章介绍的WSC，本章将继续介绍Wi-Fi联盟推出的另外一项重要技术规范Wi-Fi P2P。该规范的商品名为Wi-Fi Direct，它支持多个Wi-Fi设备在没有AP的情况下相互连接。

在Android平台的Wi-Fi相关模块中，P2P的功能点主要集中在：

- Android Framework中的WifiP2pService，其功能和WifiService类似，用于处理和P2P相关的工作。
- wpa\_supplicant中的P2P模块。

和WSC一样，本章的分析采用如下方法。

- 首先介绍P2P所涉及的基础知识。
- 然后分析和P2P相关的模块，包括Settings、WifiP2pService以及WPAS。

## 7.2 P2P基础知识

WFA定义的P2P协议文档全名为“Wi-Fi Peer-to-Peer (P2P) Technical Specification”，目前的版本为1.1，整个篇幅160页。P2P技术使得多个Wi-Fi设备在没有AP的情况下也能构成一个网络（P2P Network，也称为P2P Group）并相互通信。

Wi-Fi P2P技术是Wi-Fi Display<sup>①</sup>的基础。在Miracast应用场景中，一台支持P2P的智能手机可直接连接上一台支持P2P的智能电视，智能手机随后将自己的视频，或者媒体资源传送给电视机去显示或播放。显然，借助P2P技术，Wi-Fi设备之间的直接相连将极大拓展Wi-Fi技术的使用场景。

**注意** 根据笔者的判断，随着越来越多的设备支持P2P和Miracast，智能终端设备之间的多屏共享和互动功能将很快得以实现。另外，撰写本章之际，Google发布了Android 4.3。在这次发布盛会上，Google推出了ChromeCast设备。目前，ChromeCast的技术实现细节还不清楚，据说是Google自己定义的Google cast协议（可参考[developers.google.com/cast](http://developers.google.com/cast)）。

下面先简单介绍一下P2P架构。

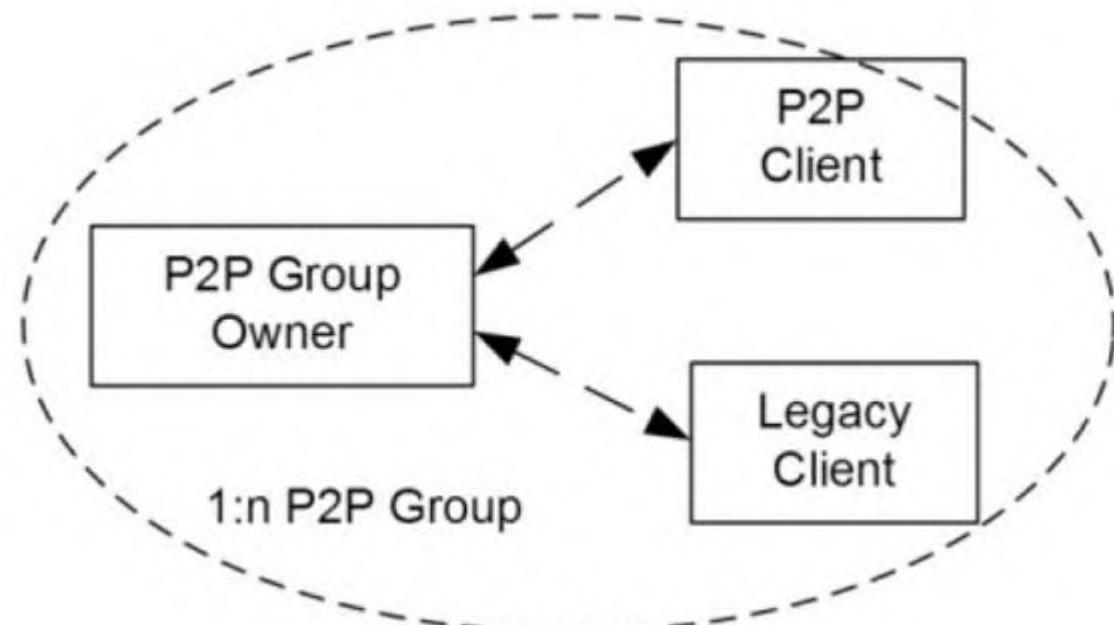
**①** 也称为Miracast，详情请参考作者的一篇博文  
<http://blog.csdn.net/innoST/article/details/8474683>。

### 7.2.1 P2P架构<sup>[1]</sup>

P2P架构中定义了三个组件，笔者将其称为“一个设备，两种角色”，分别如下。

- P2P Device: 它是P2P架构中角色的实体，可把它当做一个Wi-Fi设备。
- P2P Group Owner: Group Owner (GO) 是一种角色，其作用类似于 Infrastructure BSS中的AP。
- P2P Client: 另外一种角色，其作用类似于Infrastructure BSS中的STA。

相信读者对上面这三个组件的概念并不陌生。实际上，P2P技术模仿了 Infrastructure BSS网络结构。在组建P2P Group (即P2P Network)之前，智能终端都是一个一个的P2P Device。当这些P2P Device设备之间完成P2P协商后，其中将有一个并且只能有一个① Device来扮演 GO的角色 (即充当AP)，而其他Device来扮演Client的角色。



## 图7-1 P2P Group组织结构

图7-1展示了一个典型P2P Group的构成。和Infrastructure BSS类似，一个P2P Group中只能有一个G0。一个G0可以支持1个或多个（即图中的1：n）Client连接。

由于G0的功能类似于AP，所以周围不支持P2P功能的STA也能发现并关联到G0。这些STA称为Legacy Client。

注意 “不支持P2P功能”更准确的定义是指不能处理P2P协议。在P2P网络中，G0等同于AP，所以Legacy Client也能搜索到G0并关联上它。不过，由于Legacy Client不能处理P2P协议，所以P2P一些特有功能在这些Legacy Client中无法实现。

通过上述介绍读者会进一步发现P2P Group和Infrastructure BSS的相似性。P2P Device在构建P2P Group时，它将首先通过WSC来获取安全信息。然后，Client将利用协商好的安全设置信息去关联② G0（即P2P Group中的AP）。

这部分内容和Infrastructure BSS中STA利用WSC先协商安全信息然后再关联至AP的流程完全一样。正是这种相似性，使得P2P能充分利用现有的一些技术规范。图7-2所示为P2P及其依赖的技术项。

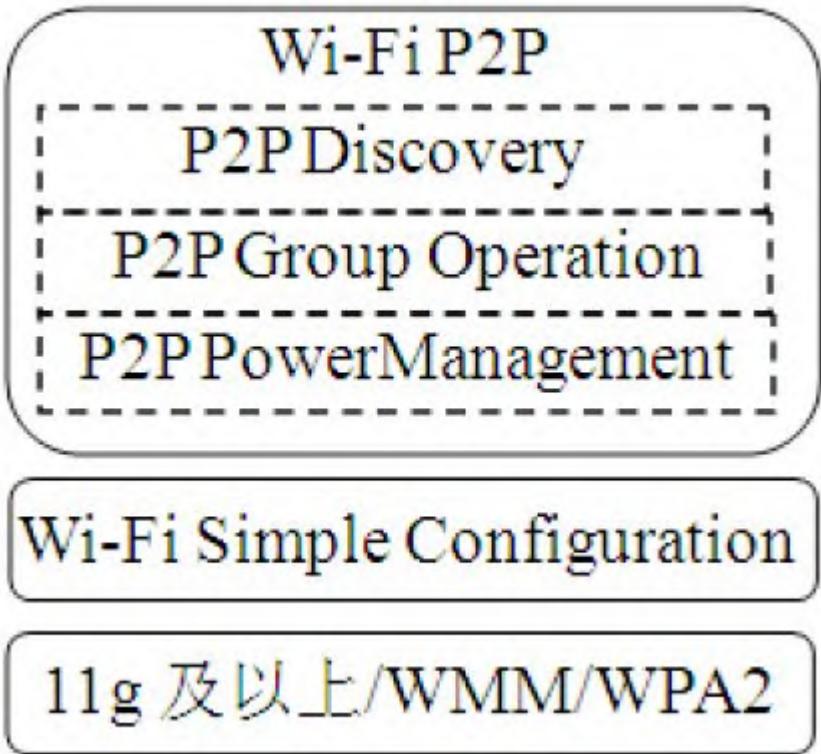


图7-2 P2P及其依赖的技术项

为了保证一定的传输速率，P2P要求P2P Device必须支持802.11g及以上的规范。其中，安全部分必须支持WPA2。由于P2P技术一个主要的应用场景就是设备之间共享媒体数据（例如前面提到的Miracast应用场景），所以P2P Device还必须支持WMM（Wi-Fi MultiMedia的缩写，一种源自802.11e的QoS服务，主要针对实时视音频数据的传输）。

P2P Client关联到G0之前，需要先通过WSC来协商安全信息，所以WSC也是P2P的依赖技术项。

在上述技术基础上，P2P规范定义了一些特有的技术项，图7-2列出了其中三种必须实现的技术项，分别是P2P Discovery、P2P Group Operation以及P2P PowerManagement。除了这三个必选技术项外，P2P规范还定义了一个可选技术项，名为Managed P2P Device Operation，该技术项定义了在企业级环境中，如何由对应的IT部门来统一配置和管理P2P设备。

在如图7-2所示的技术项中，P2P Discovery是P2P所特有的，也是其核心。本章将主要围绕它进行介绍。首先来看P2P Discovery。

**提示** P2P Group Operation讲的是G0如何管理一个Group，也就是G0的工作职责。这部分内容请读者自行学习参考资料[2]。

P2P PowerManagement和P2P设备的电源管理有关，用于节省不必要的电力损耗。由于篇幅关系，本章不讨论，感兴趣的读者自行学习参考资料[3]。

- ①** 假设这设备只组成一个P2P Network。
- ②** 注意，此处的关联指的是RSNA，其工作流程包括4-Way Handshake。

## 7.2.2 P2P Discovery技术<sup>[4]</sup>

P2P Discovery的作用很简单，就是使多个P2P Device能够互相发现并构建一个Group。根据规范，它包括四个主要技术子项。

- Device Discovery：用于P2P设备搜索周围其他支持P2P的设备。
- Service Discovery：该Device Discovery基础上，P2P还支持搜索指定的服务。这部分功能属于可选项，笔者觉得它和2.2.5节中提到的Bonjour类似。
- Group Formation：用于决定两个P2P Device谁来扮演G0，谁来扮演Client。
- P2P Invitation：用于激活一个Persistent Group（见下文解释），或者用于邀请一个Client加入一个当前已存在的Group。

提示 Group分Persistent（永久性）Group和Temporary（临时性）Group两种。我们举两个简单例子来说明二者的区别。

Temporary Group：当有文件要传给一个同事时，双方打开手机的Wi-Fi P2P功能，建立一个Group，然后传输文件，最后关闭Wi-Fi P2P。在这个过程中，G0和Client的角色分配由Group Formation来决定，这一次的G0可能是你的设备，下一次则可能是其他人的设备。对于这种Group，在建立Group过程中所涉及的安全配置信息以及和Group相关的信息（以后会见到它）都是临时的，即下一次再组建Group时，这些安全配置信息都将发生变化。

Persistent Group：在这种Group中，G0由指定设备来扮演，而且安全配置信息及Group相关信息一旦生成，后续就不会再发生变化（除非用户重新设置）。Persistent Group中的G0多见于固定用途的设备，例如打印机等。如此，除了第一次通过P2P连接到打印机时相对麻烦一点（需要利用WSC协商安全配置信息）外，后续使用的话，由于P2P设备将保存这些安全信息，所以下次再使用打印机时就能利用这些信息直接和打印机进行关联了。

由于篇幅关系，本章仅介绍上述四个知识点中最基础的Device Discovery和Group Formation，而Service Discovery和P2P Invitation的内容请读者学习完本章后再仔细研读P2P规范。

## 1. Device Discovery介绍

P2P Device Discovery虽然也是利用802.11中的Probe Request和Probe Response帧来搜索周围的P2P设备，但其步骤却比Infrastructure BSS中的无线网络搜索要复杂。举一个简单的例子，一个P2P Device除了自己要发送Probe Request帧外，还得接收来自其他设备的Probe Request帧并回复Probe Response帧。而在Infrastructure BSS中，只有AP会发送Probe Response帧。

为了加快搜索速度，P2P为Device Discovery定义了两个状态和两个阶段。

### (1) Device Discovery工作流程

先来看两个状态，分别如下。

- **Search State:** 在该状态中，P2P Device将在2.4GHz的1, 6, 11频段上分别发送Probe Request帧。这几个频段称为Social Channels。为了区别非P2P的Probe Request帧，P2P Device Discovery要求必须在Probe Request帧中包含P2P IE。
- **Listen State:** 在该状态中，P2P Device将随机选择在1, 6, 11频段中的一个频段（被选中的频段称为Listen Channel）监听Probe Request帧并回复Probe Response帧。值得指出的是，Listen Channel一旦选择好后，在整个P2P Discovery阶段就不能更改。另外，在这个阶段中，P2P Device只处理包含P2P IE信息的Probe Request帧。

再来看两个阶段，分别如下。

- **Scan Phase:** 扫描阶段。这一阶段和前面章节介绍的无线网络扫描一样，P2P Device会在各个频段上发送Probe Request帧（主动扫描）。P2P Device在这一阶段中不会处理来自其他设备的Probe Request帧。这一阶段过后，P2P Device将进入下一个阶段，即Find Phase。

- Find Phase: 虽然从中文翻译来看, Scan和Find意思比较接近, 但P2P的Find Phase却和Scan Phase大不相同。在这一阶段中, P2P Device将在Search State和Listen State之间来回切换。Search State中, P2P Device将发送Probe Request帧, 而Listen State中, 它将接收其他设备的Probe Request帧并回复Probe Response帧。

图7-3所示为两个P2P Device的Discovery流程。

- Discovery启动后, Device首先进入Scan Phase。在这一阶段, P2P设备在其支持的所有频段上都会发送Probe Request帧。
- Scan Phase完成后, Device进入Find Phase。在这一阶段中, Device将在Listen和Search State中切换。根据前面的介绍, 每一个设备的Listen Channel在Discovery开始前就已确定。例如, 图7-3中Device 1的Listen Channel是1, 而Device 2的Listen Channel是6。

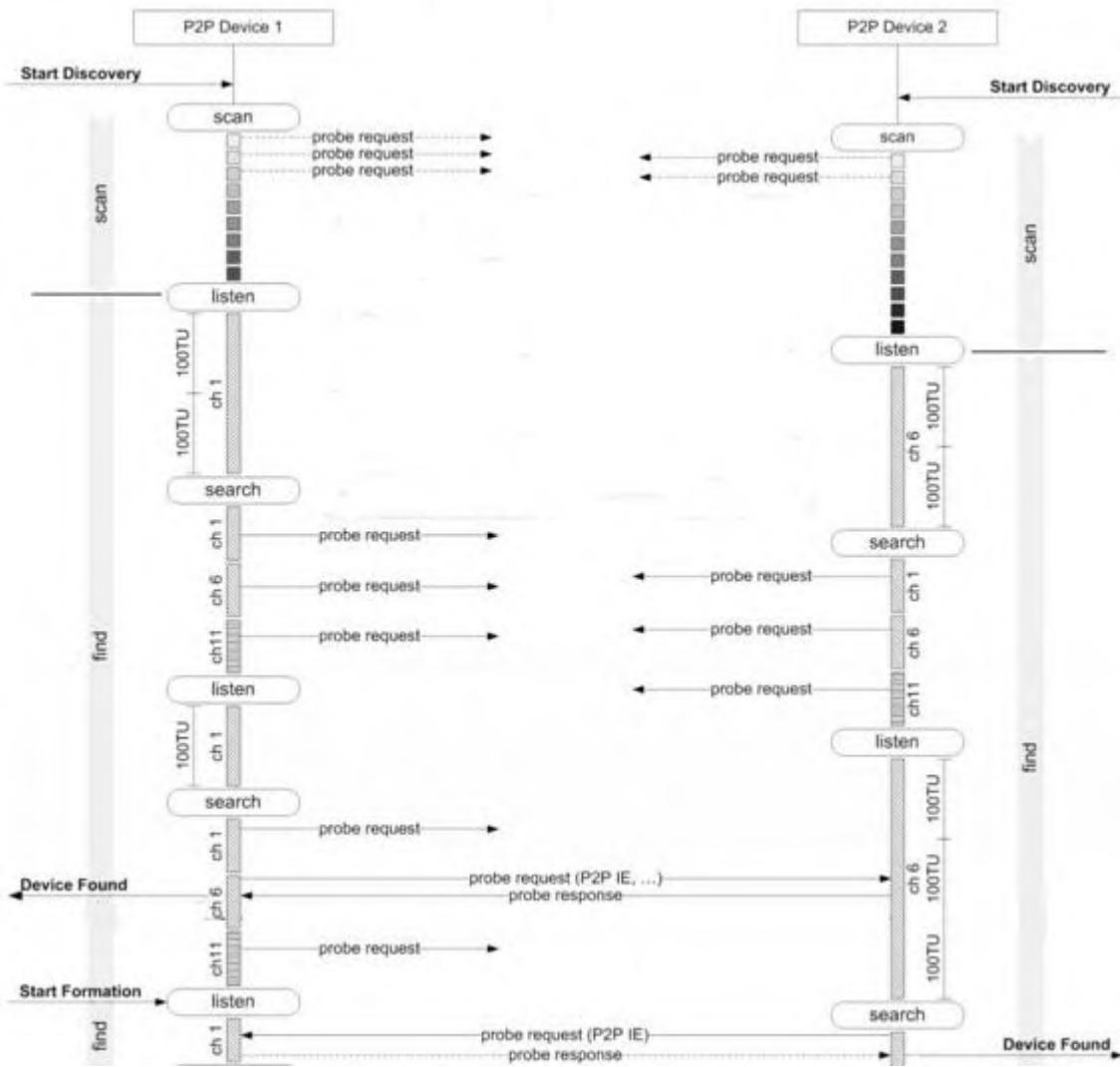


图7-3 P2P Device Discovery流程

- 在Find Phase中，P2P规范对Device处于Listen State的时间也有所规定，其时间是100TU的整数倍，倍数值是一个随机数，位于minDiscoverableInterval和maxDiscoverableInterval之间。这两个值默认为1和3，而厂商可以修改。选择随机倍数的原因是防止两个Device进入所谓的“Lock-Step怪圈”，即两个Device同时进入Listen State，等待相同的时间后又同时进入Search State。如此，双方都无法处理对方的Probe Request信息（Search State中，Device只发送Probe Request）。图7-3中，Device 1第一次在Listen State中待了2个100TU，而第二次在Listen State中待了1个100TU。

- 当Device处于Find Phase中的Search State时，它将在1、6、11频段上发送Probe Request帧。注意，只有当两个设备处于同一频段时，一方发送的帧才能被对方接收到。

**提示** P2P规范对两个状态及两个阶段的描述非常细致，甚至于对每个状态能干什么和不能干什么都有详细说明。不过，从快速掌握P2P框架的角度来看，这些内容可以省略。

了解Device Discovery的大体工作流程后，下面通过实例来看看P2P使用的Probe Request和Probe Response帧。

## (2) Probe Request帧设置

图7-4所示为Galaxy Note 2在测试P2P功能时发送的Probe Request帧。

```

+ IEEE 802.11 Probe Request, Flags: ....C
+ IEEE 802.11 wireless LAN management frame
  + Tagged parameters (267 bytes)
    ① + Tag: SSID parameter set: DIRECT-
      + Tag: Supported Rates 6(B), 9, 12(B), 18, 24(B), 36, 48, 54, [Mbit/sec]
      + Tag: HT Capabilities (802.11n D1.10)
      + Tag: DS Parameter set: Current Channel: 1
      + Tag: Vendor Specific: Microsoft: WPS
        Tag Number: Vendor Specific (221)
        Tag length: 153
        OUI: 00-50-f2 (Microsoft)
        Vendor Specific OUI Type: 4
        Type: WPS (0x04)
      + Version: 0x10
      + Request Type: Enrollee, open 802.1X (0x01)
      + Config Methods: 0x2108
      + UUID E
      + Primary Device Type
      + RF Bands: 2.4 and 5 GHz (0x03)
      + Association State: Not associated (0x0000)
      + Configuration Error: No Error (0x0000)
      + Device Password ID: PIN (default) (0x0000)
      + Manufacturer: SAMSUNG ELECTRONICS
      + Model Name: SAMSUNG MOBILE
      + Model Number: 2012
      + Device Name: Android_4aa9
      + Vendor Extension
        + Tag: Vendor Specific: wi-FiAll: P2P
          Tag Number: Vendor Specific (221)
          Tag length: 17
          OUI: 50-6f-9a (wi-FiAll)
          Vendor Specific OUI Type: 9
          + P2P Capability: Device 0x27 Group 0x0
          + Listen Channel: Operating Class 81 Channel Number 1
        + Tag: Vendor Specific: Broadcom
        + Tag: Vendor Specific: Epigram: HT Capabilities (802.11n D1.10)

```

图7-4 P2P Probe Request帧实例

图7-4所示的P2P Probe Request帧实例中有三个地方（用黑框标明）值得注意。

①中为SSID，其取值为“DIRECT-”。大家不要小看它，“DIRECT-”就是P2P规范中定义的P2P Wildcard SSID。

②中为WSC IE。WSC IE中的Device Name属性表明了发送者的设备名。另外，Probe Request发送者可以利用Primary Device Type属性来搜索指定类型的接收者（相关内容，读者可参考第6章“Configuration Methods和Primary Device Type属性”）。

③中为P2P IE。和WSC IE一样，它也属于802.11规范中Vendor自定义的IE（Element ID取值为221，参考6.3.1节WSC IE和Attribute介绍）。OUI取值为0x50-6F-9A-09。其中50-6F-9A是Wi-Fi Alliance组织的OUI，09代表P2P。P2P IE的组织结构也是由一个一个的Attribute组成。此处的P2P IE包含P2P Capability和Listen Channel两个属性，其详情见下文。

图7-4所示的P2P IE中包含了P2P Capability和Listen Channel两个属性。其中，P2P Capability的示例如图7-5所示。

图7-5中所示的P2P Capability属性表示设备对P2P各种特性支持的情况。它分为Device Capability Bitmap和Group Capability Bitmap两项考核指标。这两个Capability Bitmap长度都为1字节，每一位都代表一种P2P特性（这也是它们的名称中都带有Bitmap一词的原因）。表7-1和表7-2分别展示了这两个Capability中每一位的含义。

```
□ P2P Capability: Device 0x27 Group 0x0
  Attribute Type: P2P Capability (2)
  Attribute Length: 2
  Device Capability Bitmap: 0x27
    .... .1 = Service Discovery: 0x01
    .... .1. = P2P Client Discoverability: 0x01
    .... .1.. = Concurrent Operation: 0x01
    .... 0... = P2P Infrastructure Managed: 0x00
    ...0 .... = P2P Device Limit: 0x00
    ..1. .... = P2P Invitation Procedure: 0x01
  Group Capability Bitmap: 0x00
    .... .0 = P2P Group Owner: 0x00
    .... .0. = Persistent P2P Group: 0x00
    .... .0.. = P2P Group Limit: 0x00
    .... 0... = Intra-BSS Distribution: 0x00
    ...0 .... = Cross Connection: 0x00
    ..0. .... = Persistent Reconnect: 0x00
    .0.. .... = Group Formation: 0x00
```

图7-5 P2P Capability示例

表 7-1 Device Capability Bitmap 各个位的作用

位	名 称	说 明
0	Service Discovery	是否支持 Service Discovery
1	P2P Client Discoverability	是否支持 P2P Client Discoverability (详情见下文)
2	Concurrent Operation	是否支持 P2P 和 STA 同时工作
3	P2P Infrastructure Managed	是否支持 Managed P2P Device Operation
4	P2P Device Limit	值为 1 表示此 P2P Device 只能加入一个 P2P Group。通过 Virtual Interface, 一个 Wi-Fi 设备可以加入不同的 P2P Group 并扮演不同角色
5	P2P Invitation Procedure	表示此 P2P Device 是否能处理 Invitation Procedure。Invitation Procedure 描述了如果邀请一个 P2P Device 加入某一个 P2P Group
6-7	保留	

表7-1中的P2P Client Discoverability对应于下面所述的应用场景。

- P2P Device A已经加入了一个P2P Group 1。在Group 1中，它扮演Client的角色。
- P2P Device B（不在Group 1中）指定搜索P2P Device A。由于Device A已经扮演了Client的角色，所以它不会回复Probe Response。不过，Group 1的GO却存储有当前与它关联的Client信息，即Group 1的GO了解Device A的信息。
- 如果Device A支持Client Discoverability，那么Group 1的GO、Device A以及Device B将借助Device Discoverability Request/Response帧来获取相关信息。这部分流程比较复杂，感兴趣的读者不妨阅读参考资料[2]。

注意 通过上面关于P2P Client Discoverability的描述可知，P2P Device Discovery的内容远比图7-3所示的流程复杂。实际上，图7-3描述的只是P2P Device Discovery的一种情况。P2P规范中介绍的Device Discovery一共包含有如下几种情况。

- 图7-3所示的两个P2P Device搜索，这两个Device都支持P2P并且当前都没有加入P2P Group。
- 一个未加入Group的P2P Device搜索位于某个P2P Group中的GO。
- 一个未加入Group的P2P Device搜索位于某个P2P Group中的Client。

- Legacy Client搜索P2P Group中的GO。
- 一个P2P Device搜索另一个已经加入某个Infrastructure BSS的Device（通过Concurrent Operation来同时支持P2P和STA）。
- GO搜索周围的P2P Device和Services。

本章将只分析第一种情况，感兴趣的读者请在学完本章后再研究其他几种情况。

表7-2所示为Group Capability Bitmap各个位的作用。

表 7-2 Group Capability Bitmap 各个位的作用

位	名 称	说 明
0	P2P Group Owner	表示此 P2P Device 是否为 GO
1	Persistent P2P Group	表示此 P2P Device 是否为 Persistent Group 的 GO
2	P2P Group Limit	如果 P2P Device 为 GO，该位表示它是否还能添加新的 Client。值为 1 表示不能再添加新的 Client 到此 Group 中
3	Intra-BSS Distribution	表示是否为 Group 中的 Client 提供数据分发服务 (data distribution service between Clients)。如果该 Device 不是 GO，或者不能提供此服务 (作为 Client)，则该位被置为 0
4	Cross Connection	Cross Connection 在 Concurrent Operation 上更进一步，它使得 P2P Group 中得 Client 能访问 WLAN。只有 GO 和那些提供 Cross Connection Service 的 Client 才会设置该位为 1
5	Persistent Reconnect	如果作为 Persistent Group 的 GO，是否支持自动 (即无须用户干预) 重新连接到 Persistent Group
6	Group Formation	和 Group Formation 流程有关，表示此 Device 在 Group Formation 的 Provisioning 阶段中是否扮演 GO 的角色
7	保留	

```
□ Listen Channel: Operating Class 81 channel Number 1
Attribute Type: Listen Channel (6)
Attribute Length: 5
Country String: XX\004
Operating Class: 81
channel Number: 1
```

图7-6 Listen Channel属性

接着来看Listen Channel属性，它代表P2P Device在Listen State时将使用哪个频段，其内容如图7-6所示。可知包含三个字段。

- **Country String:** 该字段长3字节。其中，前两字节表示国家码（图7-6中的“XX”代表non-country entity，该值表示当前没有明确的国家）。最后一字节的“04”表示后面的Operating Class定义需参考资料[5]中的表J-4。
- **Operating Class:** 指明Listen State时使用的频率波段的类别，此处为81。
- **Channel Number:** 指明Listen State使用的频段。

**提示** 根据参考资料[5]的表J-4（图7-7），Operating Class 81表示频段的起始频率是2.407GHz，分为13个频段，每个频段的间隔为25MHz。另外，表J-1定义了美国的Operating Class，表J-2定义了欧洲的Operating Class，表J-3定义了日本的Operating Class，表J-4定义了其他国家的Operating Class情况。

**Table J-4—Global operating classes**

Operating class	Non-global operating class(es)	Channel starting Frequency (GHz)	Channel spacing (MHz)	Channel set	Behavior limits set
1-80		Reserved	Reserved	Reserved	Reserved
81	J-1-12, J-2-4, J-3-30	2.407	25	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13	
82	J-3-31	2.414	25	14	

图7-7 表J-4截图

**特别注意** P2P IE还定义了一个名为Operating Channel的属性，其组成结构和Listen Channel完全一样，但含义却大不相同。Operating Channel表示假设该设备扮演G0，则由它组建的P2P Group将在哪个频段上工作。并且其频段不限于Social Channels中指定的那三个频段。

除了包含P2P Capability和Listen Channel两个属性外，Probe Request中的P2P IE还可以包含其他一些属性。这部分知识见参考资料[6]。

最后，总结P2P规范中对Probe Request帧的要求。

- SSID IE必须设置为P2P Wildcard SSID，即”DIRECT-”。
- 必须包含P2P IE。
- 802.11 MAC帧头的地址域<sup>①</sup>中，Destination Address域(Address1)必须为广播地址(FF: FF: FF: FF: FF: FF)或者为目标设备的P2P Device Address(特别注意，详情见下文)，BSSID域(Address3)必须为广播地址。

**特别注意** P2P规范定义了两种类型的地址，一种是P2P Device Address，另外一种是P2P Interface Address。一个P2P Device在加入P2P Group前，将使用Device Address开展Device Discovery等工作。对一个P2P Device而言，其P2P Device Address是唯一的(作用等同于MAC地址)。而当P2P Device加入P2P Group后，它和Group中其他成员交互时将使用P2P Interface Address。另外，由于一个P2P Device可同时加入多个P2P Group，所以在每个P2P Group中，该设备必须使用不同的P2P Interface Address。最后，当一个Group结束后，Device在该Group中使用的P2P Interface Address也就相应作废了。一般而言，P2P Device Address和P2P Interface Address不同。以笔者的Galaxy Note 2为例，其P2P Device Address为92: 18: 7c: 69: 88: e2，而P2P Interface Address为92: 18: 7c: 69: 08: e2。

接着来看Probe Response帧。

### (3) Probe Response帧设置

图7-8所示的P2P Probe Response帧包含WSC IE和P2P IE。在P2P IE中，除了有P2P Capability属性外，还包含一个名为P2P Device Info的属性。P2P Device Info用于描述发送设备的一些情况。示例中，该属性的取值如图7-9所示。

```
④ IEEE 802.11 Probe Response, Flags: .....C
② IEEE 802.11 wireless LAN management frame
③ Fixed parameters (12 bytes)
③ Tagged parameters (299 bytes)
④ Tag: SSID parameter set: DIRECT-
④ Tag: Supported Rates 6(B), 9, 12(B), 18, 24(B), 36, 48, 54, [Mbit/sec]
④ Tag: DS Parameter set: Current Channel: 1
④ Tag: ERP Information
④ Tag: ERP Information
④ Tag: HT Capabilities (802.11n D1.10)
④ Tag: HT Information (802.11n D1.10)
④ Tag: Vendor Specific: Broadcom
④ Tag: Vendor Specific: Microsoft: WMM/WME: Information Element
③ Tag: Vendor Specific: Microsoft: WPS
  Tag Number: Vendor Specific (221)
  Tag length: 150
  OUI: 00-50-f2 (Microsoft)
  Vendor Specific OUI Type: 4
  Type: WPS (0x04)
④ Version: 0x10
④ Wifi Protected Setup State: Not configured (0x01)
④ Device Password ID: PIN (default) (0x0000)
④ Response Type: Enrollee, Info only (0x00)
④ UUID E
④ Manufacturer: SAMSUNG ELECTRONICS
④ Model Name: SAMSUNG MOBILE
④ Model Number: 2012
④ Serial Number: 19691101
④ Primary Device Type
④ Device Name: Android_2eab
④ Config Methods: 0x4388
④ Vendor Extension
③ Tag: Vendor Specific: wi-FiAll: P2P
  Tag Number: Vendor Specific (221)
  Tag length: 45
  OUI: 50-6f-9a (wi-FiAll)
  Vendor Specific OUI Type: 9
④ P2P Capability: Device 0x27 Group 0x0
④ P2P Device Info
```

图7-8 P2P Probe Response帧示例

```
□ P2P Device Info
  Attribute Type: P2P Device Info (13)
  Attribute Length: 33
  P2P Device address: 92:18:7c:39:05:91 (92:18:7c:39:05:91)
  Config Methods: 0x0188
    .... .... .... 0 = USBA (Flash Drive): 0x0000
    .... .... .... 0. = Ethernet: 0x0000
    .... .... .... 0.. = Label: 0x0000
    .... .... .... 1... = Display: 0x0001
    .... .... .... 0 .... = External NFC Token: 0x0000
    .... .... .... 0. .... = Integrated NFC Token: 0x0000
    .... .... .... 0.. .... = NFC Interface: 0x0000
    .... .... .... 1.... .... = PushButton: 0x0001
    .... .... 1 .... .... = Keypad: 0x0001
  Primary Device Type: 000a0050f2040005
  Primary Device Type: Category: 10
  Primary Device Type: OUI: 0050f204
  Primary Device Type: Subcategory: 5
  Number of Secondary Device Types: 0
  Device Name attribute type: 0x1011
  Device Name attribute length: 12
  Device Name: Android_2eab
```

图7-9 P2P Device Info示例

仔细观察图7-9，读者会发现，P2P Device Info包含了一些第6章介绍WSC时提到的属性，例如Primary Device Type、Config Methods等。关于WSC属性，可回顾第6章WSC IE和Attribute的相关介绍。P2P Device address字段用来描述发送设备的P2P Device Address。

**注意** 图7-9中所示的WSC属性中，Primary Device Type和Config Methods这两个属性没有包含对应的Attribute ID以及Length字段，只包含了Value字段，而Device Name属性则包含了Attribute ID、Length以及Value字段。

以上介绍了P2P Device Discovery的流程及相关的Probe Request/Response帧和P2P IE等内容。值得指出的是，本节是以两个未加入P2P Group的P2P Device互相搜索为例来介绍Device Discovery流程的，它属于P2P规范Device Discovery各种case中较简单的一种。

**提示** 根据笔者阅读P2P规范的心得，P2P Device Discovery实际所包含的内容非常丰富，而且难度比较大。在此，建议读者在学完本章

后再去研读P2P规范。

## 2. Group Formation介绍

当P2P Device A通过Device Discovery找到周围的一个P2P Device B后，Device A就可以开展Group Formation流程以准备构造一个P2P Group。Group Formation也包含两个阶段，分别如下。

- G0 Negotiation：在这一阶段中，两个Device要协商好由谁来做G0。
- Provisioning：G0和Client角色确定后，两个Device要借助WSC来交换安全配置信息。此后，Client就可以利用安全配置信息关联上G0。这个流程和第6章介绍的WSC流程一样，这部分内容请读者参考第6章。

G0 Negotiation过程中P2P设备会利用一种名为P2P Public Action类型的帧交换信息，所以下面先来认识一下P2P Public Action帧。

**提示** 除了G0 Negotiation外，P2P Invitation、Device Discoverability和Provision Discovery流程也会用到P2P Public Action帧。

### (1) P2P Public Action帧

Action帧是802.11管理帧的一种，其Type和SubType取值可参考3.3.5节帧类型、From/To DS介绍。Action帧的作用如其名称中的”Action”所示，发送方利用Action帧携带一些请求信息，从而使得接收方能对应进行一些处理。

Action帧Frame Body的结构比较简单，仅包含Category和Action Detail两个部分，Action Detail随Category的不同而变化。常用的Category<sup>[7]</sup> 如下。

- 值为0，表示Spectrum Management，用于Spectrum Measurement。
- 值为4，表示Public，P2P规范会使用这种类型的Action帧。
- 值为5，表示Radio Management，它和Radio Measurement有关。

- 值为127，表示Vendor Specific，它和具体的厂商有关。P2P规范也会使用这种类型的Action帧。

如上所述，P2P将使用Public Action和Vendor Specific这两种类型的Action帧。本节先介绍其中的Public Action帧。

802.11中Public Action帧又有多种子类型，而P2P属于Public Action中的Vendor Specific子类型。P2P使用的Action帧Frame Body的组成结构如表7-3所示。

表 7-3 P2P Public Action 结构

字段名	长度/字节	取值	说明
Category	1	0x04	表示为 Public Action 帧
Action Field	1	0x09	Public Action 帧的一种，表示 Vendor Specific

(续)

字段名	长度/字节	取值	说明
OUI	3	0x50-6F-9A	WFA OUI
OUI Type	1	0x09	09 表示 WFA P2P 1.0 版本
OUI SubType	1	见表 7-4	定义不同的 P2P Public Action 帧，见表 7-4
Dialog Token	1		对一次 Request/Response 交互的标识
Elements	可变长		具体内容随 SubType 不同而不同

表7-4所示为OUI Subtype取值，不同的Subtype代表不同的P2P Public Action帧。

表 7-4 OUI SubType 取值

Type	说明	Type	说明
0	GO Negotiation Request	4	P2P Invitation Response
1	GO Negotiation Response	5	Device Discoverability Request
2	GO Negotiation Confirmation	6	Device Discoverability Response
3	P2P Invitation Request	7	Provision Discovery Request
4	P2P Invitation Response	8	Provision Discovery Response

下面来看GO Negotiation流程，包含三次P2P Public Action帧交换。

## (2) GO Negotiation流程

图7-10为GO Negotiation涉及的三次帧交换流程。

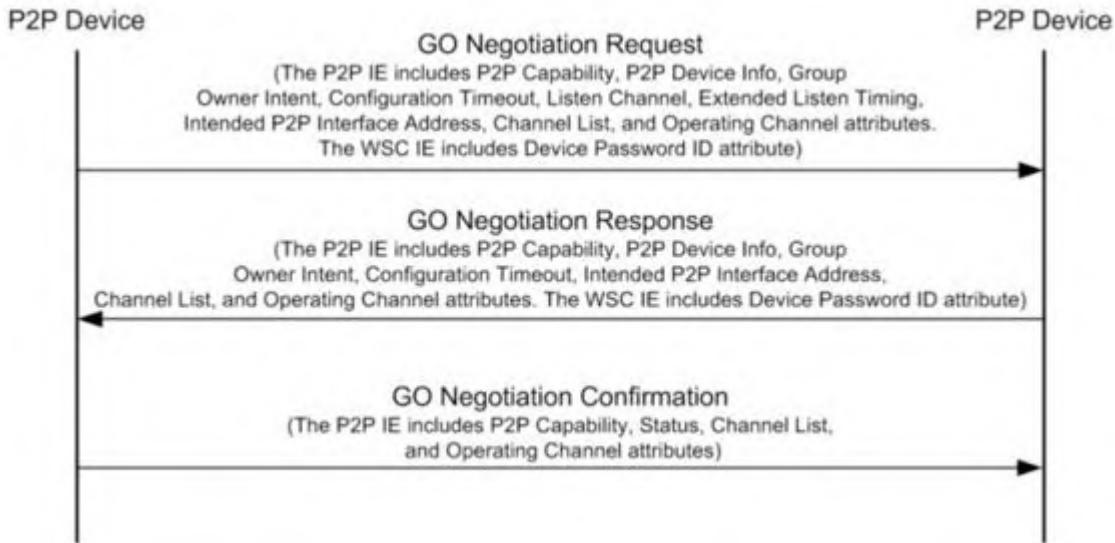


图7-10 GO Negotiation流程

由图7-10可知，GO Negotiation（以下简称GON）流程包括GON Request、GON Response和GON Confirmation三次帧交换。这三次帧交换并不涉及什么复杂的计算，只是双方交换一些信息，从而谁来扮演GO。另外，图7-10也列出了这三个帧中可包含的一些P2P和WSC属性信息。下面将直接分析这三个帧的内容。

1) 首先是GON Request，实例如图7-11所示。GON Request帧中P2P IE包含了一些之前没有提到的属性，下面分别介绍。

首先是GO Intent属性，该属性代表发送设备扮演GO的渴望程度，其内部包含一个名为GO Intent的字段。该字段长1字节，目前使用的仅是该字节的前八位。

```
□ IEEE 802.11 wireless LAN management frame
  □ Fixed parameters
    Category code: Public Action (4)
    Public Action: Vendor Specific (0x09)
    OUI: 50-6f-9a (Wi-FiAll)
    Subtype 9
    P2P Public Action Subtype: GO Negotiation Request (0)
    P2P Public Action Dialog Token: 1
  □ Tagged parameters (135 bytes)
    □ Tag: Vendor Specific: wi-FiAll: P2P
      Tag Number: Vendor Specific (221)
      Tag length: 106
      OUI: 50-6f-9a (wi-FiAll)
      Vendor Specific OUI Type: 9
      □ P2P Capability: Device 0x27 Group 0x8
      □ Group Owner Intent: Intent 7 Tie breaker 0
      □ Configuration Timeout: GO 1000 msec, client 200 msec
      □ Listen Channel: Operating class 81 channel Number 1
      □ Intended P2P Interface Address: 92:18:7c:39:85:91
      □ Channel List
      □ P2P Device Info
      □ Operating Channel: Operating class 81 Channel Number 1
    □ Tag: Vendor Specific: Microsoft: WPS
      Tag Number: Vendor Specific (221)
      Tag length: 25
      OUI: 00-50-f2 (Microsoft)
      Vendor Specific OUI Type: 4
      Type: WPS (0x04)
      □ Version: 0x10
      □ Device Password ID: PushButton (0x0004)
      □ vendor Extension
```

图7-11 GON Request实例

```
□ Group Owner Intent: Intent 7 Tie breaker 0
  Attribute Type: Group Owner Intent (4)
  Attribute Length: 1
  ...0 111. = Group Owner Intent: 7
  .... .0 = Group Owner Intent Tie Breaker: 0
```

图7-12 GO Intent属性

图7-12所示为图7-11中GO Intent属性的取值情况。

- 第0位叫做“Tie Breaker”（意思为决胜因素），Tie Breaker的取值为随机的0或1。
- 第1~7位为Intent值，取值为0~15。值越高，代表越想成为G0。15表示该发送设备必须充当G0的角色。Intent默认值为7。

在GON三次帧交换中，GON Request和GON Response都携带G0 Intent，分别代表了交互双方想成为G0的渴望程度，那么到底谁会成为G0呢？为此，规范制订了一个游戏规则，如图7-13所示。

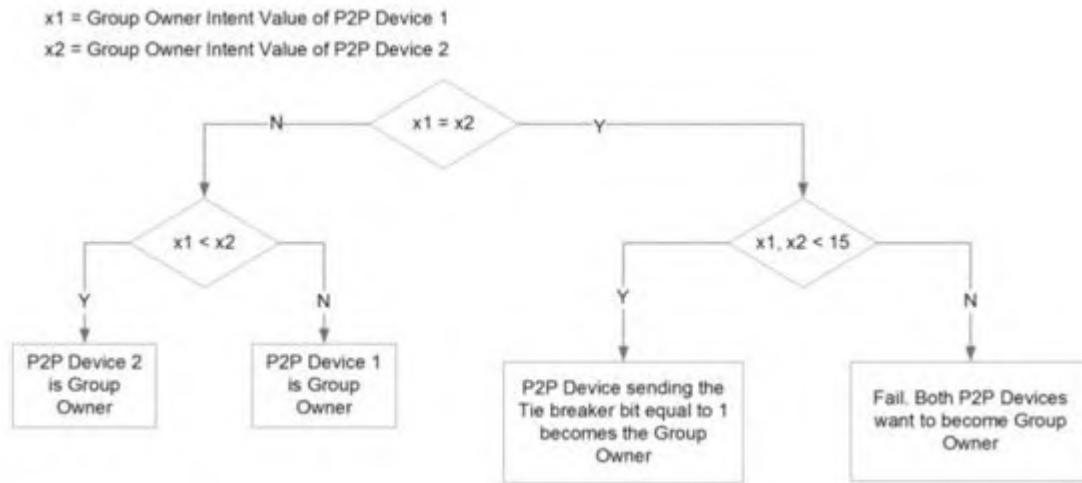


图7-13 G0角色扮演规则

由图7-13可知：

- 如果Device A的G0 Intent小于Device B的G0 Intent，则Device B将成为G0。
- 一般情况下，Device A和Device B的G0 Intent都将使用默认值（值为7）。这种情况下，Tie Breaker的取值是关键，该字段值为1的Device A将成为G0。由于Tie Breaker为随机值，所以两个设备的Tie Breaker取值相同的几率非常低。
- “一山不能容二虎”，如果两个设备都想扮演G0（如G0 Intent都为15），则GON失败，谁都成为不了G0。

来看GON Request帧中第二个P2P属性Configuration Timeout，该属性实例如图7-14所示。Configuration Timeout属性包含两个长度为1字节的字段，分别是GO Configuration Timeout和Client Configuration Timeout，它们表明Device进入GO或Client角色的超时时间（这两个字段的取值为10ms的倍数）。

例如，图7-14中的GO Configuration Timeout字段取值为100，对应超时时间为1000ms，表示Device如果要扮演GO的话，必须在1秒内完成相关准备工作。

```
Configuration Timeout: GO 1000 msec, client 200 msec
Attribute Type: Configuration Timeout (5)
Attribute Length: 2
GO Configuration Timeout: 100
Client Configuration Timeout: 20
```

图7-14 Configuration Timeout属性

```
Channel List
Attribute Type: Channel List (11)
Attribute Length: 24
Country String: XX\004
Operating Class: 81
Number of Channels: 13
Channel List: 0102030405060708090a0b0c0d
Operating Class: 124
Number of Channels: 4
Channel List: 95999da1
```

图7-15 Channel List属性

下面来看Channel List属性，它代表发送设备支持的Wi-Fi频段信息，图7-15所示为此属性的实例。

Channel List属性的组成比较简单，包括一个Country String和一个或多个Channel信息。由于笔者的Galaxy Note 2同时支持2.4GHz（对

于Operating Class为81) 和5GHz (对于Operating Class为124) 两个频段，所以Channel List包含了两个Channel信息。

**提示** 图7-15中Channel List字段中列举的是十六进制的频道号。其中，5GHz段包含4个频道，分别是149 (0x95)、153 (0x99)、157 (0x9D) 和161 (0xA1)。关于这些信息见参考资料[5]。

图7-11中最后一个比较重要的属性就是Intended P2P Interface Address，代表P2P设备加入Group后将使用的MAC地址。

在Android平台中，当某个设备收到GON Request帧后，将弹出一个如图7-16所示的提示框以提醒用户。如果用户选择“接受”，则系统将继续后续的工作，否则将终止Group Formation流程。



图7-16 GON Request接收者提示框

2) 接着来看GON Response帧，图7-17为实例。GON Response帧也包含一些新的P2P属性。首先来看其中的Status属性，该属性代表某一个Action帧的处理结果，值为0表示处理成功，其他值表示失败。

另一个比较重要的属性是P2P Group ID，其实例如图7-18所示。P2P Group ID用于唯一标示一个P2P Group，该属性必须包含P2P Device

Address以及SSID两个字段。其中，SSID的格式遵循如下规则。

- 开头必须是“DIRECT-xy”，xy为随机的两个大小写字母或数字，例如“ny”。
- 规范对规定“DIRECT-xy”之后的内容，Android中会加上设备名，例如图中的“Android\_4aa9”。

```
IEEE 802.11 wireless LAN management frame
  Fixed parameters
    Category code: Public Action (4)
    Public Action: vendor Specific (0x09)
    OUI: 50-6f-9a (wi-FiAll)
    Subtype 9
    P2P Public Action Subtype: GO Negotiation Response (1)
    P2P Public Action Dialog Token: 1
  Tagged parameters (162 bytes)
    Tag: Vendor Specific: wi-FiAll: P2P
      Tag Number: vendor Specific (221)
      Tag length: 133
      OUI: 50-6f-9a (wi-FiAll)
      Vendor Specific OUI Type: 9
      + Status: 0 (Success)
      + P2P Capability: Device 0x27 Group 0x8
      + Group Owner Intent: Intent 7 Tie breaker 1
      + Configuration Timeout: Go 1000 msec, client 200 msec
      + Operating Channel: Operating Class 81 Channel Number 1
      + Intended P2P Interface Address: 92:18:7c:69:08:e2
      + Channel List
      + P2P Device Info
      + P2P Group ID: 92:18:7c:69:88:e2
    Tag: Vendor Specific: Microsoft: WPS
      Tag Number: vendor Specific (221)
      Tag length: 25
      OUI: 00-50-f2 (Microsoft)
      Vendor Specific OUI Type: 4
      Type: WPS (0x04)
      + Version: 0x10
      + Device Password ID: PushButton (0x0004)
      + Vendor Extension
```

图7-17 GON Response帧实例

```
□ P2P Group ID: 92:18:7c:69:88:e2
  Attribute Type: P2P Group ID (15)
  Attribute Length: 28
  P2P Device address: 92:18:7c:69:88:e2 (92:18:7c:69:88:e2)
  SSID: DIRECT-ny-Android_4aa9
```

图7-18 P2P Group ID属性

注意 只有会成为GO的P2P Device在其发送的Group Response帧中才会包含P2P Group属性。

3) 最后看GON Confirmation帧，它是对GON Response的确认。图7-19为GON Confirmation帧实例。

```
□ IEEE 802.11 wireless LAN management frame
  □ Fixed parameters
    Category code: Public Action (4)
    Public Action: Vendor Specific (0x09)
    OUI: 50-6f-9a (Wi-FiAll)
    Subtype 9
    P2P Public Action Subtype: GO Negotiation Confirmation (2)
    P2P Public Action Dialog Token: 1
  □ Tagged parameters (50 bytes)
    □ Tag: Vendor Specific: Wi-FiAll: P2P
      Tag Number: Vendor Specific (221)
      Tag length: 48
      OUI: 50-6f-9a (Wi-FiAll)
      Vendor Specific OUI Type: 9
    □ Status: 0 (Success)
    □ P2P Capability: Device 0x27 Group 0x0
    □ Operating Channel: Operating Class 81 channel Number 1
    □ Channel List
```

图7-19 GON Confirmation帧实例

图7-19所示的GON Confirmation帧比较简单。不过，如果发送者将扮演GO角色，其发送的GON Confirmation帧必须包含P2P Group ID属性。

提示 P2P规范对GON三个帧包含的P2P属性及WSC属性都有明确要求，读者可通过参考资料[8]来学习相关知识。

GON流程执行完毕后，P2P Device的角色也就随之确定，下一步的工作就是Provisioning，即交互双方利用WSC来交换安全配置信息，这部分工作在第6章已经详细介绍过了，此处不赘述。细心的读者可能会发现，P2P Public Action帧中还存在着“Provision Discovery Request/Response”类型的帧，它们是干什么用的呢？来看下文。

### (3) Provision Discovery介绍

Provision Discovery也和WSC有关。第6章中曾介绍WSC中的 Configuration Methods属性，它代表了Wi-Fi设备所支持的WSC配置方法。WSC定义了一共13种配置方法。图7-20是该属性的内容。

```
Config Methods: 0x0080
Data Element Type: Config Methods (0x1008)
Data Element Length: 2
Configuration Methods: 0x0080
.... .... .... 0 = USB: 0x0000
.... .... .... 0. = Ethernet: 0x0000
.... .... .... 0.. = Label: 0x0000
.... .... .... 0... = Display: 0x0000
..0. .... .... = Virtual Display: 0x0000
.0.. .... .... = Physical Display: 0x0000
.... .... 0 .... = External NFC: 0x0000
.... .... 0. .... = Internal NFC: 0x0000
.... .... 0.. .... = NFC Interface: 0x0000
.... .... 1.... .... = Push Button: 0x0001
.... ..0. .... .... = Virtual Push Button: 0x0000
.... .0.. .... .... = Physical Push Button: 0x0000
.... ...0 .... .... = Keypad: 0x0000
```

图7-20 Config Method属性

了解上述信息后，请读者思考一个问题。两个P2P Device如何知道对方使用的是哪种WSC配置方法呢？显然，如果双方使用不同的WSC配置方法，这个Group就无法建立。

为了解决这个问题，P2P规范定义了Provision Discovery（PD）流程，该流程就是为了确定交互双方使用的WSC方法。

Provision Discovery包含PD Request和PD Response两次帧交换，其中起到决定作用的信息是WSC IE的Config Method属性。图7-21所示为PD Request和PD Response帧实例。



图7-21 PD Request/Response帧实例

图7-21中的左图为PD Request帧，右图是PD Response帧。根据P2P规范：

- PD Request帧的发送者在WSC IE的Config Method属性中设置想使用的WSC配置方法，注意，一次只能设置一种WSC配置方法。图7-21的PD Request帧发送者使用了Push Button方法。
- PD Request帧的接收者如果支持PD Request帧发送者设置的WSC配置方法，则它将在回复的PD Response帧中对应设置该WSC配置方法。例如图7-21的右图也设置了Push Button位，表示PD Response帧发送者支持PBC。
- 如果PD Request帧的接收者不支持发送者设置的WSC配置方法，它回复的PD Response帧中，Config Method值为0。这样，PD Request帧发送者将重新选择一种新的配置方法，然后再次通过PD Request帧向对方发起请求以判断对方是否支持这个新的配置方法。

简而言之，如果PD Request接收者支持发送者设置的WSC配置方法，则它在PD Response帧中将设置相同的Config Method属性值，否则设置Config Method值为0。

**提示** 让笔者颇感纳闷的一件事情是，当PD Request接收者不支持发送者设置的WSC配置方法时，它为什么不在PD Response帧中设置Config Method为自己所支持的WSC方法，而仅是通过设置Config Method值为0来简单告诉发送者其设置的配置方法无效呢？感兴趣的读者不妨对此问题展开讨论。

由于WSC配置需要用户参与，所以PD另外一个作用就是提醒用户执行相应的动作。例如让用户输入PIN码等。

注意，Provision Discovery不属于Group Formation，它的出现是为了解决如下所述的一个问题。

为了达到最好的用户体验，P2P规范要求Group Formation（即GON和Provisioning两个部分）须在15秒内完成。但WSC安全配置往往需要用户参与（例如输入PIN码）。这些操作比较费时，所以WSC规范（Provisioning遵守WSC规范）对此设置的时间限制是2分钟。也就是说，光Group Formation中的Provisioning就可能耗费最长2分钟。如何解决2分钟和15秒之间的矛盾呢？P2P规范提出了Provision Discovery这一方法，其作用如下。

- PD用于Group Formation之前，以提前邀请用户输入WSC安全配置所需信息（例如让用户输入PIN码等）。
- PD获取的安全信息（如PIN码）可直接用于后续Group Formation的Provisioning，从而避免了在Provisioning过程中让用户输入PIN码。

根据规范，Group Formation只包含两个部分。

- GO Negotiation，在此过程中，P2P Device将利用GO Intent来确定谁将扮演GO，谁将扮演Client。GO Negotiation涉及三次帧GON帧交换。
- GO和Client角色确定后就相当于确定了Infrastructure BSS中的AP和STA，下一步工作就是双方通过WSC流程交换安全配置信息。在P2P

中，该部分称为Provisioning。

Provision Discovery是为了加快Group Formation速度而设计的一种方法，它能在Group Formation正式开始前通知用户输入与WSC安全配置相关的信息。

### 7.2.3 P2P工作流程

P2P规范中附录A<sup>[9]</sup> 通过定义一个状态机介绍了P2P的整体工作流程，笔者觉得以此作为本章P2P理论知识的总结是最好不过了。该状态机的状态定义及切换如图7-22所示。

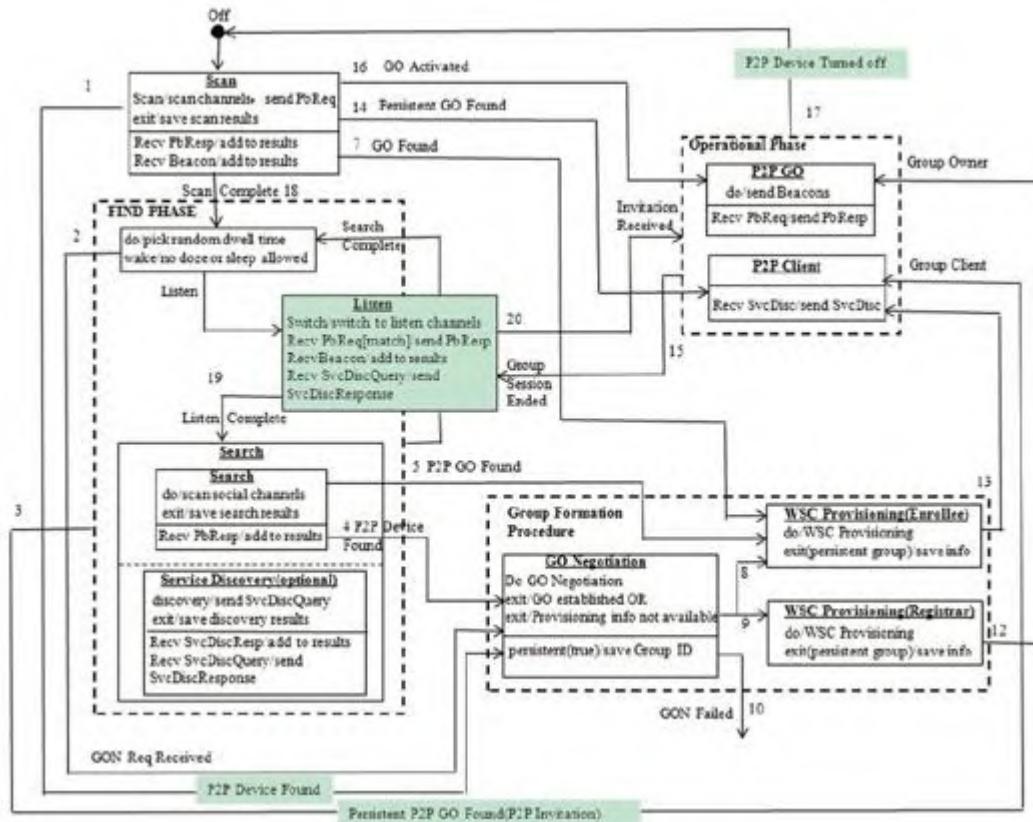


图7-22 P2P状态机

图7-22中，三个黑虚线框分别是Find Phase、Group Formation Procedure和Operational Phase，这三个Phase描述的是P2P工作流程中的一个阶段，每个阶段可包含一个或多个状态。例如Group Formation Procedure阶段包含GON、WSC Provisioning Registrar和WSC Provisioning Enrollee三个状态。

每个状态对应的状态名位于状态框顶部，其字体格式为加粗并带下划线。注意，图中Search状态包含两个子状态，分别是Search子状态以及Service Discovery子状态。由于P2P Device并不都支持Service

Discovery功能，所以Service Discovery子状态为可选(operational)状态。

每个状态都有对应的Entry Action、Exit Action和Internal Behavior。其中，EA和EXA位于状态框图的上半部分，而Internal Behavior位于状态框图的下半部分。状态之间的切换及切换条件由数字序号及箭头线表示。

下面介绍图7-22中P2P状态机的各个状态以及状态转换条件。对此，我们重点考察每个状态的EA(Entry Action)、EXA(Exit Action)、Internal Behavior以及Transition。

一个P2P Device最初的状态是Off，然后将进入Scan状态(括号中的数字对应图7-22中的数字)。

EA 和 EXA	Scan：扫描相关频道(可以是所有频道，也可以是social channels)，扫描过程中需要发送携带P2P IE信息的Probe Request帧 Exit：扫描完毕后，收集扫描结果信息。后续的Find Phase或Group Formation Procedure将会用到它
Internal Behavior	接收Probe Response帧或者Beacon帧(由GO发送)并存储相关信息
Transitions	(1) P2P Device Found：找到一个P2P Device后，P2P Device将进入GO Negotiation状态 (7) P2P GO Found：找到一个GO后，P2P Device将进入WSC Provisioning(Enrollee)状态 (14) Persistent GO Found：找到一个之前曾经连接过的Persistent GO，P2P Device可以发送P2P Invitation Request给这个GO，如果被接受，则此P2P Device将进入P2P Client状态 (16) GO Activated：P2P Device直接成为GO，故它将转入P2P GO状态 (18) Scan Complete：扫描完毕，P2P Device进入Find Phase

接着来看Find Phase，它包括Listen和Search两个状态。其中，Listen状态如下。

EA 和 EXA	Pick Random Dwell Time: 挑选一个随机等待时间 Listen on Social Channel : 在设备选择的一个 Social Channel 上监听一段时间 (时长为上面随机选择的 Dwell Time)
Internal Behavior	接收 Probe Request 帧: 如果条件满足, 则相应回复 Probe Response 帧 接收 Beacon 帧: 保存相关信息 接收 Service Discovery Query : 如果设备支持 Service Discovery, 则发送 Service Discovery Response
Transitions	( 2 ) GON Request Received: 收到 GON 请求帧后, 将转入 Group Formation Phase ( 19 ) Listen State Completed: P2P Device 转入 Search 状态 ( 20 ) Invitation Received : 该状态转换对应如下所述的一种应用场景。如果 P2P Device A 之前是 Persistent Group 1 的成员, 它在 Listen State 阶段收到了来自其他设备发出的 P2P Invitation Request 以邀请其加入 Persistent Group 1。如果 Device A 之前曾保存了 Group 1 的安全配置信息, 则可以直接转入 Operational Phase。否则将转入 WSC Provisioning Enrollee 状态 (注意, P2P Device A 在 Persistent Group 1 中可能是 GO, 也可能是 Client, 所以图中 20 号状态切换箭头只指向了 Operational Phase 框, 而未指向其中所包含的 P2P GO 或 P2P Client 状态框)

Find Phase 中另外一个状态是 Search 状态。它包含 Search 子状态和 Service Discovery 子状态。先来看 Search 子状态。

EA 和 EXA	Scan Social Channels: 在 Social Channels 上发送 Probe Request 帧 Exit: 存储搜索到的信息
Internal Behavior	接收 Probe Response 帧, 并存储相关信息

再来看 Service Discovery 子状态。

EA 和 EXA	Discovery: 向支持 Service Discovery 的设备发送 Service Discovery Request Exit: 保存其他设备通过 Service Discovery Response 帧回复的 Service 信息
Internal Behavior	接收 Service Discovery Response 帧, 并存储相关信息

Search 状态 (包括 Search 子状态和 Service Discovery 子状态) 的 Transition 情况如下。

Transitions	( 3 ) Persistent P2P GO Found : 搜索到一个 Persistent P2P GO, 如果 P2P Device 保存有该 Group 的安全配置信息, 则可通过 P2P Invitation 加入此 Group。如此, 该设备将转入 Operational Phase 的 P2P Client 状态 ( 6 ) Search State Completed: 搜索完毕, 但没有获取相关信息, 则 P2P Device 转入 Listen State ( 4 ) P2P Device Found: 搜索到另外一个 P2P Device, 则进入 Group Formation Procedure 阶段 ( 5 ) P2P GO Found : 搜索到一个 GO, 则可转入 WSC Provisioning Enrollee 状态, 或者进入 Group Formation procedure 以另外创建一个新的 Group
-------------	--

接着来看 Group Formation Procedure, 该阶段包含三个状态, 首先是 GON。

EA 和 EXA	GON: 开展 GON 流程 Exit: 如果 GON 成功, 则 GO 和 Client 角色将确立
Internal Behavior	Persistent: 如果需要建立 Persistent Group, 则该 Group 的相关信息(如 P2P Group ID 等)需要保存
Transitions	( 9 ) P2P Device 成为 GO: 转入 WSC Provisioning Registrar 状态 ( 8 ) P2P Device 成为 Client: 转入 WSC Provisioning Enrollee 状态 ( 10 ) GON fails: GON 失败, 可转入 Scan 或 Search Phase, 或者提醒用户以决定下一步工作

Group Formation Procedure另外两个状态WSC Provisioning Registrar和WSC Provisioning Enrollee比较简单, 请读者根据图7-22自行总结。

最后, 来看看Operational Phase, 它包含P2P GO和P2P Client两个状态, 首先是P2P GO状态。

EA 和 EXA	Send Beacons: GO 需要发送 Beacon 帧
Internal Behavior	Receive Probe Request: 接收其他设备的 Probe Request 帧并回复 Probe Response 帧
Transitions	( 17 ) P2P Device Turned Off: 进入 Off 状态。 ( 15 ) P2P Group Session Ended: Group 结束, 转入 Listen 状态。

再来看P2P Client状态, 它没有EA和EXA。

Internal Behavior	Receive Service Discovery Request: 接收其他设备的 Service Discovery Request, 如果此设备支持 Service Discovery, 则需回复 Service Discovery Response 帧
Transitions	( 17 ) P2P Device Turned Off: 进入 Off 状态 ( 15 ) P2P Group Session Ended: Group 结束, 转入 Listen 状态

图7-22对掌握P2P整体工作流程有重要意义, 读者不妨仔细阅读。从下一节开始, 将分析Android平台中P2P的代码实现。和WSC一样, 首先分析的是Java层中的WifiP2pSettings以及WifiP2pService。

## 7.3 WifiP2pSettings和WifiP2pService介绍

WifiP2pSettings是Settings应用中负责处理P2P相关UI/UE逻辑的主要类，与之交互的则是位于SystemServer进程中的WifiP2pService。本节先介绍WifiP2pSettings的工作流程，然后分析WifiP2pService。

### 7.3.1 WifiP2pSettings工作流程

Android平台中，P2P操作特别简单，用户只需执行如下三个步骤。

- 1) 进入WifiP2pSettings界面。
- 2) 搜索周围的P2P设备。搜索到的设备将显示在WifiP2pSettings中。
- 3) 用户选择其中的某个设备以发起连接。

下面将根据上面的使用步骤来分析WifiP2pSettings。首先来看WifiP2pSettings的onActivityCreate函数。

#### 1. WifiP2pSettings创建

WifiP2pSettings的onActivityCreated函数代码如下所示。

[-->WifiP2pSettings.java: : onActivityCreated]

```
public void onActivityCreated(Bundle savedInstanceState) {  
    addPreferencesFromResource(R.xml.wifi_p2p_settings); // 加载  
界面元素  
/*
```

和第5章介绍的wifiSettings类似， wifiP2pSettings也是通过监听广播的方式 来了解系统中

Wi-Fi P2P相关的信息及变化情况。下面这几个广播属于P2P特有的，其作用如下。

`WIFI_P2P_STATE_CHANGED_ACTION`: 用于通知系统中P2P功能的启用情况，如该功能是enable还是disable。

`WIFI_P2P_PEERS_CHANGED_ACTION`: 系统内部将保存搜索到的其他P2P设备信息，如果这些信息有变化，

则系统将发送该广播。接收者需要通过WifiP2pManager的requestPeers函数重新获取这些P2P设备的信息。

`WIFI_P2P_CONNECTION_CHANGED_ACTION`: 用于通知P2P连接情况，该广播可携带WifiP2pInfo

和NetworkInfo两个对象。相关信息可从这两个对象中获取。

`WIFI_P2P_THIS_DEVICE_CHANGED_ACTION`: 用于通知本机P2P设备信息发生了变化。

`WIFI_P2P_DISCOVERY_CHANGED_ACTION`: 用于通知P2P Device

Discovery的工作状态，如启动或停止。

WIFI\_P2P\_PERSISTENT\_GROUPS\_CHANGED\_ACTION: 用于通知之  
persistent group信息发生了变化。

\*/

```
mIntentFilter.addAction(WifiP2pManager.WIFI_P2P_STATE_CHANGED_A  
CTION);
```

```
mIntentFilter.addAction(WifiP2pManager.WIFI_P2P_PEERS_CHANGED_A  
CTION);
```

```
mIntentFilter.addAction(WifiP2pManager.WIFI_P2P_CONNECTION_CHA  
NGED_ACTION);
```

```
mIntentFilter.addAction(WifiP2pManager.WIFI_P2P_THIS_DEVICE_CHA  
NGED_ACTION);
```

```
mIntentFilter.addAction(WifiP2pManager.WIFI_P2P_DISCOVERY_CHANG  
ED_ACTION);
```

```
mIntentFilter.addAction(WifiP2pManager.WIFI_P2P_PERSISTENT_GROU  
PS_CHANGED_ACTION);
```

```
final Activity activity = getActivity();  
// 创建WifiP2pManager对象，它将和WifiP2pService交互  
mWifiP2pManager = (WifiP2pManager)  
getSystemService(Context.WIFI_P2P_SERVICE);  
if (mWifiP2pManager != null) {  
    // 初始化WifiManager并建立和WifiService的联系  
    mChannel = mWifiP2pManager.initialize(activity,  
    getActivity().getMainLooper(),null);  
}  
.....  
.....// 创建UI中按钮对应的onClickListener  
mRenameListener = new OnClickListener() {.....};  
.....  
super.onActivityResult(savedInstanceState);  
}
```

WifiP2pSettings将在onResume中注册一个广播接收对象以监听上面代码中介绍的广播事件。这部分代码很简单，请读者自行阅读。

## 2. WifiP2pSettings工作流程

### (1) WIFI\_P2P\_STATE\_CHANGED\_ACTION处理流程

打开WifiP2pSettings后，首先要等待WIFI\_P2P\_STATE\_CHANGED\_ACTION广播以判断P2P功能是否正常启动。相应的处理函数如下所示。

[-->WifiP2pSettings.java: : onReceive]

```
private final BroadcastReceiver mReceiver = new
BroadcastReceiver() {
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();
        if
(WifiP2pManager.WIFI_P2P_STATE_CHANGED_ACTION.equals(action)) {
            // 从WIFI_P2P_STATE_CHANGED_ACTION广播中获取相关状态信息
            // 以判断P2P功能是否打开
            mWifiP2pEnabled =
intent.getIntExtra(WifiP2pManager.EXTRA_WIFI_STATE,
                    WifiP2pManager.WIFI_P2P_STATE_DISABLED) ==
                    WifiP2pManager.WIFI_P2P_STATE_ENABLED;
            handleP2pStateChanged();
        }
        .....
    }
}
```

来看handleP2pStateChanged函数，代码如下所示。

[-->WifiP2pSettings.java: : handleP2pStateChanged]

```
private void handleP2pStateChanged() {
    updateSearchMenu(false); // 该函数将触发wifiP2pSettings的
    // onCreateOptionsMenu被调用
    if (mWifiP2pEnabled) {
        .....
        /*
        获得系统当前已经搜索到或者之前保存的P2P Device信息列表。Android
        为此定义了一个名为
        WifiP2pDeviceList的数据类型用来存储这些P2P Device信息。
        请读者注意requestPeers函数调用的第二个参数，该参数的类型为
        PeerListener，它是一个
        接口类，而WifiP2pSettings实现了它。WifiP2pDeviceList信息将通
        过这个接口类的
        onPeersAvailable函数返回给requestPeers的调用者。
        后文将分析onPeersAvailable函数，此处先略过。
        */
        mWifiP2pManager.requestPeers(mChannel,
```

```
WifiP2pSettings.this);
    }
}
```

根据上文的介绍，用户下一步要做的事情就是主动搜索周围的P2P设备。Android原生代码中的WifiP2pSettings界面下方有两个按钮，分别是“SEARCH”和“RENAME”。

- “RENAME”用于更改本机的P2P设备名。
- “SEARCH”用于搜索周围的P2P Device。

当P2P功能正常启用后（即上述代码中的mWifiP2pEnabled为true时），这两个按钮将被使能。此后，用户就可单击“SEARCH”按钮以搜索周围的P2P设备。该按钮对应的函数是startSearch，马上来看它。

**提示** 一些手机厂商对WifiP2pSettings界面有所更改，但大体流程没有变化。

## (2) startSearch函数

startSearch的代码如下所示。

```
[-->WifiP2pSettings.java: : startSearch]

private void startSearch() {
    if (mWifiP2pManager != null && !mWifiP2pSearching) {
        // discoverPeers将搜索周围的P2P设备
        mWifiP2pManager.discoverPeers(mChannel, new
WifiP2pManager.ActionListener() {
            ......
        });
}
```

上述代码中，WifiP2pSettings通过调用WifiManager的discoverPeers来搜索周围的设备。

当WPAS完成搜索后，WIFI\_P2P\_PEERS\_CHANGED\_ACTION广播将被发送。来看WifiP2pSettings中对该广播的处理。

## (3) WIFI\_P2P\_PEERS\_CHANGED\_ACTION处理流程

广播事件在onReceive函数中被处理，此函数的代码如下所示。

```
[-->WifiP2pSettings.java: : onReceive]

private final BroadcastReceiver mReceiver = new
BroadcastReceiver() {
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();
        .....
        // 如果搜索到新的P2P Device，则WIFI_P2P_PEERS_CHANGED_ACTION
将被发送
        } else if
(WifiP2pManager.WIFI_P2P_PEERS_CHANGED_ACTION.equals(action)) {
            mWifiP2pManager.requestPeers(mChannel,
WifiP2pSettings.this);
        }.....
        else if
(WifiP2pManager.WIFI_P2P_DISCOVERY_CHANGED_ACTION.equals(action))
) {
            int discoveryState =
intent.getIntExtra(WifiP2pManager.EXTRA_DISCOVERY_STATE,
                    WifiP2pManager.WIFI_P2P_DISCOVERY_STOPPED);
            if (discoveryState ==
WifiP2pManager.WIFI_P2P_DISCOVERY_STARTED)
                updateSearchMenu(true); // 更新“SEARCH”按钮显示的名
称
            else
                updateSearchMenu(false);
        }.....
    }
};
```

注意，startSearch还将触发系统发送  
WIFI\_P2P\_DISCOVERY\_CHANGED\_ACTION广播，WifiP2pSettings将根据  
该广播携带的信息来更新“SEARCH”按钮的界面，其规则是：如果P2P  
Discovery启动成功（即状态变为WIFI\_P2P\_DISCOVERY\_STARTED），  
则“SEARCH”按钮名显示为“Searching...”，否则该按钮名显示  
为“Search For Devices”。

当系统搜索到新的P2P Device后，WIFI\_P2P\_PEERS\_CHANGED\_ACTION广  
播将被发送，而WifiP2pSettings对于该广播的处理就是调用  
WifiP2pManager的requestPeers来获取系统保存的P2P Device信息列  
表。

前文代码中曾介绍过requestPeers，系统中所有的P2P设备信息将通过PeerListener接口类的onPeersAvailable函数返回给WifiP2pSettings。直接来看该函数，代码如下所示。

```
[-->WifiP2pSettings.java: : onPeersAvailable]

public void onPeersAvailable(WifiP2pDeviceList peers) {
    // 系统中所有的P2P设备信息都保存在这个类型为WifiP2pDeviceList的peers
    对象中
    mPeersGroup.removeAll(); // mPeersGroup类型为PerferenceGroup,
    属于UI相关的类
    mPeers = peers;
    mConnectedDevices = 0;
    for (WifiP2pDevice peer: peers.getDeviceList()) {
        // WifiP2pPeer是Perference的子类，它和UI相关
        mPeersGroup.addPreference(new WifiP2pPeer(getActivity(), peer));
        if (peer.status == WifiP2pDevice.CONNECTED)
            mConnectedDevices++;
    }
}
```

在onPeersAvailable函数中，WifiP2pDeviceList中保存的每一个WifiP2pDevice信息将作为一个Preference项添加到mPeersGroup中并显示在UI界面上。

接下来，用户就可在界面中选择某个P2P Device并与之连接。这个步骤由onPreferenceTreeClick函数来完成。

**提示** 上述代码中提到的WifiP2pDevice等数据结构将放到下文再介绍。

#### (4) onPreferenceTreeClick函数

onPreferenceTreeClick的代码如下所示。

```
[-->WifiP2pSettings.java: : onPreferenceTreeClick]

public boolean onPreferenceTreeClick(PreferenceScreen screen,
Preference preference) {
    if (preference instanceof WifiP2pPeer) {
        mSelectedWifiPeer = (WifiP2pPeer) preference; // 获取用户
```

```
指定的那个wifiP2pPeer项
    if (mSelectedWifiPeer.device.status ==
WifiP2pDevice.CONNECTED)
        showDialog(DIALOG_DISCONNECT); // 如果已经和该Device连接, 则判断是否需要与之断开连接
    else if (mSelectedWifiPeer.device.status ==
WifiP2pDevice.INVITED)
        showDialog(DIALOG_CANCEL_CONNECT);
    else {// 向对端P2P设备发起连接
        WifiP2pConfig config = new WifiP2pConfig();
        config.deviceAddress =
mSelectedWifiPeer.device.deviceAddress;
        // 判断系统是否强制使用了某种WSC配置方法
        int forceWps =
SystemProperties.getInt("wifidirect.wps", -1);
        if (forceWps != -1) config.wps.setup = forceWps;
        else {
            // 获取对端P2P Device支持的WSC配置方法, 优先考虑
PBC
            if
(mSelectedWifiPeer.device.wpsPbcSupported())
                config.wps.setup = WpsInfo.PBC;
            else if
(mSelectedWifiPeer.device.wpsKeypadSupported()) {
                config.wps.setup = WpsInfo.KEYPAD;
                else config.wps.setup = WpsInfo.DISPLAY;
            }
            // 通过wifiP2pManager的connect函数向对端P2P Device
发起连接。注意, 目标设备
                // 信息保存在config对象中
                mWifiP2pManager.connect(mChannel, config,
                    new WifiP2pManager.ActionListener()
{.....});
                }
            } .....
        }
    }
```

## (5) WIFI\_P2P\_CONNECTION\_CHANGED\_ACTION处理流程

当系统完成和指定P2P Device的连接后，WifiP2pSettings将收到WIFI\_P2P\_CONNECTION\_CHANGED\_ACTION广播，其对应的代码逻辑如下所示。

[-->WifiP2pSettings.java: : onReceive]

```

public void onReceive(Context context, Intent intent) {
    String action = intent.getAction();
    .....
} else if
(WifiP2pManager.WIFI_P2P_CONNECTION_CHANGED_ACTION.equals(action)) {
    NetworkInfo networkInfo = (NetworkInfo)
intent.getParcelableExtra(
WifiP2pManager.EXTRA_NETWORK_INFO);
    if (mWifiP2pManager != null){
        // requestGroupInfo第二个参数的类型是
GroupInfoListener, 用于返回Group信息
        mWifiP2pManager.requestGroupInfo(mChannel,
WifiP2pSettings.this);
    }
    if (networkInfo.isConnected()){
        if (DBG) Log.d(TAG, "Connected");
    } else startSearch();      // 如果没有加入某个P2P
Group, 则重新发起设备扫描
    }.....
}

```

当设备加入一个Group后，requestGroupInfo将通过其第二个参数设置的回调函数以返回此Group的信息，这个回调函数由GroupInfoListener接口类定义，WifiP2pSettings只需实现该接口类的onGroupInfoAvailable，相关代码如下所示。

[-->WifiP2pSettings.java: : onGroupInfoAvailable]

```

public void onGroupInfoAvailable(WifiP2pGroup group) {
    mConnectedGroup = group;           // 保存当前所加入的Group的
信息
    updateDevicePref();               // 更新WifiP2pPeer的界面
}

```

以上是WifiP2pSettings的大体工作流程，读者只要把握几个重要广播的处理流程即可掌握Settings应用中和Wi-Fi P2P相关的知识。

### 3. WifiP2pSettings总结

上述代码中有两个比较重要的数据结构，WifiP2pDevice和WifiP2pGroup，成员比较简单，此处不赘述。它们的类图如图7-23所

示。

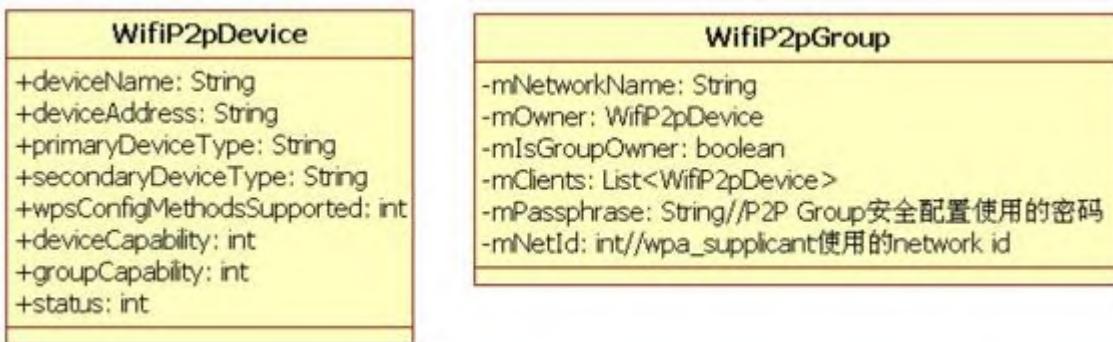


图7-23 WifiP2pDevice和WifiP2pGroup类图

通过上面几节的代码可知，WifiP2pSettings将借助WifiP2pManager和系统中的WifiP2pService交互。WifiP2pSettings主要使用WifiP2pManager的几个重要函数。

- **initialize:** 初始化WifiP2pManager，其内部将和WifiP2pService通过AsyncChannel建立交互关系。
- **discoverPeers:** 触发WifiP2pService发起P2P Device扫描工作。
- **requestPeers:** 获取系统中保存的P2P Device信息。
- **connect:** 和指定P2P Device发起连接，也就是相互协商以创建或加入一个P2P网络，即一个Group。
- **requestGroupInfo:** 获取当前连接上的Group信息。

# 读累了记得休息一会儿哦~

公众号：古德猫宁李

- 电子书搜索下载
- 书单分享
- 书友学习交流

网站：[沉金书屋 https://www.chenjin5.com](https://www.chenjin5.com)

- 电子书搜索下载
- 电子书打包资源分享
- 学习资源分享

### 7.3.2 WifiP2pService工作流程

WifiP2pService和第5章介绍的WifiService一样，都属于Android系统中负责处理Wi-Fi相关工作的核心模块。其中，WifiService处理和WLAN网络连接相关的工作，而WifiP2pService则专门负责处理和Wi-Fi P2P相关的工作。图7-24所示为WifiP2pService家族类图。

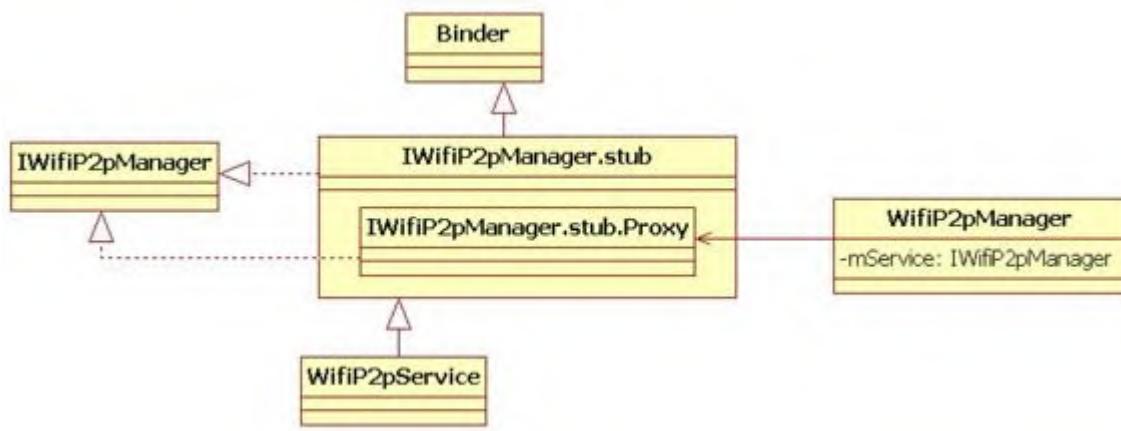


图7-24 WifiP2pService类图

图7-24所示的WifiP2pService家族类图和图5-1所示的WifiService家族类图类似，此处就不详细讨论了。直接来看WifiP2pService的代码，首先是它的构造函数，如下所示。

[-->WifiP2pService.java: : WifiP2pService]

```
public WifiP2pService(Context context) {
    mContext = context;
    mInterface = "p2p0"; // P2P使用的虚拟网络接口设备名为“p2p0”
    mActivityMgr =
        (ActivityManager) context.getSystemService(Activity.ACTIVITY_SERVICE);
    mNetworkInfo = new
        NetworkInfo(ConnectivityManager.TYPE_WIFI_P2P, 0, NETWORKTYPE,
        "");
    // 判断系统是否支持WiFi-Direct功能
    mP2pSupported =
        mContext.getPackageManager().hasSystemFeature(
```

```

PackageManager.FEATURE_WIFI_DIRECT);
// 获取PrimaryDeviceType，默认值是“10-0050F204-5”。结合图6-15可知
// “10”是Category ID，代表Telephone
// “0050F204”是WFA的OUI，最后一个“5”是Sub Category ID，在
Telephone大类里边，它代表
// 支持Dual Mode的Smartphone（规范中定义为Smart phone-Dual mode）
mThisDevice.primaryDeviceType =
mContext.getResources().getString(
com.android.internal.R.string.config_wifi_p2p_device_type);
// WiFiP2pService主要工作也是由状态机来完成的，即下面的这个
P2pStateMachine
mP2pStateMachine = new P2pStateMachine(TAG, mP2pSupported);
mP2pStateMachine.start(); // 启动P2P状态机
}

```

P2pStateMachine是WifiP2pService定义的内部类，它比第5章介绍的WifiStateMachine简单，其构造函数如下所示。

[-->WifiP2pService.java: : P2pStateMachine构造函数]

```

P2pStateMachine(String name, boolean p2pSupported) {
    super(name);
    addState(mDefaultState); // 为状态机添加状态，一共15个状态
    addState(mP2pNotSupportedState, mDefaultState);
    .....
    if (p2pSupported) setInitialState(mP2pDisabledState); // 初始
    状态为P2pDisabledState
    else setInitialState(mP2pNotSupportedState);
}

```

图7-25描述了P2pStateMachine中定义的各个状态及层级关系。

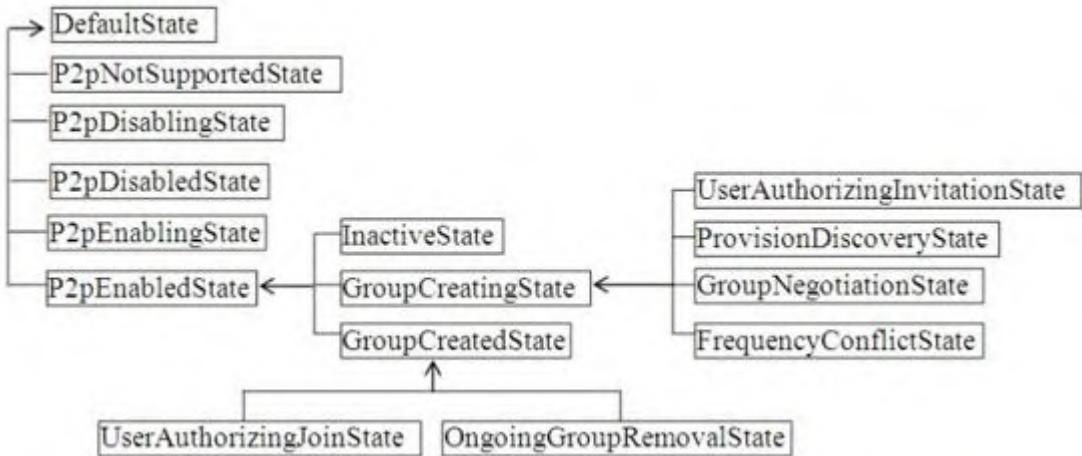


图7-25 P2pStateMachine状态机

P2pStateMachine的初始状态是P2pDisabledState，它和父状态DefaultState的Entry Action都没有执行什么有意义的事情，故此处略去对二者EA的介绍。

P2pStateMachine是WifiP2pService的核心，我们马上来介绍它的工作流程。

### 1. CMD\_ENABLE\_P2P处理流程

P2pStateMachine虽然属于WifiP2pService，但它也受WifiStateMachine的影响。通过5.2.3节“WifiStateMachine构造函数分析之二”中对WifiStateMachine InitialState EA的介绍，会发现WifiStateMachine将创建一个名为mWifiP2pChannel的AsyncChannel对象用于向P2pStateMachine发送消息。

在Android平台中，如果用户打开Wi-Fi功能，P2pStateMachine就会收到第一个消息CMD\_ENABLE\_P2P。该消息是WifiStateMachine进入DriverStartedState后，在其EA中借助mWifiP2pChannel向P2pStateMachine发送的（可参考5.3.2节SUP\_CONNECTION\_EVENT处理流程分析）[①](#)。

P2pStateMachine此时处于P2pDisabledState，它对CMD\_ENABLE\_P2P消息的处理逻辑如下所示。

[-->WifiP2pService.java: : P2pDisabledState: enter]

```

class P2pDisabledState extends State {
    public boolean processMessage(Message message) {
        switch (message.what) {
            case WifiStateMachine.CMD_ENABLE_P2P:
                try {
                    mNwService.setInterfaceUp(mInterface);
                } .....
                // 启动WifiMonitor, 它将通过wpa_ctl连接上
                wpa_supplicant。关于wpa_supplicant的启动
                // 读者可参考5.2.3节“WifiNative介绍”
                mWifiMonitor.startMonitoring();
                // 转入P2pEnablingState, 其EA未作有意义的事情, 读者可
                自行阅读它
                transitionTo(mP2pEnablingState);
                break;
            default:
                return NOT_HANDLED;
        }
        return HANDLED;
    }
}

```

处理完CMD\_ENABLE\_P2P消息后，P2pStateMachine将创建一个WifiMonitor用于接收来自wpa\_supplicant的消息，同时状态机将转入P2pEnablingState。

WifiMonitor连接wpa\_supplicant之后，WifiMonitor会发送一个SUP\_CONNECTION\_EVENT给P2pStateMachine。该消息将由P2pEnablingState处理，马上来看相关的处理流程。

## 2. SUP\_CONNECTION\_EVENT处理流程

代码如下。

[-->WifiP2pService.java: : P2pEnablingState: processMessage]

```

class P2pEnablingState extends State {
    .....
    public boolean processMessage(Message message) {
        switch (message.what) {
            case WifiMonitor.SUP_CONNECTION_EVENT:
                transitionTo(mInactiveState); // 转入InactiveState
                break;
        }
    }
}

```

```
    }  
    return NOT_HANDLED  
}  
}
```

根据5.2.1节HSM的知识，当状态机转入InactiveState后，首先执行的是其父状态P2pEnabledState的EA，然后才是InactiveState自己的EA。由于InactiveState的EA仅打印了一句日志输出，故此处仅介绍P2pEnabledState的EA，相关代码如下所示。

[-->WifiP2pService.java: : P2pEnabledState: enter]

```
class P2pEnabledState extends State {
    public void enter() {
        // 发送WIFI_P2P_STATE_CHANGED_ACTION广播，并设置
EXTRA_WIFI_STATE状态为
        // WIFI_P2P_STATE_ENABLED
        sendP2pStateChangedBroadcast(true);
        mNetworkInfo.setIsAvailable(true);
        /*
发送WIFI_P2P_CONNECTION_CHANGED_ACTION广播，它将携带
WifiP2pInfo和NetworkInfo信息。
注意，下面这个函数还会向WifiStateMachine发送
P2P_CONNECTION_CHANGED消息。读者不妨
自行研究WifiStateMachine对P2P_CONNECTION_CHANGED消息的处理流
程。
        */
        sendP2pConnectionChangedBroadcast();
        initializeP2pSettings(); // 初始化P2P的一些设置，详情见下文
    }
}
```

我们重点关注上面代码中的initializeP2pSettings函数，其代码如下所示。

[-->WifiP2pSettings.java: : initializeP2pSettings]

```
private void initializeP2pSettings() {  
    /*  
     * 发送“SET persistent_reconnect 1”给WPAS，该命令对应如下一种应用场景。  
     */  
}
```

当发现一个Persistent Group时，如果 persistent\_reconnect为1，则可利用之前保存的配置信息自动重连。

重新连接时无需用户参与。如果persistent\_reconnect为0，则需要提醒用

户，让用户来决定是否加入此

```
    persistent group.  
    /*  
     mWifiNative.setPersistentReconnect(true);  
     /*
```

获取P2P Device Name，先从数据库中查询“wifi\_p2p\_device\_name”字段的值，如果数据库中没有设置

该字段，则取数据库中“android\_id”字段值的前4个字符并在其前面加上“Android\_”字符串以

组成P2P Device Name。以Galaxy Note 2为例，数据库文件是/data/data/com.android.providers.

```
    settings/database/settings.db，所查询的表名为secure，  
    wifi_p2p_device_name字段取值为  
    "Android_4aa9"， android_id字段取值为"4aa9213016889423"。  
    /*  
     mThisDevice.deviceName = getPersistedDeviceName(); //  
     mThisDevice指向一个wifiP2pDevice对象  
     // 将P2P DeviceName保存到WPAS中  
     mWifiNative.setDeviceName(mThisDevice.deviceName);  
     // 设置P2P网络SSID的后缀。如果本设备能扮演GO，则它构建的Group对应的  
     SSID后缀就是此处设置的后缀名  
     mWifiNative.setP2pSsidPostfix("-" + mThisDevice.deviceName);  
     // 设置Primary DeviceType  
     mWifiNative.setDeviceType(mThisDevice.primaryDeviceType);  
     // 设置支持的WSC配置方法  
     mWifiNative.setConfigMethods("virtual_push_button  
     physical_display keypad");  
     // 设置STA连接的优先级高于P2P连接  
     mWifiNative.setConcurrencyPriority("sta");  
     // 从WPAS中获取P2P Device Address  
     mThisDevice.deviceAddress =  
     mWifiNative.p2pGetDeviceAddress();  
     // 更新自己的状态，并发送WIFI_P2P_THIS_DEVICE_CHANGED_ACTION消息  
     updateThisDevice(WifiP2pDevice.AVAILABLE);  
     mClientInfoList.clear();  
     // 清空WPAS中保存peer P2P Device和Service信息  
     mWifiNative.p2pFlush(); mWifiNative.p2pServiceFlush();  
     mServiceTransactionId = 0; mServiceDiscReqId = null;  
     /*
```

WPAS中会保存persistent Group信息，而P2pStateMachine也会保存一些信息，下面这个函数将根据

WPAS中的信息来更新P2pStateMachine中保存的Group信息。

P2pStateMachine通过一个名为mGroups

的成员变量（类型为WifiP2pGroupList）来保存所有的Group信息。

```
/*
```

```
        updatePersistentNetworks(RELOAD);  
    }  
}
```

至此，P2pStateMachine就算初始化完毕，接下来的工作就是处理用户发起的操作。

首先来看WifiP2pSettings中WifiP2pManager的discoverPeers函数，它将发送DISCOVER\_PEERS消息给P2pStateMachine。

### 3. DISCOVER\_PEERS处理流程

P2pStateMachine当前处于InactiveState，不过DISCOVER\_PEERS消息却是由其父状态P2pEnabledState来处理的，相关代码如下所示。

```
[-->WifiP2pService.java: : P2pEnabledState: processMessage]  
  
class P2pEnabledState extends State{  
    public boolean processMessage(Message message) {  
        switch (message.what) {  
            case WifiP2pManager.DISCOVER_PEERS:  
                clearSupplicantServiceRequest(); // 先取消Service  
                Discovery请求  
                // 发送“P2P_FIND 超时时间”给WPAS, DISCOVER_TIMEOUT_S  
                // 值为120秒  
                if (mWifiNative.p2pFind(DISCOVER_TIMEOUT_S)) {  
                    replyToMessage(message,  
                    WifiP2pManager.DISCOVER_PEERS_SUCCEEDED);  
                    // 发送WIFI_P2P_DISCOVERY_CHANGED_ACTION广播以通  
                    // 知P2P Device Discovery已启动  
                    sendP2pDiscoveryChangedBroadcast(true);  
                }.....  
                break;  
        }  
    }  
}
```

当WPAS搜索到周围的P2P Device后，将发送以下格式的消息给WifiMonitor。

```
P2P-DEVICE-FOUND fa: 7b: 7a: 42: 02: 13 p2p_dev_addr=fa: 7b: 7a:  
42: 02: 13 pri_dev_type=1-0050F204-1 name='p2p-  
TEST1'config_methods=0x188 dev_capab=0x27 group_capab=0x0
```

WifiMonitor将根据这些信息构建一个WifiP2pDevice对象，然后发送P2P\_DEVICE\_FOUND\_EVENT给P2pStateMachine。

#### 4. P2P\_DEVICE\_FOUND\_EVENT处理流程

同样，P2P\_DEVICE\_FOUND\_EVENT也由InactiveState的父状态P2pEnabledState来处理，相关代码如下所示。

```
[-->WifiP2pService.java: : P2pEnabledState: processMessage]

class P2pEnabledState extends State{
    public boolean processMessage(Message message) {
        switch (message.what) {
            .....
            case WifiMonitor.P2P_DEVICE_FOUND_EVENT:
                // WifiMonitor根据WPAS反馈的信息构建一个
                WifiP2pDevice对象
                WifiP2pDevice device = (WifiP2pDevice)
                message.obj;
                // 如果搜索到的这个P2P Device是自己（根据Device
                Address来判断），则不处理它
                if
                (mThisDevice.deviceAddress.equals(device.deviceAddress)) break;
                /*
                mPeers指向一个wifiP2pDeviceList对象。如果之前已存储
                了此Device的信息，
                更新这些信息，否则将添加一个新的WifiP2pDevice对象。
                */
                mPeers.update(device);
                sendP2pPeersChangedBroadcast(); // 发送
                WIFI_P2P_PEERS_CHANGED_ACTION广播
                break;
            } .....
        }
    }
}
```

WifiP2pSettings收到WIFI\_P2P\_PEERS\_CHANGED\_ACTION广播后，将通过WifiP2pManager的requestPeers来获得当前搜索到的P2P Device信息（即mPeers的内容）。这部分处理逻辑非常简单，请读者自行阅读相关代码。

现在，用户将选择一个P2P Device然后通过WifiP2pManager的connect函数向其发起连接。来看相关代码。

## 5. CONNECT处理流程

WifiP2pManager的connect函数将发送CONNECT消息给P2pStateMachine，该消息由InactiveState状态自己来处理，代码如下所示。

```
[-->WifiP2pSettings.java: : InactiveState: processMessage]

class InactiveState extends State {
    .....
    public boolean processMessage(Message message) {
        switch (message.what) {
            case WifiP2pManager.CONNECT:
                /*
                    WifiP2pSettings将设置一个WifiP2pConfig对象以告诉
                    P2pStateMachine该连接
                    哪一个P2P Device (参考7.3.1节
                    onPreferenceTreeClick介绍)
                */
                WifiP2pConfig config = (WifiP2pConfig)
                message.obj;
                mAutonomousGroup = false;
                // 获取该P2P Device的Group Capability信息
                int gc =
                    mWifiNative.getGroupCapability(config.deviceAddress);

                mPeers.updateGroupCapability(config.deviceAddress, gc);
                // 关键函数connect, 见下文介绍
                int connectRet = connect(config,
                TRY_REINVOCATION);
                // TRY_REINVOCATION值为true
                .....

                mPeers.updateStatus(mSavedPeerConfig.deviceAddress,
                WifiP2pDevice.INVITED);
                sendP2pPeersChangedBroadcast();
                replyToMessage(message,
                WifiP2pManager.CONNECT_SUCCEEDED);
                // 根据connectRet的值进行状态切换选择
                if (connectRet == NEEDS_PROVISION_REQ) {
                    transitionTo(mProvisionDiscoveryState); // 转入
                    ProvisionDiscoveryState
                    break;
                }
                transitionTo(mGroupNegotiationState); // 或者转入
```

```

GroupNegotiationState
    break;
    .....
}
return HANDLED
}

```

上述代码中有一个关键函数，即connect，其代码如下所示。

```

[-->WifiP2pService.java: : connect]

private int connect(WifiP2pConfig config, boolean
tryInvocation) {
    .....
    // 当前还没有保存的对端P2P Device配置信息（对应的数据类型为
    wifiP2pConfig
    // 所以isResp为false
    boolean isResp = (mSavedPeerConfig != null &&
config.deviceAddress.equals(mSavedPeerConfig.deviceAddress));
    mSavedPeerConfig = config;// 保存传入的WifiP2pConfig信息

    WifiP2pDevice dev = mPeers.get(config.deviceAddress);
    .....
    // 判断对端设备是否为GO。由于还没有开展GON，所以join为false
    boolean join = dev.isGroupOwner();
    String ssid = mWifiNative.p2pGetSsid(dev.deviceAddress);
    // 如果join为true，但对端设备不能再添加新的P2P Device，则join被
    设置为false
    if (join && dev.isGroupLimit()) join = false;
    else if (join) {// mGroups保存搜索到的GO信息，当前还没有GO，所
    以netId为-1
        int netId = mGroups.getNetworkId(dev.deviceAddress,
        ssid);
        if (netId >= 0) {// 这种情况对应于加入一个当前已经存在的
        P2P Group
            if (!mWifiNative.p2pGroupAdd(netId)) return
CONNECT_FAILURE;
            return CONNECT_SUCCESS;
        }
    }

    if (!join && dev.isDeviceLimit()) return CONNECT_FAILURE;
    // tryInvocation为true。P2P Device一般都支持Invitation
    // 下面这个if代码段处理Persistent Group的情况
}

```

```

        if (!join && tryInvocation && dev.isInvitationCapable()) {
            int netId = WifiP2pGroup.PERSISTENT_NET_ID; // PERSISTENT_NET_ID值为-2
            if (config.netId >= 0) {
                if
                (config.deviceAddress.equals(mGroups.getOwnerAddr(config.netId)))
                    netId = config.netId;
            } else netId =
            mGroups.getNetworkId(dev.deviceAddress);

            if (netId < 0) netId =
            getNetworkIdFromClientList(dev.deviceAddress);
            if (netId >= 0) { // 通过Invitation Request重新启动一个 Persistent Group
                if (mWifiNative.p2pReinvoke(netId,
                dev.deviceAddress)) {
                    mSavedPeerConfig.netId = netId;
                    return CONNECT_SUCCESS;
                } else updatePersistentNetworks(RELOAD);
            }
        }
        mWifiNative.p2pStopFind();

        if (!isResp) return NEEDS_PROVISION_REQ; // 就本例而言, connect返回NEEDS_PROVISION_REQ

        p2pConnectWithPinDisplay(config); // 发起P2P连接, 即启动 Group Formation流程
        return CONNECT_SUCCESS;
    }
}

```

就本例而言, connect将返回NEEDS\_PROVISION\_REQ, 所以 P2pStateMachine将转入ProvisionDiscoveryState, 马上来看它的 EA。

[-->WifiP2pService.java: : ProvisionDiscoveryState: enter]

```

class ProvisionDiscoveryState extends State {
    public void enter() {
        // 触发本机设备向对端设备发送Provision Discovery Request帧
        mWifiNative.p2pProvisionDiscovery(mSavedPeerConfig);
    }
}

```

注意，由于WSC配置方法为PBC，所以对端设备的P2pStateMachine将收到一个P2P\_PROV\_DISC\_PBC\_REQ\_EVENT消息。当对端设备处理完毕后，将收到一个P2P\_PROV\_DISC\_PBC\_RSP\_EVENT消息。马上来看P2P\_PROV\_DISC\_PBC\_RSP\_EVENT消息的处理流程。

## 6. P2P\_PROV\_DISC\_PBC\_RSP\_EVENT处理流程

P2pStateMachine当前处于ProvisionDiscoveryState，相关处理逻辑如下所示。

```
[-->WifiP2pService.java: : ProvisionDiscoveryState:  
processMessage]  
  
public boolean processMessage(Message message) {  
    WifiP2pProvDiscEvent provDisc;  
    WifiP2pDevice device;  
    switch (message.what) {  
        case WifiMonitor.P2P_PROV_DISC_PBC_RSP_EVENT:  
            provDisc = (WifiP2pProvDiscEvent) message.obj;  
            device = provDisc.device;  
            if  
(!device.deviceAddress.equals(mSavedPeerConfig.deviceAddress))  
break;  
  
        if (mSavedPeerConfig.wps.setup == WpsInfo.PBC) {  
            /*  
             * 下面这个函数将调用wifiNative的p2pConnect函数，此  
             * 函数将触发WPAS发送  
             * GON Request帧。接收端设备收到该帧后，将弹出图7-16  
             * 所示的提示框以提醒用户。  
            */  
            p2pConnectWithPinDisplay(mSavedPeerConfig);  
            // 转入GroupNegotiationState，其EA比较简单，请读  
            者自行阅读  
            transitionTo(mGroupNegotiationState);  
        }  
        break;  
    .....
```

上述代码中，P2pStateMachine通过p2pConnectWithPinDisplay向对端发起Group Negotiation Request请求。接下来的工作就由WPAS来处

理。当Group Formation结束后，P2pStateMachine将收到一个P2P\_GROUP\_STARTED\_EVENT消息以通知Group建立完毕，该消息的处理流程如下节所述。

## 7. P2P\_GROUP\_STARTED\_EVENT处理流程

P2P\_GROUP\_STARTED\_EVENT消息由GroupNegotiationState处理，相关代码如下所示。

```
[-->WifiP2pService.java: : GroupNegotiationState:  
processMessage]  
  
class GroupNegotiationState extends State {  
    ....  
    public boolean processMessage(Message message) {  
        switch (message.what) {  
            case WifiMonitor.P2P_NEGOTIATION_SUCCESS_EVENT:  
            case WifiMonitor.P2P_GROUP_FORMATION_SUCCESS_EVENT:  
                break; // 不处理Group Negotiation成功的消息  
            case WifiMonitor.P2P_GROUP_STARTED_EVENT:// 只处理  
Group Started消息  
                mGroup = (WifiP2pGroup) message.obj;  
                if (mGroup.getNetworkId() ==  
WifiP2pGroup.PERSISTENT_NET_ID) {  
                    updatePersistentNetworks(NO_RELOAD);  
                    String devAddr =  
mGroup.getOwner().deviceAddress;  
  
                    mGroup.setNetworkId(mGroups.getNetworkId(devAddr,  
                                            mGroup.getNetworkName()));  
                }  
                if (mGroup.isGroupOwner()) { // 如果本机P2P设备是GO,  
则启动DhcpServer  
                    // 假设本机P2P设备扮演GO, 请读者自行阅读  
startDhcpServer函数  
                    startDhcpServer(mGroup.getInterface());  
                } else {  
                    /*  
                     * 如果对端设备是GO, 则启动DhcpStateMachine用于获取一个  
IP地址, 这部分流程和  
5.3.2节NETWORK_CONNECTION_EVENT消息处理流程分析的  
ObtainingIpState工作流程类似。  
*/  
        }  
    }  
}
```

```

mWifiNative.setP2pGroupIdle(mGroup.getInterface(),
GROUP_IDLE_TIME_S);
        mDhcpStateMachine =
DhcpStateMachine.makeDhcpStateMachine(mContext,
                                         P2pStateMachine.this,
mGroup.getInterface());
mDhcpStateMachine.sendMessage(DhcpStateMachine.CMD_START_DHCP);
        WifiP2pDevice groupOwner = mGroup.getOwner();
groupOwner.update(mPeers.get(groupOwner.deviceAddress));
mPeers.updateStatus(groupOwner.deviceAddress,WifiP2pDevice.CONN
ECTED);
        sendP2pPeersChangedBroadcast();
    }
    mSavedPeerConfig = null;
    transitionTo(mGroupCreatedState); // 转入
GroupCreatedState
    break;
.....
}
}

```

P2pStateMachine将转入GroupCreatedState，其EA代码如下所示。

[-->WifiP2pService.java: : GroupCreatedState: enter]

```

class GroupCreatedState extends State {
    public void enter() {

mNetworkInfo.setDetailedState(NetworkInfo.DetailedState.CONNECT
ED, null, null);
        updateThisDevice(WifiP2pDevice.CONNECTED); // 连接成功
        if (mGroup.isGroupOwner()) {
            /*
             SERVER_ADDRESS为“192.168.49.1”，该地址也被设置到Dhcp
Server中。
另外，P2pStateMachine有一个名为mWifiP2pInfo的成员变量，其类
型为WifiP2pInfo，
下面这个函数也将GO的IP地址保存到mWifiP2pInfo中。
*/
            setWifiP2pInfoOnGroupFormation(SERVER_ADDRESS);
            sendP2pConnectionChangedBroadcast(); // 发送
WIFI_P2P_CONNECTION_CHANGED_ACTION广播
        }
    }
}

```

```
    }  
    ....  
}
```

## 8. AP\_STA\_CONNECTED\_EVENT处理流程

当对端P2P设备成功关联到本机后，WifiMonitor又将发送一个名为AP\_STA\_CONNECTED\_EVENT的消息，该消息的处理逻辑如下所示。

[-->WifiP2pService.java: : GroupCreatedState: enter]

```
public boolean processMessage(Message message) {  
    switch (message.what) {  
        case WifiMonitor.AP_STA_CONNECTED_EVENT:// 该消息表示一个  
P2P Client关联上本机GO  
            WifiP2pDevice device = (WifiP2pDevice)  
message.obj;  
            String deviceAddress = device.deviceAddress;  
            if (deviceAddress != null) {  
                ....  
                mGroup.addClient(deviceAddress); // 添加一个  
P2P Client  
                mPeers.updateStatus(deviceAddress,  
WifiP2pDevice.CONNECTED);  
                sendP2pPeersChangedBroadcast();  
            }  
            ....  
            break;  
        ....  
    }  
}
```

至此，一个P2P Device（扮演Client）就成功关联上本机的P2P Device（扮演GO）。

## 9. WifiP2pService总结

回顾上文介绍的WifiP2pService工作流程，可知P2pStateMachine初始状态为P2pDisabledState，然后：

- 1) P2pStateMachine将接收到的第一条消息，它是来自WifiStateMachine的CMD\_ENABLE\_P2P。在该消息的处理逻辑中，P2pStateMachine将创建一个WifiMonitor对象以和wpa\_supplicant进程交互。最后，P2pStateMachine转入P2pEnablingState。

- 2) 在P2pEnablingState中，P2pStateMachine将处理SUP\_CONNECT\_EVENT消息，它代表WifiMonitor成功连接上了wpa\_supplicant。该消息处理完毕后，P2pStateMachine将转入InactiveState。
- 3) InactiveState的父状态是P2pEnabledState，P2pEnabledState的EA将初始化P2P设置，这部分代码逻辑在initializeP2pSettings函数中。另外，WifiP2pSettings将收到一些P2P广播，此时P2P功能正常启动。
- 4) 用户在界面中进行操作以搜索周围的设备，这使得P2pStateMachine将收到DISCOVER\_PEERS消息。它在P2pEnabledState中被处理，wpas\_supplicant将发起P2P Device Discovery流程以搜索周围的P2P设备。
- 5) 一旦有P2P设备被搜索到，P2pStateMachine将接收到一条P2P\_DEVICE\_FOUND\_EVENT消息。该消息依然由P2pEnabledState来处理。同时，WifiP2pSettings也会相应收到信息以更新UI。
- 6) 当用户在WifiP2pSettings界面中选择连接某个P2P Device后，WifiP2pSettings将发送CONNECT消息给P2pStateMachine。该消息由InactiveState来处理。大部分情况下（除了Persistent Group或者对端设备是GO的情况下），P2pStateMachine将转入ProvisionDiscoveryState。
- 7) ProvisionDiscoveryState中，P2pStateMachine将通知WPAS以开展Provisioning Discovery流程。一切顺利的话，P2pStateMachine将接收到P2P\_PROV\_DISC\_PBC\_RSP\_EVENT消息。在该消息的处理过程中，P2pStateMachine将通过p2pConnectWithPinDisplay函数通知WPAS和对端设备启动Group Formation流程。此后，P2pStateMachine转入GroupNegotiationState。
- 8) Group Formation完成，一个Group也就创建成功，P2pStateMachine将收到P2P\_GROUP\_STARTED\_EVENT消息。该消息由GroupNegotiationState处理。如果本机扮演GO的话，它将启动一个Dhcp服务器，也就是第2章提到的dnsmasq（详情请参考2.3.8节“背景知识介绍”）。

9) 当对端P2P Client (Group建立后, 角色也就确定了) 关联上本机的GO后, AP\_STA\_CONNECTED\_EVENT消息将被发送给P2pStateMachine处理。

如果仔细阅读WifiP2pService代码, 会发现本节介绍的工作流程是WifiP2pService中最简单的一条了。经过笔者实际测试, WifiP2pService有一个工作场景的处理流程比较复杂, 即如果用户在对端设备发起connect操作, 则本机的处理相对要复杂一些。这部分流程和wpa\_supplicant的处理也有关系, 所以请读者在学完本章的基础上再自行研究它。

现在, 让我们抖擞精神来分析P2P真正的主角wpa\_supplicant。

① 注意, Android原生代码中, P2P和STA功能是能同时启用的, 但有一些手机不支持concurrent operation, 所以这些手机需要修改Wi-Fi相关的代码。

## 7.4 wpa\_supplicant中的P2P

在5.2.3节曾介绍，wpa\_supplicant进程由WifiStateMachine启动。在Android官方代码中，虽然Java层有WifiService和WifiP2pService两个几乎完全不同的Wi-Fi服务，但二者都只和Native层的唯一一个wpa\_supplicant进程交互。简单点说，Android原生代码中，一个wpa\_supplicant进程将同时支持WifiService和WifiP2pService。

上述这种设计方法使得wpa\_supplicant负担较重，所以，一些手机厂商会为WifiService和WifiP2pService各创建一个wpa\_supplicant进程，使得它们能各司其职而互不干扰。以笔者的Galaxy Note 2为例，WifiService将和wpa\_supplicant进程交互，而WifiP2pService将和一个名为p2p\_supplicant（经过笔者测试，p2p\_supplicant实际上就是wpa\_supplicant，只不过名字不同而已）的进程交互。

图7-26所示为Galaxy Note 2 init配置文件中关于p2p\_supplicant服务的示意图。

```
service p2p_supplicant /system/bin/p2p_supplicant \
    -ip2p0 -Dnl80211 -c/data/misc/wifi/p2p_supplicant.conf -e/data/misc/wifi/entropy.bin -puse_p2p_group_interface=1
    class main
    disabled
    oneshot
```

图7-26 Galaxy Note 2中p2p\_supplicant服务配置项

由图7-26可知，init配置文件定义了一个名为p2p\_supplicant的服务，该服务启动的进程为p2p\_supplicant。

p2p\_supplicant使用的配置文件名为/data/misc/wifi/p2p\_supplicant.conf，其内容如图7-27所示。

**提示** 关于init配置文件中wpa\_supplicant服务的说明，请参考4.3节wpa\_supplicant初始化流程分析。

图7-27中，p2p\_supplicant对应的ctrl\_iface路径为/data/misc/wifi/sockets。所以，如果要使用wpa\_cli和

p2p\_supplicant交互，必须指定正确的ctrl\_iface路径。图7-28所示为笔者用wpa\_cli测试p2p\_supplicant时的截图。

```
shell@android:/ # cat /data/misc/wifi/p2p_supplicant.conf
update_config=1
ctrl_interface=/data/misc/wifi/sockets
eapol_version=1
ap_scan=1
fast_reauth=1
p2p_listen_reg_class=81
p2p_listen_channel=1
p2p_oper_reg_class=115
p2p_oper_channel=48
device_name=SAMSUNG MOBILE
manufacturer=SAMSUNG ELECTRONICS
model_name=SAMSUNG MOBILE
model_number=2012
serial_number=19691101shell@android:/ #
```

图7-27 p2p\_supplicant.conf内容

```
shell@android:/ # wpa_cli -i /data/misc/wifi/sockets/p2p0
wpa_cli v2.0-devel-4.1.1
Copyright (c) 2004-2012, Jouni Malinen <j@w1.fi> and contributors

This software may be distributed under the terms of the BSD license.
See README for more details.

Interactive mode

> status
wpa_state=INACTIVE
p2p_device_address=92:18:7c:69:88:e2
address=92:18:7c:69:88:e2
<3>CTRL-EVENT-STATE-CHANGE id=-1 state=2 BSSID=00:00:00:00:00 SSID=
```

图7-28 wpa\_cli和p2p\_supplicant交互

下面来分析wpa\_supplicant中和P2P相关的代码。

注意 以Galaxy Note 2为例，p2p\_supplicant就是wpa\_supplicant，只是编译时打开了P2P相关的选项。下面的分析将以wpa\_supplicant中和P2P相关的代码及工作流程为主。

## 7.4.1 P2P模块初始化

首先来看WPAS中P2P相关模块的初始化。该初始化工作在4.3.4节“wpa\_supplicant\_init\_iface分析之五”曾提到过，其对应的函数wpas\_p2p\_init如下。

[p2p\_supplicant.c: : wpas\_p2p\_init]

```
int wpas_p2p_init(struct wpa_global *global, struct
wpa_supplicant *wpa_s)
{
    struct p2p_config p2p; // p2p变量指向一个p2p_config对象，代表
P2P模块的配置信息
    unsigned int r; int i;
    // ①WPA_DRIVER_FLAGS_P2P_CAPABLE代表wifi驱动对P2P支持的能力，
详情见下文解释
    if (!(wpa_s->drv_flags & WPA_DRIVER_FLAGS_P2P_CAPABLE))
return 0;

    if (global->p2p) return 0;
    // 如果wifi driver能完成P2P功能，就不用劳驾WPAS了
    if (wpa_s->drv_flags & WPA_DRIVER_FLAGS_P2P_MGMT) {.....}

    // ②初始化并设置p2p_config对象
    os_memset(&p2p, 0, sizeof(p2p));
    p2p.msg_ctx = wpa_s; p2p.cb_ctx = wpa_s;
    p2p.p2p_scan = wpas_p2p_scan; // P2P对应的扫描函数
    .....// 设置一些回调函数
    p2p.get_noa = wpas_get_noa; p2p.go_connected =
wpas_go_connected;
    // 设置P2P Device address。
    os_memcpy(wpa_s->global->p2p_dev_addr, wpa_s->own_addr,
ETH_ALEN);
    os_memcpy(p2p.dev_addr, wpa_s->global->p2p_dev_addr,
ETH_ALEN);
    // 设置P2P模块配置信息，包括device name、model name、uuid等
    p2p.dev_name = wpa_s->conf->device_name;
    p2p.manufacturer = wpa_s->conf->manufacturer;
    p2p.model_name = wpa_s->conf->model_name;
    p2p.model_number = wpa_s->conf->model_number;
    p2p.serial_number = wpa_s->conf->serial_number;
    if (wpa_s->wps) {
        os_memcpy(p2p.uuid, wpa_s->wps->uuid, 16);
    }
}
```

```

    p2p.config_methods = wpa_s->wps->config_methods;
}
// 设置Operation Channel信息和listen channel信息
if (wpa_s->conf->p2p_listen_reg_class &&
    wpa_s->conf->p2p_listen_channel) {
    p2p.reg_class = wpa_s->conf->p2p_listen_reg_class;
    p2p.channel = wpa_s->conf->p2p_listen_channel;
} else {.....// 设置默认值}
.....
// 设置国家码
if (wpa_s->conf->country[0] && wpa_s->conf->country[1]) {
    os_memcpy(p2p.country, wpa_s->conf->country, 2);
    p2p.country[2] = 0x04;
} else// 配置文件中没有设置国家，所以取值为"XX\x04"
    os_memcpy(p2p.country, "XX\x04", 3); // 回顾图7-7
// 判断wifi 驱动是否支持配置文件中设置的operationg channel和
listen channel
if (wpas_p2p_setup_channels(wpa_s, &p2p.channels)) {.....}

os_memcpy(p2p.pri_dev_type, wpa_s->conf-
>device_type,WPS_DEV_TYPE_LEN);

p2p.num_sec_dev_types = wpa_s->conf->num_sec_device_types;
os_memcpy(p2p.sec_dev_type, wpa_s->conf->sec_device_type,
          p2p.num_sec_dev_types * WPS_DEV_TYPE_LEN);
// 是否支持concurrent operation
p2p.concurrent_operations = !(wpa_s-
>drv_flags&WPA_DRIVER_FLAGS_P2P_CONCURRENT);

p2p.max_peers = 100;// 最多能保存100个对端P2P Device信息
/*
配置文件中没有设置p2p_ssid_postfix，但P2pStateMachine在
initializeP2pSettings函数中
将设置P2P SSID后缀。以笔者的Galaxy Note 2为例，其P2P SSID后缀
为"Android_4aa9"。
*/
if (wpa_s->conf->p2p_ssid_postfix) {.....}

p2p.p2p_intra_bss = wpa_s->conf->p2p_intra_bss;
// ③global->p2p指向一个p2p_data结构体，它是WPAS中P2P模块的代表
global->p2p = p2p_init(&p2p);
.....
for (i = 0; i < MAX_WPS_VENDOR_EXT; i++) { // 拷贝vendor厂商特
定的WSC属性信息
    if (wpa_s->conf->wps_vendor_ext[i] == NULL) continue;
    p2p_add_wps_vendor_extension(global->p2p, wpa_s->conf-

```

```

>wps_vendor_ext[i]);
}

return 0;
}

```

由上述代码可知，wpas\_p2p\_init的工作非常简单，主要包括：

- 初始化一个p2p\_config对象，然后根据p2p\_supplicant.conf文件的信息来设置其中的内容，同时还需要为P2P模块设置一些回调函数。
- 调用p2p\_init函数以初始化P2P模块。

下面来介绍上述代码中涉及的一些知识。

## 1. Driver Flags和重要数据结构

先来看上述代码中提到的drv\_flags变量。WPAS中，Wi-Fi驱动对P2P功能的支持情况就是由它来表达的。Galaxy Note 2中该变量取值为0x2EAC0，其表达的含义如下。

[-->driver.h]

```

#define WPA_DRIVER_FLAGS_AP           0x00000040 // wifi driver支
持AP。它使得P2P设备能扮演GO
/*
  标志标明association成功后，Kernel driver需要设置WEP key。
  这个标志出现的原因是Kernel API发生了变动，使得只能在关联成功后才能设置
key。
*/
#define WPA_DRIVER_FLAGS_SET_KEYS_AFTER_ASSOC_DONE
0x00000080
#define WPA_DRIVER_FLAGS_P2P_CONCURRENT      0x00000200// wifi
driver支持STA和P2P的并发运行
#define WPA_DRIVER_FLAGS_P2P_CAPABLE        0x00000800 // wifi
driver支持P2P
/*
  7.2.2节Probe Request帧设置曾提到，P2P包含Device Address和
Interface Address
  两种类型的地址。在实际实现过程中，这两个地址分别代表两个Virtual
Interface。显然，P2P中第一个
  和一直存在的是拥有Device Address的Virtual Interface。下面这个标志表
示该Virtual Interface

```

可以参与P2P管理（除P2P Group Operation之外的工作）工作以及非P2P相关的工作（例如利用这个

Virtual Interface 加入到一个BSS）。

\*/

```
#define WPA_DRIVER_FLAGS_P2P_MGMT_AND_NON_P2P 0x00002000
/*
```

该标志主要针对associate操作。当关联操作失败后，如果driver支持该选项，则表明driver能处理失败

之后的各种收尾工作（Key、timeout等工作）。否则，WPAS需要自己处理这些事情。

\*/

```
#define WPA_DRIVER_FLAGS_SANE_ERROR_CODES          0x00004000
/*
```

下面这个标志和off channel机制有关，可参考4.3.4节关于capability的介绍。  
当802.11

MAC帧通过off channel发送，下面这个标志表示driver会反馈一个发送情况（TX Report）消息给WPAS。

\*/

```
#define WPA_DRIVER_FLAGS_OFFCHANNEL_TX             0x00008000
/*
```

下面这两个标志表示Kernel中的driver是否能反馈  
Deauthentication/Disassociation帧  
发送情况（TX Report）。

\*/

```
#define WPA_DRIVER_FLAGS_DEAUTH_TX_STATUS          0x00020000
```

下面来看wpas\_p2p\_init中出现的几个重要数据结构。首先是  
p2p\_config和p2p\_data，它们的成员如图7-29所示。

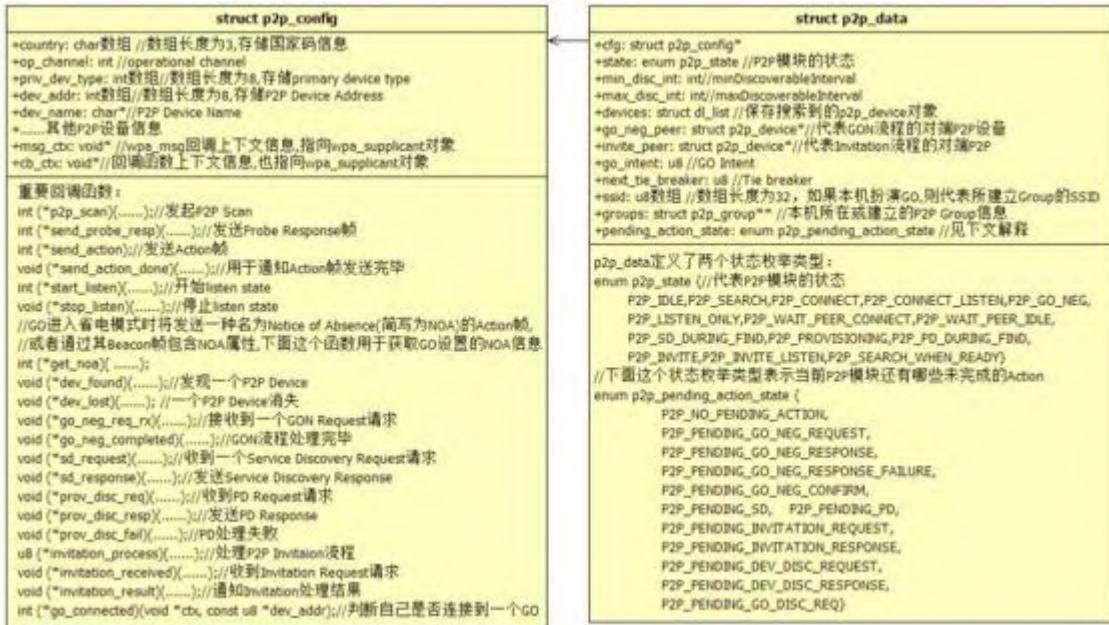


图7-29 p2p\_config和p2p\_data结构

图7-29展示了p2p\_config和p2p\_data两个数据结构中一些重要的成员。

- p2p\_config定义了20个回调函数。这些回调函数定义了P2P模块和外界交互的接口。在wpas\_p2p\_init中，这些回调函数均指向p2p\_supplicant.c中对应的函数，例如p2p\_scan指向wpas\_p2p\_scan，dev\_lost指向wpas\_dev\_lost。另外，由于回调函数的参数比较复杂，所以图中均省略了参数信息。

- p2p\_data指向一个p2p\_config对象。

下面来看另外几个重要数据结构的内容，图7-30展示了五种数据结构。

- p2p\_device代表一个P2P设备。其中设备名、Device CapabilityBitmap等信息保存在一个类型为p2p\_peer\_info的对象中。
- p2p\_group代表一个P2P Group的信息，其内部包含一个p2p\_group\_config对象和一个p2p\_group\_member链表。p2p\_group\_config表示该Group的配置信息，p2p\_group\_member代表Group Member即P2P Client的信息。

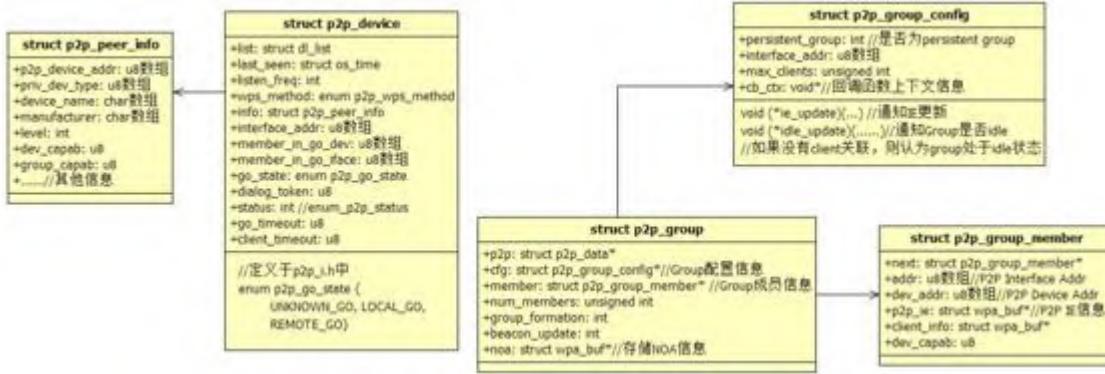


图7-30 p2p\_device及其他数据结构

**提示** WPAS中定义了非常多的数据结构类型，这极大增加了初学者的学习难度。根据笔者的经验，建议在学习过程中先简单了解这些数据结构的名字及作用，然后在具体代码分析时再结合代码逻辑来了解这些数据结构及其内部各个成员变量的具体作用。

下面来看p2p\_init函数。

## 2. p2p\_init函数

p2p\_init函数将初始化WPAS中的P2P模块，其代码如下所示。

```
[-->p2p.c: : p2p_init]

struct p2p_data * p2p_init(const struct p2p_config *cfg)
{
    struct p2p_data *p2p;
    .....
    /*

从下面这行代码可看出，一个p2p_data对象的内存分布，该内存将包含一个p2p_data的所有信息以及一个p2p_config对象的所有信息。
    */
    p2p = os_zalloc(sizeof(*p2p) + sizeof(*cfg));
    // 将p2p_data的cfg成员变量指向保存p2p_config信息的那块内存地址
    p2p->cfg = (struct p2p_config *) (p2p + 1);
    os_memcpy(p2p->cfg, cfg, sizeof(*cfg)); // 拷贝传入的p2p_config信息
    if (cfg->dev_name) p2p->cfg->dev_name = os_strdup(cfg-
>dev_name);
```

```

.....// 其他信息拷贝
#ifndef ANDROID_P2P
    p2p->min_disc_int = 2;                                // listen state的最
小时间为200毫秒
    p2p->sd_dev_list = NULL;
#else
    p2p->min_disc_int = 1;
#endif
    p2p->max_disc_int = 3;
    // 随机获取next_tie_breaker的初值
    // 第二个参数1表示next_tie_breaker的字节长度，其类型是u8
    os_get_random(&p2p->next_tie_breaker, 1);
    p2p->next_tie_breaker &= 0x01;

    // 设置本机P2P Device的device capability信息
    if (cfg->sd_request) p2p->dev_capab |=
P2P_DEV_CAPAB_SERVICE_DISCOVERY;
    p2p->dev_capab |= P2P_DEV_CAPAB_INVITATION_PROCEDURE;

    if (cfg->concurrent_operations)// 支持concurrent功能
        p2p->dev_capab |= P2P_DEV_CAPAB_CONCURRENT_OPER;
    p2p->dev_capab |= P2P_DEV_CAPAB_CLIENT_DISCOVERABILITY;

    dl_list_init(&p2p->devices);
    // 注册一个超时时间（如果定义了ANDROID_P2P宏，该时间为30ms）
    // 用来检测是否有不活跃的p2p_device
    eloop_register_timeout(P2P_PEER_EXPIRATION_INTERVAL, 0,
                           p2p_expiration_timeout, p2p, NULL);

    return p2p;
}

```

p2p模块初始化还算比较简单。

### 3. 注册Action帧监听事件

4.3.4节分析wpa\_driver\_nl80211\_finish\_drv\_init时曾介绍过，wpa\_driver\_nl80211\_set\_mode函数和P2P关系较大。为什么这么说呢？相信代码能给出最直接的解释。

[-->driver\_nl80211.c: : wpa\_driver\_nl80211\_set\_mode]

```

static int wpa_driver_nl80211_set_mode(struct i802_bss *bss,
enum nl80211_iftype nlmode)

```

```

{ // 注意，在wpa_driver_nl80211_finish_drv_init函数中，nlmode被设置
为NL80211_IFTYPE_STATION
    struct wpa_driver_nl80211_data *drv = bss->drv;
    int ret = -1; int i;
    /*
        drv->nlmode的类型为enum nl80211_iftype, 4.3.4节关于
wpa_driver_nl80211_finish_drv_init
        的分析中有对该变量的解释。drv->nlmode只有为NL80211_IFTYPE_AP或
NL80211_IFTYPE_P2P_GO时，
        is_ap_interface函数才返回非0值。很显然此时virtual interface的类
型不可能是GO。
    */
    int was_ap = is_ap_interface(drv->nlmode);
    int res;
    // 设置虚拟interface的类型为NL80211_IFTYPE_STATION
    res = nl80211_set_mode(drv, drv->ifindex, nlmode);
    if (res == 0) {
        drv->nlmode = nlmode;
        ret = 0;
        goto done; // 设置成功，直接跳转到done处
    }
    .....
done:
    .....
    if (is_ap_interface(nlmode)) {
        nl80211_mgmt_unsubscribe(bss, "start AP");
        if (nl80211_setup_ap(bss)) return -1;
    } else if (was_ap) {
        /* Remove additional AP mode functionality */
        nl80211_teardown_ap(bss);
    } else {
        // 本例将执行下面这个函数以取消监听Action帧事件
        // 由于之前并未注册，所以此时执行这个函数将没有实际作用
        nl80211_mgmt_unsubscribe(bss, "mode change");
    }

    if (!is_ap_interface(nlmode) &&
        nl80211_mgmt_subscribe_non_ap(bss) < 0) // 注册对Action帧
的监听事件
        wpa_printf(MSG_DEBUG, "nl80211: Failed to register
Action "
                    "frame processing - ignore for now");

    return 0;
}

```

n180211\_mgmt\_subscribte\_non\_ap将注册对Action帧的监听事件，其作用就是当设备收到Action帧后，Wi-Fi驱动将发送对应的netlink消息给WPAS。来看n180211\_mgmt\_subscribte\_non\_ap函数，代码如下所示。

```
[-->driver_n180211.c: : n180211_mgmt_subscribte_non_ap]

static int n180211_mgmt_subscribe_non_ap(struct i802_bss *bss)
{
    struct wpa_driver_n180211_data *drv = bss->drv;
    /*

下面这个函数将注册libnl回调事件到event loop。在4.3.4节关于
wpa_driver_n180211_init_nl与n180211_init_bss分析中曾详细介绍
过该函数。
    总之，当WPAS收到对应的netlink消息后，process_bss_event函数将被调用。
    */

    if (n180211_alloc_mgmt_handle(bss)) return -1;

#ifndef CONFIG_P2P || defined(CONFIG_INTERWORKING)
    .... // 注册对GAS Public Action帧的监听，Service Discovery和
它有关
#endif /* CONFIG_P2P || CONFIG_INTERWORKING */
#ifdef CONFIG_P2P
    /*
        注册对P2P Public Action帧的监听，第二个参数中的04-09-50-6F-9A-09
指明了P2P Public Action
        帧Frame Body的Category、Action Field、OUI、OUI-Type（参考表7-
3）的取值。即只有收到的
        Frame Body对应字段分别等于上述指定值的Action帧，Wi-Fi驱动才会发送
        netlink消息给WPAS。
    */
    if (n180211_register_action_frame(bss, (u8 *)
"\x04\x09\x50\x6f\x9a\x09", 6) < 0)
        return -1;
    /*
        注册对P2P Action帧的监听，第二个参数中7F-50-6F-9A-09指明了Action
帧Frame Body的
        Category和OUI。根据802.11规范，7F代表Vendor Specific，50-6F-9A
是WFA的OUI，最后一个
        09代表P2P。
    */
    if (n180211_register_action_frame(bss, (u8 *
")"\x7f\x50\x6f\x9a\x09", 5) < 0) return -1;
```

```

#endif /* CONFIG_P2P */
#ifndef CONFIG_IEEE80211W
.....
#endif /* CONFIG_IEEE80211W */
#ifndef CONFIG_TDLS
.....
#endif /* CONFIG_TDLS */
.....// 其他感兴趣帧的注册
return 0;
}

```

由上述代码可知n180211\_mgmt\_subscribe\_non\_ap在P2P方面注册了两种类型的帧监听事件。

- P2P Public Action帧监听事件：根据P2P规范，目前使用的均是802.11 Public Action帧，即Category的值为0x04。目前GON、P2P Invitation、Provision Discovery以及Device Discoverability使用P2P Public Action帧。
- P2P Action帧监听事件：这种类型的帧属于802.11 Action帧的一种，其Category取值为0x7F，OUI指定为WFA的OUI（即50-6F-9A），而OUI-Type指定为P2P（取值为0x09）。目前Notice of Absence、P2P Presence、GO Discoverability使用P2P Action帧。

注意 上述注册的Action帧监听事件对应的处理函数是process\_bss\_event。

至此，P2P模块以及Action帧监听事件注册等工作都已完成，WPAS马上可为WifiP2pService提供P2P相关的服务了。下面将结合7.2节中介绍的如下几个重要的P2P工作流程来分析代码。

- 搜索周围的P2P设备。
- 向某个P2P设备发起Provision Discovery流程。
- 对端设备开展Group Formation流程，重点关注其中的Group Negotiation。

提示 GON结束后，如果本机设备扮演Client，则后续工作包括Provisioning（即WSC安全配置协商，本机充当Enrollee，参考第6

章) 和加入Group (类似STA加入AP, 参考第4章)。如果本机扮演GO, 则后续工作也分为Provisioning (充当Registrar) 和处理Client加入Group (扮演AP的角色)。本书不讨论和GO相关的知识, 请读者在阅读完相关章节基础上自行研究它们。

## 7.4.2 P2P Device Discovery流程分析

根据7.3.2节中对DISCOVER\_PEERS命令的代码分析可知，P2pStateMachine将发送“P2P\_FIND 120”命令给WPAS以触发P2P Device Discovery流程。处理该命令的代码如下所示。

```
[-->ctrl_iface.c: : wpa_supplicant_ctrl_iface_process]

char * wpa_supplicant_ctrl_iface_process(struct wpa_supplicant
*wpa_s,
                                         char *buf, size_t *resp_len)
{
    char *reply; const int reply_size = 4096;
    int ctrl_rsp = 0; int reply_len;
    .....
    #ifdef CONFIG_P2P
        // 处理带参数的P2P_FIND命令
    } else if (os_strncmp(buf, "P2P_FIND ", 9) == 0) {
        // 注意"P2P_FIND "多了一个空格
        if (p2p_ctrl_find(wpa_s, buf + 9)) reply_len = -1;
    } else if (os_strcmp(buf, "P2P_FIND") == 0) {
        // 处理不带参数的P2P_FIND命令
        if (p2p_ctrl_find(wpa_s, "")) reply_len = -1;
    } .....// 其他P2P命令处理
#endif
    .....
}
```

不论P2P\_FIND命令是否携带参数，其最终的处理函数都将对应为p2p\_ctrl\_find，如下所示。

```
[-->ctrl_iface.c: : p2p_ctrl_find]

static int p2p_ctrl_find(struct wpa_supplicant *wpa_s, char
*cmd)
{
    unsigned int timeout = atoi(cmd);
    enum p2p_discovery_type type = P2P_FIND_START_WITH_FULL;
    // 搜索方式，见下文解释
    u8 dev_id[ETH_ALEN], *_dev_id = NULL;
    char *pos;
    // 设置搜索方式，见下文解释
```

```

    if (os strstr(cmd, "type=social")) type =
P2P_FIND_ONLY_SOCIAL;
    else if (os strstr(cmd, "type=progressive")) type =
P2P_FIND_PROGRESSIVE;

    pos = os strstr(cmd, "dev_id");// dev_id代表peer端device的
MAC地址
    if (pos) {....}
    // wpas_p2p_find内部将调用p2p_find, 下文将直接分析p2p_find
    return wpas_p2p_find(wpa_s, timeout, type, 0, NULL,
    dev_id);
}

```

P2P\_FIND支持三种不同的Discovery Type，分别如下。

- P2P\_FIND\_START\_WITH\_FULL：默认设置。表示先扫描所有频段，然后再扫描social channels。这种搜索方式如图7-3所示。
- P2P\_FIND\_ONLY\_SOCIAL：只扫描social channels。它将跳过“扫描所有频段”这一过程。这种搜索方式能加快搜索的速度。
- P2P\_FIND\_PROGRESSIVE：它和P2P\_FIND\_START\_WITH\_FULL类似，只不过在Search State阶段将逐个扫描所有频段。为什么在search state阶段会扫描所有频段呢？请读者参考图7-22中的状态切换路线14。当周围已经存在Group的时候，如果在最初的“扫描所有频段”这一过程中没有发现Group，则在后续的search state逐个扫描频段过程中就有可能发现之前那些没有找到的Group。

注意 GO将工作在Operation Channel，而Listen Channel只在最初的P2P Device Discovery阶段使用。

## 1. P2P设备扫描流程

P2P设备扫描流程从wpas\_p2p\_find开始，其代码如下所示。

[-->p2p\_supplicant.c: : wpas\_p2p\_find]

```

int wpas_p2p_find(struct wpa_supplicant *wpa_s, unsigned int
timeout,
                    enum p2p_discovery_type type, unsigned int
num_req_dev_types,
                    const u8 *req_dev_types, const u8 *dev_id)

```

```

{
    /*
        取消还未发送的Action帧数据。WPAS中，待发送的Action帧数据保存在
        wpa_supplicant对象的
        pending_action_tx变量中，它指向一块数据缓冲区。
    */
    wpas_p2p_clear_pending_action_tx(wpa_s);
    wpa_s->p2p_long_listen = 0;
    /*
        如果wifi driver能直接处理P2P管理，则主要工作将由wifi driver来完成。
    Galaxy Note 2
    不支持WPA_DRIVER_FLAGS_P2P_MGMT。
    */
    if (wpa_s->drv_flags & WPA_DRIVER_FLAGS_P2P_MGMT)
        return wpa_drv_p2p_find(wpa_s, timeout, type);
    .....
    // 取消计划扫描任务
    wpa_supplicant_cancel_sched_scan(wpa_s);
    // 调用p2p_find函数
    return p2p_find(wpa_s->global->p2p, timeout, type,
                    num_req_dev_types, req_dev_types, dev_id);
}

```

来看p2p\_find函数，其代码如下所示。

```

[-->p2p.c: : p2p_find]

int p2p_find(struct p2p_data *p2p, unsigned int timeout, enum
p2p_discovery_type type,
             unsigned int num_req_dev_types, const u8
*req_dev_types, const u8 *dev_id)
{
    int res;
    p2p_free_req_dev_types(p2p);
    if (req_dev_types && num_req_dev_types) {.....// 本例没有设
置request dev type属性}

    if (dev_id) {
        os_memcpy(p2p->find_dev_id_buf, dev_id, ETH_ALEN);
        p2p->find_dev_id = p2p->find_dev_id_buf;
    } else p2p->find_dev_id = NULL;
    // 注意下面这个P2P_AFTER_SCAN NOTHING标志，它表示P2P设备完成scan动
作后，无须做其他动作
    p2p->start_after_scan = P2P_AFTER_SCAN NOTHING;
    p2p_clear_timeout(p2p);
}

```

```

p2p->cfg->stop_listen(p2p->cfg->cb_ctx); // 停止监听
p2p->find_type = type;

p2p_device_clear_reported(p2p);
p2p_set_state(p2p, P2P_SEARCH);           // 设置P2P模块的状态为
P2P_SEARCH

eloop_cancel_timeout(p2p_find_timeout, p2p, NULL);
p2p->last_p2p_find_timeout = timeout;
// 注册一个扫描超时处理任务
if (timeout) eloop_register_timeout(timeout, 0,
p2p_find_timeout, p2p, NULL);
switch (type) {
case P2P_FIND_START_WITH_FULL:
case P2P_FIND_PROGRESSIVE:           // p2p_scan函数指针指向
wpas_p2p_scan
    res = p2p->cfg->p2p_scan(p2p->cfg->cb_ctx,
P2P_SCAN_FULL, 0,
p2p->num_req_dev_types, p2p->req_dev_types,
dev_id);
    break;
case P2P_FIND_ONLY_SOCIAL:
    res = p2p->cfg->p2p_scan(p2p->cfg->cb_ctx,
P2P_SCAN_SOCIAL, 0,
p2p->num_req_dev_types, p2p->req_dev_types,
dev_id);
    break;
default:
    return -1;
}
if (res == 0) {
    // 设置p2p_scan_running值为1，该变量后面用到的地方比较多，请读者注意
    p2p->p2p_scan_running = 1;
    eloop_cancel_timeout(p2p_scan_timeout, p2p, NULL);
    eloop_register_timeout(P2P_SCAN_TIMEOUT, 0,
p2p_scan_timeout, p2p, NULL);
} .....           // 略去res为其他值的处理情况
return res;
}

```

上述代码中p2p\_config对象的p2p\_scan函数指针变量将指向p2p\_supplicant.c中的wpas\_p2p\_scan，马上来看它，代码如下所示。

[-->p2p\_supplicant.c: : wpas\_p2p\_scan]

```
static int wpas_p2p_scan(void *ctx, enum p2p_scan_type type,
int freq,
    unsigned int num_req_dev_types, const u8
*req_dev_types, const u8 *dev_id)
{
    struct wpa_supplicant *wpa_s = ctx;
    // 扫描参数
    struct wpa_driver_scan_params params;
    int ret; struct wpabuf *wps_ie, *ies;
    int social_channels[] = { 2412, 2437, 2462, 0, 0 };
    size_t ielen; int was_in_p2p_scan;

    os_memset(&params, 0, sizeof(params));

    params.num_ssids = 1; // 设置SSID参数,
P2P_WILDCARD_SSID的值为“DIRECT-”
    params.ssids[0].ssid = (u8 *) P2P_WILDCARD_SSID;
    params.ssids[0].ssid_len = P2P_WILDCARD_SSID_LEN;

    wpa_s->wps->dev.p2p = 1;
    // 构造Probe Request帧中WSC IE信息
    wps_ie = wps_build_probe_req_ie(0, &wpa_s->wps->dev, wpa_s-
>wps->uuid,
        WPS_REQ_ENROLLEE, num_req_dev_types,
        req_dev_types);

    ielen = p2p_scan_ie_buf_len(wpa_s->global->p2p);
    ies = wpabuf_alloc(wpabuf_len(wps_ie) + ielen);
    .....
    // 构造P2P IE信息, 感兴趣的读者不妨自行阅读p2p_scan_ie函数
    p2p_scan_ie(wpa_s->global->p2p, ies, dev_id);

    params.p2p_probe = 1;
    params.extra_ies = wpabuf_head(ies); params.extra_ies_len =
wpabuf_len(ies);

    switch (type) {
        case P2P_SCAN_SOCIAL: // 只扫描social channels的话, 将设置
params.freqs变量
            params.freqs = social_channels;
            break;
        case P2P_SCAN_FULL:
            break;
        .....// 其他扫描频段控制
```

```

    }

    was_in_p2p_scan = wpa_s->scan_res_handler ==
wpas_p2p_scan_res_handler;
    // 设置P2P扫描结果处理函数
    wpa_s->scan_res_handler = wpas_p2p_scan_res_handler;
    // 发起P2P设备扫描，该函数内部将调用driver_nl80211.c的
wpa_driver_nl80211_scan函数
    ret = wpa_drv_scan(wpa_s, &params);
    wpabuf_free(ies);
    .....
    return ret;
}

```

读者可比较本节和4.5.3节无线网络扫描流程分析的内容。总体而言，P2P设备扫描的代码逻辑比无线网络扫描的代码逻辑要简单得多。图7-31为WPAS中P2P设备扫描所涉及的几个重要函数调用。

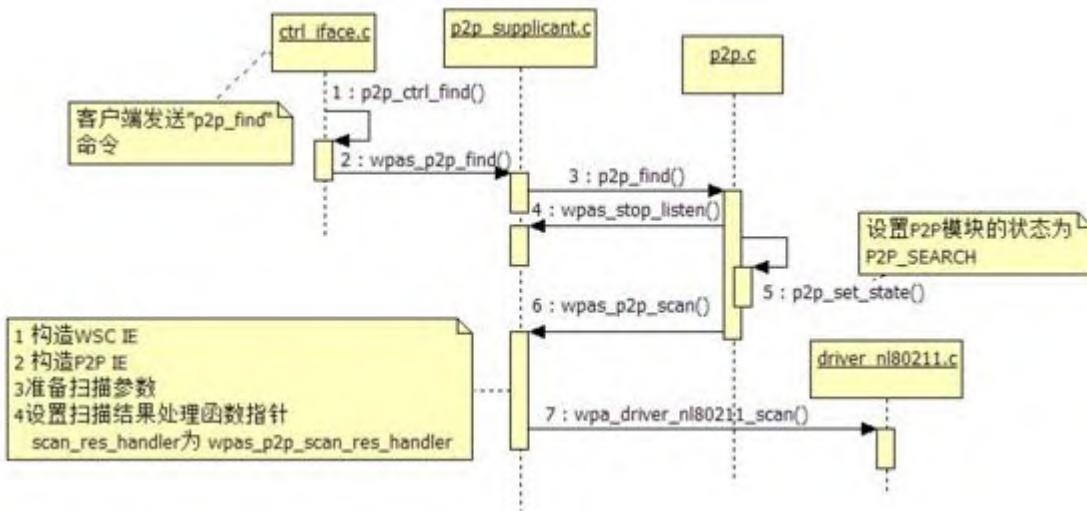


图7-31 P2P Device扫描流程

下面来看P2P设备扫描结果的处理流程。

## 2. P2P设备扫描结果处理流程

由4.5.3节“\_wpa\_supplicant\_event\_scan\_results分析之二”中的代码可知，当scan\_res\_handler不为空的时候，扫描结果将交给scan\_res\_handler来处理。由图7-31可知，对P2P设备扫描时将设置scan\_res\_handler为wpas\_p2p\_scan\_res\_handler，其代码如下所示。

```
[-->p2p_supplicant.c: : wpas_p2p_scan_res_handler]

static void wpas_p2p_scan_res_handler(struct wpa_supplicant
*wpa_s,
                                     struct wpa_scan_results *scan_res)
{
    size_t i;
    .....
    for (i = 0; i < scan_res->num; i++) {
        struct wpa_scan_res *bss = scan_res->res[i];
        // ①对每一个扫描结果调用p2p_scan_res_handler函数
        if (p2p_scan_res_handler(wpa_s->global->p2p, bss-
>bssid, // 处理扫描结果
                                bss->freq, bss->level, (const u8 *) (bss + 1),
bss->ie_len) > 0)
            break;
    }
    p2p_scan_res_handled(wpa_s->global->p2p); // ②处理完毕后调用
p2p_scan_res_handled
}
```

wpas\_p2p\_scan\_res\_handler中有两个关键函数，先来看第一个。

### (1) p2p\_scan\_res\_handler函数

p2p\_scan\_res\_handler的代码如下所示。

```
[-->p2p.c: : p2p_scan_res_handler]

int p2p_scan_res_handler(struct p2p_data *p2p, const u8 *bssid,
int freq,
           int level, const u8 *ies, size_t ies_len)
{
    // 添加一个P2P Device, 详情见下文代码分析
    p2p_add_device(p2p, bssid, freq, level, ies, ies_len);
/*
go_neg_peer代表GON的对端设备，如果go_neg_peer不为空而且设备扫描时由
发现了它，则直接通过
    p2p_connect_send向其发送GON Request帧以开展GON流程。
*/
    if (p2p->go_neg_peer && p2p->state == P2P_SEARCH &&
        os_memcmp(p2p->go_neg_peer->info.p2p_device_addr,
bssid, ETH_ALEN) == 0) {
        p2p_connect_send(p2p, p2p->go_neg_peer);
        return 1;
    }
}
```

```

    }
    return 0;
}

```

上述代码中最重要的是p2p\_add\_device函数，其代码如下所示。

[-->p2p.c: : p2p\_add\_device]

```

int p2p_add_device(struct p2p_data *p2p, const u8 *addr, int
freq, int level,
                     const u8 *ies, size_t ies_len)
{
    struct p2p_device *dev;  struct p2p_message msg;
    const u8 *p2p_dev_addr;  int i;

    os_memset(&msg, 0, sizeof(msg));
    // 解析扫描结果中的IE信息，解析完的结果保存在一个p2p_message对象中
    if (p2p_parse_ies(ies, ies_len, &msg)) {.....// 解析扫描结
果}

    // p2p device info中的属性
    if (msg.p2p_device_addr) p2p_dev_addr =
msg.p2p_device_addr;
    else if (msg.device_id) p2p_dev_addr = msg.device_id;
    .....
    // 过滤那些被阻止的P2P Device
    if (!is_zero_ether_addr(p2p->peer_filter) &&
        os memcmp(p2p_dev_addr, p2p->peer_filter, ETH_ALEN) != 0) {.....}
    // 构造一个p2p_device对象，并将其加入p2p_data结构体的devices链表
    // 中
    dev = p2p_create_device(p2p, p2p_dev_addr); // 创建一个P2P
Device对象
    .....
    os_get_time(&dev->last_seen); // 设置发现该P2P Device的时间
    // p2p_device的flags变量代表该p2p_device的一些信息
    dev->flags &= ~(P2P_DEV_PROBE_REQ_ONLY |
P2P_DEV_GROUP_CLIENT_ONLY);

    if (os memcmp(addr, p2p_dev_addr, ETH_ALEN) != 0)
        os_memcpy(dev->interface_addr, addr,
ETH_ALEN);
    .... // 处理ssid、listen channel等内容
    dev->listen_freq = freq;
    // 如果对端P2P Device是GO，它回复的Probe Response帧P2P IE信息中

```

将包含Group Info属性

```
if (msg.group_info) dev->oper_freq = freq;
dev->info.level = level;
p2p_copy_wps_info(dev, 0, &msg); // 复制WSC IE
.....// 处理Vendor相关的IE信息
// 根据Group Info信息添加Client。就本例而言，周围还不存在GO
p2p_add_group_clients(p2p, p2p_dev_addr, addr, freq,
msg.group_info, msg.group_info_len);

p2p_parse_free(&msg);
// 判断是否有Service Discovery Request，如果有，需要为flags设置
P2P_DEV_SD_SCHEDULE标志位
if (p2p_pending_sd_req(p2p, dev)) dev->flags |=
P2P_DEV_SD_SCHEDULE;
// P2P_DEV_REPORTED表示WPAS已经向客户端汇报过该P2P Device信息了
if (dev->flags & P2P_DEV_REPORTED) return 0;
// P2P_DEV_USER_REJECTED表示用户拒绝该P2P Device信息
if (dev->flags & P2P_DEV_USER_REJECTED) {return 0;}
/*
dev_found函数指针指向wpas_dev_found，该函数将向WifiMonitor发送消息以告知我们找到了一个
P2P Device，该消息也称为P2P Device Found消息。
*/
p2p->cfg->dev_found(p2p->cfg->cb_ctx, addr, &dev->info,
!(dev->flags & P2P_DEV_REPORTED_ONCE));
// 下面这两个标志表示该P2P Device已经向客户端汇报过并且汇报过一次了
dev->flags |= P2P_DEV_REPORTED | P2P_DEV_REPORTED_ONCE;
return 0;
}
```

图7-32为WPAS向其客户端汇报的P2P Device Found消息的格式示例。

```
/* P2P-DEVICE-FOUND fa:7b:7a:42:02:13 p2p_dev_addr=fa:7b:7a:42:02:13 pri_dev_type=l-0050F204-1
... name='p2p-TEST1' config_methods=0x188 dev_capab=0x27 group_capab=0x0 */
private static final String P2P_DEVICE_FOUND_STR = "P2P-DEVICE-FOUND";
```

图7-32 P2P Device Found消息

## (2) p2p\_scan\_res\_handled函数

接着来看第二个关键函数p2p\_scan\_res\_handled。

[-->p2p.c: : p2p\_scan\_res\_handled]

```

void p2p_scan_res_handled(struct p2p_data *p2p)
{
    .....
    p2p->p2p_scan_running = 0;           // 设置p2p_scan_running
值为0
    eloop_cancel_timeout(p2p_scan_timeout, p2p, NULL); // 取消扫
描超时处理任务
/*
    还记得p2p_find函数中介绍的P2P_AFTER_SCAN_NOTHING标志吗（参考
7.4.2节）？它将用在
    下面这个p2p_run_after_scan函数中。由于指定了
P2P_AFTER_SCAN_NOTHING标志，所以下面这个
    函数返回0。感兴趣的读者可自行研究p2p_run_after_scan函数。
*/
    if (p2p_run_after_scan(p2p)) return;
    if (p2p->state == P2P_SEARCH)           // 就本例而言，将满足此if条
件
        p2p_continue_find(p2p);
}

```

p2p\_continue\_find的代码如下所示。

[-->p2p.c: : p2p\_continue\_find]

```

void p2p_continue_find(struct p2p_data *p2p)
{
    struct p2p_device *dev;
#ifdef ANDROID_P2P
    int skip=1;
#endif
    p2p_set_state(p2p, P2P_SEARCH);
    .... // Service Discovery和Provision Discovery处理。就本例而
言，这部分代码逻辑意义不大
    p2p_listen_in_find(p2p);           // 进入listen state。来看
此函数的代码
}

```

[-->p2p.c: : p2p\_listen\_in\_find]

```

static void p2p_listen_in_find(struct p2p_data *p2p)
{
    unsigned int r, tu;
    int freq;
    struct wpabuf *ies;

```

```

    // 根据p2p_supplicant.conf中listen_channel等配置参数获取对应的频
段
    freq = p2p_channel_to_freq(p2p->cfg->country,
        p2p->cfg->reg_class, p2p->cfg->channel);
    .....
    // 计算需要在listen state等待的时间
    os_get_random((u8 *) &r, sizeof(r));
    tu = (r % ((p2p->max_disc_int - p2p->min_disc_int) + 1) +
        p2p->min_disc_int) * 100;

    p2p->pending_listen_freq = freq;
    p2p->pending_listen_sec = 0;
    p2p->pending_listen_usec = 1024 * tu;
/*
构造P2P Probe Response帧，当我们在Listen state收到其他设备发来的
Probe Request帧后，wifi
驱动将直接回复此处设置的P2P Probe Response帧。
*/
    ies = p2p_build_probe_resp_ies(p2p);
    // start_listen指向wpas_start_listen函数
    if (p2p->cfg->start_listen(p2p->cfg->cb_ctx, freq, 1024
*tu/1000, ies)<0){.....}
    wpabuf_free(ies);
}

```

由上述代码可知，p2p\_listen\_in\_find的内部实现完全遵循了7.2.2节介绍的和listen state相关的理论知识。

下面来看wpas\_start\_listen函数。

[-->p2p\_supplicant.c: : wpas\_start\_listen]

```

static int wpas_start_listen(void *ctx, unsigned int freq,
                             unsigned int duration, const struct wpabuf
*probe_resp_ie)
{
    struct wpa_supplicant *wpa_s = ctx;
    /*
    调用driver_n180211.c的wpa_driver_set_ap_wps_p2p_ie函数，它用于
    将Probe Response帧信息和
    Association Response帧信息。此函数为Android新增的功能，由厂商实
    现。
    */
    wpa_drv_set_ap_wps_ie(wpa_s, NULL, probe_resp_ie, NULL);
    /*

```

调用driver\_n180211.c的wpa\_driver\_n180211\_probe\_req\_report函数，其目的是让wifi driver

收到Probe Request帧后，返回一个EVENT\_RX\_PROBE\_REQ netlink消息给WPAS。

```
*/
```

```
if (wpa_drv_probe_req_report(wpa_s, 1) < 0) {.....}
```

```
wpa_s->pending_listen_freq = freq;
```

```
wpa_s->pending_listen_duration = duration;
```

```
/*
```

调用driver\_n180211.c的wpa\_driver\_n180211\_remain\_on\_channel函数，其目的是让

wlan设备在指定频段（第二个参数freq）上停留duration毫秒。注意，这个函数的返回值只是表示

wifi driver是否成功处理了这个请求，它不能用于判断wifi driver是否已经切换到了指定频段。

如果一切正常，当wifi driver切换到指定频段后，它将发送一个名为EVENT\_REMAIN\_ON\_CHANNEL的

```
netlink消息给WPAS。
```

```
*/
```

```
if (wpa_drv_remain_on_channel(wpa_s, freq, duration) < 0)  
{.....}
```

```
wpa_s->off_channel_freq = 0;
```

```
wpa_s->roc_waiting_drv_freq = freq;
```

```
return 0;
```

```
}
```

wpas\_start\_listen比较简单，不过有一些知识点请读者注意。

- wpa\_drv\_set\_ap\_wps\_ie为wifi driver设置了P2P IE信息。如果 wifi driver自己处理Probe Request帧（即不发送 EVENT\_RX\_PROBE\_REQ消息给WPAS），则wifi driver将把此处设置的 P2P IE信息填写到Probe Response帧中。

- wpa\_drv\_probe\_req\_report要求wifi driver收到Probe Request帧后，发送EVENT\_RX\_PROBE\_REQ消息给WPAS。WPAS内部将处理此消息，最终会回复一个Probe Response帧。这部分代码请读者自行阅读。

- wpa\_drv\_remain\_on\_channel要求wifi driver在指定频段工作一段时间。当wifi driver切换到指定频段后，会发送 EVENT\_REMAIN\_ON\_CHANNEL消息给WPAS，WPAS内部将处理一些事情。这部分代码也请读者自行阅读。

**提醒** 笔者感觉wpa\_drv\_set\_ap\_wps\_ie和wpa\_drv\_probe\_req\_report功能重复。不过由于driver.h对set\_ap\_wps\_ie的功能描述不是特别清晰，请了解细节的读者和我们分享相关的知识。另外，请读者认真阅读EVENT\_RX\_PROBE\_REQ和EVENT\_REMAIN\_ON\_CHANNEL消息的处理代码。

### (3) P2P设备扫描结果处理流程总结

图7-33展示了P2P设备扫描结果处理的流程。

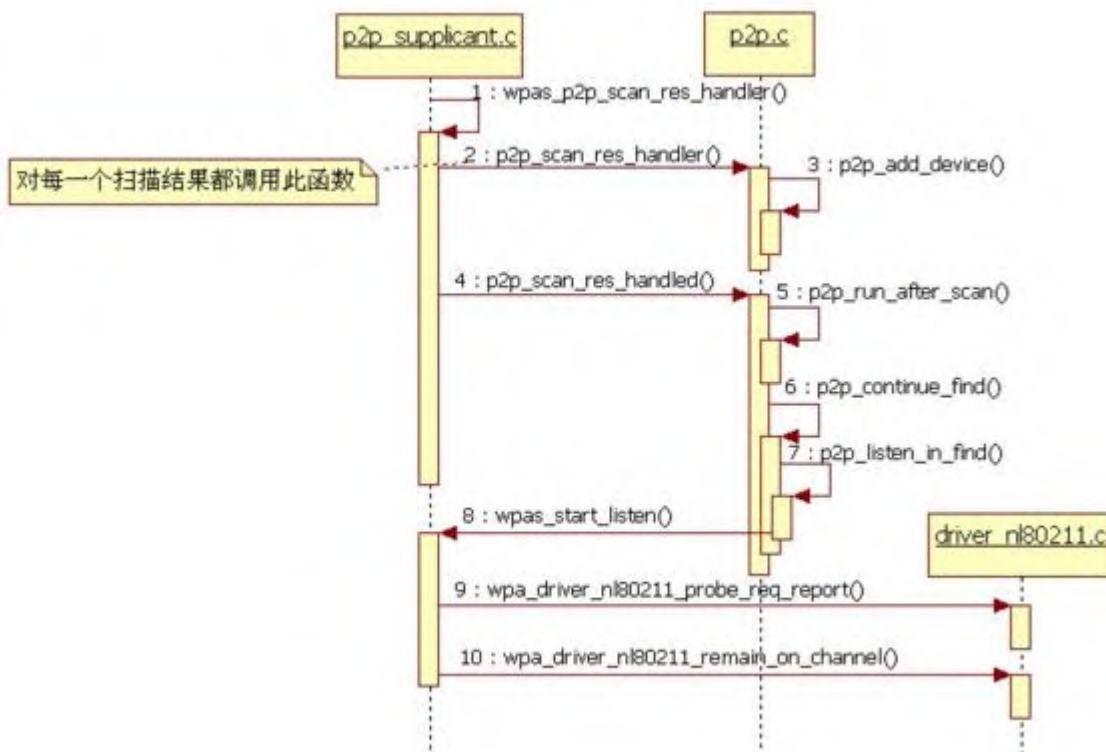


图7-33 P2P设备扫描结果处理流程

**注意** 图7-33中，只有满足一定条件，第6个及以后的函数才能执行。请读者结合p2p\_scan\_res\_handled的代码来加深对图7-32的体会。另外，由于Android平台中wpa\_driver\_set\_ap\_wps\_ie由厂商实现，故图7-33没有列出该函数。

设备找到以后，下一步工作就是发起Provision Discovery流程，马上来看它。

### 7.4.3 Provision Discovery流程分析

P2pStateMachine的ProvisionDiscoveryState在其EA中将发送形如“P2P\_PROV\_DISC 8a: 32: 9b: 6c: d1: 80 pbc”的命令给WPAS（请参考7.3.2节“CONNECT处理流程分析”）去执行，其核心处理函数是p2p\_ctrl\_prov\_disc，代码如下所示。

```
[-->ctrl_iface.c: : p2p_ctrl_prov_disc]

static int p2p_ctrl_prov_disc(struct wpa_supplicant *wpa_s,
char *cmd)
{
    u8 addr[ETH_ALEN];  char *pos;
    if (hwaddr_aton(cmd, addr)) return -1;
    .....// 参数处理。P2P_PROV_DISC命令的完整参数形式为"<addr>
<config method> [join]"
    // 其最后一个join参数为可选项。WifiP2pService没有使用它
    // 调用wpas_p2p_prov_disc，其内部将调用p2p_prov_disc_req，我们
    直接来看它
    return wpas_p2p_prov_disc(wpa_s, addr, pos, os_strstr(pos,
"join") != NULL);
}
```

#### 1. PD Request帧发送流程

p2p\_prov\_disc\_req的代码如下所示。

```
[-->p2p.c: : p2p_prov_disc_req]

int p2p_prov_disc_req(struct p2p_data *p2p, const u8
*peer_addr,
                      u16 config_methods, int join, int
force_freq)
{
    struct p2p_device *dev;

    dev = p2p_get_device(p2p, peer_addr); // 根据目标设备地址找到对
    应的p2p_device对象
    if (dev == NULL)
        dev = p2p_get_device_interface(p2p, peer_addr);
    .....
```

```

dev->wps_prov_info = 0;
dev->req_config_methods = config_methods;
// 就本例而言, join的值为0
if (join) dev->flags |= P2P_DEV_PD_FOR_JOIN;
else dev->flags &= ~P2P_DEV_PD_FOR_JOIN; // 取消dev->flags中的
的P2P_DEV_PD_FOR_JOIN标志
.....
p2p->user_initiated_pd = !join;

if (p2p->user_initiated_pd && p2p->state == P2P_IDLE)
    p2p->pd_retries = MAX_PROV_DISC_REQ_RETRIES;
// 最后调用p2p_send_prov_disc_req发送数据
return p2p_send_prov_disc_req(p2p, dev, join, force_freq);
}

```

p2p\_send\_prov\_disc\_req比较简单，代码如下。

```

[-->p2p_pd.c: : p2p_send_prov_disc_req]

int p2p_send_prov_disc_req(struct p2p_data *p2p, struct
p2p_device *dev,
                           int join, int force_freq)
{
    struct wpabuf *req; int freq;
    // 确定对端设备所在的工作频段
    if (force_freq > 0) freq = force_freq;
    else freq = dev->listen_freq > 0 ? dev->listen_freq : dev-
>oper_freq;
    .....
    dev->dialog_token++; // 还记得表7-3关于Dialog Token的描述吗
    if (dev->dialog_token == 0) dev->dialog_token = 1;
    // 构造Provision Discovery Request帧内容
    req = p2p_build_prov_disc_req(p2p, dev->dialog_token,
                                   dev->req_config_methods, join ? dev :
NULL);
    .....
    p2p->pending_action_state = P2P_PENDING_PD; // 该标志表明当前
pending的Action是PD
    // p2p_send_action内部将调用wpas_send_action。这部分内容比较简
单，请读者自行研究
    // 另外，第7.4.2节也会提到wpas_send_action函数，读者也可学习完本章
后再来研究它
    if (p2p_send_action(p2p, freq, dev->info.p2p_device_addr,
                        p2p->cfg->dev_addr, dev->info.p2p_device_addr,
                        wpabuf_head(req), wpabuf_len(req), 200) < 0)

```

```

{.....}
// 保存对端P2P设备地址
os_memcpy(p2p->pending_pd_devaddr, dev-
>info.p2p_device_addr, ETH_ALEN);

wpabuf_free(req);
return 0;
}

```

上述代码中，PD Request帧最终将通过p2p\_send\_action函数发送出去。不过，p2p\_send\_action并不简单，它将涉及Off Channel发送以及处理对应netlink消息的过程。做为本书Wi-Fi部分的最后一章，请读者在学习完本章后，再自行研究这个函数。

图7-34展示了PD Request帧发送流程中的重要函数调用序列。

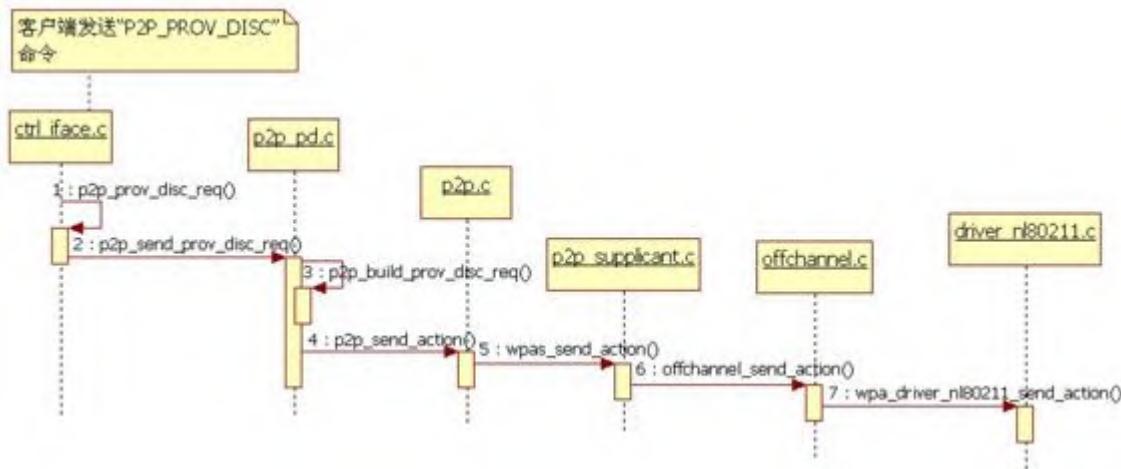


图7-34 PD Request帧发送流程

图7-34列出了wpas\_send\_action中的几个重要函数调用，请读者自行研究时候注意相关内容。

下面来看PD Response帧的处理流程。由于PD Response属于Action帧，所以我们将介绍WPAS中Action帧的接收流程，然后再分析PD Response的处理流程。

## 2. Action帧接收流程

PD Response帧属于Public Action帧的一种，而根据7.4.1节“注册Action帧监听事件”的分析可知，当收到对端设备发来的PD Response帧后，process\_bss\_event函数将被调用。此函数的代码如下所示。

```
[-->driver_nl80211.c: : process_bss_event]

static int process_bss_event(struct nl_msg *msg, void *arg)
{
    struct i802_bss *bss = arg; struct genlmsghdr *gnlh =
nlmsg_data(nlmsg_hdr(msg));
    struct nla_attr *tb[NL80211_ATTR_MAX + 1];

    nla_parse(tb, NL80211_ATTR_MAX, genlmsg_attrdata(gnlh, 0),
              genlmsg_attrlen(gnlh, 0), NULL);

    switch (gnlh->cmd) {
        case NL80211_CMD_FRAME:                      // 收到对端发送的帧
        case NL80211_CMD_FRAME_TX_STATUS:           // 对应本机所发送的管理帧的
TX Report
            mlme_event(bss->drv, gnlh->cmd, tb[NL80211_ATTR_FRAME],
                        tb[NL80211_ATTR_MAC],
                        tb[NL80211_ATTR_TIMED_OUT],
                        tb[NL80211_ATTR_WIPHY_FREQ],
                        tb[NL80211_ATTR_ACK],
                        tb[NL80211_ATTR_COOKIE]);
            break;
        .....
    }
    return NL_SKIP;
}
```

由上述代码可知，不论是代表由本机所发送的管理帧TX Report的NL80211\_CMD\_FRAME\_TX\_STATUS消息，还是代表本机接收到对端发来的管理帧事件的NL80211\_CMD\_FRAME消息，最终都会调用mlme\_event函数，其代码如下所示。

```
[-->driver_nl80211.c: : mlme_event]

static void mlme_event(struct wpa_driver_nl80211_data *drv,
enum nl80211_commands cmd,
                      struct nla_attr *frame, struct nla_attr *addr,
struct nla_attr *timed_out,
                      struct nla_attr *freq, struct nla_attr *ack, struct
nla_attr *cookie)
```

```

{
    .....
    switch (cmd) {
        case NL80211_CMD_AUTHENTICATE:
            mlme_event_auth(drv, nla_data(frame), nla_len(frame));
            break;
        case NL80211_CMD_ASSOCIATE:
            mlme_event_assoc(drv, nla_data(frame), nla_len(frame));
            break;
        .....
        case NL80211_CMD_FRAME:
            mlme_event_mgmt(drv, freq, nla_data(frame),
nla_len(frame));
            break;
        case NL80211_CMD_FRAME_TX_STATUS:
            mlme_event_mgmt_tx_status(drv, cookie, nla_data(frame),
nla_len(frame), ack);
        .....
    }
}

```

`mlme_event`将处理各种类型的帧事件。对于本例而言，此时将调用`mlme_event_mgmt`函数，其代码如下所示。

```

[-->driver_nl80211.c: : mlme_event_mgmt]

static void mlme_event_mgmt(struct wpa_driver_nl80211_data
*drv,
                           struct nlattr *freq, const u8 *frame, size_t
len)
{
    const struct ieee80211_mgmt *mgmt;
    union wpa_event_data event;
    u16 fc, stype;

    mgmt = (const struct ieee80211_mgmt *) frame;
    .....
    fc = le_to_host16(mgmt->frame_control);
    stype = WLAN_FC_GET_STYPE(fc);
    os_memset(&event, 0, sizeof(event));
    if (freq) {
        event.rx_action.freq = nla_get_u32(freq);
        drv->last_mgmt_freq = event.rx_action.freq;
    }
    if (stype == WLAN_FC_STYPE_ACTION) {
        event.rx_action.da = mgmt->da;

```

```

    event.rx_action.sa = mgmt->sa;
    event.rx_action.bssid = mgmt->bssid;
    event.rx_action.category = mgmt->u.action.category;
    event.rx_action.data = &mgmt->u.action.category + 1;
    event.rx_action.len = frame + len -
event.rx_action.data;
        // EVENT_RX_ACTION帧代表ACTION帧
    wpa_supplicant_event(drv->ctx, EVENT_RX_ACTION,
&event);
} else {
    event.rx_mgmt.frame = frame;
    event.rx_mgmt.frame_len = len;
    // EVENT_RX_MGMT代表其他类型的管理帧
    wpa_supplicant_event(drv->ctx, EVENT_RX_MGMT, &event);
}
}
}

```

wpa\_supplicant\_event处理EVENT\_RX\_ACTION的内容比较丰富，不过对于P2P来说，wpa\_supplicant\_event将调用wpas\_p2p\_rx\_action，而wpas\_p2p\_rx\_action又会调用p2p\_rx\_action，所以此处直接看p2p\_rx\_action函数即可，其代码如下所示。

[-->p2p.c: : p2p\_rx\_action]

```

void p2p_rx_action(struct p2p_data *p2p, const u8 *da, const u8
*sa,
                    const u8 *bssid, u8 category, const u8 *data, size_t
len, int freq)
{
    if (category == WLAN_ACTION_PUBLIC) { // 处理Public Action帧
        p2p_rx_action_public(p2p, da, sa, bssid, data, len,
freq);
        return;
    }
    .....// 参数检查
    switch (data[0]) { // P2P规范使用的其他非Public类型的Action帧
        case P2P_NOA:
            break;
        case P2P_PRESENCE_REQ:
            p2p_process_presence_req(p2p, da, sa, data + 1, len -
1, freq);
            break;
        case P2P_PRESENCE_RESP:
            p2p_process_presence_resp(p2p, da, sa, data + 1, len -
1);
    }
}

```

```

        break;
    case P2P_GO_DISC_REQ:
        p2p_process_go_disc_req(p2p, da, sa, data + 1, len - 1,
freq);
        break;
    .....
}
}

```

上述代码中专门处理Public Action帧的p2p\_rx\_action\_public函数代码如下所示。

```
[-->p2p.c: : p2p_rx_action_public]

static void p2p_rx_action_public(struct p2p_data *p2p, const u8
*da,
                                const u8 *sa, const u8 *bssid, const u8
*data, size_t len, int freq)
{
    .....
    switch (data[0]) {
    case WLAN_PA_VENDOR_SPECIFIC:// P2P Public Action满足此条件
        .....
        p2p_rx_p2p_action(p2p, sa, data + 1, len - 1, freq);
        break;
    case WLAN_PA_GAS_INITIAL_REQ:
        p2p_rx_gas_initial_req(p2p, sa, data + 1, len - 1,
freq);
        break;
    .... // 其他类型的Public Action帧处理
    }
}

```

p2p\_rx\_p2p\_action函数是P2P模块中Public Action帧得到分类处理的最后一关，其代码如下所示。

```
[-->p2p.c: : p2p_rx_p2p_action]

static void p2p_rx_p2p_action(struct p2p_data *p2p, const u8
*sa,
                                const u8 *data, size_t len, int rx_freq)
{
    switch (data[0]) { // P2P支持的Public Action帧在此处
得到分类和相应处理
    case P2P_GO_NEG_REQ: // 处理GON Request帧

```

```

        p2p_process_go_neg_req(p2p, sa, data + 1, len - 1,
rx_freq);
        break;
    case P2P_GO_NEG_RESP: // 处理GON Response帧
        p2p_process_go_neg_resp(p2p, sa, data + 1, len - 1,
rx_freq);
        break;
    case P2P_GO_NEG_CONF: // 处理GON Confirmation
帧
        p2p_process_go_neg_conf(p2p, sa, data + 1, len - 1);
        break;
    case P2P_INVITATION_REQ: // 处理Invitation Request帧
        p2p_process_invitation_req(p2p, sa, data + 1, len -
1, rx_freq);
        break;
    case P2P_INVITATION_RESP: // 处理Invitation Response帧
        p2p_process_invitation_resp(p2p, sa, data + 1, len -
1);
        break;
    case P2P_PROV_DISC_REQ: // 处理PD Request帧
        p2p_process_prov_disc_req(p2p, sa, data + 1, len - 1,
rx_freq);
        break;
    case P2P_PROV_DISC_RESP: // 处理PD Response帧
        p2p_process_prov_disc_resp(p2p, sa, data + 1, len - 1);
        break;
    case P2P_DEV_DISC_REQ: // 处理Device Discoverability
Request帧
        p2p_process_dev_disc_req(p2p, sa, data + 1, len - 1,
rx_freq);
        break;
    case P2P_DEV_DISC_RESP: // 处理Device Discoverability
Response帧
        p2p_process_dev_disc_resp(p2p, sa, data + 1, len - 1);
        break;
    .....// default语句
}
}

```

至此，Action帧的接收流程就介绍完了。整体而言，这部分代码难度不大，但是调用函数却比较多。图7-35总结了这部分流程所涉及的一些重要函数。

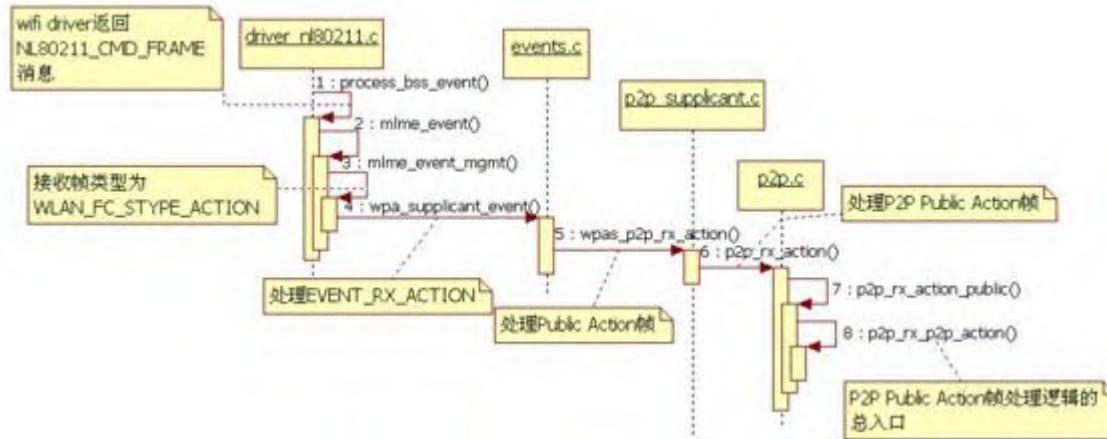


图7-35 Action帧接收流程

由图7-35可知，p2p\_rx\_p2p\_action为P2P Public Action帧处理逻辑的总入口，如果后文分析时碰到其他类型的P2P Public Action帧，我们将直接转入该函数来分析。

### 3. PD Response帧处理流程

由上述的p2p\_rx\_p2p\_action可知，PD Response帧对应的处理函数是p2p\_process\_prov\_disc\_resp，其代码如下所示。

[-->p2p\_pd.c : : p2p\_process\_prov\_disc\_resp]

```

void p2p_process_prov_disc_resp(struct p2p_data *p2p, const u8
*sa,
                                const u8 *data, size_t len)
{
    struct p2p_message msg;  struct p2p_device *dev; u16
report_config_methods = 0;
    // 解析PD Response帧
    if (p2p_parse(data, len, &msg)) return;
    // 获取对应的P2P Device对象
    dev = p2p_get_device(p2p, sa);
    .....
/*
    当前我们pending的action是PD，由于已经收到了PD Response，所以可以置
pending_action_state
    变量为P2P_NO_PENDING_ACTION。
*/
    if (p2p->pending_action_state == P2P_PENDING_PD) {
        os_memset(p2p->pending_pd_devaddr, 0, ETH_ALEN);
    }
}

```

```

    p2p->pending_action_state = P2P_NO_PENDING_ACTION;
}

if (dev->dialog_token != msg.dialog_token) return;

if (p2p->user_initiated_pd &&
    os_memcmp(p2p->pending_pd_devaddr, sa, ETH_ALEN) == 0)
    p2p_reset_pending_pd(p2p);
/*
如果所要求的WSC方法和PD Response返回的WSC方法不一致，则表明对端P2P
设备不支持所要求的WSC方法。
*/
if (msg.wps_config_methods != dev->req_config_methods) {
    // 调用wpas_prov_disc_fail，以处理PD失败的情况
    // 不过WPAS中，该函数没有干什么有意义的事情
    if (p2p->cfg->prov_disc_fail)
        p2p->cfg->prov_disc_fail(p2p->cfg->cb_ctx,
sa, P2P_PROV_DISC_REJECTED);
    p2p_parse_free(&msg);
    goto out;
}
report_config_methods = dev->req_config_methods;
dev->flags &= ~(P2P_DEV_PD_PEER_DISPLAY |
P2P_DEV_PD_PEER_KEYPAD);
.....
dev->wps_prov_info = msg.wps_config_methods;

p2p_parse_free(&msg);

out:
    dev->req_config_methods = 0;
    p2p->cfg->send_action_done(p2p->cfg->cb_ctx); // 请读者自行研
究send_action_done函数
    if (p2p->cfg->prov_disc_resp) // prov_disc_resp指向
wpas_prov_disc_resp
        p2p->cfg->prov_disc_resp(p2p->cfg->cb_ctx,
sa, report_config_methods);
}

```

马上来看wpas\_prov\_disc\_resp函数，其代码如下所示。

[-->p2p\_supplicant.c: : wpas\_prov\_disc\_resp]

```

void wpas_prov_disc_resp(void *ctx, const u8 *peer, u16
config_methods)
{

```

```

struct wpa_supplicant *wpa_s = ctx;
unsigned int generated_pin = 0;
/*
pending_pd_before_join变量对应于这样一种场景：即GON已经完成，但
WSC配置方法还没有确定。
在后文分析GON时，我们将见到这种场景。
*/
if (wpa_s->pending_pd_before_join &&
    (os_memcmp(peer, wpa_s->pending_join_dev_addr,
ETH_ALEN) == 0 || 
     os_memcmp(peer, wpa_s->pending_join_iface_addr,
ETH_ALEN) == 0)) {
    wpa_s->pending_pd_before_join = 0;
    wpas_p2p_join_start(wpa_s);
    return;
}

if (config_methods & WPS_CONFIG_DISPLAY)
    wpas_prov_disc_local_keypad(wpa_s, peer, "");
else if (config_methods & WPS_CONFIG_KEYPAD) {
    generated_pin = wps_generate_pin();
    wpas_prov_disc_local_display(wpa_s, peer, "", 
generated_pin);
} else if (config_methods & WPS_CONFIG_PUSHBUTTON)
    wpa_msg(wpa_s, MSG_INFO, P2P_EVENT_PROV_DISC_PBC_RESP
MACSTR, MAC2STR(peer));
.....
}

```

对于WSC PBC方法而言，`wpa_msg`将发送  
`P2P_EVENT_PROV_DISC_PBC_RESP`（字符串，值为“P2P-PROV-DISC-PBC-  
RESP”）消息给客户端，这也触发了7.3.2节分析  
`P2P_PROV_DISC_PBC_RSP_EVENT`处理流程中所描述的工作流程。

现在来看本章关于P2P的最后一到工序。

## 7.4.4 GO Negotiation流程分析

P2pStateMachine收到P2P\_PROV\_DISC\_PBC\_RSP\_EVENT消息后，将在ProvisionDiscoveryState中调用p2pConnectWithPinDisplay，该函数内部将发送P2P\_CONNECT命令给WPAS。马上来看该命令的处理流程。

### 1. P2P\_CONNECT处理流程

P2P\_CONNECT命令的参数比较多，而本例中P2pStateMachine发送的命令格式如下。

```
P2P_CONNECT 8a: 32: 9b: 6c: d1: 80 pbc go_intent=7
//其中，"8a: 32: 9b: 6c: d1: 80"代表对端P2P设备地址
//"pbc"指定了WSC配置方法为PBC， "go_intent=7"设置GO Intent值为7
```

P2P\_CONNECT对应的处理函数为p2p\_ctrl\_connect，其代码如下所示。

```
[-->ctrl_iface.c: : p2p_ctrl_connect]

static int p2p_ctrl_connect(struct wpa_supplicant*wpa_s,
                           char *cmd, char *buf, size_t
buflen)
{
    u8 addr[ETH_ALEN]; char *pos, *pos2;
    char *pin = NULL; enum p2p_wps_method wps_method;
    int new_pin; int ret; int persistent_group;
    int join; int auth;
    int go_intent = -1; int freq = 0;

    if (hwaddr_aton(cmd, addr)) return -1;
    .....// 参数处理，最终调用的函数为wpas_p2p_connect
    new_pin = wpas_p2p_connect(wpa_s, addr, pin, wps_method,
                               persistent_group, join, auth, go_intent, freq);
    .....
    os_memcpy(buf, "OK\n", 3);
    return 3;
}
```

wpas\_p2p\_connect的代码如下所示。

```
[-->p2p_supplicant.c: : wpas_p2p_connect]
```

```

int wpas_p2p_connect(struct wpa_supplicant *wpa_s, const u8
*peer_addr,
                      const char *pin, enum p2p_wps_method wps_method,
                      int persistent_group, int join, int auth, int
go_intent, int freq)
{
    int force_freq = 0, oper_freq = 0;
    u8 bssid[ETH_ALEN];
    int ret = 0;
    enum wpa_driver_if_type iftype;
    const u8 *if_addr;
    .....
    if (go_intent < 0) go_intent = wpa_s->conf->p2p_go_intent;
    wpa_s->p2p_wps_method = wps_method;

    wpa_s->p2p_pin[0] = '\0';
    // 本例中，由于GON还未完成，GO角色未能确定，所以join为0
    if (join) {.....}
    .....// 频段设置
/*
注意下面这个wpas_p2p_create_iface函数，它将判断是否需要创建一个新的
virtual interface，还记得
7.4.1节介绍Driver Flags和重要数据结构时提到的
WPA_DRIVER_FLAGS_P2P_MGMT_AND_NON_P2P
标志吗？就本例而言，wifi driver flags中包含了该标志，所以下面这个函
数的返回值为1，表示需要单独创建
一个新的virtual interface供P2P使用。这个virtual interface的地址
应该就是P2P Interface Address。
*/
    wpa_s->create_p2p_iface = wpas_p2p_create_iface(wpa_s);

    if (wpa_s->create_p2p_iface) {                                // 本例满足
此if条件
        iftype = WPA_IF_P2P_GROUP;                                // 设置
interface type
        if (go_intent == 15) iftype = WPA_IF_P2P_GO;           // 本例
的go_intent为7
/*
下面这个函数将创建此virtual interface，并获取其interface
address。以Galaxy Note 2
为例。P2P Device Address是“92:18:7c:69:88:e2”，而P2P
Interface Address是
“92:18:7c:69:08:e2”。
wpas_p2p_add_group_interface内部将调用driver_nl80211.c的
wpa_driver_nl80211_if_add

```

函数。感兴趣的读者不妨自行研究。

```

*/
    if (wpas_p2p_add_group_interface(wpa_s, iftype) < 0)
return -1;
    if_addr = wpa_s->pending_interface_addr;
} else
    if_addr = wpa_s->own_addr;
.....
// 下面这个函数内部将调用p2p_connect，我们将直接分析
if (wpas_p2p_start_go_neg(wpa_s, peer_addr, wps_method,
                           go_intent, if_addr,
force_freq,persistent_group) < 0) {.....}
    return ret;
}

```

## 2. GON Request发送流程

来看p2p\_connect函数，其代码如下所示。

[-->p2p.c: : p2p\_connect]

```

int p2p_connect(struct p2p_data *p2p, const u8 *peer_addr,
                  enum p2p_wps_method wps_method,
                  int go_intent, const u8
*own_interface_addr,
                  unsigned int force_freq,
                  int persistent_group)
{
    struct p2p_device *dev;
    // 如果指定了工作频段，则需要判断是否支持该工作频段
    if (p2p_prepare_channel(p2p, force_freq) < 0) return -1;

    p2p->ssid_set = 0;
    dev = p2p_get_device(p2p, peer_addr);
    .... // 设置dev的一些信息
    p2p->go_intent = go_intent;
    os_memcpy(p2p->intended_addr, own_interface_addr,
ETH_ALEN);
    // 如果P2P模块的状态不为P2P_IDLE，则先停止find工作
    if (p2p->state != P2P_IDLE) p2p_stop_find(p2p);
    ....
    dev->wps_method = wps_method;
    dev->status = P2P_SC_SUCCESS;
    ....
    if (p2p->p2p_scan_running) {

```

```

/*
如果当前P2P还在扫描过程中，则设置start_after_scan为
P2P_AFTER_SCAN_CONNECT标志，
当scan结束后，在扫描结果处理流程中，该标志将通知P2P进入connect
处理流程。
*/
p2p->start_after_scan = P2P_AFTER_SCAN_CONNECT;
os_memcpy(p2p->after_scan_peer, peer_addr, ETH_ALEN);
return 0;
}
p2p->start_after_scan = P2P_AFTER_SCAN_NOTHING;
// 下面这个函数将发送GON Request帧，直接来看该函数
return p2p_connect_send(p2p, dev);
}

```

[-->p2p\_go\_neg.c: : p2p\_connect\_send]

```

int p2p_connect_send(struct p2p_data *p2p, struct p2p_device
*dev)
{
    struct wpabuf *req;  int freq;
    .....
    req = p2p_build_go_neg_req(p2p, dev);
    p2p_set_state(p2p, P2P_CONNECT);      // 设置P2P模块的状态为
P2P_CONNECT
    // 设置pending_action_state为P2P_PENDING_GO_NEG_REQUEST
    p2p->pending_action_state = P2P_PENDING_GO_NEG_REQUEST;
    p2p->go_neg_peer = dev;              // 设置GON对端设备
    dev->flags |= P2P_DEV_WAIT_GO_NEG_RESPONSE;
    dev->connect_reqs++;
#ifdef ANDROID_P2P
    dev->go_neg_req_sent++;
#endif
    // 发送GON Request帧
    if (p2p_send_action(p2p, freq, dev->info.p2p_device_addr,
                        p2p->cfg->dev_addr, dev->info.p2p_device_addr,
                        wpabuf_head(req), wpabuf_len(req), 200) < 0)
{.....}
.....
return 0;
}

```

至此，GON Request帧就将发送给对端P2P设备。图7-36描述了P2P\_CONNECT命令的处理流程。

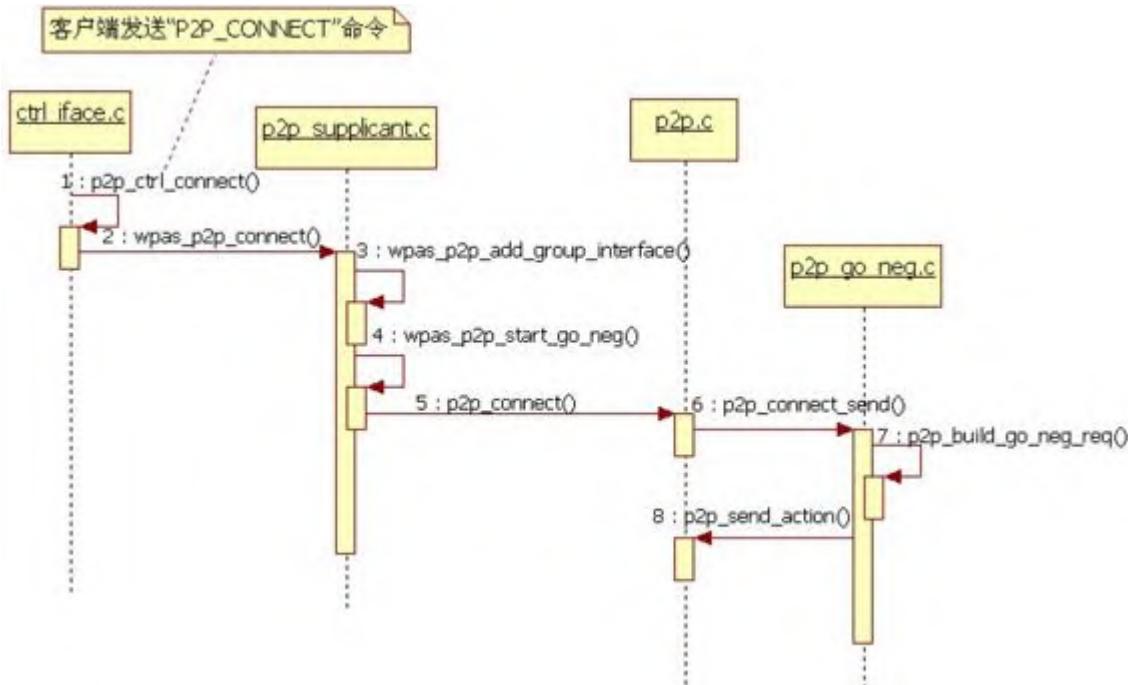


图7-36 P2P\_CONNECT命令处理流程

### 3. GON Response帧处理流程

根据前面对Action帧接收流程的分析可知，收到的GON Response帧将在p2p\_process\_go\_neg\_resp函数中被处理。该函数的代码如下所示。

```

[-->p2p_go_neg.c: : p2p_process_go_neg_resp]

void p2p_process_go_neg_resp(struct p2p_data *p2p, const u8
*sa,
                           const u8 *data, size_t len, int rx_freq)
{
    struct p2p_device *dev; struct wpabuf *conf; int go = -1;
    struct p2p_message msg; u8 status = P2P_SC_SUCCESS;
    int freq;
    dev = p2p_get_device(p2p, sa);
    // 解析GON Response帧
    if (p2p_parse(data, len, &msg)) return;
    .....// 参数检查
/*
下面这个函数将利用图7-13计算谁来扮演GO。返回值大于0，表示本机扮演GO，  

返回-1表示双方都想成为GO，返回值为0，表示对端扮演GO。本例中，假设GO为  

本机设备。
*/
  
```

```

go = p2p_go_det(p2p->go_intent, *msg.go_intent);
.....// 参数检查，内容比较繁杂，感兴趣的读者请自行研究
if (go) {.....// 处理工作频段}

p2p_set_state(p2p, P2P_GO_NEG); // 设置P2P模块的状态为
P2P_GO_NEG
p2p_clear_timeout(p2p);
.....
fail:
    // 构造GON Confirmation帧
    conf = p2p_build_go_neg_conf(p2p, dev, msg.dialog_token,
status,
                                msg.operating_channel, go);
    p2p_parse_free(&msg);
    .....
    if (status == P2P_SC_SUCCESS) {
        p2p->pending_action_state = P2P_PENDING_GO_NEG_CONFIRM;
        dev->go_state = go ? LOCAL_GO : REMOTE_GO; // 本机扮演GO
    } else
        p2p->pending_action_state = P2P_NO_PENDING_ACTION;
    .....
    // 发送GON Confirmation帧
    if (p2p_send_action(p2p, freq, sa, p2p->cfg->dev_addr, sa,
wpabuf_head(conf), wpabuf_len(conf), 200) < 0)
{.....}
    wpabuf_free(conf);
}

```

p2p\_process\_go\_neg\_resp实际的代码比较复杂，建议初学者先了解上面代码所涉及的大体流程。

当GON Confirmation帧发送出去后，wifi driver将向WPAS发送一个NL80211\_CMD\_FRAME\_TX\_STATUS消息，而该消息将导致driver wrapper发送EVENT\_TX\_STATUS消息给WPAS。下面我们直接来看EVENT\_TX\_STATUS的处理流程。

#### 4. EVENT\_TX\_STATUS处理流程

在events.c中，和P2P以及EVENT\_TX\_STATUS相关的处理函数是offchannel\_send\_action\_tx\_status，该函数的代码如下所示。

[-->offchannel.c: : offchannel\_send\_action\_tx\_status]

```

void offchannel_send_action_tx_status(struct wpa_supplicant
*wpa_s, const u8 *dst,
        const u8 *data, size_t data_len, enum
offchannel_send_action_result result)
{
    .....// 参数检查
    wpabuf_free(wpa_s->pending_action_tx);
    wpa_s->pending_action_tx = NULL;
    /*
    注意下面这个pending_action_tx_status_cb参数，它是一个函数指针，
P2P每次发送Action的时候，
    都会设置该变量。其真实的函数为wpas_p2p_send_action_tx_status(在
wpas_send_action
    函数中设置)
    */
    if (wpa_s->pending_action_tx_status_cb) {
        wpa_s->pending_action_tx_status_cb( wpa_s, wpa_s-
>pending_action_freq,
                wpa_s->pending_action_dst, wpa_s-
>pending_action_src,
                wpa_s->pending_action_bssid, data, data_len,
result);
    }
}

```

来看wpas\_p2p\_send\_action\_tx\_status，其代码如下所示。

```

[-->p2p_supplicant.c: : wpas_p2p_send_action_tx_status]

static void wpas_p2p_send_action_tx_status(struct
wpa_supplicant *wpa_s,
        unsigned int freq, const u8 *dst, const u8 *src,
const u8 *bssid,
        const u8 *data, size_t data_len, enum
offchannel_send_action_result result)
{
    enum p2p_send_action_result res = P2P_SEND_ACTION_SUCCESS;
    .....
    // 重要函数: p2p_send_action_cb
    p2p_send_action_cb(wpa_s->global->p2p, freq, dst, src,
bssid, res);
    .....
}

```

p2p\_send\_action\_cb的代码如下所示。

```

[-->p2p.c: : p2p_send_action_cb]

void p2p_send_action_cb(struct p2p_data *p2p, unsigned int
freq, const u8 *dst,
                      const u8 *src, const u8 *bssid, enum
p2p_send_action_result result)
{
    enum p2p_pending_action_state state;
    int success;
    success = result == P2P_SEND_ACTION_SUCCESS;
    // 读者还记得该变量的值吗？它应该是P2P_PENDING_GO_NEG_CONFIRM
    state = p2p->pending_action_state;

    p2p->pending_action_state = P2P_NO_PENDING_ACTION;
    switch (state) {
        case P2P_NO_PENDING_ACTION:
            break;
        case P2P_PENDING_GO_NEG_REQUEST:
            // 读者可自行研究此函数。当发送完GON Request帧后，此函数也会被
触发生效
            p2p_go_neg_req_cb(p2p, success);
            break;
        .....
        case P2P_PENDING_GO_NEG_CONFIRM:
            p2p_go_neg_conf_cb(p2p, result); // 分析这个函数
            break;
        .... // 其他case处理
    }
}

```

来看p2p\_go\_neg\_conf\_cb函数，代码如下所示。

```

[-->p2p.c: : p2p_go_neg_conf_cb]

static void p2p_go_neg_conf_cb(struct p2p_data *p2p,
                               enum p2p_send_action_result result)
{
    struct p2p_device *dev;
    /*
    每次收到TX Report后，都需要调用send_action_cb，其对应的真实函数是
wpas_send_action_done。
    */
    p2p->cfg->send_action_done(p2p->cfg->cb_ctx);
    .....
    dev = p2p->go_neg_peer;

```

```

    if (dev == NULL)    return;
    p2p_go_complete(p2p, dev);
}

```

p2p\_go\_complete函数比较简单，其代码如下所示。

[-->p2p.c: : p2p\_go\_complete]

```

void p2p_go_complete(struct p2p_data *p2p, struct p2p_device
*peer)
{
    struct p2p_go_neg_results res; int go = peer->go_state ==
LOCAL_GO;
    struct p2p_channels intersection; int freqs;
    size_t i, j;
    .....// 设置频段等参数

    p2p_set_state(p2p, P2P_PROVISIONING); // 进入Group Formation
的Provisioning阶段
    // go_neg_completed对应的函数是wpas_go_neg_completed
    p2p->cfg->go_neg_completed(p2p->cfg->cb_ctx, &res);
}

```

[-->p2p\_supplicant.c: : wpas\_go\_neg\_compeleted]

```

void wpas_go_neg_completed(void *ctx, struct p2p_go_neg_results
*res)
{
    struct wpa_supplicant *wpa_s = ctx;
    .....
    // 下面这个函数将导致P2pStateMachine收到
P2P_GO_NEGOTIATION_SUCCESS_EVENT消息
    wpa_msg(wpa_s, MSG_INFO, P2P_EVENT_GO_NEG_SUCCESS);
    wpas_notify_p2p_go_neg_completed(wpa_s, res);

    if (wpa_s->create_p2p_iface) { // 本例满足此if条件
        /*
        再创建一个wpa_supplicant对象。感兴趣的读者可自行研究下面这个函
数。其内部将调用4.4.3节
        分析的wpa_supplicant_add_iface函数。
        */
        struct wpa_supplicant *group_wpa_s =
            wpas_p2p_init_group_interface(wpa_s, res->role_go);
        .....
    }
}

```

```

        // 如果本机扮演GO，则启动WSC Registrar功能，否则启动
Enrollee功能
        // 下面这两个函数留给读者自行分析
        if (res->role_go)  wpas_start_wps_go(group_wpa_s, res,
1);
        else    wpas_start_wps_enrollee(group_wpa_s, res);
    } .....
wpa_s->p2p_long_listen = 0;
eloop_cancel_timeout(wpas_p2p_long_listen_timeout, wpa_s,
NULL);
eloop_cancel_timeout(wpas_p2p_group_formation_timeout,
wpa_s, NULL);
        // Group Formation的超时时间为15秒左右
        eloop_register_timeout(15 + res->peer_config_timeout / 100,
(res->peer_config_timeout % 100) * 10000,
wpas_p2p_group_formation_timeout, wpa_s,
NULL);
}

```

当Group Negotiation完成后，WPAS将新创建一个wpa\_supplicant对象，它将用于管理和操作专门用于P2P Group的virtual interface，此处有几点请读者注意。

- 4.3.1节中介绍过，一个interface对应一个wpa\_supplicant对象。
- 此处新创建的wpa\_supplicant对象用于GO，即扮演AP的角色，专门处理和P2P Group相关的事情，其MAC地址为P2P Interface Address。
- 之前使用的wpa\_supplicant用于非P2P Group操作，其MAC地址为P2P Device Address。

至此，整个EVENT\_TX\_STATUS处理流程就分析完毕了，其内容比较复杂。图7-37整理了此过程中一些重要的函数调用。

注意，图7-37实际上描述的是GON Confirmation帧对应的EVENT\_TX\_STATUS处理流程，和其他EVENT\_TX\_STATUS处理流程相比，前面7个函数调用都是一样的。

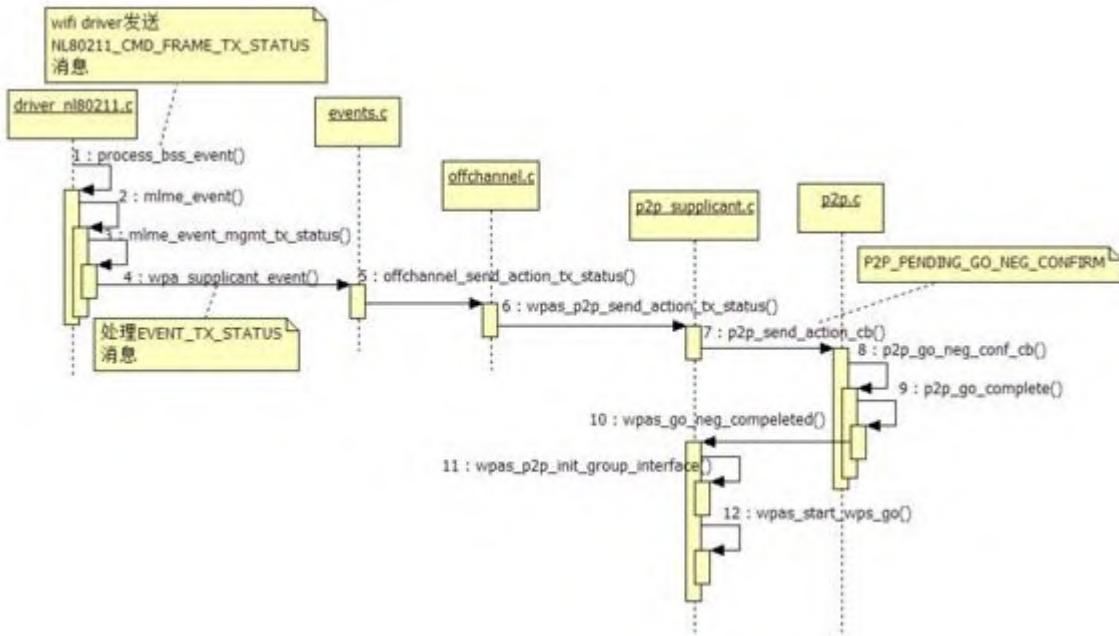


图7-37 EVENT\_TX\_STATUS处理流程

## 7.5 本章总结和参考资料说明

### 7.5.1 本章总结

本章对Wi-Fi P2P进行了详细介绍，主要内容如下。

- P2P理论知识。从完整性来说，本章介绍的内容只是P2P规范中最基础的部分。但对于初学者而言，这部分的难度也不算小。就笔者的学习经历而言，这部分内容需要反复琢磨和研究，有时候还需要结合代码分析才能真正掌握其精髓。
- 在学习完P2P理论知识后，对WifiP2pSettings以及WifiP2pService进行了介绍。这部分内容比较简单，读者可轻松掌握相关知识。
- 最后，对WPAS中的P2P模块及运行机制进行了介绍。就所涉及的知识而言，这些内容并不复杂，但由于P2P以及和wifi driver在具体实现时有诸多考虑（例如off channel的情况，TX Report的处理），所以其工作流程反倒显得比较烦琐。

最后，希望读者在本章的基础上，完成下列任务。

- 通读P2P规范，了解Group Operation、Invitation、Device Discoverability以及P2P Power Management相关知识。
- 继续研究wpas\_start\_wps\_go代码，掌握WPAS中WSC Registrar以及AP的工作流程。

## 7.5.2 参考资料说明

### 1. P2P基础知识介绍

本章参考资料主要是Wi-Fi P2P规范1.1版，可从  
<http://www.doc88.com/p-908280242988.html>下载协议全文。

### 2. P2P架构介绍

- [1] "Wi-Fi P2P"的第2章"Architectural Overview"
- [2] "Wi-Fi P2P"的3.2节"P2P Group Operation"
- [3] "Wi-Fi P2P"的3.3节"P2P Power Management"

### 3. P2P Device Discovery和Group Formation

- [4] "Wi-Fi P2P"的3.1节"P2P Discovery"
- [5] Part11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications附录J"Country information element and regulatory classes"

说明：该文档下载地址为  
[http://download.csdn.net/download/s\\_bird0529/2553723](http://download.csdn.net/download/s_bird0529/2553723)。附录J详细描述了国家码和管制信息方面的内容。

- [6] "Wi-Fi P2P"的4.2节"Management Frames"
- [7] "802.11-2012"的8.4.1.11节"Action Field"

说明：该节介绍了Action帧Category字段的取值情况。

- [8] "Wi-Fi P2P"的4.1节"P2P Information Elements"
- [9] "Wi-Fi P2P"附录A"P2P State Machine"

# 第8章 深入理解NFC

本章所涉及的源代码文件名及位置

- ForegroundDispatch. java  
development/samples/ApiDemos/src/com/example/android/apis/nfc/  
/ForegroundDispatch. java
- Beam. java  
development/samples/AndroidBeamDemo/src/com/example/android/b  
eam/Beam. java
- NfcService. java  
packages/apps/Nfc/src/com/android/nfc/NfcService. java
- P2pLinkManager. java  
packages/apps/Nfc/src/com/android/nfc/P2pLinkManager. java
- NativeNfcManager. java  
packages/apps/Nfc/nxp/src/com/android/nfc/dhimpl/NativeNfcMan  
ager. java
- P2pEventManager. java  
packages/apps/Nfc/src/com/android/nfc/P2pEventManager. java
- SnepServer. java  
packages/apps/Nfc/src/com/android/nfc/SnepServer. java
- NfcDispatcher. java  
packages/apps/Nfc/src/com/android/nfc/NfcDispatcher. java
- SendUi. java  
packages/apps/Nfc/src/com/android/nfc/SendUi. java
- SnepClient. java  
packages/apps/Nfc/src/com/android/nfc/SnepClient. java

读累了记得休息一会儿哦~

公众号：古德猫宁李

- 电子书搜索下载
- 书单分享
- 书友学习交流

网站：[沉金书屋 https://www.chenjin5.com](https://www.chenjin5.com)

- 电子书搜索下载
- 电子书打包资源分享
- 学习资源分享

## 8.1 概述

NFC（Near Field Communication，近场通信）也叫做近距离无线通信技术。该技术最早由Philips和Sony两家公司于2002年末联合推出。2004年，Nokia、Philips、Sony等公司还共同组建了一个名为NFC Forum的非盈利性组织来推广和发展NFC技术。NFC Forum的职责和Wi-Fi Alliance类似，它不但负责制定NFC相关的技术标准，同时还通过NFC认证测试<sup>①</sup>来保证各厂家的NFC产品符合NFC规范。

从原理上说，NFC和Wi-Fi类似，二者都利用无线射频技术来实现设备之间的通信。不过，和Wi-Fi相比，NFC的工作频率为13.56MHz，有效距离为4cm左右，目前所支持的数据传输速率有106kbps、212kbps和424kbps三种。

**提示** 通过NFC无线射频参数的介绍可知，NFC所针对的应用场景和Wi-Fi明显不同。以NFC有效距离为4cm为例，这么短的有效距离本身就要求交互双方必须有某种程度的相互信任。否则，一个用户不会随便让另外一个用户的设备这么靠近自己的设备。NFC还有其他非常多广泛的应用场景，感兴趣的读者请阅读参考资料[1]。

NFC技术从创建到现在已超过10年，在技术层面上已相当完善。但NFC至今未能像Wi-Fi一样被普及，其中一个重要原因就是大众消费者没有一个合适的载体来使用它。显然，随着越来越多携带NFC功能的Android智能终端的出现，NFC这种有价无市的状况有望很快得以改善。

**提示** 很多专家预测2014年或2015年是NFC技术推广和普及的元年。但奇怪的是iPhone却迟迟没有支持NFC，这不免给它的前景蒙上了一层阴影。不过，最近有消息称苹果秘密申请了一项和NFC相关的专利。

本章将从以下几个方面来介绍NFC以及它在Android平台中的实现。

- 首先介绍NFC基础知识，这是本章的核心内容。相对Wi-Fi而言，本章介绍的NFC理论知识相对比较简单，相信读者能轻松掌握。

- 然后介绍Android平台中NFC实现，这部分内容包括NFC客户端示例以及NFC系统模块。
- 最后探讨目前一些开源NFC相关模块的实现情况。

先来看NFC基础知识。

① 关于NFC认证测试，请参考<http://www.nfc-forum.org/certification/certification-testing/>。

## 8.2 NFC基础知识

简而言之，NFC是从多种不同技术基础上综合发展而来，图8-1展示了NFC技术的演化历程。

### 8.2.1 NFC概述

通过图8-1所示的NFC技术演化历程可知，NFC融合了三条主要的技术发展路线<sup>[2]</sup>。

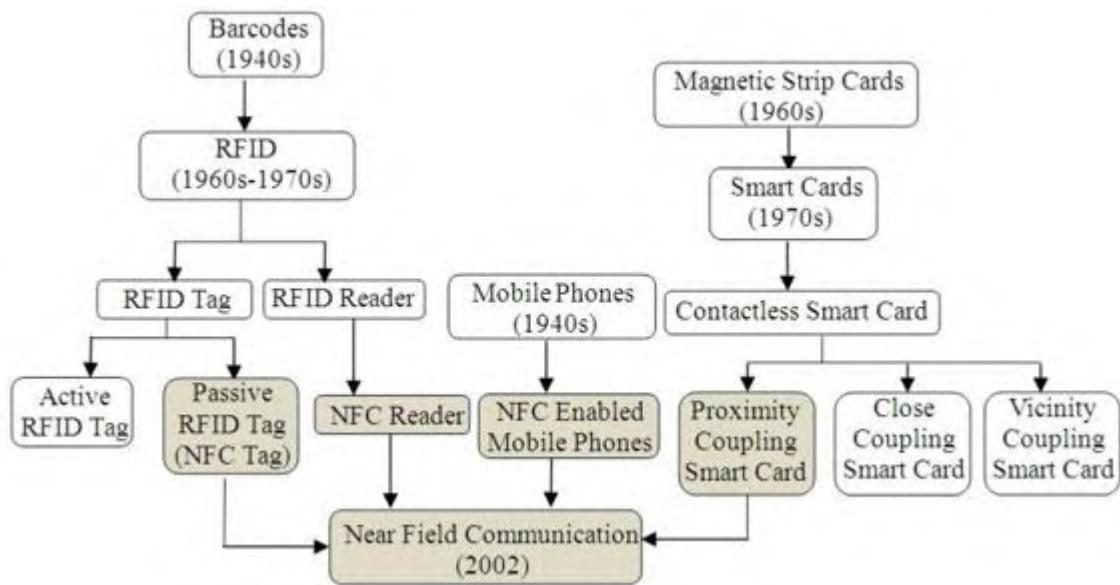


图8-1 NFC技术演化历程

• RFID技术路线，即无线射频识别技术（图左边）。该技术路线发源于条形码（Barcodes），然后发展出了RFID，最终出现了NFC中的两个重要组件NFC Tag（标签）和NFC Reader。NFC Tag的作用和Barcodes类似，它是一种用于存储数据的被动式（Passive）RFID Tag，其最重要的特征就是NFC Tag自身不包含电源组件，所以它工作时必须依靠其他设备（比如NFC Reader）通过电磁感应的方式向其输送电能。和NFC Tag相对应的组件是NFC Reader，它首先通过电磁感应向NFC Tag输送电能使其工作，然后根据相关的无线射频通信协议来存取NFC Tag上的数据。

• 磁条卡（Magnetic Strip Cards）技术路线（图右边）。该路线最终演化了NFC使用的Proximity Coupling Smart Card技术（有效距离为10cm，对应的规范为ISO/IEC 14443。注意，图中的Close Coupling Smart Card的有效距离为1cm，对应的规范为ISO/IEC 10536。Vicinity Coupling Smart Card的有效距离为1m，对应的规范为

ISO/IEC 15693）。粗略来看Smart Card和RFID Tag类似，例如二者都只存储一些数据，而且自身都没有电源组件，但Smart Card在安全性上的要求远比RFID Tag严格。另外，Smart Card上还能运行一些小的嵌入式系统（如Java Card OS）或者应用程序（Applets）以完成更为复杂的工作。

- 移动终端线路，演化了携带NFC功能的终端设备（图中间）。随着移动终端越来越智能，NFC和这些设备也融合得更加紧密，使得NFC的应用场景得到了较大的拓展。本书第6章在介绍Wi-Fi Simple Configuration时（6.1节）曾介绍过一个例子，即智能手机可通过NFC来和AP交换安全配置信息。一个与之类似的例子是NFC Connection Handover技术，它描述了两台智能终端如何通过NFC相关协议来选择合适的数据传输方式（例如Bluetooth或Wi-Fi，受限于传输速率以及有效距离，NFC本身不适合大数据量传输）。

了解NFC技术的演化历程之后，我们来看看NFC现在的样子。图8-2所示为NFC技术框架<sup>[3]</sup>。

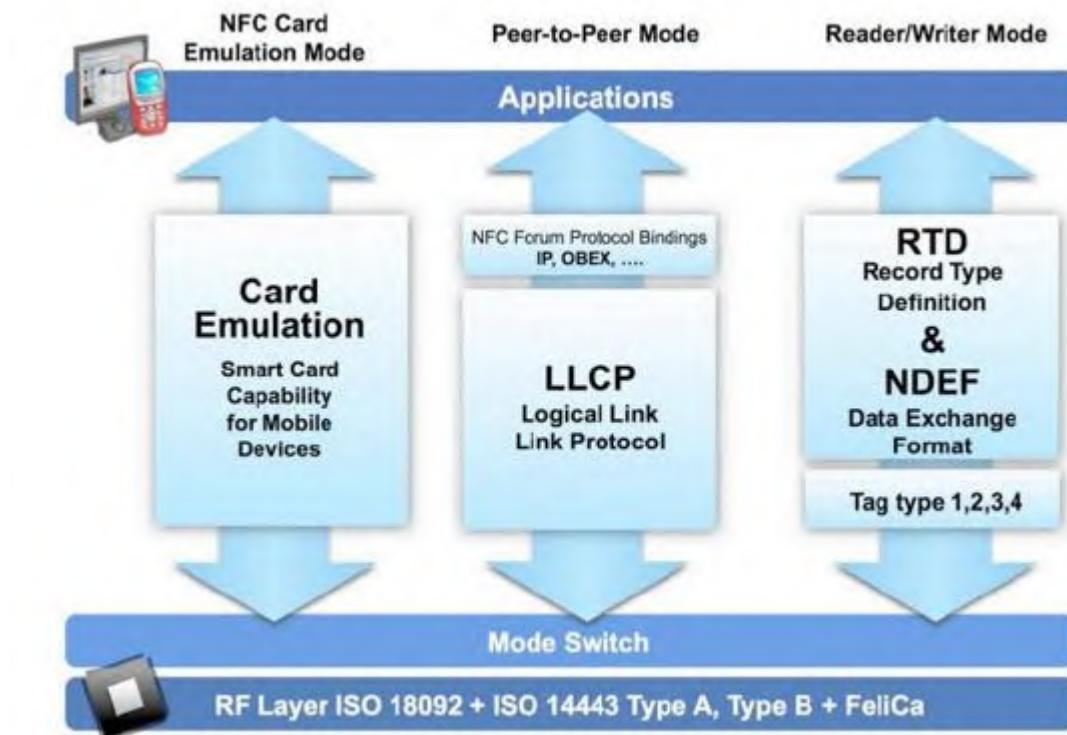


图8-2 NFC技术框架

由图8-2可知，从用户角度（即图中的Applications层之上）来看，NFC有三种运行模式（operation mode）。

- Reader/Write模式：简称R/W，和NFC Tag/NFC Reader相关。
- Peer-to-Peer模式：简称P2P，它支持两个NFC设备交互。
- NFC Card Emulation模式：简称CE，它能把携带NFC功能的设备模拟成Smart Card，这样就能实现诸如手机支付、门禁卡之类的功能。

Application之下的三个箭头描述了三种运行模式所使用的协议栈。这部分内容将留待下文分析。

NFC使用的是无线射频技术。在RF层，与之相关的规范是ISO 18092（NFC Interface and Protocol I，简称NFCIP-1，该规范定义了NFC RF层的工作流程）和ISO 14443 Type A、Type B，以及Felica。

ISO 14443全称为非接触式IC卡标准，它从RF层面定义了如何与不同的非接触式IC卡（其实物可以是NFC Tag、RFID Tag、Smart Cards）交互。ISO 14443定义了Type A和Type B两种非接触式IC卡。

- Type A最早由Philips公司制订（其生产的芯片商标名为MIFARE，现在由从Philips独立出来的NXP公司拥有，目前世界上70%左右的非接触式IC卡都使用了MIFARE芯片，例如北京市的公交卡）。
- Type B（主要用在法国市场）由其他公司制订，二者最终都成为ISO标准。
- Felica（也称为Type F）由Sony开发，它最终没有成为ISO标准，而成为日本工业标准JIS X6319-4，所以Felica主要用于日本市场。

Type A、B和F主要区别在于RF层的信号调制解调方法、传输速率及数据编码方式上。关于ISO 14443和Felica之间的区别，请读者阅读参考资料[4]。

RF层之上是Mode Switch，用于确定对端NFC Device的类型并选择合适的RF层协议与之通信。

**提示** 由于NFC是从多种技术综合发展而来，所以读者在学习NFC时将会碰到很多规范，如上文所提到的ISO 18092以及ISO 14443、Felica等。除了ISO等标准组织制定的规范外，NFC Forum也制定了一系列的标准和规范。由于篇幅问题，本章仅介绍NFC Forum定义的一些规范。对ISO相关规范感兴趣的读者可在本章基础之上自行阅读。

图8-3所示为与NFC技术框架相对应的NFC Forum所定义的规范框架<sup>[3]</sup>。

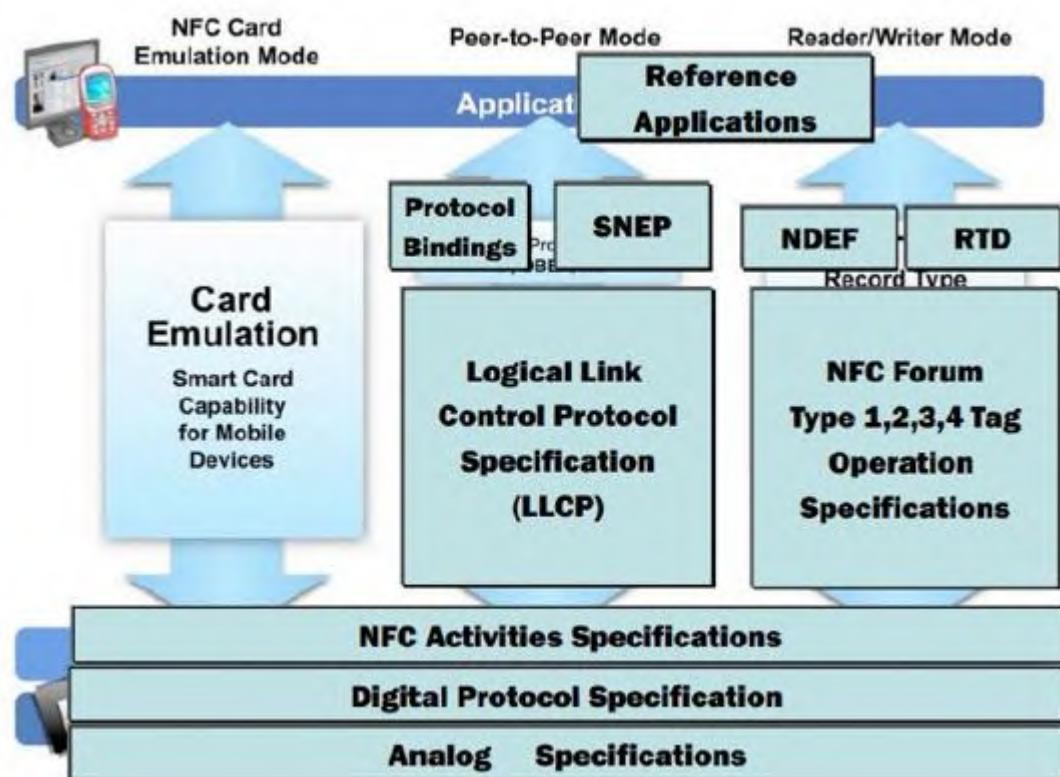


图8-3 NFC Forum规范框架

由图8-3所示的NFC Forum规范框架可知，NFC Forum本身只定义了P2P模式和R/W模式相关的规范，规范的细节将留待下文详细介绍。CE模式比较复杂，下文也会讨论和它相关的一些知识。

在RF层，NFC Forum定义了三个主要规范。

- **Analog Specifications:** 该规范描述了NFC设备RF层的电气特性。

- Digital Protocol Specification: 该规范在ISO 18092、ISO 14443及JIS X6319-4之上定义了NFC设备之间的数字通信协议，它使得基于不同底层协议例如Type A或Type F的NFC设备之间或者NFC设备与其他使用ISO 18092等规范的设备之间能够交互。
- NFC Activities Specification: 该规范为各运行模式对应的协议栈提供支持，例如P2P模式下两个NFC设备如何建立链接，R/W模式下NFC Device如何操作NFC Tag。

图8-3最上层的Reference Applications表示NFC Forum在应用层面所定义的一些规范。目前有两个规范。

- Connection Handover: 两个NFC设备通过它来协商用蓝牙或Wi-Fi来开展后续的数据传输工作。
- Personal Health Device Communication: 该规范定义了如何利用NFC技术在个人健康设备之间交换数据信息。

另外，除了图8-3所示的规范外，NFC还制定了一个NCI（NFC Controller Interface）规范，该规范制定了一套交互接口，使得主机设备（Device Host，以手机为例，NFC芯片被集成到某个手机中，那么手机就是Device Host）能够使用这套接口来和NFC芯片交互。

下面，先讨论NFC三种运行模式，而NCI相关知识将留待本节最后介绍。

提示 关于NFC Forum制定的各种规范及简要说明见参考资料[5]。

## 8.2.2 NFC R/W运行模式

以支持NFC功能的智能终端为例，NFC R/W运行模式所包含的组件如图8-4<sup>[6]</sup>所示。

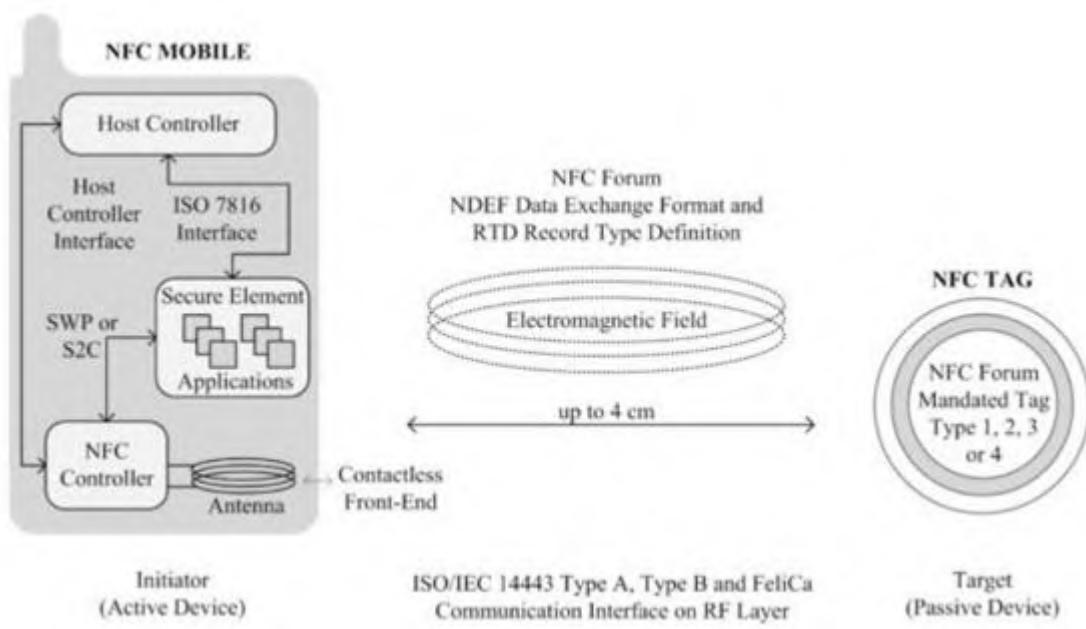


图8-4 R/W运行模式组件

图8-4展示了一个包含NFC芯片的智能终端与NFC Tag交互所涉及的组件。

- 左边的智能终端扮演NFC Reader角色。位于其内部的NFC芯片包含NFC Controller（NFC控制器，它可与Device Host或Secure Element安全单元交互）、Antenna（天线）和Contactless Front-End（CLF，非接触式前端，负责射频信号的调制解调等工作）三个部分。注意，图中所示的SWP等内容将在8.2.4节介绍。
- 在R/W模式中，交互操作的发起方只能是NFC Reader，因此它也称为Initiator或Active Device。
- 右边的NFC Tag，由于需要NFC Reader通过电磁感应为其提供电能，所以在R/W模式中，NFC Tag只能作为交互操作的Target（也称为

Passive Device)。

NFC Forum定义了四种类型的Tag，分别为Type 1、Type 2、Type 3和Type 4。这四种类型NFC Tag的区别在于存储空间大小、数据传输率以及底层使用的协议。表8-1列举了它们的不同点。

NFC Forum定义了两个通用的数据结构用于NFC Device之间（包括R/W模式中的NFC Reader和NFC Tag）传递数据。这两个通用数据结构分别是NFC Data Exchange Format（NDEF）以及NFC Record。

我们先来看NFC四种不同类型的Tag有何区别，如表8-1<sup>[7]</sup> 所示。

表 8-1 NFC Tag Type 说明

参数	Type 1	Type 2	Type 3	Type 4
对应规范	ISO 14443 Type A	ISO 14443 Type A	FeliCa	ISO 14443 Type A, Type B
常见芯片名	Topaz	MIFARE	FeliCa	MIFARE-DESFire
存储容量	最大 1KB	最大 2KB	最大 1MB	最大 64KB
读写速率	106kbps	106kbps	212kbps	106 ~ 424kbps
价格	低	低	高	中等 / 高
安全性	数字签名保护	不安全	数字签名保护	可选
说明	Topaz 由 Innovision 公司推出	MIFARE 由 NXP 公司推出	由 Sony 公司推出，价格比较高	这类芯片在出厂时就被配置好是否只读或可读写

注意 这里需要特别指出的是：虽然NFC Froum只有四种类型的Tag，但由于NFC本身源自RFID技术，二者在一些底层协议上也相互兼容，所以很多RFID Tag也能被NFC Reader识别和操作。为了书写方便，除非特别说明，本章所指的NFC Tag也包括那些和NFC相关规范兼容的RFID Tag。

虽然NFC Tag有四种不同类型（由上文可知，实际上能被NFC Reader读写的RFID Tag还远不止四种），但为了保证最大兼容性，NFC Forum建议NFC设备之间尽量使用通用数据结构NDEF和NFC Record来交换信息。

NFC R/W模式涉及的规范比较多，包括：

- NFC Reader如何与不同类型的Tag交互，这部分内容涉及非常底层的一些协议。
- NDEF和一些常用数据类型定义。

出于篇幅和实用性考虑，本书仅介绍NDEF和相关的数据类型，感兴趣的读者可自行研究NFC Reader和Tag之间的交互协议。

## 1. NDEF和NFC Record

### (1) NDEF和NFC Record<sup>[8][9]</sup> 之间的关系

根据NFC Forum的定义，R/W模式下，NFC设备之间每一次交互的数据都会封装在一个NDEF Message中，而一个NDEF Message可以包含多个NFC Record，真正的数据则封装在NFC Record中。图8-5展示了NDEF Message和NFC Record之间的关系。

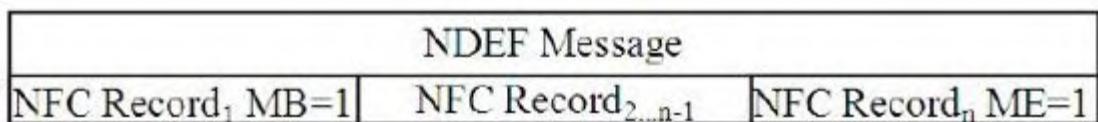


图8-5 NDEF Message和NFC Record的关系

由图8-5可知，一个NDEF Message可包含一个或多个NFC Record。在一个NDEF Message中，第一个NFC Record需设置其MB位（Message Begin）为1，表示它是该消息中第一个NFC Record，最后一个NFC Record需设置ME位（Message End）位为1，表示它是此消息中最后一个NFC Record。

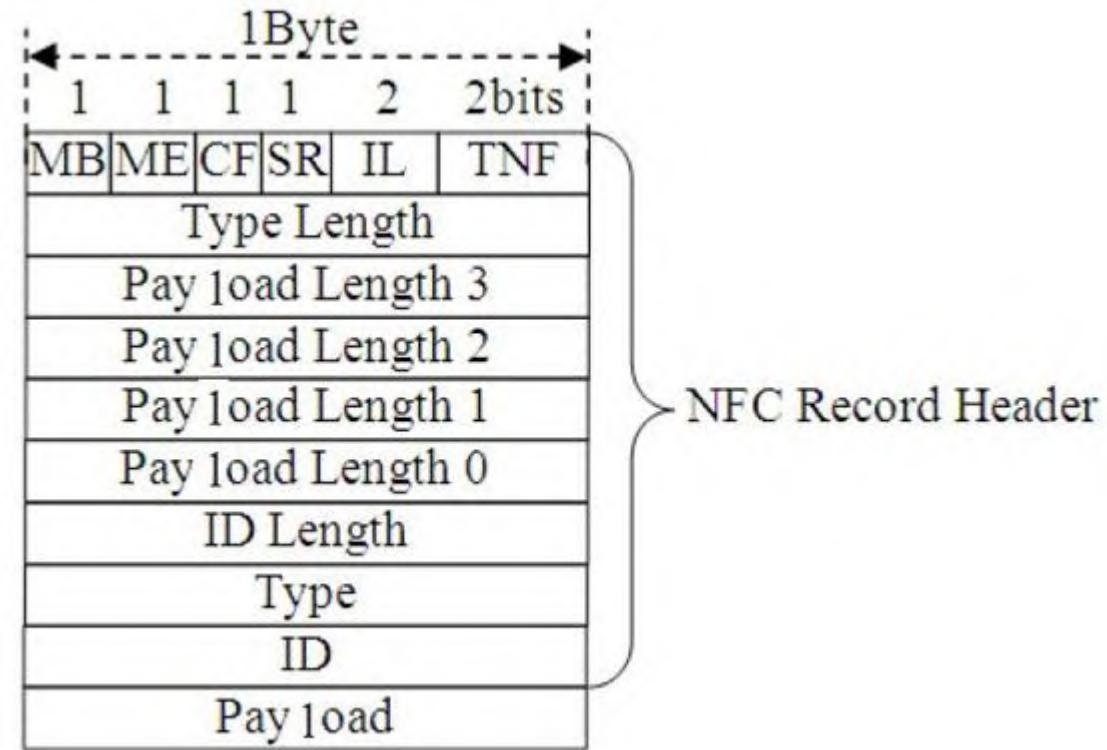


图8-6 NFC Record组织结构

NFC Record本身的组织结构如图8-6所示。NFC Record分为NFC Record Header（头部信息）和Payload（数据载荷）两大部分。

Record Header中最重要的是其第一字节。该字节有6个标志信息，分别如下。

- MB (Message Begin标志)
- ME (Message End标志)
- CF (Chunk Flag标志，表示该Record是否为分片Record)
- SR (Short Record标志。如果该标志被设置，则图中的4个Payload Length字段仅需一个，这表明Payload数据长度将限制在255字节以内)
- IL (ID\_LENGTH标志，它用于指明Header中是否包含ID Length和ID这两个字段)

- TNF (Type Name Format标志，用于指明Payload的类型，NFC Forum 定义了一些常用的Payload类型，详情见下文分析)

其他字节如下。

- Type Length指明Record Header中Type字段的长度。
- Payload Length 3～Payload Length 0这4个字段共同指明Payload 字段的长度。如果SR标志被设置，则Record Header仅包含一个 Payload length字段。
- ID Length指明ID字段的长度。如图所示IL标志未设置，则ID Length和ID字段都不存在。
- Type字段表明Payload的类型，NFC Forum定义了诸如URI、MIME等类型的Type，其目的是方便不同的应用来处理不同Type的数据，例如URI 类型的数据就交给浏览器来处理。
- ID需要配合URI类型的Payload一起使用，它使得一个NFC Record能 通过ID来指向另外一个NFC Record。

NFC Record中，常令初学者感到困惑的是TNF字段，其作用是什么？来看下文。

## (2) TNF和RTD

TNF用于描述一个NFC Record中数据（Payload）的类型，为了方便应 用程序能正确解析NFC Record中的数据，NFC Forum规定了一些常用的数据类型，如表8-2所示。

表 8-2 TNF 取值

TNF 名	取值	TNF 名	取值
Empty	0x00	NFC Forum External Type	0x04
NFC Forum Well-Known Type	0x01	Unknown	0x05
MIME	0x02	Unchanged	0x06
Absolute URI	0x03	Reserved	0x07

目前NFC支持七种数据类型。

- Empty：表示该Record中没有数据，即相当于一个空的NFC Record。

- NFC Forum Well-Known Type: 由NFC Forum定义的一些较为常用的数据类型，包括URI、TEXT等，其格式遵循NFC Forum RTD (Record Type Definition) 规范。下文将详细介绍它。
- MIME: 它是Multipurpose Internet Mail Extensions的缩写，遵循RFC2046规范。例如，当TNF取值为MIME时，其Type字段取值可为"text/plain"或"image/png"等。
- Absolute URI: 绝对URI，遵循RFC 3986规范。例如某文件的绝对URI为"http://android.com/robots.txt"，而其相对URI则为"robots.txt"。
- NFC Forum External Type: 也由NFC Forum的RTD规范定义，下文将介绍它。
- Unknown: 代表Payload中的数据类型未知，它和MIME类型"application/octet-stream"有些类似，这种类型的数据由相应的应用程序来解析。
- Unchanged: 这种类型的数据用于NFC Record分片。例如一个大的数据需要通过多个NFC Record来承载，除第一个NFC Record分片外，该数据对应的其他NFC Record分片都必须设置TNF为Unchanged。关于这部分内容，读者可参考NDEF规范的2.3.3节"Record Chunks"。

在TNF七大类型中，NFC Forum通过RTD规范定义了其中的WKT (Well-Known Type) 和External Type两种类型。虽然RTD规范全长只有20来页，但阅读起来比较枯燥，在此，笔者总结其核心内容。

简单点说，WKT就是NFC Forum自己定义的一些常用数据类型，目前常用类型如下。

- URI Record Type: 用于存储URI数据，对应Type字段取值为"U"。
- Text Record Type: 用于存储文本数据，对应Type字段取值为"T"。
- Signature Record Type: 用于存储数字签名数据，对应Type字段取值为"Sig"。

- Smart Poster Record Type: 智能海报，用于存储与该海报相关的一些资讯信息，如图片、相关介绍等，对应Type字段取值为”Sp”。
- Generic Control Record Type: 用于传递控制信息，对应Type字段取值为”Gc”。
- External Type: 为第三方组织定义的类型，目前NFC Forum没有定义相关的数据类型。

**提示** NFC Forum目前定义的所有WKT类型列表可参考  
[http://www.nfc-forum.org/specs/nfc\\_forum\\_assigned\\_numbers\\_register](http://www.nfc-forum.org/specs/nfc_forum_assigned_numbers_register)。

掌握了上述理论知识后，下面将通过两个实例来看看NFC Record各个字段到底该如何设置。

## 2. NFC Record实例<sup>[10][11]</sup>

本节这两个实例分别来自URI Record Type规范和TEXT Record Type规范。先来看URI Record Type实例。

### (1) URI Record Type实例

URI Record Type属于NFC Forum Well-known Type的一种，其对应的Type字段取值为”U”。对于这种类型的NFC Record，其Payload组织结构如表8-3所示。

表 8-3 URI Record Payload 组织结构

字段名	长度	说 明
Identifier Code	1字节	指明 URI 的 ID，详情见下文
URI	N字节	URI 的值

在URI Record Payload中，第一个字节指明URI的ID码，表8-4为NFC Forum定义的几种ID码。

表 8-4 ID Code 示意

ID Code	含    义	ID Code	含    义
0x00	无前缀	0x05	tel:
0x01	http://www	0x06	mailto:
0x02	https://www	0x07	ftp://anonymous:anonymous@
0x03	http://	0x08	ftp://ftp.
0x04	https://	0x09	ftps://

了解上述信息后，我们来看“`http://www.nfc.com`”这样的信息该如何封装为一个NDEF消息，图8-7所示为NDEF消息各字段的取值情况。

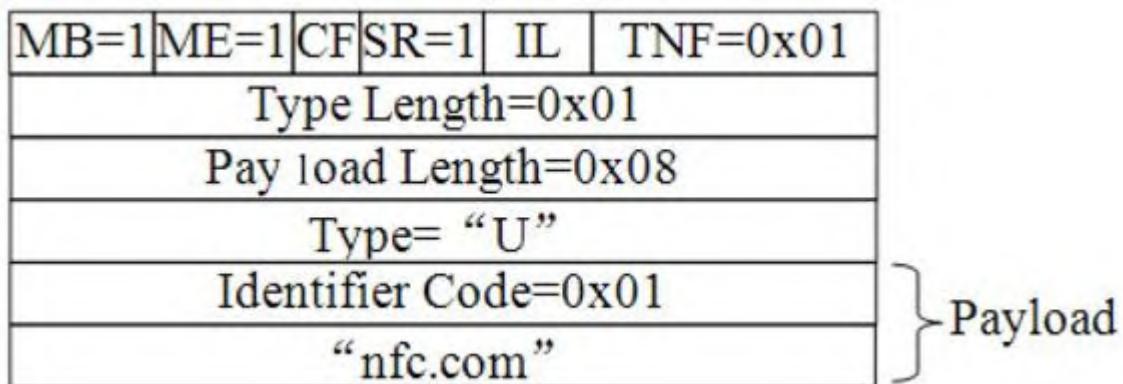


图8-7 URI Record实例

由于该NDEF消息只包含一个NFC Record，所以这个唯一的NFC Record将设置MB和ME标志位为1。另外，由于数据量小于255字节，所以SR标志位为1。最后，该Record携带的数据属于URI类型，它为Well-Known Type的一种，所以TNF取值为0x01。

Type Length字段取值为0x01，对应的Type字段取值为“U”，代表URI Record Type。

根据本节对URI Record的介绍，这种类型的Record的Payload包含ID Code和data两个部分。ID Code取值为0x01占据1字节（代表“`http://www`”），而data为“`nfc.com`”占据7字节，所以整个Payload长度为8字节，故Payload length字段取值为0x08。

当应用程序获取Payload信息后，将根据ID Code和Data的取值最终计算出对应的URI为“`http://www.nfc.com`”。

相信本节所述的URI Record实例能帮助读者更加直观得了解NDEF和NRC Record，下面再来看一个实例。

## (2) Text Record Type实例

Text Record Type和URI Record Type类似，其Payload组织结构如表8-5所示。

表 8-5 Text Record Payload 组织结构

字段名	长 度	说 明
Status	1字节	第7位为1，表示后面Text字段中的字符串采用UTF-8编码，否则为UTF-16编码。第0~5位表示语言码字段的长度
语言码	由status第0~5位决定	指明Text字段中字符串使用的语言，可选值有“zh”“en-US”“jp”等。语言码本身用ASCII字符串表示
Text		实际的Text内容，编码方式由Status第7位决定

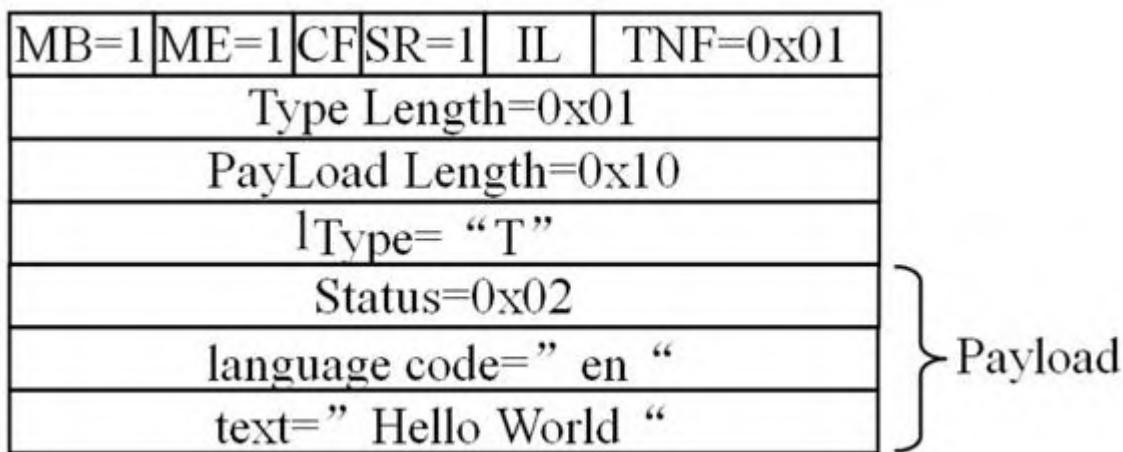


图8-8 TEXT Record实例

图8-8所示为携带“Hello World”字符串信息的NDEF消息各字段的取值情况。

实例比较简单，请读者根据本节对TEXT Record知识的介绍来自行解释图8-8各个字段的取值。

至此，NFC R/W运行模式介绍完毕。在R/W模式下，对应用程序而言最重要的工作就是解析NDEF消息。NFC Forum定义了七种数据类型，其中内容比较丰富的属于NFC Forum Well Known Type。本节介绍了WKT中

最简单的URI Record和TEXT Record。读者可在本节基础上自行研究其他几种数据类型。

### 8.2.3 NFC P2P运行模式[12]

在前面介绍的R/W模式中，NFC Device只能单向和NFC Tag交互，即只能NFC Device单方对NFC Tag发起操作，而NFC所基于的无线射频技术实际上可以支持NFC Device之间互相传递数据。为了满足NFC Device之间双向交互的需求，NFC Forum定义了P2P（Peer-to-Peer）运行模式。

图8-9展示了IEEE 802参考模型、OSI参考模型及NFC P2P的协议栈参考模型。由此图可知，NFC P2P协议栈最高层为LLC（Logical Link Control，逻辑链路控制层）。这一层使用的协议称为LLCP（LLC Protocol）。

在OSI参考模型中，LLC比较偏底层，其更多考虑的是物理地址寻址、链路管理，以及数据传输方面的事情（参考3.3.1节关于OSI/RM的介绍）。所以，NFC也在LLC层之上添加了一些对使用者更为方便和友好的协议。图8-10所示为NFC P2P协议栈的全貌。

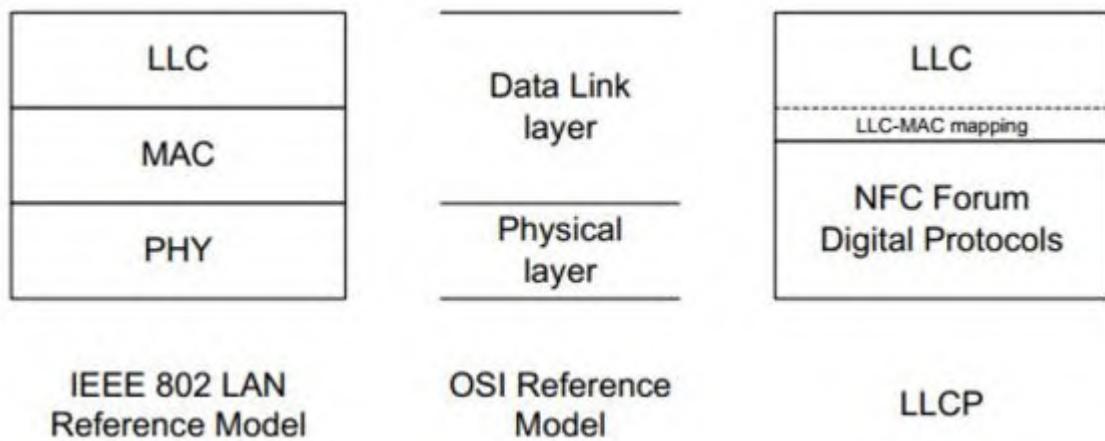


图8-9 NFC P2P协议栈

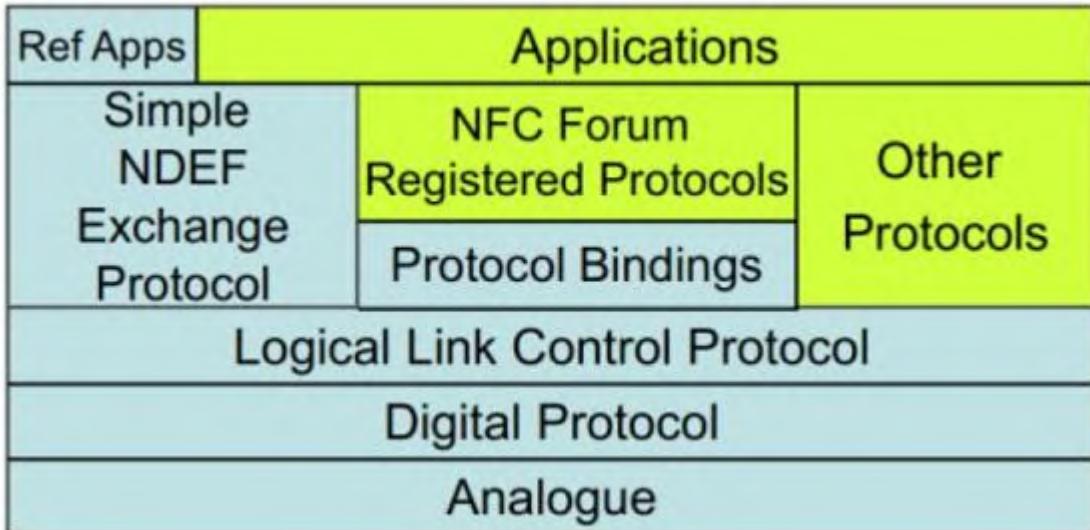


图8-10 NFC P2P协议栈参考模型全貌

- SNEP (Simple NDEF Exchange Protocol) 紧接LLC层。该协议使得两个NFC Device之间能直接交换NDEF消息。
- 通过Protocol Bindings, NFC可支持其他高层次并且用途更加广泛的协议。根据参考资料[3]所示的内容, NFC可支持IP和OBEX (Object Exchange, 对象交换) 协议, 但经过调查发现NFC Forum官网目前只有LLCP–OBEX–Binding协议的草案, 而LLCP和IP协议如何绑定还在研究当中。
- Other Protocols中目前比较常用的是CHP (Connection Handover Protocol) 。

目前, Android 4.2中的NFC P2P模块支持SNEP和CHP。本章将重点分析SNEP, 而CHP则请读者学完本章后再自行研究。下面先介绍LLCP, 然后再介绍SNEP。

## 1. LLCP介绍

NFC LLCP比较简单, 对应的规范全长也只有40来页。关于LLCP, 从以下两个方面来介绍。

- LLCP的数据封包格式。对学习通信协议来说, 掌握数据包格式非常重要。

- NFC LLCP对上层提供无链接（Connectionless）和面向链接（Connection-oriented）的两种数据传输服务。其中，无链接的数据传输服务和UDP类似，上层的收发双方无须事先建立逻辑链接关系即可收发数据。面向链接的数据传输服务和TCP类似，收发双方发送数据前，需要在LLC层先建立逻辑链接关系（即类似TCP协议中的connect和accept）。同时，LLC层还会处理数据包丢失、重传以及接收确认等方面的事情。目前SNEP和CHP均使用了LLC提供的面向链接的数据传输服务，故我们将重点介绍它。

### (1) LLCP数据包格式

NFC LLC层数据封包格式如图8-11所示。

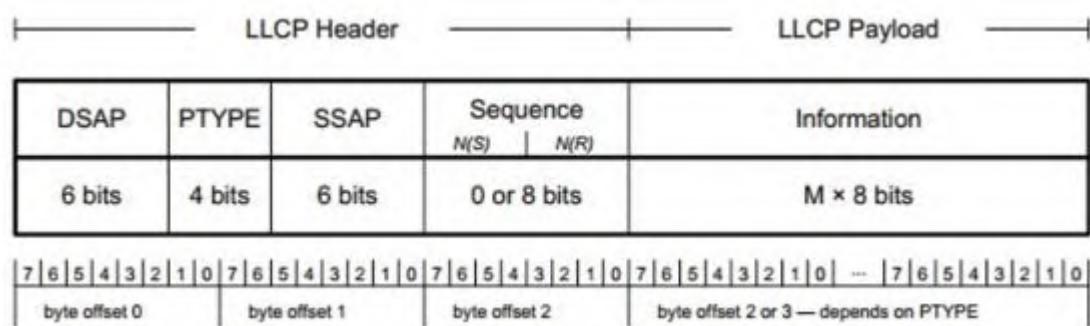


图8-11 NFC LLCP数据包格式

由图8-11可知，LLCP数据包前3字节为LLCP Header。LLCP Header之后就是Payload，其长度由PTYPE来决定。

- DSAP和SSAP分别代表Destination和Source Service Access Point（目标和源服务接入点）。DSAP和SSAP的作用类似于TCP/UDP中的端口号。注意，使用NFC LLCP时，DSAP和SSAP可唯一确定通信双方。读者可能有疑问，使用TCP/UDP时，除了指明端口号外，还需要指明对端设备的IP地址，但NFC LLCP数据包中却没有这样的信息。另外，和图3-24所示的LLC数据封装格式比起来，NFC LLCP数据包也没有MAC地址这样的字段。也就是说，LLCP只需要通信双方所使用的端口号即可，而无需MAC或IP地址这样的信息，这是因为NFC近距离作用的特点使得通信双方从进入有效距离内开始就已彼此确定，故无需再通过MAC地址指明谁是接收设备，谁是发送设备。而当上层通过LLC发送数

据或者LLC向上层传递接收到的数据时则需要通过类似端口这样的SSAP和DSAP来进一步确定发送模块和接收模块到底是谁。

- PTYPE字段指明LLCP包的类型。NFC LLCP定义了多种不同类型的包，下文将结合面向链接的数据传输服务来学习相关的LLCP包。
- Sequence字段指明LLCP包的序号，它可分为Send端和Receiver端。由于有一些类型的LLCP包无需Sequence字段，所以Sequence字段长度有可能为0。例如，无链接的数据包就不需要Sequence字段，而面向链接的数据包需要该字段来处理数据接收确认或丢失重传等方面的事情。

由上述介绍可知，DSAP和SSAP类似TCP/UDP的端口号，决定了收发模块到底是谁。表8-6所示为NFC中SAP取值情况。

表 8-6 LLCP SAP 取值说明

SAP 取值	说 明
0x00 ~ 0x0F	NFC 定义了一些常用服务（Well-Known Service, WKS）的端口号。例如 0x00 用于和 LLC 层中的链路管理模块通信，而 0x01 用于 SDP（Service Discovery Protocol）。SNEP 服务对应的端口是 0x04
0x10 ~ 0x1F	本地服务的端口，远端设备可通过 SDP 搜索到它们
0x20 ~ 0x3F	本地服务的端口，远端设备不能通过 SDP 搜索到它们

以SNEP的使用为例：

- 位于NFC Device A的服务端模块在SSAP为0x04的端口上进行监听。
- 位于NFC Device B的客户端模块选择一个合适的SSAP，设置DSAP为0x04。然后该客户端模块发送数据包，LLC负责将数据包打包传递给NFC Device B（假设这两个设备都在彼此的有效距离内）。
- NFC Device A的LLC层接收到数据包后发现DSAP为0x04，而其上刚好有一个服务模块工作在0x04端口，故LLC层将把数据包传递给这个在0x04端口上监听的服务模块。

下面将通过分析面向链接数据传输服务的工作流程来进一步研究LLCP。

## （2）面向链接数据传输服务

假设Device A和Device B打开了NFC功能。当二者进入有效距离后，它们的LLC模块将进入Link Activation（链路激活）阶段，在此阶段中，A和B的交互过程如图8-12所示。

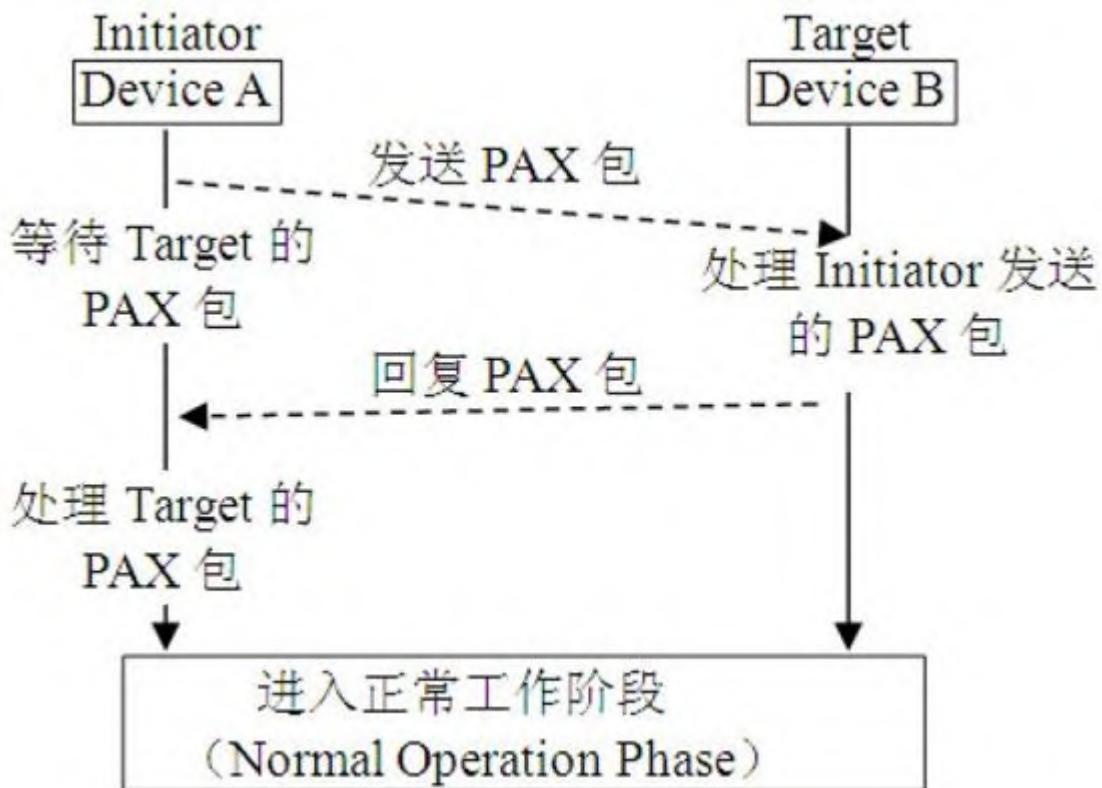


图8-12 Link Activation工作流程

- 进入Link Activation时，Device A和Device B将分别扮演Initiator和Target角色，参考资料[13]可用于确定谁来扮演Initiator或Target。
- Initiator发送PAX数据包给Target。PAX全称为Parameter Exchange，它用于在两个设备间交换彼此的LLC层配置信息（如协议版本等，详情见下文）。
- Target收到Initiator的PAX包后需要相应处理，例如判断协议版本是否匹配等。Target处理完后，它需要发送自己的LLC层配置信息给Initiator。

- Initiator 检查 Target 的 LLC 层配置参数，如果一切正常，双方 Logical Link 成功建立，随后可进入正常工作阶段。

根据上述内容，双方需要通过 PAX 交换 LLC 层的配置信息。PAX 属于 LLCP 数据包的一种，其格式如图 8-13 所示。

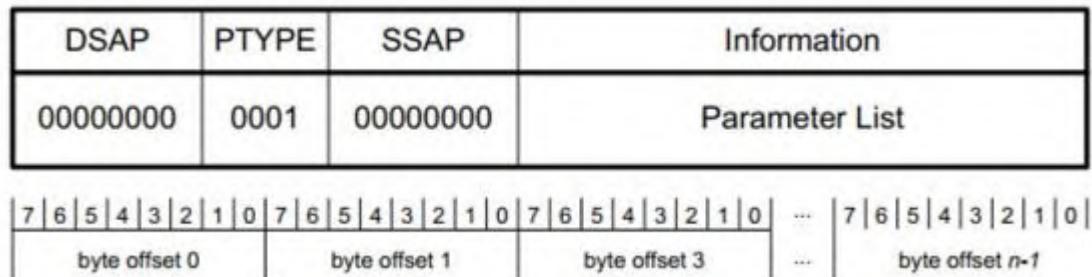


图 8-13 PAX 数据包

LLC 层的配置信息保存在图 8-13 中的参数列表中，正常情况下 PAX 携带的参数信息及作用如表 8-7 所示。（注意，并非所有参数都会包含在图 8-13 的参数列表中。）

表 8-7 PAX 参数说明

参数名	Type 值	说 明
VERSION	0x01	用于指明 LLC 协议的版本号
MIUX	0x02	全称为 Maximum Information Unit Extension。图 8-11 中，每个 NFC Device 所能处理的 LLC 数据包 Information 字段的长度是有限制的，其最大长度叫 MIU。MIU 默认为 128 字节。MIUX 用于告诉对端设备自己的 MIU 是多少。注意，MIUX= 真实 MIU-128
WKS	0x03	全称为 Well-Known Service List，用于告诉对端设备当前本机哪些 Well-Known 服务端口上有模块在监听。该字段长 16 位，和表 8-6 所示的 WKS 取值范围（0x00 ~ 0x0F，也是 16 位）一致，该字段每一位对应一个 SAP
LTO	0x04	全称为 Link Timeout，链接超时时间，基本单位为 10ms，LTO 取值为 10ms 的倍数
OPT	0x07	全称为 Option，目前只有 Link Service Class 选项，该选项表明本设备支持数据传输的类型，例如无链接、有链接还是二者都支持

以上知识有一些细节需要读者注意。

- 当 Device A 和 Device B 进入有效距离后，Link Activation 将被触发，而 Device A 和 B 分开后，Link Deactivation 将被触发。从使用角度来看，Link Activation/Deactivation 可能会频繁被触发。

- WKS用于告知本机设备哪些Well-Known服务端口上有模块在监听。这表明在Link Activation被触发前，使用者就必须在感兴趣的WKS端口上进行监听。这和笔者之前所认为的设备先进入Link Activation，然后再监听WKS端口不同。以后分析Android平台中SNEP的代码时读者将看到相关的处理。

Link被激活后，Device A和Device B将先建立面向链接的关系，然后再开展数据交互，这一流程如图8-14所示。

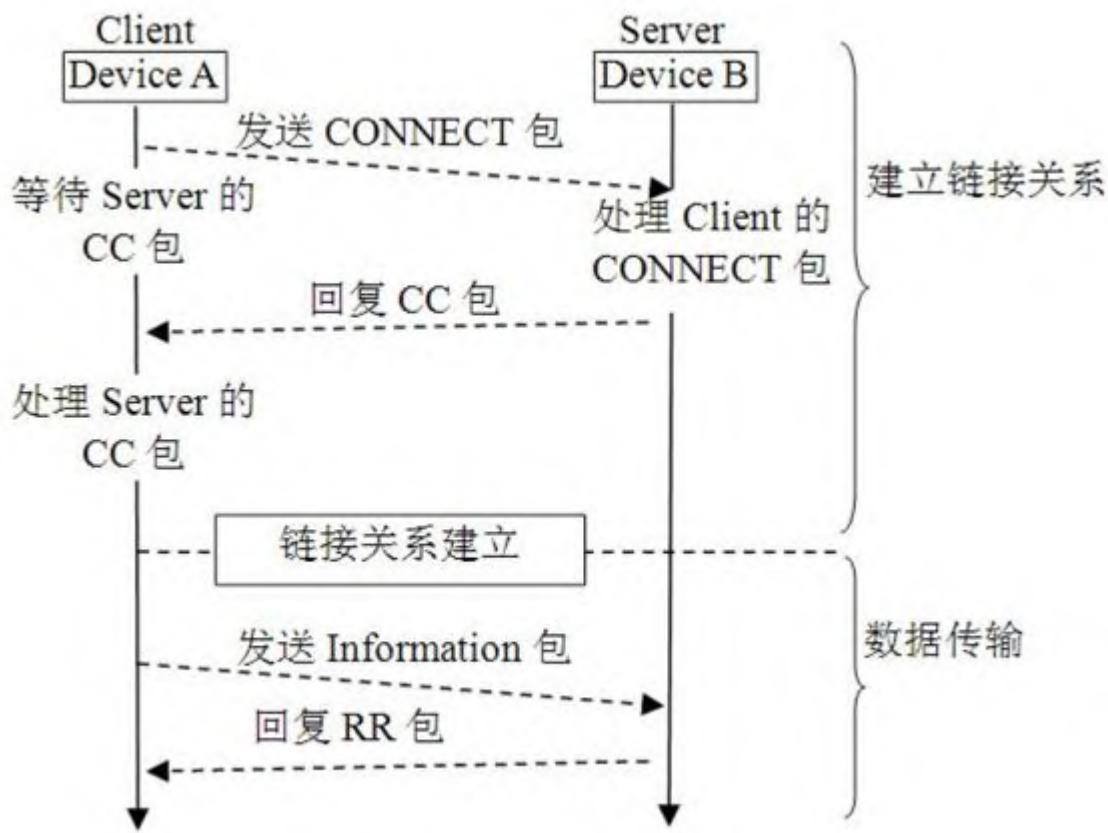


图8-14 面向链接工作流程

假设Device A扮演Client角色，Device B扮演Server角色。Client先通过CONNECT包向Server发起链接请求。CONNECT包对应的格式如图8-15所示。

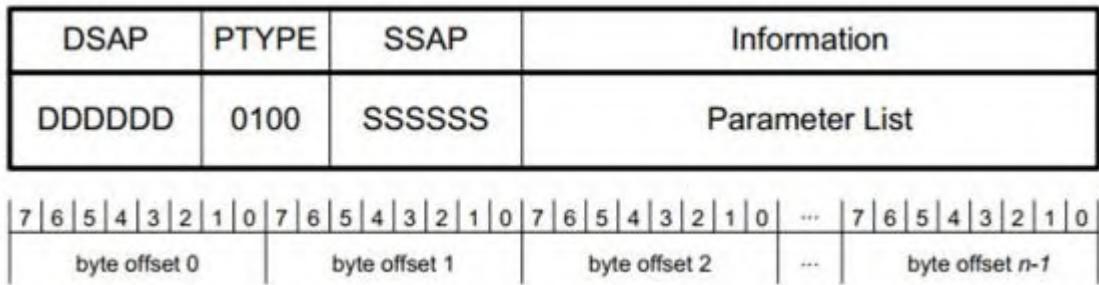


图8-15 CONNECT包格式

CONNECT包需要携带一些参数信息，常见的参数如表8-8所示。

表 8-8 CONNECT 参数说明

参数名	Type 值	说 明
MIUX	0x01	MIUX 可以在 Link Activation 阶段传递，也可以在 CONNECT 时传递
RW	0x05	全称为 Receive Window Size。该参数和数据接收确认有关。RW 为 1，表示数据接收方每收到一个数据包就会回复确认信息
SN	0x06	全称是 Service Name。LLCP 中，Client 支持两种方法来链接 Server 端。 • Client 发起链接时可直接指明 DSAP。以 SNEP 为例，只要设置 DSAP 参数为 0x04 • 设置 DSAP 为 0x01，然后设置 SN 参数为指定服务的名称。以 SNEP 为例，SN 参数设置为 “urn:nfc:sn:snep”

当服务器端成功处理CONNECT包后，它将回复CC（Connection Complete）包给客户端。如此，Client和Server就建立了链接关系。CC包内容非常简单，请读者自行研究参考资料[12]。此后，Client和Server就可通过Information（规范中简称为I）包和RR（Receive Ready）包来传输数据，其中：

- I包用于承载具体的数据。
- RR包用来确认接收方确实收到了数据。

I和RR包比较简单，这部分内容也请读者自行研究参考资料[12]。

总体而言，LLCP比较简单，不过直接使用LLCP还是稍显复杂。所以，在LLCP基础上，NFC Forum定义了SNEP协议用于在两个NFC Device之间传输NDEF消息。下面来学习SNEP。

## 2. SNEP介绍<sup>[14]</sup>

SNEP (Simple NDEF Exchange Protocol) 支持在两个NFC Device之间交换NDEF消息。SNEP是一种基于面向链接的数据传输协议，作为Well-Known Service的一种，其服务端口号为0x04，服务名为“urn: nfc:sn: snep”。

SNEP属于Request/Response方式，其工作过程如图8-16所示。

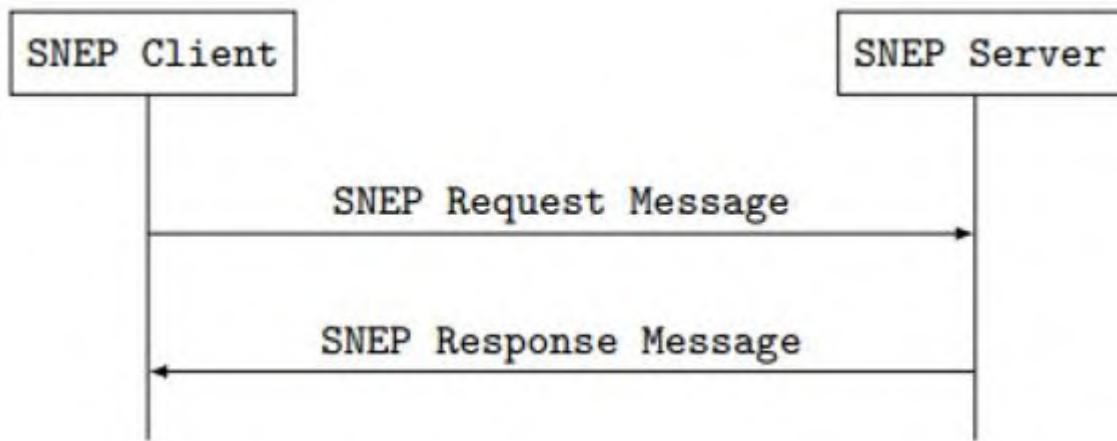


图8-16 SNEP工作方式

SNEP的工作流程非常简单，主要包括两个步骤。

- 1) SNEP客户端发送SNEP Request消息给服务端进行处理。
- 2) SNEP服务端回复SNEP Response消息给客户端以告知处理结果。

SNEP Request消息和Response消息的格式如图8-17所示。

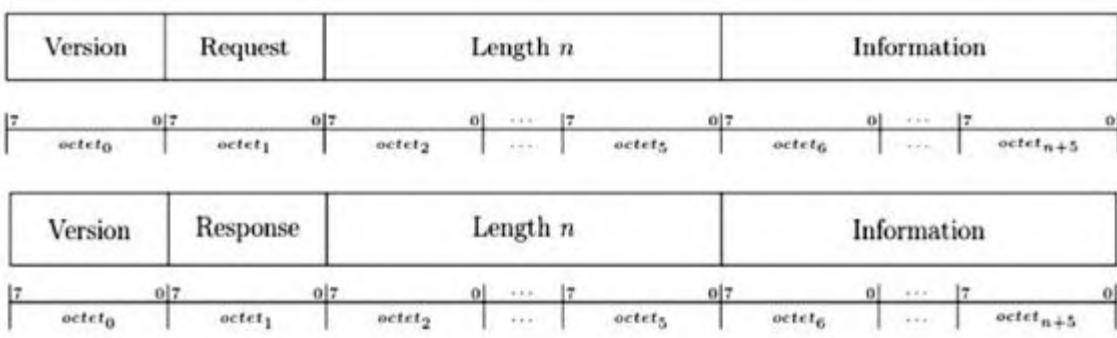


图8-17 SNEP Request/Response消息格式

图8-17中，SNEP Request/Response消息开头都是1字节的Version字段，Version字段之后分别是Request字段和Response字段。Request字段表示请求类型，Response字段表示处理结果。

表8-9所示为SNEP当前支持的Request类型。

表 8-9 Request 类型说明

Request 名	取 值	说 明
Continue	0x00	继续发送分片 SNEP 数据
Get	0x01	客户端请求从服务端返回一个 NDEF 消息
Put	0x02	客户端发送一个 NDEF 消息给服务端
Reject	0x7F	停止发送分片 SNEP 消息

以SNEP Put请求消息为例，其对应的格式如图8-18所示。

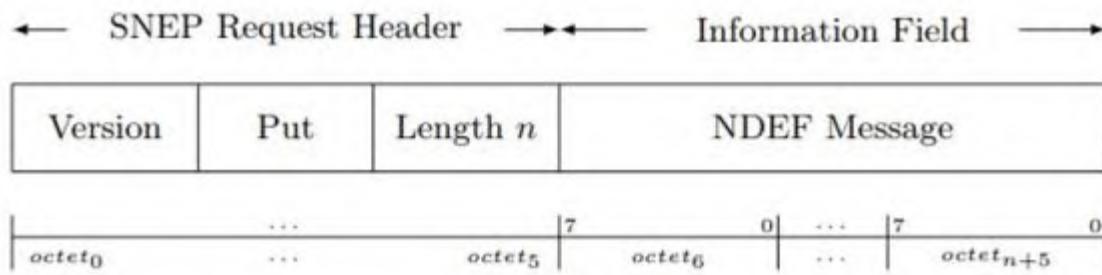


图8-18 SNEP Put消息格式

SNEP协议本身非常简单，此处不详细介绍，本章下文将结合代码来介绍Android中SNEP的实现。

至此，我们对NFC LLCP进行了相关介绍。这部分难度不大，读者需要重点掌握的部分包括LLCP协议本身，尤其是其数据封包格式、各种参数信息、常见SAP等。在LLCP基础上，读者可学习SNEP这种比较常用的协议。另外，读者还可在本节基础上自行学习Connection Handover，它是另外一种基于LLCP面向链接数据传输服务的协议。

**提示** 以后在分析Android NFC模块代码时也会碰到Connection Handover，请读者自己来分析。

下面来看NFC运行模式中最后一种，即NFC CE（Card Emulation）模式。

## 8.2.4 NFC CE运行模式<sup>[15][16]</sup>

NFC CE运行模式使得携带NFC芯片的设备能充当智能卡（例如信用卡）使用。该运行模式所支持的应用场景极具吸引力，例如用支持该功能的Android智能手机来完成购票、支付，甚至充当门禁卡，汽车钥匙、公交卡等。

图8-19为CE运行模式示意图。

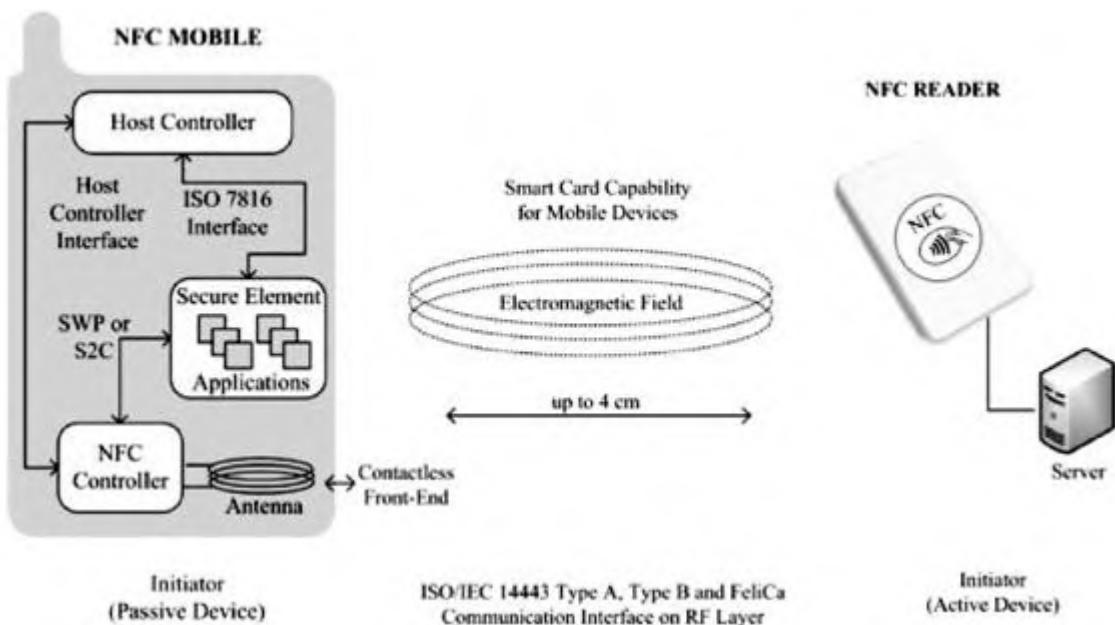


图8-19 CE运行模式

由图8-19可知，SE和NFC芯片（主要是指NFC Controller，简称NFCC）通过SWP (Single Wire Protocol) 或者S<sup>2</sup> C (SignalIn/SignalOut Connection Interface, 也叫NFC Wired Interface, 简称NFC-WI) 来交互。一般来说，SE上面运行了一些特殊的应用程序，NFC负责将数据通过SWP或S<sup>2</sup> C传递给SE中的应用来处理。

NFCC通过HCI协议和NFC Mobile交互，而SE也可通过ISO 7816协议和NFC Mobile交互。

在CE模式中，NFC Mobile被NFC Reader识别成一个智能卡。NFC Reader通过相关规范发送数据或控制命令给NFC Mobile中的NFCC。

当NFCC收到数据或控制命令后，将交给相关的应用程序来处理。由于CE相关的应用场景针对支付、门禁等这类对安全性要求非常高的情况，以Android手机NFC支付为例，一个完整的支付应用程序包括一个为用户提供操作界面的APK以及一些运行在安全性有绝对保障的SE中的应用程序。

总之，SE在CE模式中扮演了非常重要的角色，目前SE和NFC的组合有三种方式，如图8-20所示。这三种组合方式从上到下分别如下。

- SE为一个嵌入式安全芯片，该芯片在手机出厂前就已经安装在其内部，而且无法被替换。该芯片上运行着一个小系统能够处理支付或安全方面的工作。目前，这种形式的SE还没有标准规范，可参考的模型有NXP公司的pn65芯片模块示意（如图8-21<sup>[17]</sup> 所示）。

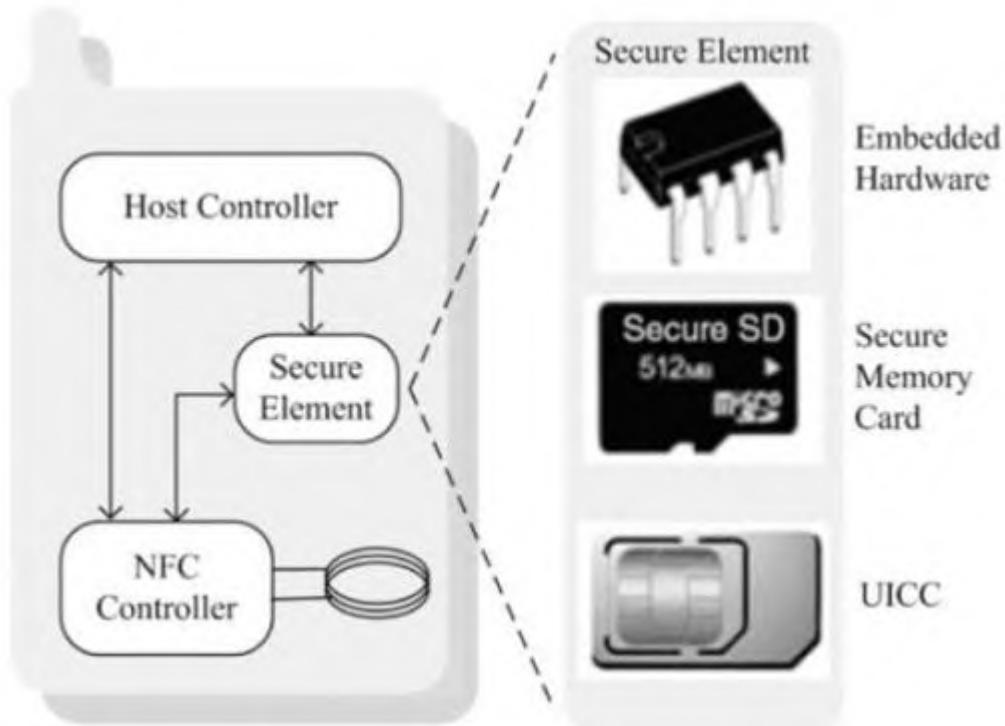


图8-20 SE和NFC的组合方式

- SE为一个支付型SD卡，这种卡实际上是在SD卡上嵌入了安全模块，相关应用可在这张卡上运行。该种组合方式所对应的方案也称为NFC-SD方案，这方面的国际标准有ISO 7816。中国的银联曾经主推过NFC-SD卡支付解决方案。
- SE为UICC，也就是常说的手机SIM卡，这种组合方式所对应的方案也称为NFC-SIM方案，目前由运营商主推。前面提到的北京市利用NFC手机充当一卡通所使用的方案就是NFC-SIM，它需要使用者先到移动运营商那换一个特殊的SIM卡。

图8-21中，NXP公司pn65 NFC芯片自身就包含一个Secure Element，即图中的SmartMX模块，该模块中运行着一个名为Java Card OS的操作系统。在Java Card OS上，用户可以安装和运行一些应用程序（称为Applets）。除了SmartMX内置的SE外，pn65也支持使用外部的SE，即图8-21中的UICC。

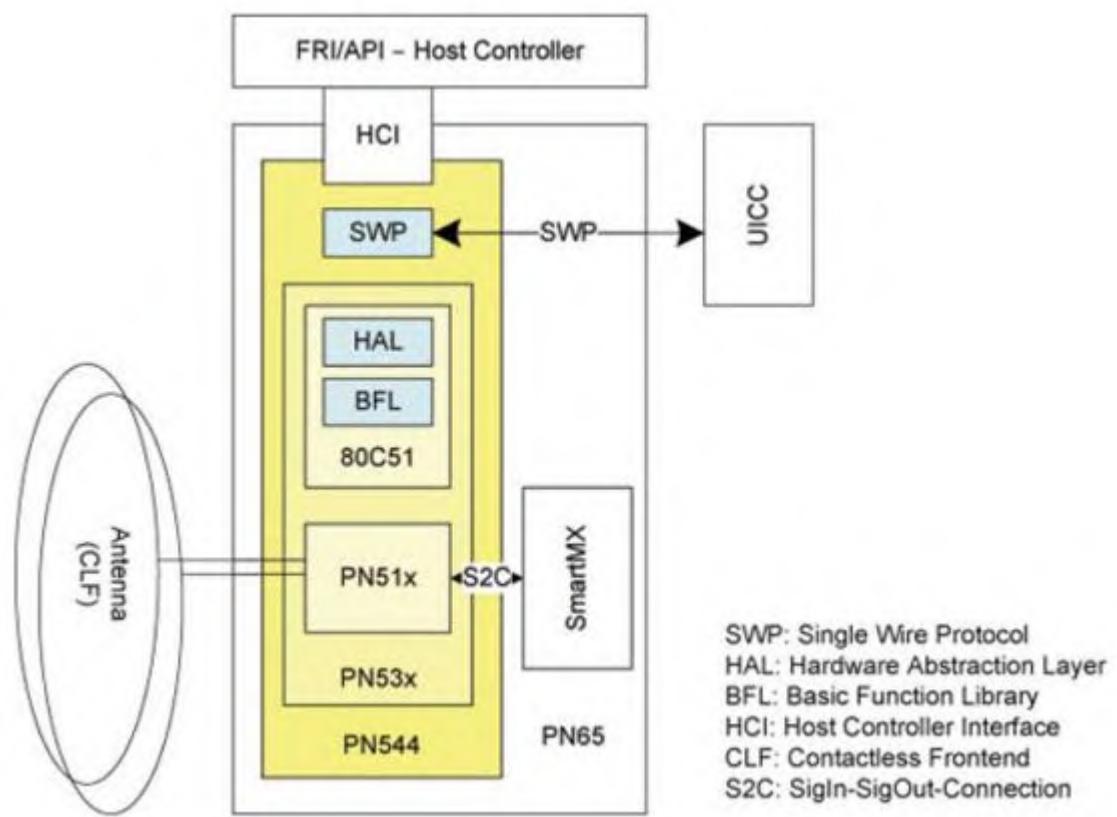


图8-21 NXP pn65芯片模块

**提示** 从参考资料[18]和[19]来看，目前国际上大多使用NFC-SIM方案，而中国的运营商和银联也将联合推广它，其对应的商品名叫“闪付”。

SE和NFC控制器连接所使用的S2C和SWP协议中，NFC-SIM方案将采用SWP，其对应的规范是ETSI TS 102613。NFC和UICC使用SWP的连接如图8-22所示。

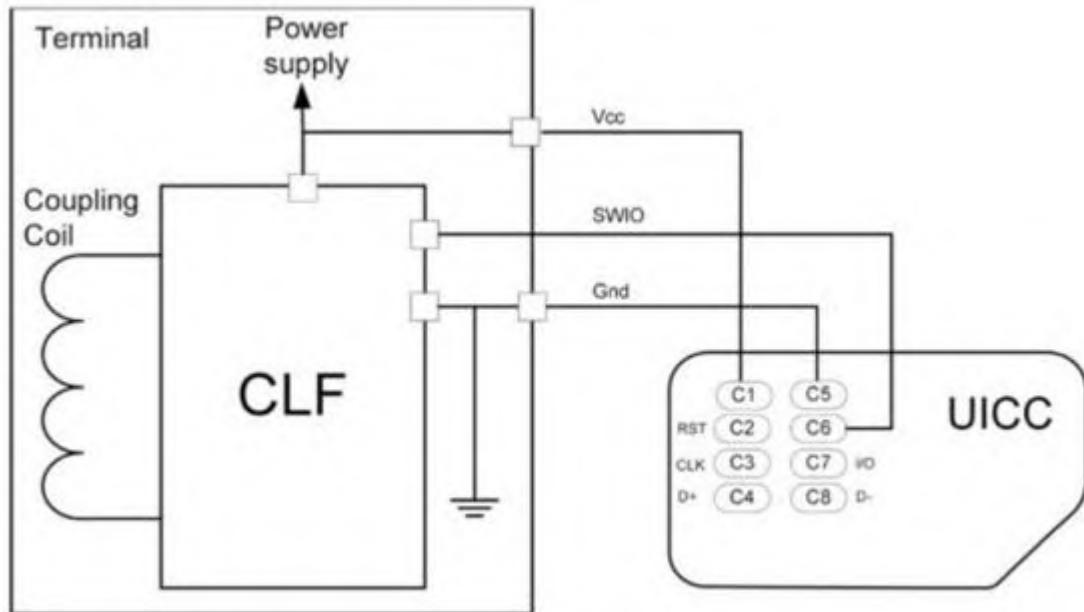


图8-22 CLF-UICC连线

CLF（NFC Contactless Front-End缩写）和UICC通过三条线相连。Gnd接地，Vcc提供电源。SWIO为CLF和UICC的数据连接线，数据传输率在212kbps~1.6Mbps之间，每次传输的数据包小于30字节。

注意，图中UICC的电源由CLF来提供，而非直接由手机电源来提供。这种设计方案使得手机在电池耗尽的情况下，也可通过外部电磁感应（由NFC Reader或其他NFC设备）来给CLF和UICC供电，从而确保支付请求不受手机本身的电源影响。

**提示** 关于SWP的细节，读者可参考ETSI TS 102613。

NFC Forum没有和CE相关的规范，所以读者先了解本节所述的知识，后文在NFC CE示例中将进一步介绍与之相关的内容。

至此，NFC Device最后一种运行模式就介绍完毕，下面来介绍NFC理论知识的最后一部分，即NCI。

### 8.2.5 NCI原理[20]

NCI (NFC Controller Interface) 是NFC Forum于2012年制定的一个规范，其主要关注点为DH (Device Host, 主机设备) 如何控制并与NFCC (NFC Controller) 交互。图8-23所示为NFCC、NCI和DH三者之间的关系。

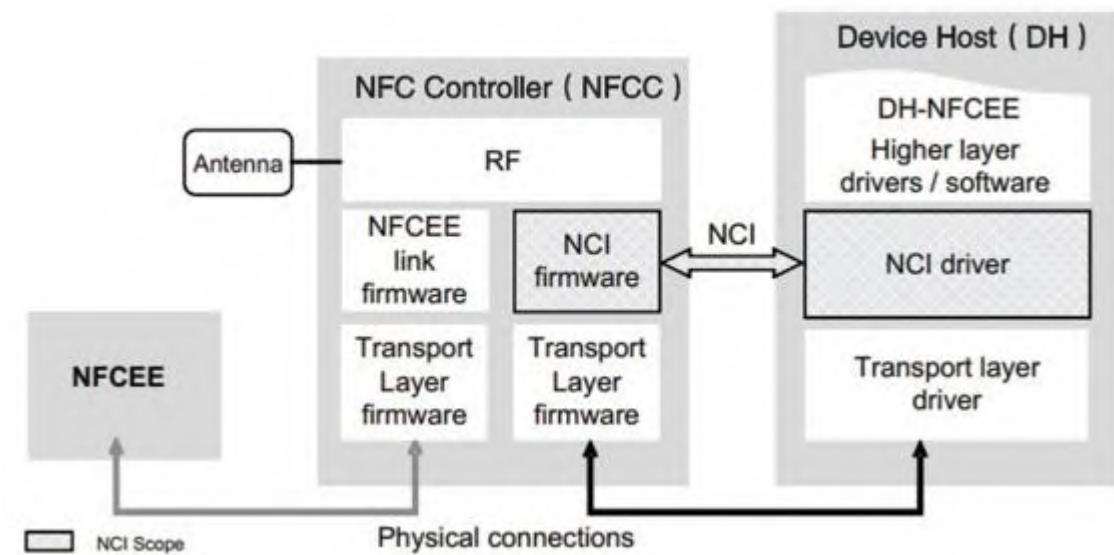


图8-23 NFCC、NCI和DH三者之间的关系

在图8-23中，NFCC和DH通过物理连线相连，物理连线对应为Transport Layer（传输层）。目前，NFCC和DH在传输层这一块支持SPI、I<sup>2</sup> C、UART和USB等。

在图右边的DH中，所有和NFC相关的应用程序都可被视为DH-NFCEE（EE是Execution Environment的缩写）。图左边有一个NFCEE模块，该模块也可运行着一些和NFC相关的程序或系统（以图8-21为例，它的SmartMX Secure Element就是此处所说的EE）。NFCEE模块可直接集成在NFCC中，也可作为单独的芯片模块通过物理连线与NFCC相连。

NCI负责处理DH和NFCC之间的交互。NCI包含多个模块，详情见下文。图8-24所示为NCI的模块结构。

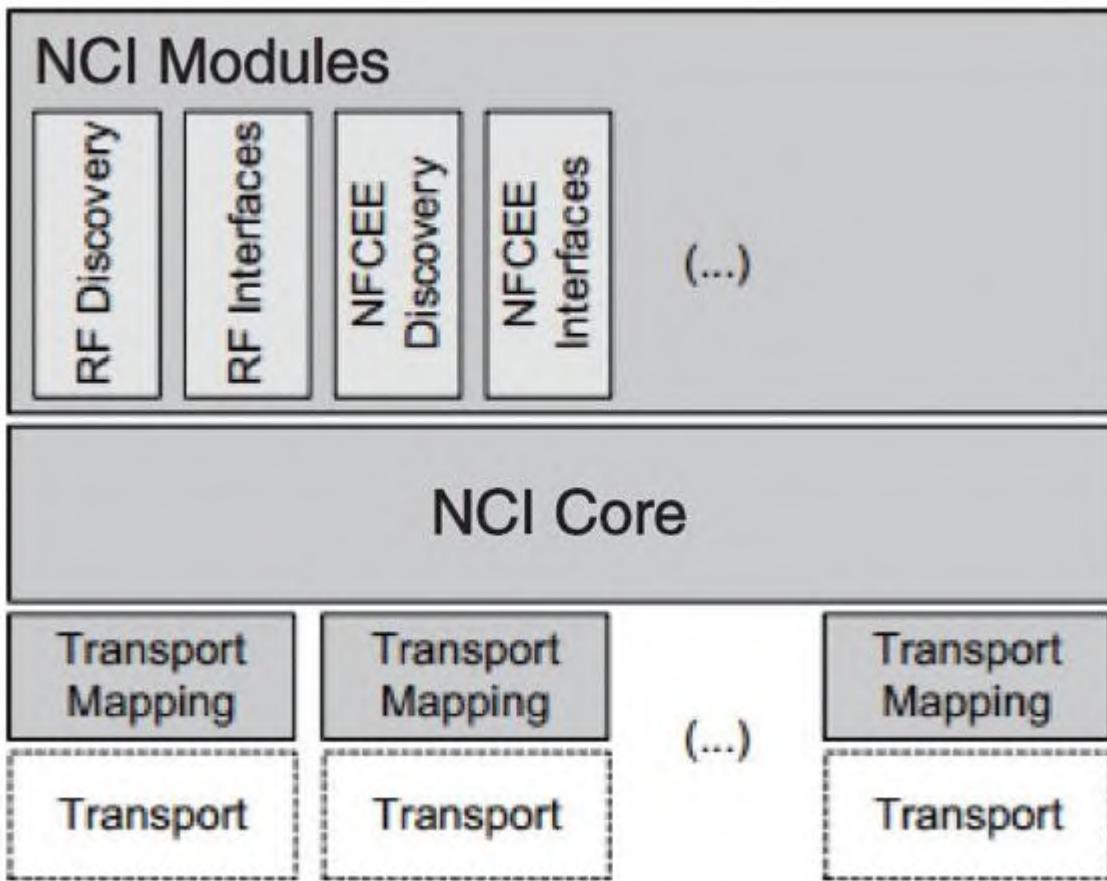


图8-24 NCI模块结构

- NCI Core模块负责DH和NFCC之间交互的基本功能，包括控制消息（Control Message）和数据消息（Data Message）的传递、DH初始化、重置和配置NFCC等。
- Transport Mapping用于在NFC Core和传输层之间转换数据格式，例如将NCI Core使用的控制消息和数据消息转换成对应传输层使用的数据格式。
- NCI Module包含多个功能模块，例如RF Discovery模块用于搜索周围的其他NFC Device、RF Interface用于和对端的NFC Device交互。

使用NCI的NFC Device中，DH和NCI的工作原理如图8-25所示。

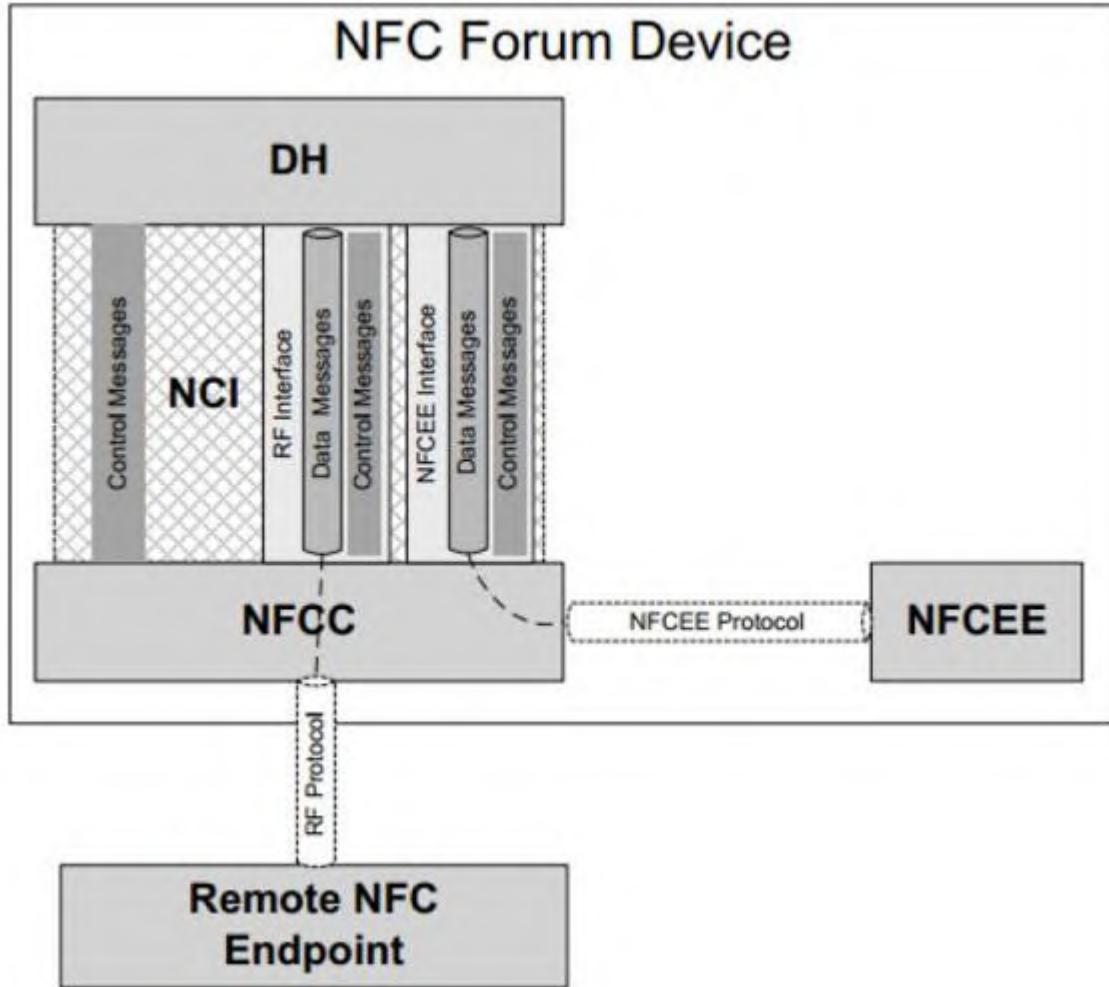


图8-25 NCI工作原理

图8-25中：

- DH通过NCI规范定义的Control Message来控制NFCC。目前规范定义的Control Message包括Commands（请求命令，包括初始化NFCC、重置NFCC、设置NFCC配置参数等）、Responses（回复）和Notifications（通知）。这些Message都封装在NCI Control Packages中。其中，Commands只能由DH发送给NFCC。
- DH通过RF Interface和对端NFC设备（图中的Remote NFC Endpoint）交互，也可通过NFCEE Interface和本设备的NFCEE交互。交互数据包括Control Message和Data Message。

NCI规范一共有140多页，是NFC Forum众多规范中比较复杂的一个。根据笔者的理解，NCI的一个很重要的作用就是统一Android中NFC HAL层的实现，即通过一套标准的方法来实现对NFCC的控制以及数据交互。

不过，由于NCI规范推出的时间比较晚（该协议最终版的时间为2012年11月6日），所以占据最大市场份额的NXP公司在其Android平台的NFC HAL层中还没有使用NCI。

**提示** 8.4节将专门讨论Android平台中NFC HAL层的实现状况。从Android 4.2的代码来看，NXP公司使用了自己的一套NFC HAL层实现方式，而博通公司的NFC HAL层的实现参考了NCI规范。但实际上这两家公司NFC HAL层的代码处处透露着它们与特定芯片的紧密关系，这使不了解芯片细节的读者很难真正看懂NFC HAL层的代码。随着NFC的重要性和普及程度日益加大，开发者已经在Linux Kernel 3.8<sup>[21]</sup> 中增加了一个名为NFC的子系统，它使得以后的NFC HAL层只需通过netlink消息就可和位于Kernel空间的NFC驱动交互。因目前NFC HAL层这些被不同芯片所“绑架”的代码就可从用户空间移除，而那些和芯片相关的代码就可通过NFC驱动的形式运行在Kernel之内。

## 8.2.6 NFC相关规范

至此，我们对NFC理论知识进行了一番介绍，读者应重点关注NFC的三种运行模式以及相关的数据类型定义、协议栈和工作方式。另外，对目前NFC HAL层实现感兴趣的读者不妨仔细研究NCI规范。

NFC涉及的规范非常多，表8-10总结了相关的规范<sup>[22]</sup>。

表 8-10 NFC 相关规范总结

发布机构	规 范 名	说 明
ISO/IEC ( International Organization for Standardization/ International Electrotechnical Commission, 国际标准化组织 / 国际电工委员会)	ISO/IEC 18092	NFC Interface and Protocol (NFCIP-1)
	ISO/IEC 21481	NFC Interface and Protocol (NFCIP-2)
	ISO/IEC 28361	NFC Wired Interface (NFC-WI)
	ISO/IEC 14443	Contactless Proximity Smart Cards and their technical features
	ISO/IEC 15693	Contactless Proximity Smart Cards and their technical features
ETSI ( European Telecommunications Standards Institute, 欧洲电信标准化协会)	ETSI TS 102 190	NFC Interface and Protocol (NFCIP-1)
	ETSI TS 102 312	NFC Interface and Protocol (NFCIP-2)
	ETSI TS 102 541	NFC Wired Interface (NFC-WI)
	ETSI TS 102 613	Contactless front end ( CLF ) interface to UICC, physical and data link layer characteristics; Single Wire Protocol ( SWP )
	ETSI TS 102 622	Contactless front end ( CLF ) interface to UICC, Host Controller Interface ( HCI )
ECMA ( European Computer Manufacturers Association, 欧洲计算机制造联合会)	ECMA 340	NFC Interface and Protocol (NFCIP-1)
	ECMA 352	NFC Interface and Protocol (NFCIP-2)
	ECMA 356	NFCIP-1 RF Interface Test Methods
	ECMA 362	NFCIP-1 Protocol Test Methods
	ECMA 373	NFC Wired Interface (NFC-WI)
	ECMA 385	NFC-SEC:NFCIP-1 Security Services and Protocol
	ECMA 386	NFC-SEC-01:NFC-SEC Cryptography Standard using ECDH and AES
NFC Forum	ECMA 390	Front-End Configuration Command for NFC-WI
		请参考 <a href="http://www.nfc-forum.org/specs/spec_list/">http://www.nfc-forum.org/specs/spec_list/</a>

## 8.3 Android中的NFC

Android平台中，NFC系统模块运行在一个名为“com.android.nfc”的应用进程中，该应用程序的代码位于packages/apps/Nfc下。由于目前NFC HAL层的实现还没有统一接口，所以该应用程序对应的组织结构如图8-26所示。

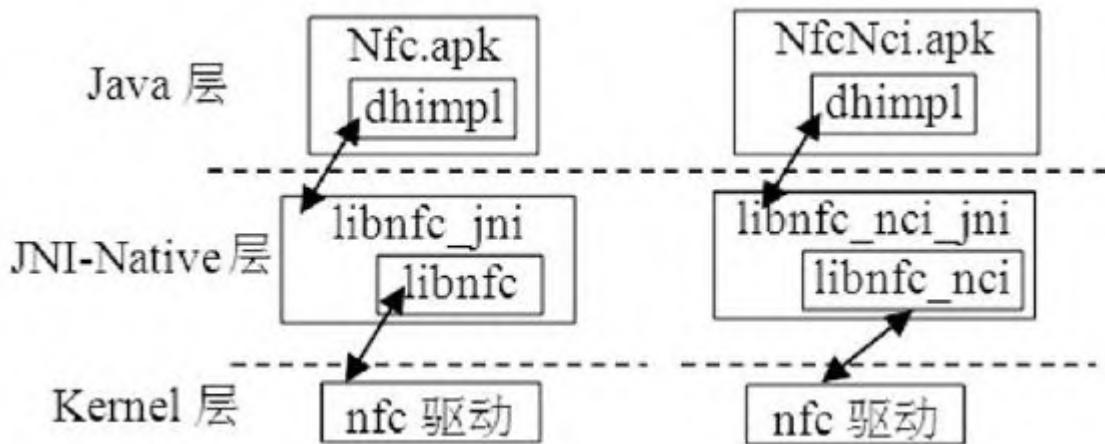


图8-26 Android平台中Nfc模块结构

如果使用NXP公司pn系列的NFC芯片，则Nfc模块结构如左图所示，即最终的APK文件名为Nfc.apk，它通过packages/apps/Nfc/nxp目录下dhimpl模块与libnfc\_jni以及libnfc这两个动态库交互。libnfc的代码位于external/libnfc-nxp目录下，由NXP公司提供以用于操作NXP公司的NFC芯片。

如果使用博通公司2079x系列的NFC芯片，则Nfc模块结构如右图所示，即最终的APK文件名为NfcNci.apk，它通过packages/apps/Nfc/nci目录下的dhimpl模块与libnfc\_nci\_jni以及libnfc\_nci这两个动态库交互。libnfc\_nci的代码位于external/libnfc-nci目录下，由博通公司提供以用于操作博通公司的NFC芯片。

**提示** 图8-26所述的Nfc模块结构对应的Android系统版本为4.2，而Android 4.1只支持NXP公司的芯片。

如果看过libnfc\_jni或libnfc\_nci\_jni的代码，会发现它们分别使用了NXP和博通公司封装得用于和各自NFC芯片交互的API，代码可读性非常差。这种情况出现的原因正是前文所说当前Linux Kernel中还没有一种统一的方法让用户空间的进程和NFC驱动交互。当然，此问题有望通过完善NFC Subsystem和对应的netlink消息机制得以解决。

基于上述原因，本书不打算介绍任何与特定芯片平台结合过于紧密的模块。所以，本章分析重点将以图8-26中Nfc.apk为主，它包含了Android平台中NFC的一些核心知识。读者在掌握的基础上，可尝试结合pn544芯片的数据手册来自行分析dhimpl、libnfc-jni和libnfc。

下面将开始NFC代码分析之旅，包括两条分析路线。

- 先分析NFC相关的应用程序，从客户端角度介绍如何使用Android系统提供的NFC服务。
- 然后介绍Nfc.apk，展示NFC系统模块的核心内容。

读累了记得休息一会儿哦~

公众号：古德猫宁李

- 电子书搜索下载
- 书单分享
- 书友学习交流

网站：[沉金书屋 https://www.chenjin5.com](https://www.chenjin5.com)

- 电子书搜索下载
- 电子书打包资源分享
- 学习资源分享

### 8.3.1 NFC应用示例

Android平台中，NFC应用的类型和NFC三种运行模式有关，我们先来看一个使用NFC R/W模式读取NFC Tag的示例。

#### 1. NFC R/W模式示例

根据前文对NFC基础知识的介绍可知，和R/W模式相关的应用场景就是使用者利用NFC手机（充当NFC Reader的角色）来读取目标NFC Tag中的信息。Android平台为NFC R/W模式设计了“Tag分发系统”（Tag Dispatch System）的机制，描述了NFC系统模块如何向应用进程分发与目标NFC Tag相关的Intent（该Intent中包含了Tag中的数据或是一个代表目标NFC Tag的Tag对象）。

##### (1) NFC Tag分发系统

Tag分发系统的工作机制如图8-27所示。

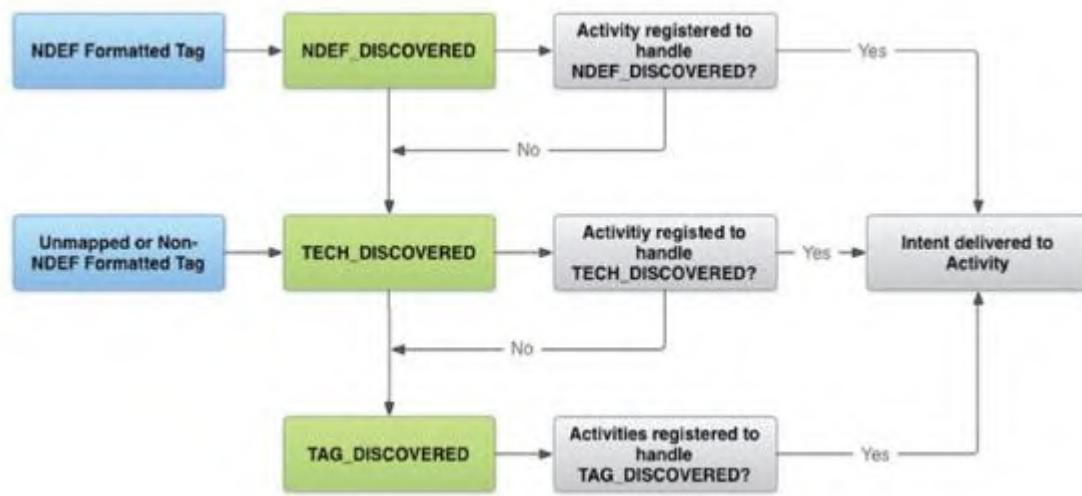


图8-27 Tag分发系统的工作机制

Tag分发系统的工作机制如下。

- 1) 当本机扫描到一个NFC Tag后，NFC系统模块将首先尝试直接读取该Tag中的数据。

2) 如果这些数据封装在NDEF消息中并且能映射成Android系统直接支持的数据类型（目前仅支持MIME和URI这两大类数据类型，详情见表8-11），则NFC系统模块将发送一个ACTION\_NDEF\_DISCOVERED的Intent给那些注册了对ACTION\_NDEF\_DISCOVERED通知感兴趣的Activity。如果找到目标Activity，则将此Intent（携带Tag中的NDEF消息和一个代表该NFC Tag的Tag对象）派发给它。如果NFC系统模块没有找到目标Activity，则将尝试发送一个ACTION\_TECH\_DISCOVERED的Intent（包含一个代表目标NFC Tag的Tag对象）。

3) 如果NFC Tag中的数据不能转换成系统直接支持的类型，或者NFC Tag中的数据没有使用NDEF消息格式或者没有目标Activity对ACTION\_NDEF\_DISCOVERED通知感兴趣，则NFC系统模块将发送一个ACTION\_TECH\_DISCOVERED的Intent（包含一个Tag对象）。如果找到对该Intent感兴趣的Activity，则此Intent将派发给它。

4) 如果没有Activity对ACTION\_TECH\_DISCOVERED感兴趣，则NFC系统模块将最后尝试发送一个ACTION\_TAG\_DISCOVERED的Intent（包含一个Tag对象）。如果有对ACTION\_TAG\_DISCOVERED感兴趣的Activity，则此Intent将派发给它。

上述的Tag分发系统看起来很复杂，实际上其核心内容可概况成三个步骤。

步骤1 如果目标NFC Tag包含了系统支持的NDEF消息，则NFC系统模块将直接把这个NDEF消息分发给感兴趣的Activity。如果有目标Activity，则直接分发给它，否则转步骤2。分支转换的判断标准是NFC Tag是否包含了系统支持的NDEF消息以及同时是否有目标Activity注册了ACTION\_NDEF\_DISCOVERED通知。

步骤2 如果目标NFC Tag包含了系统不支持的NDEF消息或者步骤1中没有目标Activity，则NFC系统模块将尝试分发一个ACTION\_TECH\_DISCOVERED通知。NFC系统模块在分发此通知时，将首先分析目标NFC Tag所支持的Tag Technology（它代表目标NFC Tag所使用的技术，详情见下文分析），然后寻找注册了支持这些Tag Technology的目标Activity并将Intent分发给它。如果没有合适的目标Activity，则转入步骤3。

步骤3 NFC系统模块将分发ACTION\_TAG\_DISCOVERED通知给注册了对该通知感兴趣的目标Activity。

除了Tag分发系统外，Android系统还有一个“前台分发系统”（Foreground Dispatch System）。其规则和Tag分发系统类似，二者区别主要集中在选择目标Activity上。

- Tag分发系统中，Activity在其AndroidManifest.xml中设置Intent分发条件，即设置对应的IntentFilter。在这种分发系统中，不考虑目标Activity是否在前台还是后台。只要找到目标Activity，NFC系统就会启动它。
- 前台分发系统中，当前活跃（即所谓的前台）的Activity在其启动过程中设置Intent分发条件。如果NFC Tag满足前台Activity设置的分发条件，NFC系统模块首先会把Intent分发给前台这个Activity。当该Activity退到后台时，它需要取消前台分发功能，即它不再是目标Activity。

简而言之，前台分发系统只检查当前显示的Activity是否满足分发条件，而Tag分发系统则会搜索系统内所有满足条件的Activity。

## (2) Tag分发通知

下面我们分别来看看这三个不同作用的Tag分发通知。

1) ACTION\_NDEF\_DISCOVERED：由上文可知，NFC系统模块首先尝试将NFC Tag中的数据映射成系统直接支持的数据格式。表8-11列举了Android系统直接支持的NFC Forum数据格式。

表 8-11 Android 系统直接支持的 NFC Forum 数据格式

NFC Forum 中的格式	Android 系统映射后的数据格式
Absolute URI	URI
NFC Forum External Type	原格式为类似 <domain>:<service> 这样的 URI，映射后变成 vnd.android.nfc:// ext/<domain>:<service>
MIME	MIME
NFC Forum Well-Known Type	<ul style="list-style-type: none"><li>• TEXT Record Type 映射成 “text/plain”</li><li>• URI Record Type 映射成 URI 格式</li><li>• Smart Poster Record Type 也映射成 URI 格式</li></ul>

由表8-11可知，如果NFC Tag中数据格式能映射成功，则NFC系统模块将发送一个ACTION\_NDEF\_DISCOVERED Intent给目前Activity，而该Intent将包含此NFC Tag中的NDEF消息。

除了NFC Forum定义的数据类型外，Android还新增了一个名为AAR（Android Application Record）的数据类型，它其实是在一个NDEF的消息中封装了某个应用的package名。对AAR来说，分发系统的工作流程如下。

- 分发系统首先尝试使用IntentFilter来寻找目标Activity。如果和IntentFilter匹配的Activity同时和AAR匹配（即二者的package名一样），就启动该Activity。
- 如果Activity跟AAR不匹配，或者是有多个Activity能够处理该Intent，或者是没有能够处理该Intent的Activity，NFC系统模块将启动由AAR指定的应用程序。
- 如果系统中没有安装该AAR对应的应用程序，NFC系统模块将从Google Play下载该应用程序。

AAR的好处是能让某个公司部署的NFC标签只能由该公司开发的客户端（通过在NDEF中设置AAR）来处理。后文代码分析时候读者还将看到上述AAR的工作流程。

2) ACTION\_TECH\_DISCOVERED：如果系统不能映射NFC Tag中的数据，我们该如何处理呢？

提示 ACTION\_TECH\_DISCOVERED触发的另一个原因是没有Activity对ACTION\_NDEF\_DISCOVERED感兴趣。

该问题的直观答案就是应用程序自己去读取并解析Tag中的数据。不过，由于NFC Tag的类型有四种之多，甚至同一个厂商还生产了基于不同底层协议的NFC Tag，导致Android系统无法提供一种通用的接口来操作所有种类的NFC Tag。为了解决此问题，Android提供了一个名为“android.nfc.tech”的Java包来帮助应用程序操作对应的NFC Tag。表8-12为android.nfc.tech包中的几个重要成员类。

表 8-12 android.nfc.tech 支持的 Tag 种类

类 名	说 明
NfcA	用于操作 NFC-A (ISO 14443-3A) 类型的 Tag
NfcB	用于操作 NFC-B (ISO 14443-3B) 类型的 Tag
NfcF	用于操作 NFC-F (Felica/JIS 6319-4X) 类型的 Tag
NfcV	用于操作 NFC-V (ISO 15693) 类型的 Tag
IsoDep	用于操作 ISO-DEP (ISO 14443-4) 类型的 Tag
NdefFormattable	能将其中数据转换成 NDEF 消息的 Tag。由于 NFC Forum 没有相关规范，所以目前还没有固定的方法知道哪些 Tag 种类能将其包含的数据转换成 NDEF 消息
NfcBarcode	处理数据信息为二维码的 Tag
MifareClassic	处理 MIFARE Classic Tag，它为 NXP 公司 NFC 芯片系列中的一种
MifareUltralight	处理 MIFARE Ultralight Tag，它为 NXP 公司 NFC 芯片系列中的一种
Ndef	处理 NFC Type1 到 Type4 的 Tag，该类提供了往这些 Tag 中写数据的方法

NFC 系统模块将在 ACTION\_TECH\_DISCOVERED Intent 中携带一个 Tag 对象，应用程序可调用该 Tag 对象的 getTechList 来获取该 Tag 所使用的 Technology。注意，一个 Tag 可能同时支持表 8-12 中多种 Technology。例如图 8-28 所示为笔者测试北京市公交卡时所得到的 Tag Technology 信息。

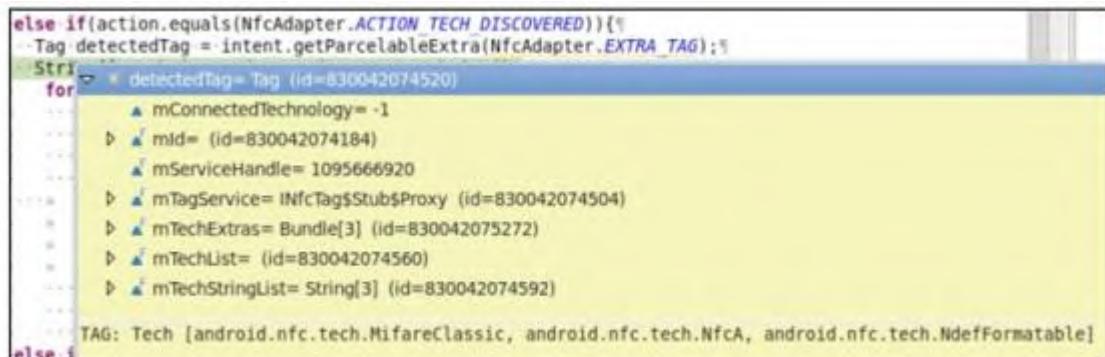


图 8-28 北京市公交卡 Tag Technology 示例

由图 8-28 可知，北京市公交卡同时支持 MifareClassic、NfcA 和 NdefFormattable 这三种类型的 Tag Technology。应用程序接着可根据目标 NFC Tag 所支持的 Tag Technology 来创建表 8-12 中的对象来和 NFC Tag 交互。

特别注意 严格来说，表 8-12 中所列的类不仅仅是用来读写对应类型的 NFC Tag，它还支持一些控制操作以至于能在 NFC Tag 上实现一些特定的协议。以北京市公交卡为例，其内部肯定有一个相关的协议使得

应用程序可通过这些协议来完成公交卡充值，付费等操作。“小木公交”软件即可读取多个城市公交卡的信息，读者不妨下载试试。

3) ACTION\_TAG\_DISCOVERED：如果目标NFC Tag不属于表8-12中的一种，则NFC系统模块将发送ACTION\_TAG\_DISCOVERED Intent并携带一个Tag对象传递给感兴趣的Activity。Activity将根据Tag的ID（调用Tag的getId函数）或该Tag使用的技术（调用Tag的getTechList）来创建合适的处理对象。

下面通过一个示例来了解上述三种通知的用法。

### (3) 示例分析

本节将通过一个前台分发示例来看看应用程序如何处理上述三种Intent。

[-->ForegroundDispatch.java: : onCreate]

```
public class ForegroundDispatch extends Activity { //  
    ForegroundDispatch是一个Activity  
    .....// 定义一些成员变量  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        // NFC客户端必须调用下面这个函数以获得一个NfcAdapter对象，该对象用于和NFC系统模块交互  
        mAdapter = NfcAdapter.getDefaultAdapter(this);  
        // 构造一个PendingIntent供NFC系统模块派发  
        mPendingIntent = PendingIntent.getActivity(this, 0,  
            new Intent(this,  
                getClass()).addFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP), 0);  
        // 监听ACTION_NDEF_DISCOVERED通知，并且设置MIME类型为“*/*”  
        // 对任何MIME类型的NDEF消息都感兴趣  
        IntentFilter ndef = new  
        IntentFilter(NfcAdapter.ACTION_NDEF_DISCOVERED);  
        ndef.addDataType("*/*");  
  
        // 我们同时还监听ACTION_TECH_DISCOVERED和ACTION_TAG_DISCOVERED  
        // 通知  
        mFilters = new IntentFilter[] {  
            ndef,  
            new  
            IntentFilter(NfcAdapter.ACTION_TECH_DISCOVERED),  
            new
```

```

IntentFilter(NfcAdapter.ACTION_TAG_DISCOVERED),
};

// 对ACTION_TECH_DISCOVERED通知来说，还需要注册对哪些Tag
Technology感兴趣
mTechLists = new String[][] {
    new String[] { NfcF.class.getName() }, // 假设本
例支持NfcF
    new String[] { MifareClassic.class.getName() } }; // 假设本例支持MifareClassic
}
}
}

```

在上述onCreate函数中，同时监听了三种Tag Intent通知，最终效果如下。

- 如果目标Tag中包含MIME类型的NDEF消息，则Tag分发系统将给我们传递一个ACTION\_NDEF\_DISCOVERED Intent。
- 如果目标Tag使用的Tag Technology为NfcF或MifareClass，则Tag分发系统将给我们传递一个ACTION\_TECH\_DISCOVERED Intent。
- 最后，Tag分发系统将不满足上述条件的其他所有Tag通过ACTION\_TAG\_DISCOVERED Intent传递给我们。

接下来，由于本例使用了NFC的前台分发系统，故需要将onCreate中设置的配置信息传递给NFC系统模块，相关代码如下所示。

[-->ForegroundDispatch.java: : onResume]

```

public void onResume() {
    super.onResume();
    // 调用NfcAdapter的enableForegroundDispatch函数启动前台分发系统
    // 同时需要将分发条件传递给NFC系统模块
    mAdapter.enableForegroundDispatch(this, mPendingIntent,
    mFilters, mTechLists);
}

```

当NFC系统模块扫描到一个NFC Tag时，前台分发系统通过将触发ForegroundDispatch这个Activity的onNewIntent函数，该函数的代码如下所示。

[-->ForegroundDispatch.java: : onNewIntent]

```
public void onNewIntent(Intent intent) {  
    String action = intent.getAction();  
    // 处理ACTION_NDEF_DISCOVERED消息  
    if(action.equals(NfcAdapter.ACTION_NDEF_DISCOVERED)){  
        NdefMessage[] ndefMsgs = null;  
        // 获取该Intent中的NdefMessage数组。绝大部分情况下该数组的长度为1  
        NdefMessage[] ndefMsgs = (NdefMessage[]) intent.  
  
        getParcelableArrayExtra(NfcAdapter.EXTRA_NDEF_MESSAGES);  
    } // 处理ACTION_TECH_DISCOVERED通知  
    else if (action.equals(NfcAdapter.ACTION_TECH_DISCOVERED)) {  
        // 获取该Intent中的Tag对象  
        Tag detectedTag =  
        intent.getParcelableExtra(NfcAdapter.EXTRA_TAG);  
        String[] techList = detectedTag.getTechList(); // 获取该  
        Tag使用的Technology  
        for(String tech: techList){  
            if(tech.equals(NfcF.class.getName())){// 假设该Tag支  
持NfcF  
                // 创建NfcF对象和该Tag交互  
                NfcF nfcF = NfcF.get(detectedTag);  
                nfcF.connect(); // 向目标Tag发起I/O操作前需要先连接上  
                它  
                .....// 调用NfcF类的其他函数，例如transceive向NFC  
                Tag发送命令  
                nfcF.close(); // 关闭连接  
            }.....  
        }  
    } // 处理ACTION_TAG_DISCOVERED通知  
    else if(action.equals(NfcAdapter.ACTION_TAG_DISCOVERED)){  
        Tag tag =  
        (Tag) intent.getParcelableExtra(NfcAdapter.EXTRA_TAG);  
        String[] arry = tag.getTechList();  
        .....// 根据Tag使用的Technology来构造相应的处理对象  
    }  
}
```

当ForegroundActivity退出时，需要在onPause函数中停止使用前台分发系统，相关代码如下所示。

```
public void onPause() {  
    super.onPause();
```

```
mAdapter.disableForegroundDispatch(this); // 停止前台分发系统  
}
```

通过上述介绍可知，在R/W模式中：

- 如果应用程序仅用于读取NFC Tag中所包含的数据，则应尽量通过注册ACTION\_NDEF\_DISCOVERED通知来获取自己感兴趣的数据。
- 如果应用程序希望能和NFC Tag交互以实现自己的一套协议或者希望能直接读写NFC Tag，则可通过注册ACTION\_TECH\_DISCOVERED通知来获得代表NFC Tag的Tag对象。应用程序接着要根据该Tag使用的Technology来构造对应的TagTechnology对象来操作此NFC Tag。
- 如果应用程序处理的NFC Tag不满足表8-12中的一种，则需要监听ACTION\_TAG\_DISCOVERED通知，然后再构造自己的TagTechnology对象来操作此NFC Tag。

提示 关于Android中NFC的分发系统，请读者阅读Android SDK关于NFC的介绍，相关资料位于  
<http://developer.android.com/guide/topics/connectivity/nfc/nfc.html>。

## 2. NFC P2P模式示例

Android平台中的NFC P2P模式使用了前文介绍的SNEP协议。在SNEP协议基础上，Android设计了“Android Beam”技术架构，该架构使得NFC客户端程序能非常容易得在两个NFC设备间传递NDEF消息。

提示 除了SNEP外，Android还定义了一个与之类似的NPP（Ndef Push Protocol），该协议对应的服务端SAP为0x10，服务名为“com.android.npp”。Android Beam中，系统首先使用SNEP进行传输。如果一些老旧的设备不支持SNEP，则系统将使用NPP。

和R/W模式一样，Android Beam的使用也需要绑定到一个Activity中，下面我们直接通过一个例子来看看如何使用Android Beam。

### （1）发送端处理

```
[-->Beam.java: : onCreate]
```

```

public class Beam extends Activity implements // Beam是一个
Activity
    CreateNdefMessageCallback, // 用于从源Activity中得到需要传
递的NDEF消息
    OnNdefPushCompleteCallback // 用于通知NDEF消息传送完毕
{
    NfcAdapter mNfcAdapter;
    TextView mInfoText;
    private static final int MESSAGE_SENT = 1;
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        mInfoText = (TextView) findViewById(R.id.textView);
        // 得到一个NfcAdapter对象，用于和NFC系统模块交互
        mNfcAdapter = NfcAdapter.getDefaultAdapter(this);
        /*
        下面这两个函数调用非常重要。
        setNdefPushMessageCallback: 设置一个回调对象。如果该回调对象
不为空，则NFC系统模块将
        为Beam这个Activity启用Android Beam。该回调对象的作用是：当NFC
系统模块通过SNEP协议
        发现另外一个NFC设备时，系统会弹出如图8-29所示的“数据发送通知
框”。如果用户选择本机发送数据，
        则NFC系统模块将通过这个回调对象获取需要发送的数据。
        setOnNdefPushCompleteCallback: 设置一个数据发送完毕通知回调
对象，当NFC系统模块发送完
        本Activity所设置的NDEF消息时，该回调对象对应的函数将被调用。
    */
    mNfcAdapter.setNdefPushMessageCallback(this, this);
    mNfcAdapter.setOnNdefPushCompleteCallback(this, this);
}

```

如图8-29所示，左图为数据发送通知框。在Android平台中，两个互相靠近的NFC设备都会弹出类似左图这样的数据发送通知框。至于最终是谁来发送数据则需要用户点击触摸屏来决定。当对端设备收到示例Beam所发送的NDEF消息后，对端设备的NFC系统模块将解析该NDEF消息然后通过Tag分发系统或前台分发系统找到目标Activity。图8-29右图即为对端设备收到本例中Beam发送的数据后的处理结果。



图8-29 Beam示例截图

下面来看createNdefMessage函数，它实现了CreateNdefMessageCallback接口类。当用户在图8-29左图中点击触摸屏后，NFC系统模块将通过该函数获取应用程序需要发送的NDEF消息，其代码如下所示。

[-->Beam.java: : createNdefMessage]

```
public NdefMessage createNdefMessage(NfcEvent event) {  
    Time time = new Time();  
    time.setToNow();  
    // 设置一些信息  
    String text = ("Beam me up!\n\n" +"Beam Time: " +  
    time.format("%H:%M:%S"));  
    // 构造一个MIME Type的NDEF消息  
    NdefMessage msg = new NdefMessage(NdefRecord.createMime(  
        "application/com.example.android.beam",  
        text.getBytes()));  
    return msg;  
}
```

当本机NFC系统模块成功发送了NDEF消息后，onNdefPushComplete将被调用以通知数据发送的情况。在Beam示例中，该函数的代码如下所示。

[-->Beam. java: : onNdefPushComplete]

```
public void onNdefPushComplete(NfcEvent arg0) {
    // 发送一个MESSAGE_SENT消息。注意，onNdefPushComplete运行在
    Binder线程
    mHandler.obtainMessage(MESSAGE_SENT).sendToTarget();
}
```

## (2) 接收端处理

当对端NFC设备接收到此NDEF消息时，将通过Tag分发系统来处理它。Beam示例在其AndroidManifest. xml设置了如图8-30所示的IntentFilter。

```
<intent-filter>
    <action android:name="android.nfc.action.NDEF_DISCOVERED"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <data android:mimeType="application/vnd.com.example.android.beam"/>
</intent-filter>
```

图8-30 Beam设置的IntentFilter

根据前文对Tag分发系统的介绍，对端设备的Beam将被启动。启动过程中几个重要函数的代码如下所示。

[-->Beam. java: : onNewIntent/onResume/processIntent]

```
public void onNewIntent(Intent intent) {
    setIntent(intent); // Beam使用了SINGLE_TOP启动模式。setIntent
    用于保存Intent
}
// Beam启动时，onResume将被调用
public void onResume() {
    super.onResume();
    if
(NfcAdapter.ACTION_NDEF_DISCOVERED.equals(getIntent().getAction()
())) {
        processIntent(getIntent()); // getIntent获取setIntent
        设置的那个Intent对象
    }
}
```

```
// 处理ACTION_NDEF_DISCOVERED通知
void processIntent(Intent intent) {
    // 取出对端Beam发送的NDEF消息
    Parcelable[] rawMsgs =
        intent.getParcelableArrayExtra(NfcAdapter.EXTRA_NDEF_MESSAGES);
    NdefMessage msg = (NdefMessage) rawMsgs[0];
    // 设置Text控件，最终结果如图8-29右图所示
    mInfoText.setText(new String(msg.getRecords()
[0].getPayload())));
}
```

至此，通过一个示例展示了NFC客户端程序如何使用Android Beam技术。Android Beam的本质是利用NFC P2P模式的SNEP协议在两个NFC设备间传递NDEF消息。除了NfcAdapter的setOnNdefPushCompleteCallback函数外，NfcAdapter还有其他方式能发送NDEF消息。关于这部分内容，请读者务必阅读SDK中的介绍(<http://developer.android.com/guide/topics/connectivity/nfc/nfc.html#p2p>)。

另外，Android Beam除了能发送NDEF消息外，它还支持发送URI Scheme为“file”或“content”类型的数据，也就是文件或数据库中的内容。这些数据的量可能比较大，所以Android Beam将使用Handover并选择蓝牙来传输它们。

### 3. NFC CE模式示例

在Android平台中，NFC CE的使用比较特殊，主要体现在两点。

- Android SDK没有直接提供Card Emulation相关的API，但Android系统内部提供了一个名为“com.android.nfc\_extras.jar”的Java动态库。在这个动态库中，Android封装了和CE相关的API。应用程序需要主动加载这个nfc\_extras库才能使用CE模式。
- 由于CE通常用于支付等方面的工作，所以Android系统在nfc\_extras动态库的使用上有着非常严格的权限管理。

下面介绍相关知识。

#### (1) nfc\_extras和nfcee\_access.xml

根据上文所述，应用程序如何才能使用CE模式呢？我们先来看看如何在应用程序中使用nfc\_extras动态库。图8-31所示的AndroidManifest.xml指明了必要的做法。

```
<uses-permission android:name="android.permission.NFC" />
<application android:icon="@drawable/ic_launcher">
    <!-- android:label="@string/app_name" -->
    <uses-library android:name="com.android.nfc_extras" />
```

图8-31 AndroidManifest.xml设置

使用NFC CE的应用必须通过<uses-permission>标签申明“android.permission.NFC”权限。同时还需通过<uses-library>申明使用动态库“com.android.nfc\_extras”。这样，当应用程序运行时，系统会为它加载com.android.nfc\_extras.jar包。该包对应的文件位于/system/framework目录下。

接着，客户端在需要使用nfc\_extras API的Java类文件中通过import语句导入相关的类，如图8-32所示。

```
import com.android.nfc_extras.NfcAdapterExtras;
import com.android.nfc_extras.NfcAdapterExtras.CardEmulationRoute;
import com.android.nfc_extras.NfcExecutionEnvironment;
```

图8-32 nfc\_extras动态库相关类

nfc\_extras主要包含三个类，其用法将留待下节的示例代码中再来介绍。

客户端导入相关类后，下一步要解决的问题就是编译。由于Android SDK没有提供这些类，故需要手动解决编译问题。目前有两种方法解决。

一种方法是为应用程序编写Android.mk，然后添加以下内容。

```
LOCAL_JAVA_LIBRARIES := com.android.nfc_extras
```

该方法要求在Android源码下编译此应用程序。

另外一种方式是在Eclipse中为应用程序手动添加一个编译路径，如图8-33所示。

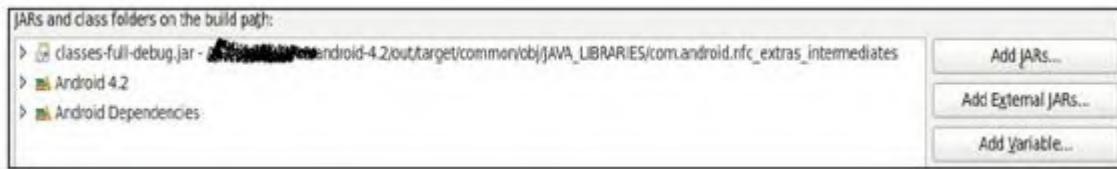


图8-33 Eclipse设置编译路径

图8-33中，笔者在测试示例中添加了nfc\_extras动态库（即classes-full-debug.jar，它是Android系统编译nfc\_extras时生成的中间JAR文件包）。

注意 无论哪种方法，都需要有Android系统的源码。

通过上述步骤，应用程序可编译成功，但它此时依然没有权限操作NFC CE。这是因为在Android系统中，除了“android.permission.NFC”权限外，NFC系统模块针对NFC CE这种重要运行模式还需要检查另外一个权限，即客户端程序的签名信息。只有拥有系统指定签名的应用程序才能使用NFC CE模式。

Android系统所指定的签名信息都保存在/etc/nfcee\_access.xml文件中，图8-34所示为Galaxy Note 2中该文件的内容。

```

<resources xmlns:xliff="urn:oasis:names:tc:xliff:document:1.2">
    <!-- Applications granted NFCEE access on user builds
        See packages/apps/Nfc/etc/sample_nfcee_access.xml for full documentation.
        -->

    <!-- Google wallet release signature -->
    <signer android:signature="3082044c30820334a003020102020900a8cd17c93da5d990300d06092a
407130d4d6f756e7461696e205669657731143012060355040a130b476f6f676c6520496e632e3110300e0603
8303830393031303635335a3077310b3009060355040613025553311330110603550408130a43616c69666f72
50355040b1307416e64726f6964311330110603550403130a476f6f676c65204e464330820120300d06092a86
ad870dee821a53e1f5b170fc96245a3c982a7cb4527053be35e34f396d24b2291ec0c528d6e26927465e66875
fe0621f918d1f35a82489252c6fa6b63392a7686b3e48612d06a9cf6f49bff11d5d96289c9dfe14ac57624396
50b705a86e6e83ee2ae37fe5701bcdb26feefdf860f6a5bdfb5b64793020103a381dc3081d9301d0603551d
30a0c72e08cc96da17ba4793077310b3009060355040613025553311330110603550408130a43616c69666f72
50355040b1307416e64726f6964311330110603550403130a476f6f676c65204e4643820900a8cd17c93da5d9
96b9843724931d75d4ca10c321520d33ccfed2aa65462234c9ef9b6f910cc676b99cb7f9895d6c06763574fb
873dbe64b9c9e74f8f2c2f6c40124aaa8d17880d18512b540add28b3e9581971a4170dd868cf5f31e44712b2c2
9586a5ccb7faf7198a84bcf744310e9e927597f00a23dd00660800c2238d90b2fb372dfdbba75bd852e" />

    <!-- Samsung wallet signature -->
    <signer android:signature="3082037830820260a00302010202044fd98476300d06092a864886f70d
875776f6e2043697479311c301a060355040a131353616d73756e6720436f72706f726174696f6e310b300906
13339313033313036323830365a307e310b3009060355040613024b52311430120603550408130b536f757468
4696f6e310b3009060355040b13024547311930170603550403131053616d73756e67204e4643204365727430
ecacf22c3a8c402670f7777a776f367da3756e31c8ac72a9801439236db9cc5cdf11509531648d9f82ed08e31
beff6c436236e76bfebdb288ed0f30f97b1d9a5a509c3d8e3fa402ecf74b2d4be0cb70d5041310e18a3a29ec41
:c0b20c27a555f3b8818404a50eb055eb61552d711f6a7bb551cae4d2b6436f69d63ffdb2b64ad020301000130
13745e05fa086d9fd784b8735145670910ffa36f2617bd4be1d24a9ea25c6c8705cdd27c1303b8ac13a3426a9
:40b6dff9c54b22afce0ddc3680f05c70183f70876d5cb25c7845fe5970cc90aa71708870915a8acfcd87a27
ff4ebea47fdb3d24fc31c4c1f8cf0c1fddb7998af2bc697bf" />
<signer android:signature="308201e53082014ea0030201020204523a504d300d06092a864886f70d0101
4204465627567301e170d3133303931393031313535375a170d3433303931323031313535375a3037310b3009
f300d06092a864886f70d010101050003818d0030818902818100977ed4e012bf23af7f8205977ae9720d2f20
84983337744f2f49743d421e8d057add7703d2fb622f9b63b416e8b84d964f852817e726b3a26fd3204f2d308
b7d1b5553d2c5ea23587995f9207f1ccc1b25f5b5cf4add5c1fc36c8f24ablebefc3686492b460a3bc1d0198f
2fc40d95a4f9045c0d18500e6">
    <package android:name="com.example.android.beam" />
</signer>

```

图8-34 nfcee\_access.xml示例

图8-34中所示nfcee\_access.xml包含三个签名（由signer标签指定，图中由黑框标示）。

- 第一个签名为Google Wallet相关应用拥有。
- 第二个签名为Samsung Wallet相关应用拥有。
- 第三个签名为笔者测试时用的签名信息。

签名信息检查的工作流程如下。

1) 客户端程序开展CE相关操作前，必须先获得一个NfcAdapterExtras对象。

2) 在获取该对象时，NFC系统模块先检查应用程序是否拥有"android.permission.NFC"权限，接着检查该应用程序的签名信息。只有调用程序的签名信息在nfcee\_access.xml有记录，该应用才能得到一个NfcAdapterExtras对象。

提示 显然，nfcee\_access.xml要么由手机厂商在出厂前设置，要么在root的手机上修改。

了解上述知识后，通过一个示例来介绍nfc\_extras相关的API及使用方法。

## (2) 示例分析

本节使用的示例通过修改AndroidBeamDemo而来。

[-->示例]

```
class NfcEEActivity extends Activity{
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    try {// 先从PackageManager那获取自己的签名信息
        PackageManager pm = getPackageManager();
        PackageInfo info =
pm.getPackageInfo("com.example.android.beam",
                           PackageManager.GET_SIGNATURES);
        // 将签名信息打印出来，开发者需要将此签名信息保存
        // 到/etc/nfcee_access.xml中
        Log.e("NfcEE", "signature = " +
info.signatures[0].toCharsString());
        mContext = this;
        // 调用NfcAdapterExtras.get函数获取一个NfcAdapterExtras
        对象
        mAdapterExtras =
NfcAdapterExtras.get(NfcAdapter.getDefaultAdapter(
                           mContext));
        /*
        获取与NFC芯片中Execution Environment模块交互的对象。注意：
        此处的EE一般情况下就是指
        Secure Element。以图8-21 NXP pn65芯片模块图为例，SE可以是
```

其内部的SmartMX模块，

也可以是外部的UICC。对UICC来说，不是所有手机都支持将UICC连接到NFC芯片。除此之外，

通过NFC操作UICC还需要相关驱动的支持。

mEe的类型为NfcExecutionEnvironment，通过它可以和SE交互。

```
/*
 * 
 mEe =
mAdapterExtras.getEmbeddedExecutionEnvironment();
}.....
// 创建一个新的线程，相关测试工作放在此线程中进行
Thread testThread = new Thread() {
    public void run() {
        /*
         使用CE模式前需要先设置Route，系统目前有
ROUTE_ON_WHEN_SCREEN_ON（屏幕打开
时启用CE）和ROUTE_OFF（关闭CE）这两种选项。
*/
        mAdapterExtras.setCardEmulationRoute(
            new
CardEmulationRoute(CardEmulationRoute.ROUTE_ON_WHEN_SCREEN_ON,
mEe));
        // 创建和SE交互的通道。该通道创建后，NFC其他功能将被禁止（如R/W
或P2P）
        mEe.open();
        /*
         发送命令给EE去执行。SELECT_CARD_MANAGER_COMMAND存储了相关的命
令信息。
注意：EE命令的格式遵循ISO 7816-4规范。不同应用需要具体芯片的情况
使用对应的命令，这部分是
CE模式的难点。
*/
        byte[] out =
mEe.transceive(SELECT_CARD_MANAGER_COMMAND);
        .....
        mEe.close();// 关闭与SE交互的通道
        // 关闭Card Emulation功能
        mAdapterExtras.setCardEmulationRoute(
            new
CardEmulationRoute(CardEmulationRoute.ROUTE_OFF, null));
    }
};

testThread.start(); // 启动工作线程
}
.....// 其他代码
}
```

从上述示例代码来看，`nfc_extras`的API似乎比较简单。但对一个实际应用程序而言，其最大难度却在于处理相关命令上。由于不同NFC芯片以及所使用的SE不同，其定义的命令也不尽相同。关于SE命令的格式，读者可参考ISO 7816-4规范。

至此，我们对NFC CE进行了一些简单介绍，并围绕`nfc_extras`动态库的使用进行了相关讨论。根据笔者的研究，CE模式的内容远比R/W及P2P模式复杂。为此，强烈建议读者继续阅读参考资料[23]、[24]和[25]。

### 8.3.2 NFC系统模块

本节开始时介绍，Android平台中，NFC系统模块运行在com.android.nfc进程中，该进程对应的应用程序文件名为Nfc.apk。NFC系统模块包含的组件非常多，所以通过以下几条分析路线来介绍。

- NFC系统模块的核心NfcService和一些重要成员的作用及之间的关系。
- R/W模式下NFC Tag的处理。
- Android Beam的实现。
- CE模式相关的处理。

#### 1. NfcService介绍

Nfc.apk源码中包含一个NfcApplication类。当该应用启动时，NfcApplication的onCreate函数将被调用。正是在这个onCreate函数中，NFC系统模块的核心成员NfcService得以创建。我们直接来看NfcService的构造函数。

[-->NfcService.java: : NfcService]

```
public NfcService(Application nfcApplication) {
    // NFC系统模块重要成员
    mNfcTagService = new TagService(); // TagService用于和NFC Tag交互
    // NfcAdapterService用于和Android系统中其他使用NfcService的客户端交互
    mNfcAdapter = new NfcAdapterService();
    // NfcAdapterExtrasService用于和Android系统中使用Card Emulation模式的客户端交互
    mExtrasService = new NfcAdapterExtrasService();
    sService = this; mContext = nfcApplication;
    // NativeNfcManager由dhimpl模块实现，用于和具体芯片厂商提供的NFC模块交互
    mDeviceHost = new NativeNfcManager(mContext, this);
    // HandoverManager处理Connection Handover工作
```

```

    HandoverManager handoverManager = new
    HandoverManager(mContext);
    // NfcDispatcher用于向客户端派发NFC Tag相关的通知
    mNfcDispatcher = new NfcDispatcher(mContext,
    handoverManager);
    // P2pLinkManager用于处理LLCP相关的工作
    mP2pLinkManager = new P2pLinkManager(mContext,
    handoverManager,
                mDeviceHost.getDefaultLlcpMiu(),
                mDeviceHost.getDefaultLlcpRwSize());
    // NativeNfcSecureElement用于和SE交互，它也由dhimpl模块实现
    mSecureElement = new NativeNfcSecureElement(mContext);
    mEeRoutingState = ROUTE_OFF;
    /*
     NfceeAccessControl用于判断哪些应用程序有权限操作NFCEE。它将读
取/etc/nfcee_access.xml文件的
内容。nfcee_access.xml内容比较简单，请参考Nfc目录下的
etc/sample_nfcee_access.xml来学习。
 */
    mNfceeAccessControl = new NfceeAccessControl(mContext);
    .....
    // 向系统注册一个“nfc”服务。注意，SERVICE_NAME的值为“nfc”。该服务对
应的对象为mNfcAdapter
    ServiceManager.addService(SERVICE_NAME, mNfcAdapter);
    /*
     注册广播事件监听对象。NfcService对屏幕状态、应用程序安装和卸载等广播
事件感兴趣。这部分内容请读者
自行研究。
 */
    .....
    // EnableDisableTask为一个AsyncTask，TASK_BOOT用于NfcService其他
初始化工作
    new EnableDisableTask().execute(TASK_BOOT);
}

```

由上述代码可知，NfcService在其构造函数中，首先创建了NFC系统模块的几个核心成员。下文将详细介绍它们的作用及之间的关系。  
NfcService向Binder系统添加了一个名为“nfc”的服务，该服务对应的Binder对象为mNfcAdapter，类型为NfcAdapter。通过一个AysncTask（代码中的EnableDisableTask）完成NfcService其他初始化工作。

下面马上来看NFC系统模块核心成员。

## (1) NfcService核心成员

图8-35所示为NfcAdapter、TagService等相关成员的类信息。

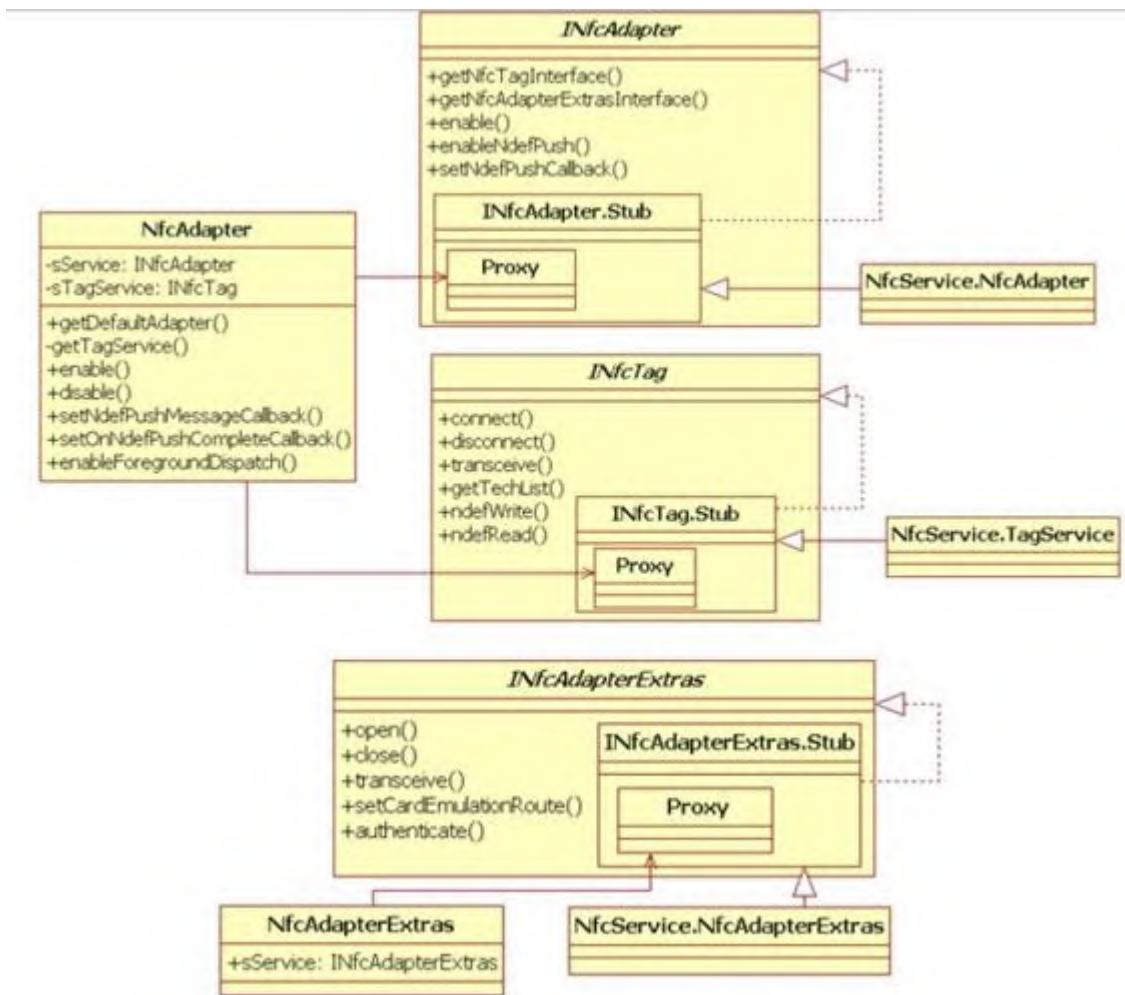


图8-35 NfcAdapter、TagService及相关类成员结构图

图8-35中，NfcAdapter包含一个类型为INfcAdapter的sService成员变量，该变量通过Android Binder机制来和NfcService内部类NfcAdapter的实例（即上述代码中的mAdapter）交互。NfcService内部的NfcAdapter对象即是注册到Android系统服务中的那个名为“nfc”的Binder服务端对象。

NfcAdapter还包含一个类型为INfcTag的sTagService成员变量，该变量通过Android Binder机制来和NfcService内部类TagService的实例（即上述代码中的mNfcTagService）交互。INfcTag接口封装了对Tag

操作相关的函数。注意，前面所示的Nfc客户端示例中并没有直接使用INfcTag接口的地方，但表8-12所列的各种Tech类内部需要通过ITag接口来操作NFC Tag。

NfcAdapterExtras包含一个类型为INfcAdapterExtras的sService成员变量，也通过Android Binder机制来和NfcService内部类NfcAdapterService的实例（即上述代码中的mExtrasService）交互。

接着来看NfcService和NativeNfcManager，它们的类家族如图8-36所示。

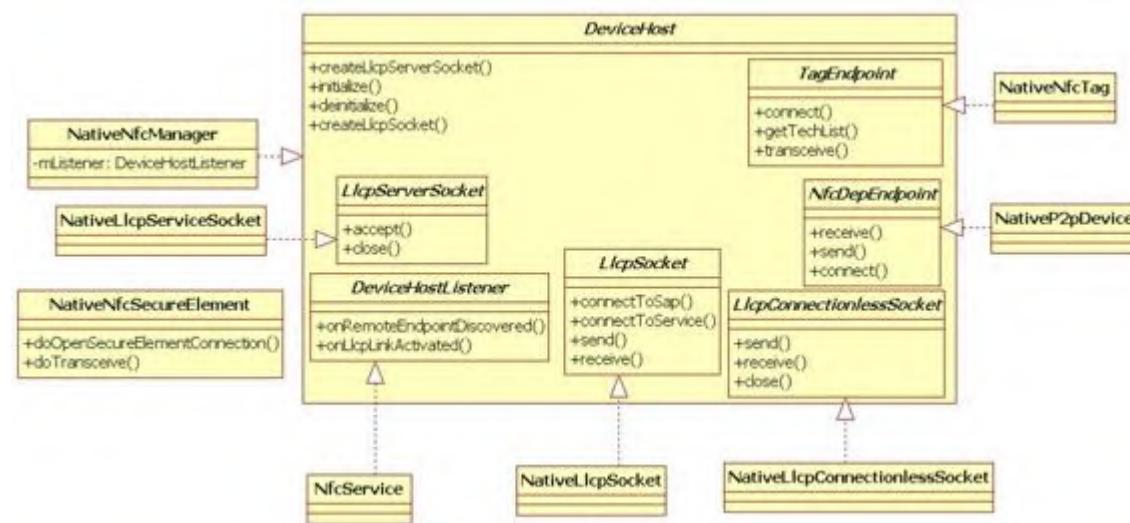


图8-36 NativeNfcManager和NfcService类家族

Android NFC系统模块通过接口类DeviceHost和其内部的接口类L1cpServerSocket、DeviceHostListener、L1cpSocket、L1cpConnectionlessSocket、NfcDepEndpoint、TagEndpoint将NFC系统模块中和NFC芯片无关的处理逻辑，以及和芯片相关的处理逻辑进行了有效解耦。图8-36中以”Native”开头的类均定义在packages/app/Nfc/nxp目录下，所以它们和NXP公司的NFC芯片相关。

DeviceHost接口中，DeviceHost类用于和底层NFC芯片交互，TagEndpoint用于和NFC Tag交互，NfcDepEndpoint用于和P2P对端设备交互，L1cpSocket和L1cpServerSocket分别用于LLCP中有连接数据传输服务的客户端和服务器端，L1cpConnectionlessSocket用于LLCP中无连接数据传输服务。另外，DeviceHostListener也非常重要，它用

于NativeNfcManager往NfcService传递NFC相关的通知事件。例如其中的onRemoteEndpointDiscovered代表搜索到一个NFC Tag、onL1cpActivited代表本机和对端NFC设备进入Link Activation（链路激活）阶段。

NativeNfcManager实现了DeviceHost接口，以NXP公司的NativeNfcManager为例，它将通过libnfc\_jni及libnfc和NXP公司的NFC芯片交互。

NativeNfcSecureElement用来和Secure Element交互。

接下来要出场的是HandoverManager以及P2pLinkManager，它们的家族关系如图8-37所示。

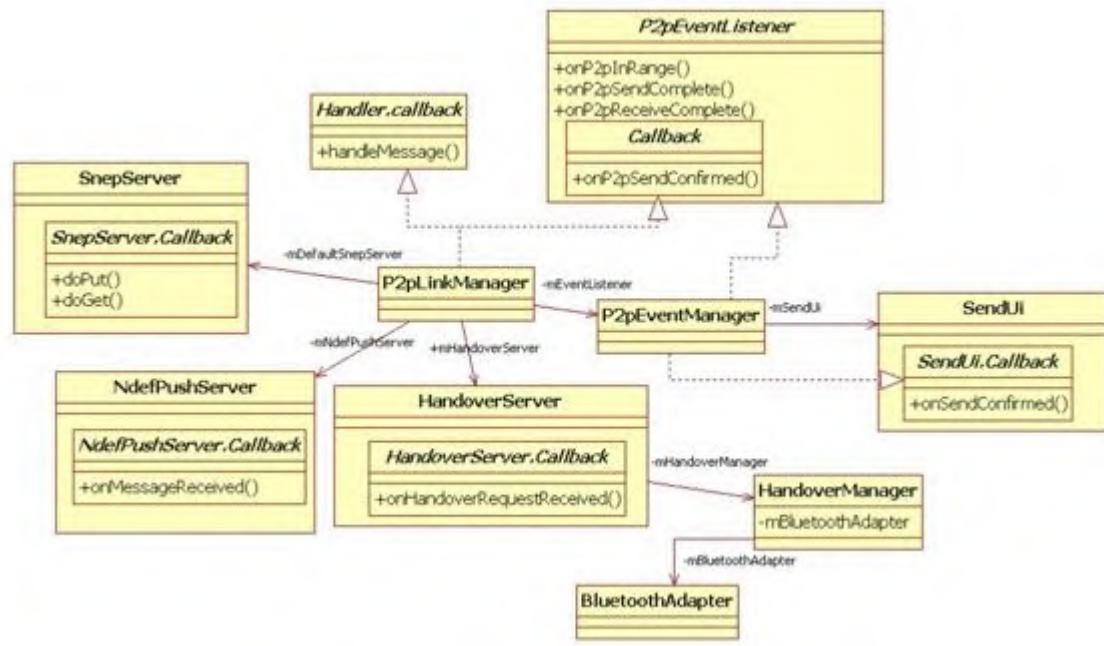


图8-37 P2pLinkManager家族关系

图8-37所示的P2pLinkManager家族关系非常复杂，图中的各成员说明如下。

- P2pLinkManager包含三个和传输相关的Server，分别是SnepServer、NdefPushServer以及HandoverServer。每一个Server都定义了相关的回调接口类（如SnepServer.callback），这些回调接口类的实现均由P2pLinkManager的内部类来实现。

- HandoverServer负责处理NFC Connection Handover协议，而具体的数据传输则由HandoverManager来实现。由图中HandoverManager的mBluetoothAdapter可知，Android默认使用蓝牙来传输数据（提示：这部分内容请读者学完本章后自行研究）。
- P2pEventManager用于处理NFC P2P中和用户交互相关的逻辑。例如当搜索到一个P2P设备时，手机将会震动并发出提示音。

SendUi实现了类似图8-29左图的界面及用户触摸事件处理。在Android平台中，当两个设备LLCP链路被激活时，SendUi将显示出来。其界面组成部分包括两个部分，一个是SendUi界面显示之前手机的截屏图像，另外一个是“触摸即可发送”的文本提示信息。当用户触摸SendUi界面时，数据将被发送出去。

## (2) enableInternal函数

介绍完NfcService的几个核心成员后，马上来看NfcService构造函数中最后创建的EnableDisableTask，由于设置了参数为TASK\_BOOT，故最终被执行的函数为enableInternal。

[-->NfcService.java: : EnableDisableTask: enableInternal]

```
boolean enableInternal() {
    .....
    // 启动一个WatchDog线程用来监视NFC底层操作是否超时
    WatchDogThread watchDog = new
    WatchDogThread("enableInternal",
                    INIT_WATCHDOG_MS);
    watchDog.start();
    try {
        mRoutingWakeLock.acquire();
        try {
            // 初始化NFC底层模块，这部分内容请读者自行阅读
            if (!mDeviceHost.initialize()) {.....}
        } finally {
            mRoutingWakeLock.release();
        }
    } finally {
        watchDog.cancel();
    }
    synchronized(NfcService.this) {
        mObjectMap.clear();
    }
}
```

```

        // mIsNdefPushEnabled判断是否启用NPP协议，可在
        Settings中设置

    mP2pLinkManager.enableDisable(mIsNdefPushEnabled, true);
        updateState(NfcAdapter.STATE_ON);
    }
    initSoundPool(); // 创建SoundPool，用于播放NFC相关事件的
通知音
    applyRouting(true); // 启动NFC Polling流程，一旦搜索到周
围的NFC设备，相关回调将被调用
    return true;
}

```

我们重点关注上面代码中和P2pLinkManager相关的enableDisable函数，其代码如下所示。

[-->P2pLinkManager.java: : enableDisable]

```

public void enableDisable(boolean sendEnable, boolean
receiveEnable) {
    synchronized (this) { // 假设参数sendEnable和receiveEnable
为true
        if (!mIsReceiveEnabled && receiveEnable) {
            /*
             * 启动SnepServer、NdefPushServer和HandoverServer。
             * 下面这三个成员变量均在P2pLinkManager的构造函数中被创建，
             * 这部分内容请读者自行阅读
             * 本章将只分析SnepServer。
             */
            mDefaultSnepServer.start();
            mNdefPushServer.start();
            mHandoverServer.start();
            if (mEchoServer != null) // EchoServer用于测试，以后
            代码分析将忽略它
                mHandler.sendMessage(MSG_START_ECHO_SERVER);
            }.....
            mIsSendEnabled = sendEnable;
            mIsReceiveEnabled = receiveEnable;
        }
    }

```

(3) SnepServer的start函数

SnepServer的start函数将创建一个ServerThread线程对象，其run函数代码如下所示。

[-->SnepServer.java: : ServerThread: run]

```
public void run() { // 注意：为了方便阅读，此处代码省略了synchronized  
和try/catch等一些代码逻辑  
    boolean threadRunning;  
    threadRunning = mThreadRunning;  
    while (threadRunning) {  
        // 创建一个LlcpServerSocket，其中mServiceSap值为0x04，  
mServiceName为“urn:nfc:sn:sne”  
        // mMiу和mRwSize为本机NFC LLCP层的MIU和RW大小。1024为内部缓  
冲区大小，单位为字节  
        mServerSocket =  
NfcService.getInstance().createLlcpServerSocket(mServiceSap,  
                                              mServiceName, mMiу, mRwSize,  
1024);  
        LlcpServerSocket serverSocket;  
        serverSocket = mServerSocket;  
        // 等待客户端的链接  
        LlcpSocket communicationSocket =  
serverSocket.accept();  
        if (communicationSocket != null) {  
            // 获取客户端设备的MIU  
            int miу = communicationSocket.getRemoteMiу();  
            /*  
             * 判断分片大小。mFragmentLength默认为-1。MIU非常重要。例  
如本机的MIU为1024，而  
对端设备的MIU为512，那么本机在向对端发送数据时，每次发送的  
数据不能超过对端  
             */  
            MIU即512字节。  
            int fragmentLength = (mFragmentLength == -1) ?  
                                  miу : Math.min(miу,  
mFragmentLength);  
            // 每一个连接成功的客户端对应一个ConnectionThread，其内  
容留待下文详细分析  
            new ConnectionThread(communicationSocket,  
fragmentLength).start();  
        }  
    }  
    mServerSocket.close();  
}
```

NfcService初始化完毕后，手机中的NFC模块就进入工作状态，一旦有Tag或其他设备进入其有效距离，NFC模块即可开展相关工作。

下面先来分析NFC Tag的处理流程。

## 2. NFC Tag处理流程分析

### (1) notifyNdefMessageListeners流程

当NFC设备检测到一个NFC Tag时，NativeNfcManager的notifyNdefMessageListeners函数将被调用（由libnfc\_jni在JNI层调用），其代码如下所示。

[-->NativeNfcManager.java: : notifyNdefMessageListeners]

```
private void notifyNdefMessageListeners(NativeNfcTag tag) {  
    /*  
     * mListener指向NfcService，它实现了DeviceHostListener接口。  
     * 注意，notifyNdefMessageListeners的参数类型为NativeNfcTag，tag对象  
     * 由jni层直接创建  
     * 并返回给Java层。  
     */  
    mListener.onRemoteEndpointDiscovered(tag);  
}
```

上述代码中，mListener指向NfcService，它的onRemoteEndPointDiscovered函数代码如下所示。

[-->NfcService.java: : onRemoteEndpointDiscovered]

```
public void onRemoteEndpointDiscovered(TagEndpoint tag) {  
    // 注意，onRemoteEndpointDiscovered的参数类型是TagEndpoint  
    // 由图8-36可知，NativeNfcTag实现了该接口  
    sendMessage(NfcService.MSG_NDEF_TAG, tag); // 发送一个  
    MSG_NDEF_TAG消息  
}
```

NfcService的onRemoteEndpointDiscovered将给自己发送一个MSG\_NDEF\_TAG消息。NfcService内部有一个NfcServiceHandler专门用来处理这些消息。其处理函数如下所示。

[-->NfcService.java: : NfcServiceHandler: handleMessage]

```
final class NfcServiceHandler extends Handler {
    public void handleMessage(Message msg) {
        switch (msg.what) {
            .....// 其他消息处理
            case MSG_NDEF_TAG:
                TagEndpoint tag = (TagEndpoint) msg.obj;
                playSound(SOUND_START); // 播放一个通知音
                /*
                    从该NFC Tag中读取NDEF消息，NativeNfcTag的
                    findAndReadNdef比较复杂，
                    其大体工作流程是尝试用该tag支持的Technology来读取其中的
                    内容。
                */
                NdefMessage ndefMsg = tag.findAndReadNdef();
                // 注意下面这段代码，无论ndefMsg是否为空，
                dispatchTagEndpoint都会被调用
                if (ndefMsg != null) {
                    tag.startPresenceChecking(); // 检测目标Tag是否
                    还在有效距离内
                    dispatchTagEndpoint(tag); // 重要函数，详情见下
                    节
                } else {
                    if (tag.reconnect()) { // 重新链接到此NFC Tag
                        tag.startPresenceChecking();
                        dispatchTagEndpoint(tag);
                    }
                }
                break;
            .....
        }
    }
}
```

由上述代码可知，NfcService先调用TagEndpoint的findAndReadNdef函数来读取Tag中的数据，然后NfcService将调用dispatchTagEndpoint做进一步处理。

提示 findAndReadNdef的实现和具体的NFC芯片有关，而NXP公司的实现函数在NativeNfcTag类中，内容比较复杂，感兴趣的读者可以阅读。

## (2) dispatchTagEndpoint流程

代码如下。

```
[-->NfcService.java: : NfcServiceHandler.dispatchTagEndpoint]

private void dispatchTagEndpoint(TagEndpoint tagEndpoint) {
    // 构造一个Tag对象。前面的示例中已经见过Tag。对客户端来说，它代表目标NFC Tag
    Tag tag = new Tag(tagEndpoint.getUid(),
tagEndpoint.getTechList(),
tagEndpoint.getTechExtras(),
tagEndpoint.getHandle(), mNfcTagService);
    registerTagObject(tagEndpoint); // 保存此tagEndpoint对象
    // mNfcDispatcher的类型是NfcDispatcher，调用它的dispatchTag函数来分发Tag
    if (!mNfcDispatcher.dispatchTag(tag)) {.....}
```

```
[-->NfcDispatcher.java: : dispatchTag]
```

```
public boolean dispatchTag(Tag tag) {
    NdefMessage message = null;
    Ndef ndef = Ndef.get(tag); // 构造一个Ndef对象，Ndef属于Tag Technology的一种
    /*
     * 从Ndef获取目标Tag中的NDEF消息。如果目标Tag中保存的是系统支持的NDEF消息，则message不为空。
     */
    特别注意：在前面代码中见到的findAndReadNdef函数内部已经根据表8-11进行了相关处理。
    if (ndef != null) message = ndef.getCachedNdefMessage();

    PendingIntent overrideIntent;
    IntentFilter[] overrideFilters;
    String[][] overrideTechLists;
    // ①构造一个DispatchInfo对象，该对象内部有一个用来触发Activity的Intent
    DispatchInfo dispatch = new DispatchInfo(mContext, tag,
message);
    synchronized (this) {
        // 下面三个变量由前台分发系统相关的NfcAdapter enableForegroundDispatch函数设置
        overrideFilters = mOverrideFilters;
        overrideIntent = mOverrideIntent;
        overrideTechLists = mOverrideTechLists;
    }
    // 恢复App Switch，详情可参考《深入理解Android：卷II》6.3.3节关于
```

resume/stopAppSwitches的介绍

```
resumeAppSwitches();  
  
    // 如果前台Activity启用了前台分发功能，则只需要处理前台分发相关工作即可  
    if (tryOverrides(dispatch, tag, message, overrideIntent,  
                      overrideFilters, overrideTechLists)) return true;  
    // 处理Handover事件  
    if (mHandoverManager.tryHandover(message)) return true;  
  
    // ②下面是Tag分发系统的处理，首先处理ACTION_NDEF_DISCOVERED  
    if (tryNdef(dispatch, message)) return true;  
  
    // 如果tryNdef处理失败，则接着处理ACTION_TECH_DISCOVERED  
    if (tryTech(dispatch, tag)) return true;  
    // 如若tryTech处理失败，则处理ACTION_TAG_DISCOVERED  
  
    // 设置DispatchInfo对象的内部Intent对应的ACTION为  
    ACTION_TAG_DISCOVERED  
    dispatch.setTagIntent();  
    // 首先从PackageManagerService查询对ACTION_TAG_DISCOVERED感兴趣的Activity，如果有则启动它  
    if (dispatch.tryStartActivity()) return true;  
    return false;  
}
```

上述代码中有①②两个重要函数，我们先来看第一个。

[-->NfcDispatcher.java: : DispatchInfo构造函数]

```
public DispatchInfo(Context context, Tag tag, NdefMessage  
message) {  
    // 这个Intent的内容将派发给NFC的客户端  
    intent = new Intent();  
    /*  
        不论最终Intent的Action是什么，NFC系统模块都会将tag对象和tag的ID包含  
        在Intent中传递  
        给客户端。  
    */  
    intent.putExtra(NfcAdapter.EXTRA_TAG, tag);  
    intent.putExtra(NfcAdapter.EXTRA_ID, tag.getId());  
    // 如果NDEF消息不为空，则把它也保存在Intent中  
    if (message != null) {  
        intent.putExtra(NfcAdapter.EXTRA_NDEF_MESSAGES, new  
NdefMessage[] {message});  
    }  
}
```

```

       ndefUri = message.getRecords()[0].toUri();
       ndefMimeType = message.getRecords()[0].toMimeType();
    } else {
       ndefUri = null;
       ndefMimeType = null;
    }
/*
rootIntent用来启动目标Activity。NfcRootActivity是Nfc.apk中定义的一个Activity。
目标Activity启动的过程如下。NFC系统模块先启动NfcRootActivity，然后再由NfcRootActivity启动目标Activity。由于NfcRootActivity设置了启动标志
(FLAG_ACTIVITY_NEW_TASK和
FLAG_ACTIVITY_CLEAR_TASK)，所以目标Activity将单独运行在一个Task中。关于ActivityManager
这部内容，感兴趣的读者可阅读《深入理解Android：卷II》第6章。
*/
rootIntent = new Intent(context, NfcRootActivity.class);
// 将Intent信息保存到rootIntent中。
rootIntent.putExtra(NfcRootActivity.EXTRA_LAUNCH_INTENT,
intent);
rootIntent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK |
Intent.FLAG_ACTIVITY_CLEAR_TASK);
this.context = context;
packageManager = context.getPackageManager();
}

```

接着来看第二个关键函数tryNdef，代码如下所示。

[-->NfcDispatcher.java: : tryNdef]

```

boolean tryNdef(DispatchInfo dispatch, NdefMessage message) {
    if (message == null) return false;
    /* setNdefIntent:
       设置Dispatcher内部intent对象的Action为
ACTION_NDEF_DISCOVERED。
       如果Dispatch对象的ndefUri和ndefMimeType都为null，则函数返回
null。
    */
    Intent intent = dispatch.setNdefIntent();
    if (intent == null) return false;

    // 如果message中包含了AAR信息，则取出它们。AAR信息就是应用程序的包名
    List<String> aarPackages = extractAarPackages(message);

```

```

for (String pkg : aarPackages) {
    dispatch.intent.setPackage(pkg);
    /*
     tryStartActivity先检查目标Activity是否存在以及目标Activity
     的IntentFilter
     是否匹配intent。注意，下面这个tryStartActivity没有参数。
     */
    if (dispatch.tryStartActivity()) return true;
}

// 上面代码对目标Activity进行了精确匹配，如果没有找到，则尝试启动AAR
// 指定的应用程序
if (aarPackages.size() > 0) {
    String firstPackage = aarPackages.get(0);
    PackageManager pm;
    /*
     下面这段代码用于启动目标应用程序的某个Activity，由于AAR只是指定了
     应用程序的包名而没有指定
     Activity，所以getLaunchIntentForPackage将先检查目标应用程序
     中是否有Activity的Category
     为CATEGORY_INFO或CATEGORY_LAUNCHER，如果有则启动它。
     */
    UserHandle currentUser = new
UserHandle(ActivityManager.getCurrentUser());
    pm = mContext.createPackageContextAsUser("android", 0,
                                              currentUser).getPackageManager();
    Intent appLaunchIntent =
pm.getLaunchIntentForPackage(firstPackage);
    if (appLaunchIntent != null &&
dispatch.tryStartActivity(appLaunchIntent))
        return true;

    /*
     如果上述处理失败，则获得能启动应用市场去下载某个应用程序的的
     Intent，该Intent的
     Data字段取值为“market:// details?id=应用程序包名”。
     */
    Intent marketIntent = getAppSearchIntent(firstPackage);
    if (marketIntent != null &&
dispatch.tryStartActivity(marketIntent))
        return true;
}
// 处理没有AAR的NDEF消息
dispatch.intent.setPackage(null);
if (dispatch.tryStartActivity()) return true;

```

```

        return false;
    }
}

```

### (3) NFC Tag处理流程总结

NFC Tag的处理流程还算简单，下面总结其中涉及的重要函数调用，如图8-38所示。

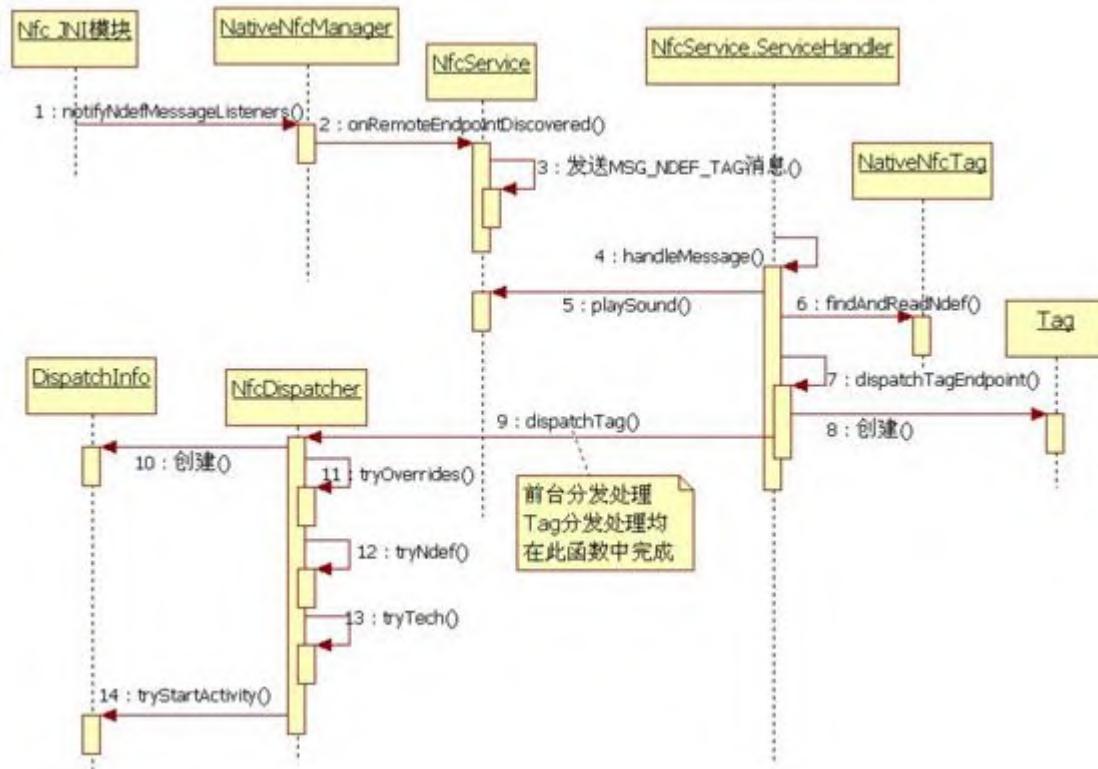


图8-38 NFC Tag处理流程

NFC Tag的处理流程比较简单，其中有的代码逻辑比较复杂，这部分内容主要集中在NativeNfcTag的findAndReadNdef函数中。它和具体NFC芯片厂商的实现有关，故把它留给感兴趣的读者自己来研究。

下面我们将研究Android Beam的工作流程。

### 3. Android Beam工作流程分析

当本机检测到某个NFC设备进入有效距离并且能处理LLCP协议后，将通过notifyLlcpLinkActivation通知我们。本节就从这个函数开始分析。

## (1) notifyLlcpLinkActivation流程

notifyLlcpLinkActivation代码如下所示。

[-->NativeNfcManager. java: : notifyLlcpLinkActivation]

```
private void notifyLlcpLinkActivation(NativeP2pDevice device) {  
    // mListener指向NfcService  
    // 它的onLlcpLinkActivated函数将发送一个  
    MSG_LLCP_LINK_ACTIVATION消息  
    mListener.onLlcpLinkActivated(device);  
}
```

MSG\_LLCP\_LINK\_ACTIVATION消息由NfcService的内部类  
NfcServiceHandler处理，它将调用llcpActivated函数，代码如下所  
示。

[-->NfcService. java: : NfcServiceHandler: llcpActivated]

```
private boolean llcpActivated(NfcDepEndpoint device) {  
    /*  
     * NfcDepEndpoint代表对端设备，其真实类型是NativeP2pDevice。不论对端设  
     * 备是Target还是  
     * Initiator，P2pLinkManager的onLlcpActivated函数都将被调用。  
     */  
    if (device.getMode() == NfcDepEndpoint.MODE_P2P_TARGET) {  
        if (device.connect()) {// 如果对端是Target，则需要连接上它  
            if (mDeviceHost.doCheckLlcp()) {  
                if (mDeviceHost.doActivateLlcp()) {  
                    synchronized (NfcService.this) {  
                        mObjectMap.put(device.getHandle(),  
device);  
                    }  
                }  
            }  
        }  
        mP2pLinkManager.onLlcpActivated();  
        return true;  
    }.....  
    } else if (device.getMode() ==  
    NfcDepEndpoint.MODE_P2P_INITIATOR) {  
        if (mDeviceHost.doCheckLlcp()) {  
            if (mDeviceHost.doActivateLlcp()) {  
                synchronized (NfcService.this) {  
                    mObjectMap.put(device.getHandle(),  
device);  
                }  
            }  
        }  
    }
```

```

        mP2pLinkManager.onLlcpActivated();
        return true;
    } .....
}
return false;
}

```

P2pLinkManager的onLlcpActivated函数代码如下所示。

[-->P2pLinkManager.java: : onLlcpActivated]

```

public void onLlcpActivated() {
    synchronized (P2pLinkManager.this) {
        .....
        switch (mLinkState) {
            case LINK_STATE_DOWN:// 如果之前没有LLCP相关的活动，则
mLinkState为LINK_STATE_DOWN
                mLinkState = LINK_STATE_UP;
                mSendState = SEND_STATE NOTHING_TO_SEND;
                /*
                 * mEventListener指向P2pEventManager，它的
onP2pInRange函数中将播放
                 * 通知音以提醒用户，同时它还会通过SendUi截屏。请读者自行
阅读该函数。
                */
                mEventListener.onP2pInRange();
                prepareMessageToSend(); // ①准备发送数据
                if (mMessageToSend != null ||
                    (mUriToSend != null &&
mHandoverManager.isHandoverSupported())) {
                    mSendState = SEND_STATE NEED_CONFIRMATION;
                    // ②显示提示界面，读者可参考图8-29的左图
                    mEventListener.onP2pSendConfirmationRequested();
                }
                break;
            .....
        }
    }
}

```

onLlcpActivated有两个关键函数，我们先来看第一个函数prepareMessageToSend，其代码如下所示。

[-->P2pLinkManager.java: : prepareMessageToSend]

```

void prepareMessageToSend() {
    synchronized (P2pLinkManager.this) {
        .....
        // 还记得NFC P2P模式示例程序吗？可通过
        setNdefPushMessageCallback设置回调函数
        if (mCallbackNdef != null) {
            try {
                mMessageToSend =
                    mCallbackNdef.createMessage(); // 从回调函数那获取要发送的数据
                /*
                 * getUrис和NfcAdapter的setBeamPushUrисCallback函数有
                 * 关，它用于发送file或
                 * content类型的数据。由于这些数据需要借助Handover技术，请
                 * 读者自己来分析。
                */
                mUrисToSend = mCallbackNdef.getUrис();
                return;
            } .....
        }
        // 如果没有设置回调函数，则系统会尝试获取前台应用进程的信息
        List<RunningTaskInfo> tasks =
        mActivityManager.getRunningTasks(1);
        if (tasks.size() > 0) {
            // 获取前台应用进程的包名
            String pkg =
            tasks.get(0).baseActivity.getPackageName();
            /*
             * 应用程序可以在其AndroidManifest中设
             * 置“android.nfc.disable_beam_default”
             * 标签为false，以阻止系统通过Android Beam来传递与该应用相关的
             * 信息。
            */
            if (beamDefaultDisabled(pkg)) mMessageToSend = null;
            else {
                /*
                 * createDefaultNdef将创建一个NDEF消息，该消息包含两个NFC
                 * Record。
                */
                mMessageToSend = createDefaultNdef(pkg); // 包名为pkg
                的应用程序
            }
        }
    }
}

```

```
        }else
            mMessageToSend = null;
    }
}
```

prepareMessageToSend很有意思，其主要工作如下。

- 如果设置回调对象，系统将从回调对象中获取要发送的数据。
- 如果没有回调对象，系统会获取前台应用程序的包名。如果前台应用程序禁止通过Android Beam分享信息，则prepareMessageToSend直接返回，否则它将创建一个包含了两个NFC Record的NDEF消息。

接下来看第二个关键函数onP2pSendConfirmationRequested，其代码如下所示。

[-->P2pEventManager.java: : onP2pSendConfirmationRequested]

```
public void onP2pSendConfirmationRequested() {
    // 对于拥有显示屏幕的Android设备来说，mSendUi不为空
    if (mSendUi != null) mSendUi.showPreSend(); // 显示类似图8-29左
    图所示的界面以提醒用户
    else mCallback.onP2pSendConfirmed();
    /*
    对于那些没有显示屏幕的Android设备来说，直接调用onP2pSendConfirmed。
    mCallback指向
    P2pLinkManager。待会将分析这个函数。
    */
}
```

在onP2pSendConfirmationRequested函数中，SendUi的showPreSend函数将绘制一个通知界面，如图8-29所示。至此，notifyLlcpLinkActivation的工作完毕。在继续分析Android Beam之前，先来总结notifyLlcpLinkActivation的工作流程，如图8-39所示。

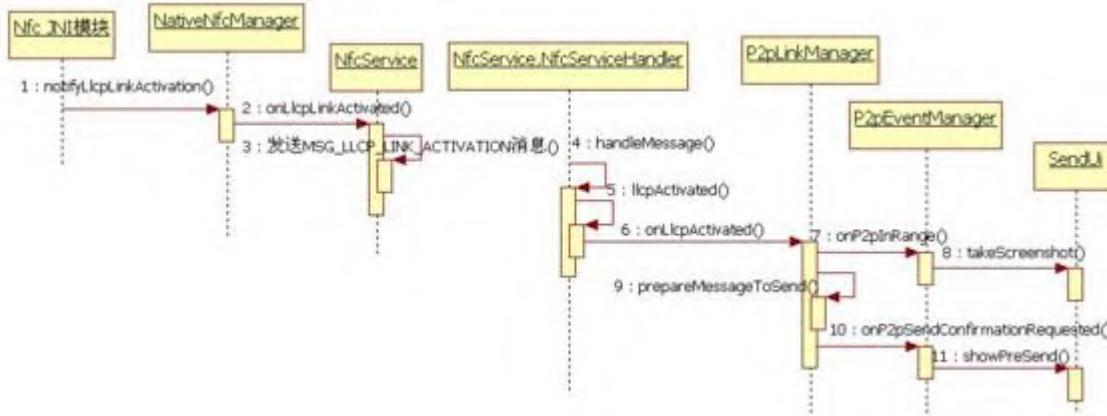


图8-39 notifyLlcpLinkActivation流程

## (2) Beam数据发送流程

SendUi界面将提醒用户触摸屏幕以发送数据，触摸屏幕这一动作将导致SendUi的onTouch函数被调用，其代码如下所示。

[-->SendUi.java: : onTouch]

```

public boolean onTouch(View v, MotionEvent event) {
    .....
    mCallback.onSendConfirmed(); // mCallback指向P2pEventManager
    return true;
}
  
```

[-->P2pEventManager.java: : onSendConfirmed]

```

public void onSendConfirmed() {
    if (!mSending) {
        if (mSendUi != null) mSendUi.showStartSend(); // 显示数据发送动画
        mCallback.onP2pSendConfirmed(); // mCallback指向P2pLinkManager
    }
    mSending = true;
}
  
```

[-->P2pLinkManager.java: : onP2pSendConfirmed]

```

public void onP2pSendConfirmed() {
    synchronized (this) {
}
  
```

```

    .....
    mSendState = SEND_STATE_SENDING;
    if (mLinkState == LINK_STATE_UP)    sendNdefMessage(); // 关
键函数
}
}

```

sendNefMessage将创建一个类型为SendTask的AsyncTask实例以处理数据发送相关的工作，我们来看这个SendTask，相关代码如下所示。

[-->P2pLinkManager. java: : SendTask: doInBackground]

```

final class SendTask extends AsyncTask<Void, Void, Void> {
    public Void doInBackground(Void... args) {
        NdefMessage m; Uri[] uris; boolean result;
        m = mMessageToSend; uris = mUristsToSend; // 设置要发送的数据
        long time = SystemClock.elapsedRealtime();
        try {
            int snepResult = doSnepProtocol(mHandoverManager, m,
uris,
                mDefaultMiu, mDefaultRwSize);
        .....
        } catch (IOException e) {
            // 如果使用SNEP发送失败，则将利用NPP协议再次尝试发送
            if (m != null) // 请读者自行研究和NPP相关的代码
                result = new NdefPushClient().push(m);
            .....
        }
        time = SystemClock.elapsedRealtime() - time;
        if (result) onSendComplete(m, time); // 发送完毕。请读者自行阅
读该函数
        return null;
    }
}

```

上述代码中的doSnepProtocol函数内容如下。

[-->P2pLinkManager. java: : SendTask: doSnepProtocol]

```

static int doSnepProtocol(HandoverManager handoverManager,
                           NdefMessage msg, Uri[] uris, int miu, int rwSize)
throws IOException {
    // 创建一个SnepClient客户端
    SnepClient snepClient = new SnepClient(miu, rwSize);
    try {

```

```

        snepClient.connect(); // ①连接远端设备的SnepServer
    } .....
    try {
        if (uris != null) {// 如果uris不为空，需要使用
HandoverManager, 这部分内容请读者自行阅读
        .....
        } else if (msg != null) {
            snepClient.put(msg); // ②利用SNEP的PUT命令发送数
据
        }
        return SNEP_SUCCESS;
    } catch .....
    finally {
        snepClient.close();
    }
    return SNEP_FAILURE;
}

```

重点介绍SnepClient的connect函数以及put函数。connect函数的代码如下所示。

[-->SnepClient.java: : connect]

```

public void connect() throws IOException {
    .....
    LlcpSocket socket = null;
    SnepMessenger messenger; // SnepMessenger用于处理数据发送和接
收
    try {
        socket = NfcService.getInstance().createLlcpSocket(0,
mMiu, mRwSize, 1024);
        if (mPort == -1)
socket.connectToService(mServiceName); // 通过服务名来连接服务端
        else socket.connectToSap(mPort); // 通过SAP连接服务端
        // 获取远端设备的MIU
        int miu = socket.getRemoteMiu();
        int fragmentLength = (mFragmentLength == -1) ? miu :
Math.min(miu, mFragmentLength);
        messenger = new SnepMessenger(true, socket,
fragmentLength);
    } .....
    .....
}

```

put函数代码如下所示。

[-->SnekClient.java: : put]

```
public void put(NdefMessage msg) throws IOException {
    SnekMessenger messenger;
    messenger = mMessenger;
    synchronized (mTransmissionLock) {
        try {
            /*
             * 获取SNEP PUT命令对应的数据包，然后发送出去。
             */
            messenger.sendMessage(SnekMessage.getPutRequest(msg));
            messenger.getMessage();
        } . . .
    }
}
```

图8-40总结了本节所述的Beam数据发送流程。

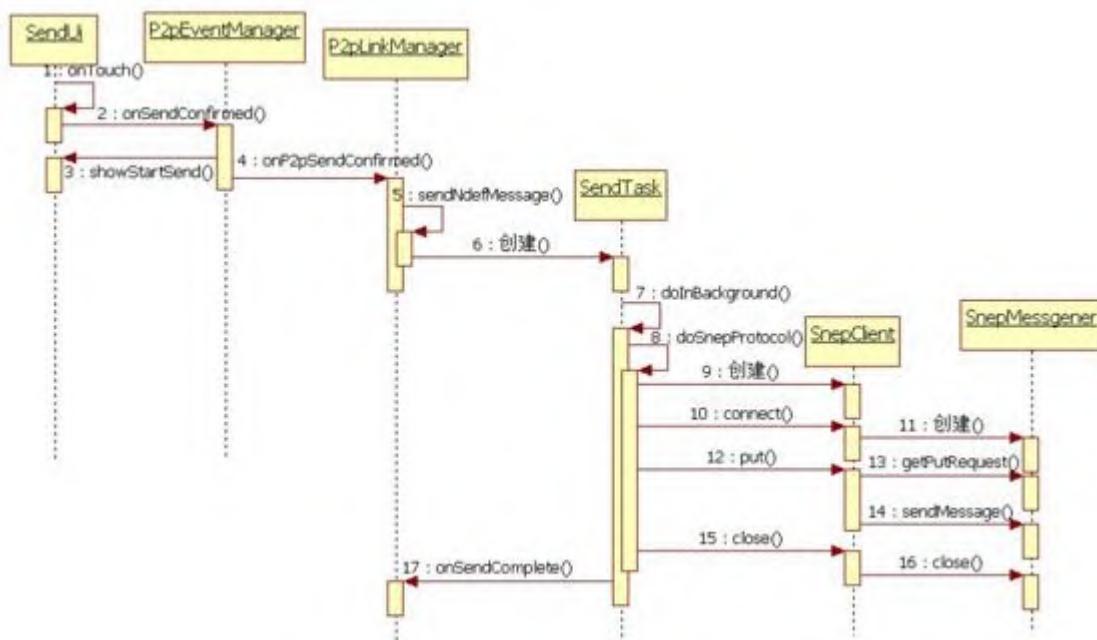


图8-40 Beam数据发送流程

接着来看Beam数据接收流程。

### (3) Beam数据接收流程

8.3.2节中曾介绍，SnepServer每接收一个客户端的连接后均会创建一个ConnectionThread，它的代码如下所示。

[-->SnepServer.java: : ConnectionThread]

```
private class ConnectionThread extends Thread {  
    private final LlcpSocket mSock;  
    private final SnepMessenger mMessenger;  
    ConnectionThread(LlcpSocket socket, int fragmentLength) {  
        super(TAG);  
        mSock = socket;  
        // 也创建一个SnepMessenger用来处理具体的数据接收  
        mMessenger = new SnepMessenger(false, socket,  
fragmentLength);  
    }  
    public void run() {  
        .....// 省略一些try/catch逻辑  
        while (running) {  
            if (!handleRequest(mMessenger, mCallback)) break;  
            .....  
        }  
        mSock.close();  
    }  
}
```

[-->SnepServer.java: : handleRequest]

```
static boolean handleRequest(SnepMessenger messenger,  
                             Callback callback) throws IOException {  
    SnepMessage request;  
    request = messenger.getMessage();  
    .....  
    if (((request.getVersion() & 0xF0) >> 4) !=  
        SnepMessage.VERSION_MAJOR) {  
        messenger.sendMessage(SnepMessage.getMessage(  
            SnepMessage.RESPONSE_UNSUPPORTED_VERSION));  
    } else if (request.getField() == SnepMessage.REQUEST_GET) {  
        // 处理GET命令，callback类型为SnepServer的内部类Callback  
  
        messenger.sendMessage(callback.doGet(request.getAcceptableLengt  
h(),  
                                         request.getNdefMessage()));  
    } else if (request.getField() == SnepMessage.REQUEST_PUT) {  
        // 处理PUT命令  
  
        messenger.sendMessage(callback.doPut(request.getNdefMessage()))  
    }  
}
```

```
;  
} ....  
return true;  
}
```

来看SnepServer. callback的doPut函数，它由P2pLinkManager的内部类实现，代码如下所示。

[-->P2pLinkManager. java: : SnepServer. Callback]

```
final SnepServer.Callback mDefaultSnepCallback = new  
SnepServer.Callback() {  
    public SnepMessage doPut(NdefMessage msg) {  
        onReceiveComplete(msg);  
        return  
        SnepMessage.getMessage(SnepMessage.RESPONSE_SUCCESS);  
    }  
}
```

onReceiveComplete的代码如下所示。

[-->P2pLinkManager. java: : onReceiveComplete]

```
void onReceiveComplete(NdefMessage msg) {  
    // 发送一个MSG_RECEIVE_COMPLETE消息  
    mHandler.obtainMessage(MSG_RECEIVE_COMPLETE,  
msg).sendToTarget();  
}
```

MSG\_RECEIVE\_COMPLETE消息由P2pLinkManager的handleMessge处理，相关代码逻辑如下所示。

[-->P2pLinkManager. java: : ]

```
public boolean handleMessage(Message msg) {  
    switch (msg.what) {  
        ....  
        case MSG_RECEIVE_COMPLETE:  
            NdefMessage m = (NdefMessage) msg.obj;  
            synchronized (this) {  
                ....  
                mSendState = SEND_STATE NOTHING_TO_SEND;  
            }  
            mEventListener.onP2pReceiveComplete(true); // 取消本机显示的SendUi
```

界面

```
// sendMockNdefTag将发送一个MSG_MOCK_NDEF消息  
给NfcService的NfcServiceHandler  
  
NfcService.getInstance().sendMockNdefTag(m);  
}  
break;  
....  
}  
}
```

[-->NfcService.java: : NfcServiceHandler: handleMessage]

```
final class NfcServiceHandler extends Handler {  
    public void handleMessage(Message msg) {  
        switch (msg.what) {  
            case MSG_MOCK_NDEF: {  
                NdefMessage ndefMsg = (NdefMessage) msg.obj;  
                Bundle extras = new Bundle();  
                extras.putParcelable(Ndef.EXTRA_NDEF_MSG,  
ndefMsg);  
                extras.putInt(Ndef.EXTRA_NDEF_MAXLENGTH, 0);  
                extras.putInt(Ndef.EXTRA_NDEF_CARDSTATE,  
Ndef.NDEF_MODE_READ_ONLY);  
                extras.putInt(Ndef.EXTRA_NDEF_TYPE,  
Ndef.TYPE_OTHER);  
                // 创建一个模拟Tag对象  
                Tag tag = Tag.createMockTag(new byte[] { 0x00 },  
                    new int[] { TagTechnology.NDEF }, new  
Bundle[] { extras });  
                // 直接通过分发系统分发这个Tag  
                boolean delivered =  
mNfcDispatcher.dispatchTag(tag);  
                ....  
                break;  
            }  
            ....  
        }  
    }  
}
```

Beam接收端的处理流程比较巧妙，系统将创建一个模拟的Tag对象，然后利用分发系统来处理它。图8-41总结了Beam接收端的处理流程。

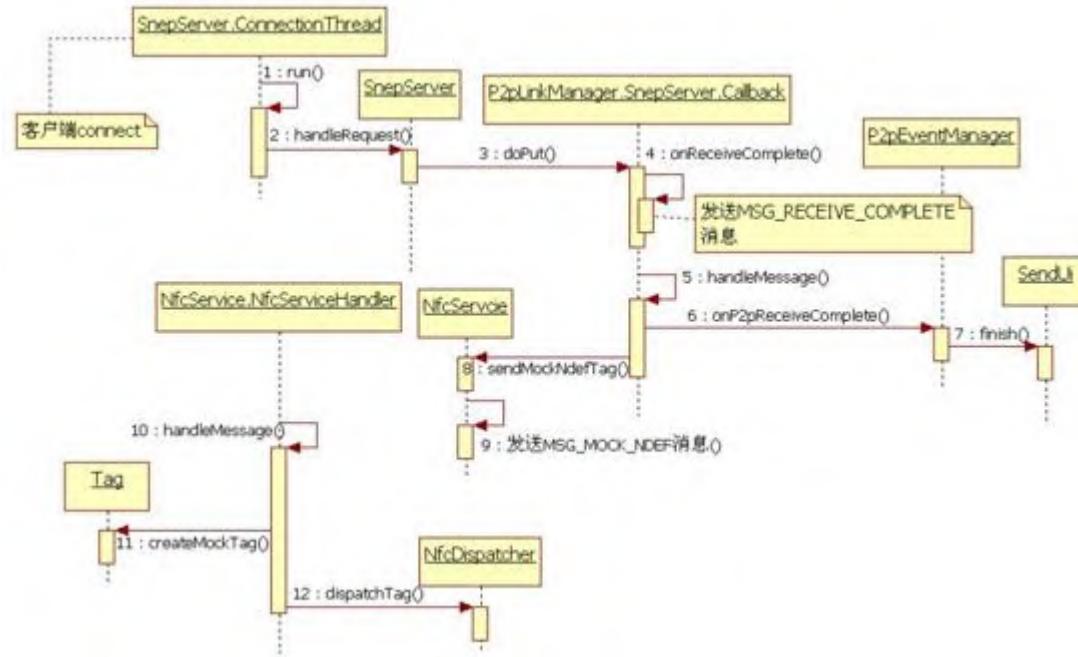


图8-41 Beam数据接收流程

#### 4. CE模式

NFC系统模块对CE模式的支持集中在以下两点。

- 通过NfcAdapterExtras为CE模式的客户端提供INfcAdapterExtras功能实现。这部分内容所涉及的调用流程比较简单，请读者自行研读。当然，我们略过了和芯片相关的部分。
- 当NFC设备进入CE模式后，和它交互的另一端是诸如NFC Reader这样的设备。此时对端发起一些操作，而NativeNfcManagement的一些回调函数将被触发。下面代码展示了和CE相关的这些回调函数。

[-->NativeNfcManager.java: : CE相关回调函数]

```
// 根据NFCIP-1协议，该函数表示对端设备发送了DESELECT命令给我们。它属于
deactivation阶段的命令
private void notifyTargetDeselected() { // mListern指向
    NfcService
        mListner.onCardEmulationDeselected();
    }
/*
用于通知SE上某个应用程序开始和对端设备进行交互。AID为Application ID的缩
```

写，它代表SE上的某个应用程序（称为Applet）。

```

*/
private void notifyTransactionListeners(byte[] aid) {
    mListener.onCardEmulationAidSelected(aid);
}
// 用于通知SE模块被激活
private void notifySeFieldActivated() {
    mListener.onRemoteFieldActivated();
}
// 用于通知SE模块被禁止
private void notifySeFieldDeactivated() {
    mListener.onRemoteFieldDeactivated();
}
// 收到对端设备的APDU命令
private void notifySeApduReceived(byte[] apdu) {
    mListener.onSeApduReceived(apdu);
}
/*
EMV是EMV标准是由国际三大银行卡组织Europay(欧陆卡，已被万事达收购)、
MasterCard(万事达卡)和
Visa(维萨)共同发起制定的银行卡从磁条卡向智能IC卡转移的技术标准，是基于IC卡的金融支付标准，
目前已成为公认的全球统一标准。下面这个函数用于通知EMV Card进入Removal阶段。
该函数只适用于NXP公司的相关芯片。
*/
private void notifySeEmvCardRemoval() {
    mListener.onSeEmvCardRemoval(); // NfcService如何处理它呢
}
// 用于通知MIFARE SMX Emulation被外部设备访问。该函数只适用于NXP公司的相关芯片
private void notifySeMifareAccess(byte[] block) {
    mListener.onSeMifareAccess(block);
}

```

NfcService是如何处理这些回调通知的呢？以notifySeEmvCardRemoval中的onSeEmvCardRemoval为例，NfcService将发送一个MSG\_SE\_EMV\_CARD\_REMOVAL消息，而这个消息的处理函数代码如下所示。

[-->NfcService.java: : NfcServiceHandler: handleMessage]

```
.....
    case MSG_SE_EMV_CARD_REMOVAL:           // CE相关的消息全是类型的
处理方式
        Intent cardRemovalIntent = new Intent();
        cardRemovalIntent.setAction(ACTION_EMV_CARD_REMOVAL);
        sendSeBroadcast(cardRemovalIntent); // 发送广播
.....
```

由上述代码的注释可知，NfcService对CE相关的处理非常简单，就是接收来自底层的通知事件，然后将其转化为广播事件发送给系统中感兴趣的应用程序。

## 5. Android中的NFC总结

本节介绍了Android平台中NFC系统模块NfcService及其他重要组件。从整体上来说，NfcService以及与底层芯片无关的模块难度不大，而与底层芯片相关的模块则难度较大（位于com.android.nfc.dhimpl包中）。本章没有讨论dhimpl包的具体代码，希望感兴趣的读者能结合芯片手册自行研究它们。

另外，本节还对NFC R/W模式及P2P模块的工作流程进行了相关介绍，这部分难度不大，相信读者能轻松掌握。同时，作为课后作业，请读者在本节基础上自行学习Handover相关的处理流程。

最后，本节简单介绍了NFC CE模式的处理流程，NfcService本身对CE相关的处理比较简单，它仅根据CE相关的操作向系统发送不同的广播，而这些广播则会由感兴趣的应用程序来处理，例如Google Wallet。

**提示** NFC CE模式其实内容相当复杂，涉及很多规范。笔者会在博客上继续介绍NFC CE相关的知识，敬请读者关注。

## 8.4 NFC HAL层讨论

Android在Hardward目录下为NFC定义了一个nfc.h头文件用于支持NFC HAL操作，但读者如果看过libnfc或libnfc-nci代码会发现，libnfc和libnfc-nci没有太多使用nfc.h定义的接口，而是大量引用各自公司定义的一套API。这种做法无可厚非，但它使得其他更上层的模块很难做到与底层平台或硬件解耦合。相信图8-26已经让读者直观感受到到这种做法恶果了。

**注意** 与NFC这种状况形成鲜明对比，本书前面浓墨重彩介绍的Wi-Fi模块，借助n180211机制或历史更悠久的wireless extension API解决了上层模块与底层平台或硬件的解耦合问题。

表8-13列举了当前知名的几个NFC HAL层实现。

表 8-13 NFC HAL 实现情况

名 称	相关公司	说 明
libnfc-nxp	NXP	只支持 NXP 公司的 NFC 芯片。注意，Android 中使用的 libnfc 实际上是 libnfc-nxp。请读者注意和此表中的 libnfc 区别开来
opennfc	Inside Secure	<a href="http://open-nfc.org/wp/">open-nfc.org/wp/</a> ，由 Inside Secure 公司赞助，开发文档非常完善，支持 WineCE、Windows Mobile 7、Android、Linux、MeeGo 等平台
libnfc-nci	Broadcom	只支持 Broadcom 公司的芯片
libnfc	社区支持	<a href="http://www.libnfc.org/community">www.libnfc.org/community</a> 。该项目和 wpa_supplicant 非常类似，文档也做得比较完善
nfcpy	社区支持	<a href="http://nfcpy.org">nfcpy.org</a> 。用 Python 编写的 NFC 模块。据说该项目由 Sony 赞助

笔者研究了表8-13中的除nfcpy之外的几个NFC HAL层模块代码，感觉和wpa\_supplicant比起来还是有一定差距。不过，根据参考资料[21]和[26]的介绍，未来Linux系统中，NFC整个软件架构将变成如图8-42所示。

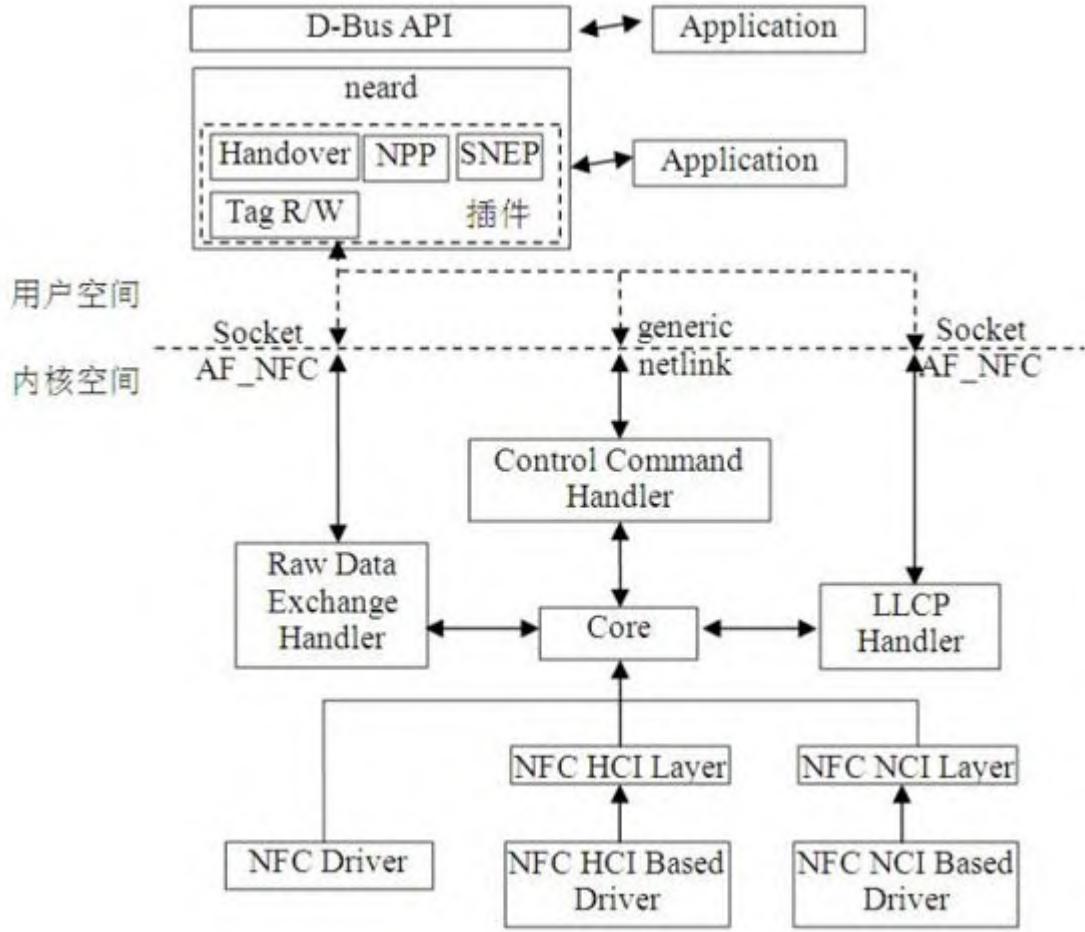


图8-42 NFC软件架构展望

图8-42中，用户空间运行一个名为neard的NFC Daemon进程，它通过AF\_NFC socket以及Generic Netlink机制和内核空间的NFC子系统通信。neard通过不同的插件来支持NFC的协议，例如Handover、NPP、SNEP等。

内核空间中，NFC子系统包括Control Command Handler、LLCP Handler、Raw Data Handler以及Core等核心模块。

不同NFC芯片厂商只要实现相关的NFC驱动即可。至于Core模块如何与NFC驱动交互，则可使用基于NFC Forum定义的NCI规范（抽象为NFC NCI Layer）、HCI规范（抽象为NFC HCI Layer），或者直接操作NFC Driver。

**提示** 在此，笔者希望NFC软件架构尽快完善，同时也希望国内的公司能积极参与到这一过程中来以提高我们的话语权。

## 8.5 本章总结和参考资料说明

### 8.5.1 本章总结

本章对NFC进行了详细介绍，主要内容如下。

- NFC理论知识，这部分内容主要围绕NFC三种运行模式进行了相关讨论。
- Android系统中NFC模块的结构，通过三个示例程序向读者展示了Android中NFC API的一些知识。
- 对NFC系统模块NfcService及其他重要成员进行了介绍，详细分析了NFC Tag分发以及Android Beam的实现细节。

最后，希望读者在本章的基础上，完成下列任务。

- 学习NFC Forum中的其他规范。
- 结合NFC Forum中的Connection Handover规范，分析Android中NFC Handover的实现代码。

## 8.5.2 参考资料说明

### 1. 概述

[1] 《Near Field Communication From Theory to Practice》第1章和第2章

说明：这本书是笔者目前所阅读的关于NFC最为详尽的资料，建议初学者仔细阅读，尤其是前三章。

### 2. NFC概述

[2] 《Near Field Communication From Theory to Practice》图2-1（略有修改）

[3] NFC Technology Overview

说明：下载地址为[http://www.nfc-forum.org/resources/presentations/NFCForum\\_Technical\\_WIMA09.pdf](http://www.nfc-forum.org/resources/presentations/NFCForum_Technical_WIMA09.pdf)。该资料为NFC Forum官方提供，介绍了NFC技术。

[4] NFC vs ISO 14443 vs Felica

说明：该文档介绍了NFC、ISO 14443和Felica之间的区别，文档下载地址为  
<http://developer.nokia.com/Community/Blogs/resources/300066/Philips-NFC-vs-ISO14443-vs-Felica-SLIDES.pdf>。

[5] [http://www.nfc-forum.org/specs/spec\\_list/](http://www.nfc-forum.org/specs/spec_list/)

说明：该网页介绍了当前NFC Forum官方各个技术文档的主要内容，建议读者下载NFC Forum技术文档前先阅读此网页。

### 3. NFC R/W运行模式

[6] 《Near Field Communication From Theory to Practice》3.5节“Reader/Writer Operating Mode Essentials”

说明：该节对NFC R/W运行模式进行了相关介绍。

[7] [http://www.nfc-forum.org/resources/white\\_papers/NXP\\_BV\\_Type\\_Tags\\_White\\_Paper-Apr\\_09.pdf](http://www.nfc-forum.org/resources/white_papers/NXP_BV_Type_Tags_White_Paper-Apr_09.pdf)

说明：该文档可在NFC Forum官网上下载，属于NXP公司的一篇介绍NFC Tag Type的白皮书，通俗易懂，建议不熟悉的读者仔细研究。

#### 4. NDEF和NFC Record

[8] NFC Data Exchange Format Technical Specification

[9] NFC Record Type Definition Technical Specification

[10] URI Record Type Definition Technical Specification

[11] Text Record Type Definition Technical Specification

说明：NFC Forum官方文档，难度都比较小。

#### 5. NFC P2P运行模式

[12] Logical Link Control Protocol Technical Specification

说明：LLCP的官方协议，建议读者先阅读本章相关章节后再去看它。

[13] NFC Digital Protocol Technical Specification

说明：阅读此规范前，最好看看ISO 18092 (<http://www.docin.com/p-586980527.html>)。

[14] Simple NDEF Exchange Protocol Technical Specification

说明：SNEP官方协议，非常简单。

#### 6. NFCCE运行模式

[15] 《Near Field Communication From Theory to Practice》 3.7 节

[16] 《Near Field Communication From Theory to Practice》 3.3 节

说明：详细介绍了NFC Enabled Phone和Card Emulation Mode，读者可在此基础上去理解。

[17] <http://www.nfc.cc/technology/nxp-nfc-chips/>

说明：NXP公司pn65 NFC系列芯片模块图。

[18] <http://www.chinaz.com/biz/2011/0827/207232.shtml>

[19] <http://kan.weibo.com/con/3616344461572955>

说明：中国市场上运营商和银联这两大利益集团联合推广NFC-SIM卡方案。

## 7. NCI介绍

[20] NFC Controller Interface (NCI) Specification

说明：NCI官方文档，长达140多页。不过读者无须了解其细节，只要掌握NCI架构及相关模块的功能即可。

[21] <https://github.com/charsyam/linux-kernel-3.8/blob/master/Documentation/networking/nfc.txt>

说明：Linux Kernel 3.8中关于NFC Subsystem的介绍。

## 8. NFC规范列表

[22] 《Professional NFC Application Development for Android》表1-2

说明：此书与参考资料[1]由同一团队编写，对Android上如何开发NFC应用进行了详细介绍。

## 9. NFC CE示例

[23] <http://stackoverflow.com/questions/15065172/nfcee-execution-environment-hardware-or-library-module>

说明：这个资料介绍了Android中如何操作NFC EE，读者不妨看看。

[24] <http://nelenkov.blogspot.jp/2012/08/accessing-embedded-secure-element-in.html>

[25] <http://nelenkov.blogspot.de/2012/08/android-secure-element-execution.html>

说明：以上两个资料非常详尽地介绍了Android SE方面的知识，文章质量非常高。

## 10. NFC HAL层探讨

[26]

[http://elinux.org/images/d/d1/Near\\_Field\\_Communication\\_with\\_Linux.pdf](http://elinux.org/images/d/d1/Near_Field_Communication_with_Linux.pdf)

说明：内容和参考资料[21]类似。关于NFC认证测试，请参考<http://www.nfc-forum.org/certification/certification-testing/>。

注意 图8-26对应的Android版本为4.2。根据审稿专家的意见，NFC Tag Technologies分为supported和optional supported两种。

optional supported表示某些Tag Technology在某些平台上不受支持。笔者此处采用的是Android SDK关于Tag Technology的解释，详情见网址

<http://developer.android.com/reference/android/nfc/tech/package-summary.html>。

2013年5月起，北京可用支持NFC功能的手机当公交卡乘坐地铁和公交，该措施无疑为NFC的推广起到了积极作用。

# 第9章 深入理解GPS

本章所涉及的源代码文件名及位置

- LocationActivity. java  
development/samples/training/location-aware/src/com/example/android/location/LocationActivity. java
- LocationManagerService. java  
framework/base/services/java/com/android/server/LocationManagerService. java
- LocationProviderProxy. java  
framework/base/services/java/com/android/server/location/LocationProviderProxy. java
- GpsLocationProvider. java  
framework/base/services/java/com/android/server/location/GpsLocationProvider. java
- com\_android\_server\_location\_GpsLocationProvider. cpp  
framework/base/services/jni/com\_android\_server\_location\_GpsLocationProvider. cpp
- gps. h      hardware/libhardware/include/hardware/gps. h
- LocationSettings. java  
packages/apps/Settings/src/com/android/settings/LocationSettings. java

## 9.1 概述

GPS (Global Positioning System, 全球定位系统) 源自美国军方的一个项目，其主要作用是为陆海空三大领域提供实时、全天候和全球性的导航服务。和GPS相对应的还有一个词，GNSS (Global Navigation Satellite System, 全球导航卫星系统)。GPS是GNSS的一种具体实现形式。目前，世界上的GNSS除了美国的GPS外，还有欧盟的GALILEO、俄罗斯的GLONASS以及中国的北斗导航系统。

近几年来，随着新一代移动智能平台的普及，支持GPS及其他GNSS系统几乎是当下所有智能手机的标准功能，而在GPS或其他能提供位置信息的服务之上，人们更是构建了一个市场规模达数十亿美金的LBS (Location Based Service, 基于位置的服务<sup>①</sup>)。

随着位置信息获取技术的多样化，Android平台在这些技术之上抽象出了一套名为Location Manager (位置管理) 的软件架构。当然，作为该框架中最重要的位置提供服务模块，GPS功能由Android系统直接提供。

和本书其他章节类似，本章也会从两个方面来介绍Android平台中LM相关的功能。

- 首先介绍和GPS相关的一些基础知识。从原理上看，GPS与GLONASS或北斗等其他卫星导航系统类似，所以本章将仅围绕GPS开展讲解。而读者在掌握GPS知识的基础上，能轻松将它们运用到其他GNSS系统中。
- 在了解GPS相关原理的基础上，介绍Android平台中位置管理的软件架构及代码实现。

**提示** 希望读者在本章基础上深入钻研GPS相关知识并能和其他读者分享自己的成果。

<sup>①</sup> 根据参考资料[1]，LBS源于几起悲剧事件。

## 9.2 GPS基础知识

与GPS相关的知识非常多，市面上也有很多专业的书籍。不过，对于本书的读者来说，笔者将挑选并介绍一些比较实用的内容。笔者将这些知识归纳为如下三个部分。

- 卫星导航基本原理：主要介绍卫星导航的一些基础知识。
- GPS工作原理：介绍GPS的工作原理和相关的数据格式。
- OMA-SUPL协议：介绍OMA-SUPL方面的知识。

注意 如何选择合适的知识点向读者介绍是本书编写过程中一项非常重要及困难的工作。以GPS为例，其专业书籍涉及较多的数学计算和公式推导。显然，这些内容对于当今已成熟并高度集成化的GPS模块来说太过基础。笔者的经验是，对于一门陌生的技术和专业，初学者首先要掌握其基本原理和相关的概念。这些基本原理和概念将是这门技术或专业的主要框架和脉络。只有在掌握专业知识框架的基础上，才能开展更进一步的学习和研究。从这个角度出发，本章将综合下文实际代码分析的需求，集中介绍相关的GPS原理和概念。

### 9.2.1 卫星导航基本原理

本节介绍测距、参考坐标系、时间系统、卫星轨道四个方面的基础知识。

#### 1. 测距原理<sup>[2]</sup>

GPS（包括其他的GNSS系统）使用的测距原理非常简单，它的工作过程如图9-1所示。

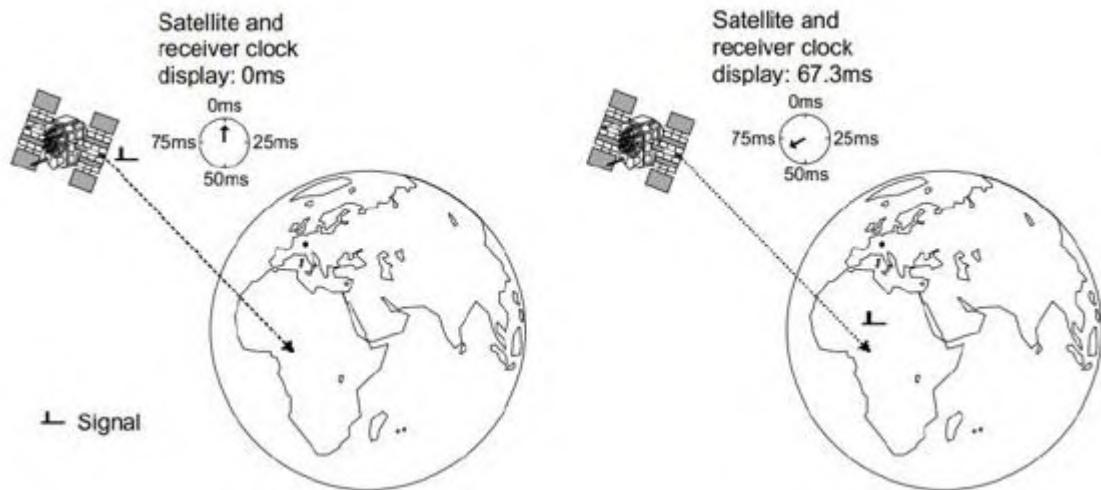


图9-1 卫星测距原理

如图9-1所示，卫星和地面接收器各自有一个时钟。假设卫星和接收器的时钟能完美同步（注意这个假设，以后我们还会讲到它）。在0ms时刻，卫星向接收器发送了一串信号。在67.3ms时，接收器收到了该信号。卫星离接收器的距离就是信号传播速度乘以传播时间。

用公式一来表示图9-1的卫星测距原理。

[公式一]

$$D = \Delta \tau * c$$

该公式中，c为光速， $\Delta t$ 为信号传输时间，D为距离。

有了公式一，我们可以计算接收器到任意卫星的距离。不过，距离（Range）和位置（Location）显然是两个不同的概念。如何根据距离得到位置信息呢？

原来，位置需要放在某个坐标系中来考察，下一节将专门讨论坐标系。假设现在已经有一个坐标系，图9-2就能回答刚才提出的问题。

和图9-1比起来，图9-2有如下特点。

- 卫星和接收器的位置都置于一个统一的二维坐标系中来考察。
- 接收器离两个卫星的距离都由公式一计算得到，分别是 $D_1$  和 $D_2$  。
- 如果以卫星为圆心，以接收器到卫星的距离为半径，可以得到图9-2中的两个圆。这两个圆的相交点到卫星1的距离为 $D_1$ ，到卫星2的距离为 $D_2$ 。也就是说，这两个点就是接收器的可能位置。
- 如果接收器的Y坐标值不能高于卫星的Y坐标值，接收器的实际位置只能是图9-2中的 $(X_p, Y_p)$ 。

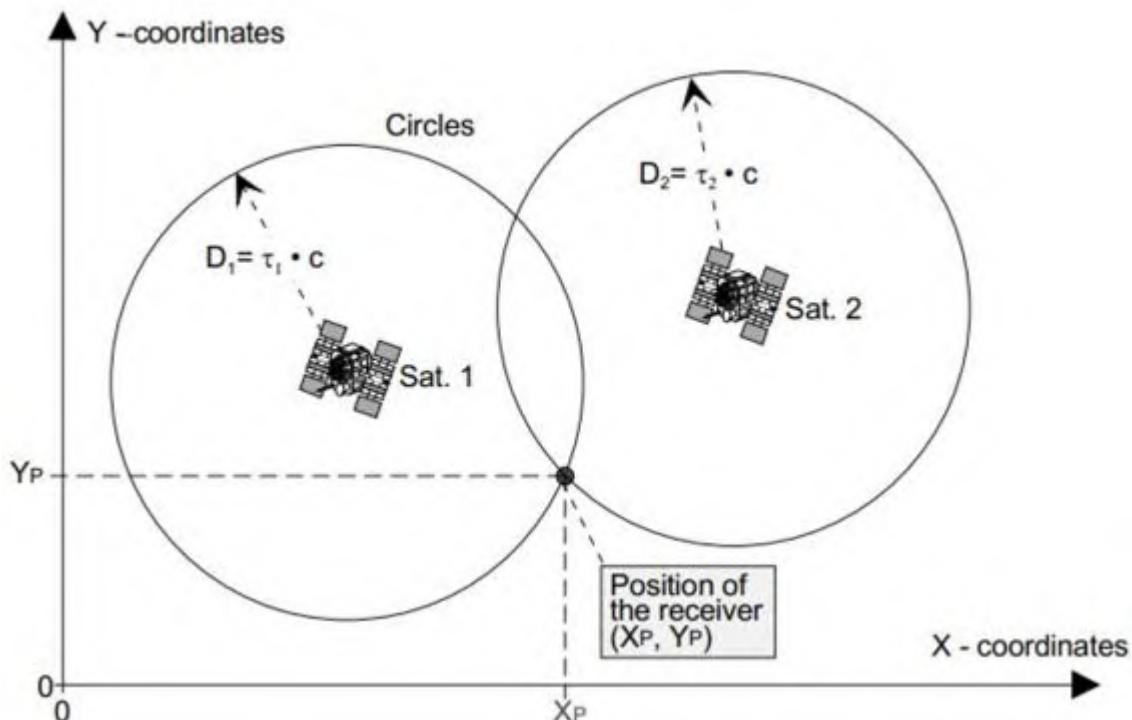


图9-2 二维坐标系中接收器位置计算示意图

掌握了二维坐标系中接收器的位置计算方法，只要再增加一颗卫星，就很容易推导出接收器在三维坐标系中的位置了。

从理想情况来说，定位（Positioning）计算就这么简单，但现实情况却相当复杂。例如，在上述的讨论中还有两个重要的潜在问题没有解决。

- 如何选择坐标系？

- 出于成本、便携性等各方面的考虑，接收器的时钟精度远不如卫星的时钟精度，所以在计算信号传输时间时会造成较大的偏差。由于信号传播速度是光速，所以哪怕这个时间偏差为0.1ms，距离偏差都会达到30km。

这两个问题如何解决的呢？下两节将分别介绍坐标系和时间系统。时间偏差的问题则通过引入第四颗GPS卫星参与定位计算来解决（详情见9.2.2节）。

## 2. 坐标系

### (1) ECI/ECEF/WGS-84<sup>[2]</sup>

根据上一节的内容可知，坐标系对于位置计算非常重要。坐标系有很多个，甚至不同的国家都可能建立更加符合本国实际情况的坐标系。但在GPS中，相关的坐标系主要有两个。

- 地心惯性坐标系（Earth Centered Inertial, ECI）：用于描述GPS卫星的位置信息。在这种坐标系中，原点为地球的质心，卫星围绕质心运动，并遵守牛顿运动定律。
- 地心地球固连坐标系（Earth Centered, Earth Fixed, ECEF）：用于描述地面接收器的位置信息。ECEF最大的特点是它会随着地球而旋转。

**提示** 在GPS的定位计算过程中，需要先把卫星在ECI坐标系的位置转换成它在ECEF坐标系的位置。

图9-3展示了ECI和ECEF坐标系。

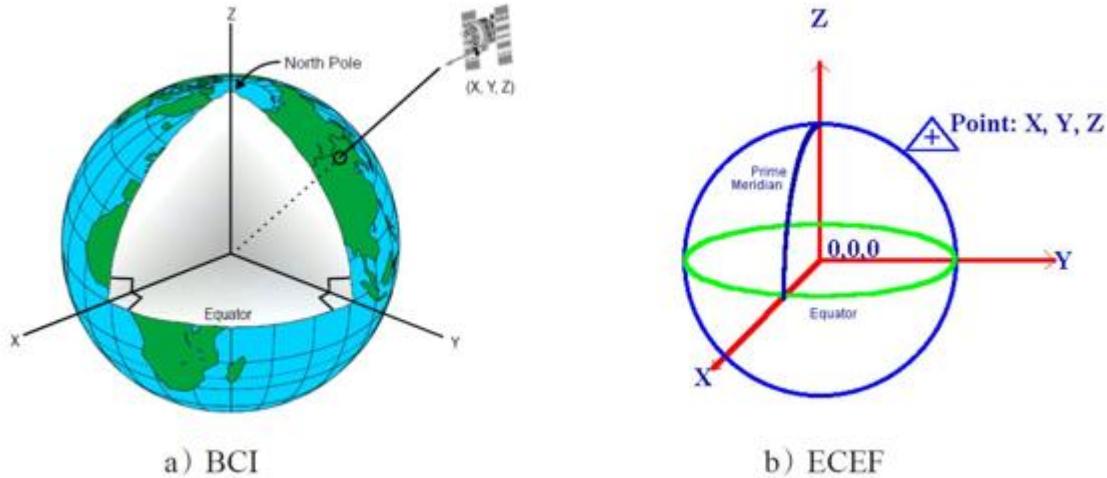


图9-3 ECI和ECEF坐标系

ECI坐标系中，XY平面与地球赤道面重合。X轴指向天球（Celestial Sphere，一种假想的无限大的球，它和地球同心。所以ECI坐标系不受地球旋转的影响）的某个位置。Z轴与XY平面垂直并指向北极。ECI坐标系属于笛卡尔坐标系，故卫星的位置由(X, Y, Z)表示。

ECEF坐标系的原点为地球中心（这就是Earth Centered一词的缘由）。XY平面也与地球赤道面重合。不过其X轴指向0经度方向，Y轴指向东经90度的方向。所以ECEF坐标系实际上是随着地球一起旋转的。ECEF坐标系也属于笛卡尔坐标系，故接收器的位置也由(X, Y, Z)表示。

ECEF是一个笛卡尔坐标系，而我们实际使用的位置信息却是由经纬度来表示的，如何将笛卡尔坐标系中的X, Y, Z值转换成经纬度呢？

该转换工作涉及另外一个重要的概念，即标准地球模型。GPS参考的地球模型为WGS-84（World Geodetic System 1984，由美国国防部建立）。WGS-84模型如图9-4所示。

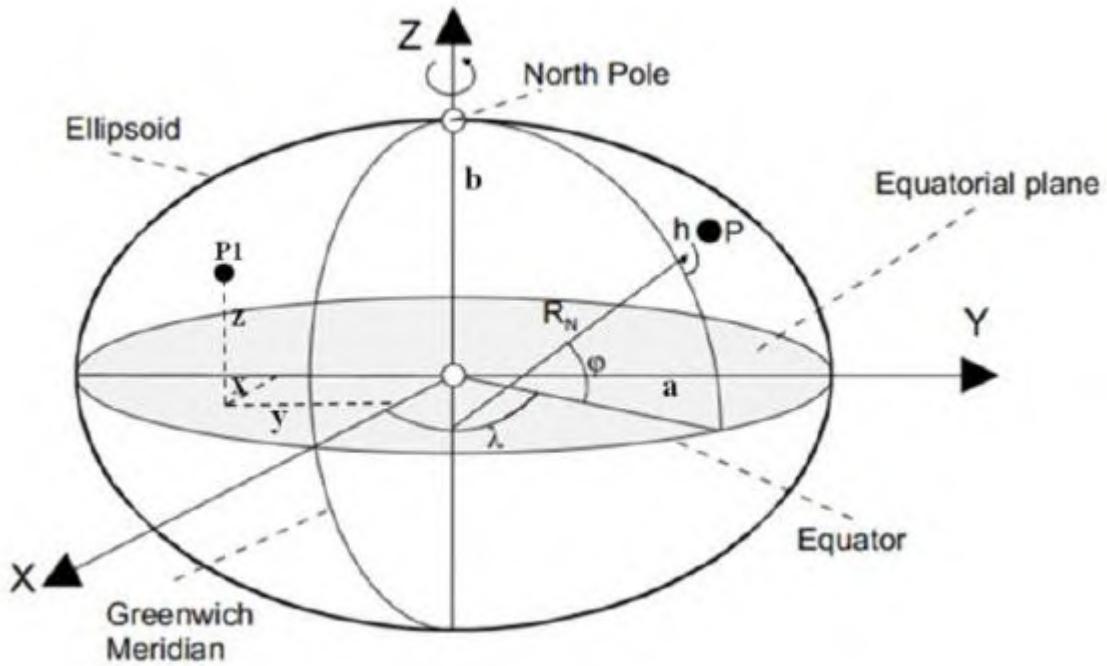


图9-4 WGS-84模型

图9-4所示的标准大地模型中，地球被看做一个椭球体。该椭球体的半长轴（Semi Major Axis，实际长度为6378137.00m）为a，半短轴（Semi Minor Axis，实际长度为6356752.31m）为b。根据a和b的值，该椭球体的偏心率①（Eccentricity）可由以下公式计算得到。

$$f = \frac{a-b}{a}$$

图中的Equatorial Plane为赤道面。赤道面和椭球体相交得到的椭圆为赤道（Equator），它就是纬度为0的地方。图中的Greenwich Meridian为格林尼治子午线，即经度为0的地方。椭球体的表面叫椭球面，即图中的Ellipsoid。

图中的P1点的位置采用了笛卡尔坐标系，其值为(x, y, z)，而P点的位置则由椭球坐标系确定，其值为(φ, λ, h)。注意，此处的h是P点与椭球面的高度，即GPS概念中的高度。

根据相关的公式<sup>[2]</sup>，椭球坐标系和笛卡尔坐标系能相互转化。

## (2) 高度计算

根据上节关于椭球坐标系中 $h$ 坐标值的解释，GPS中的高度是指它和椭球面（Ellipsoid）的距离。但值得特别注意的是，这个高度和日常生活中所说的海拔高度不是同一个概念。日常生活中所说的海拔高度不是基于Ellipsoid，而是基于大地水准面（Geoid）的。

大地水准面是一个重力等位面。简单点说，静止海水在大地水准面上不会因为重力原因而流动。大地水准面和地球的质量分布等有重要关系。相比椭球面而言，大地水准面的数学模型非常复杂，很难用数学公式来描述。

大地水准面和椭球面之间的区别影响了我们对高度的计算。图9-5所示为GPS高度与海拔高度的区别<sup>[3]</sup>。

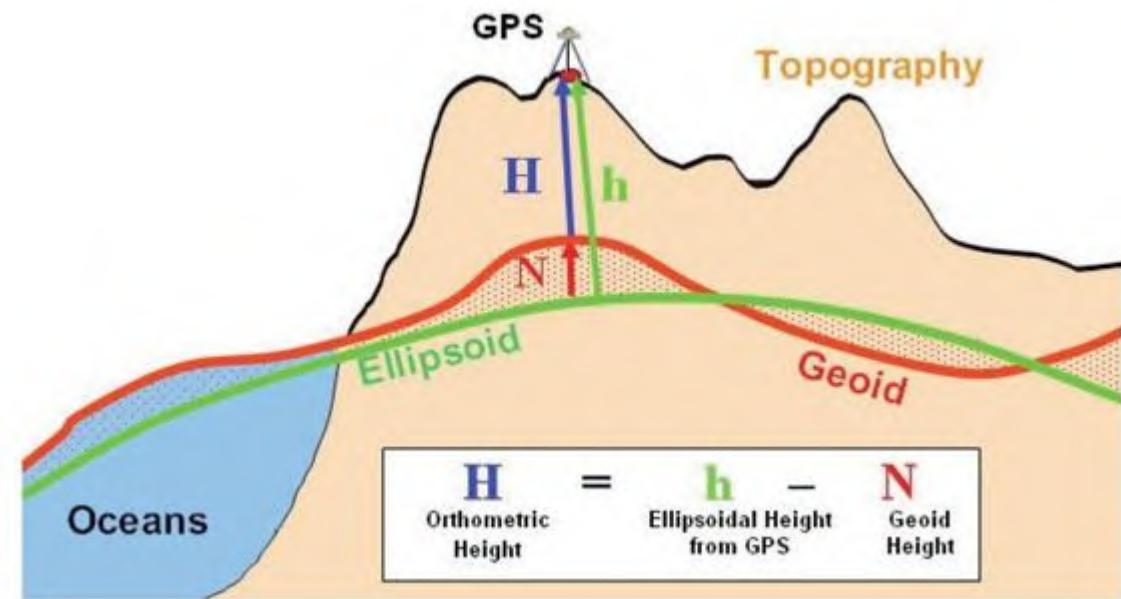


图9-5 高度计算的区别

图9-5中，地球真实的表面由大海和高山组成，这个表面叫地形（Topography）。

GPS测量的高度（也叫大地高，Ellipsoidal Height）为 $h$ ，而日常所说的海拔高度（也叫正高，Orthometric Height）为 $H$ 。 $h$ 和 $H$ 之间的差（也叫大地水准面高，Geoid Height）为 $N$ 。

注意 对于高精度的测绘需求，往往需要把 $h$ 值转换成 $H$ ，不过一般情况下二者的差别不大。

了解了GPS的坐标系统，马上来看与GPS相关的另外一个非常重要的系统。

### 3. 时间系统 [②](#)

和GPS相关的时间系统有四种，分别是国际原子时（International Atomic Time, IAT，注意，其对应的法语名为Temps Atomique International，所以其常用缩写也为TAI。笔者此处采用英文缩写 IAT）、协调世界时间（Coordinated Universal Time, UTC）、GPS时间（GPS Time, GPST）和本地时间（Local Time）。四种时间系统的特点如下。

#### (1) IAT

1967年，人们利用铯原子振荡周期极为规律的特性研制出了高精度的原子钟，并将铯原子能级跃迁辐射9192631770周所经历的时间定为1s。IAT起始时间从1958年1月1日0时0分0秒开始，其精度能达到每日数纳秒。细心的读者可能会问到，在原子钟出现之前，人们如何定义秒呢？原来，在原子钟出现之前，人们使用基于地球自转的天文测量得到的世界时（Universal Time, UT）作为时间计量单位。和原子时比起来，UT会由于地球自转的不稳定（由地球物质分布不均匀和其他星球的摄动力等引起的）而带来时间上的差异，该差异大概在三年内会增加到1s左右。

#### (2) UTC

也叫世界统一时间、世界标准时间。TAI的精度为每日数纳秒，而UT的精度为每日数毫秒。对于这种情况，“协调世界时”于1972年面世。UTC以原子秒长为基础，在时刻上尽量接近UT。UT和UTC之间的间隔不能超过0.9s，所以在有需要的情况下会在UTC内加上正或负闰秒（Leap second）。因此，协调世界时与国际原子时之间会出现若干整数秒的

差别，而位于巴黎的国际地球自转事务中央局将决定何时加入闰秒以减少UTC和IAT之间的差别。UTC时间系统用途很广。目前几乎所有国家发播的时号都以UTC为基准。另外，互联网使用的网络时间协议（Network Time Protocol, NTP）获取的时间就是UTC。UTC的时间格式为：年（y）月（m）日（d）时（h）分（min）秒（s）。

### （3）GPST

GPST也使用IAT中的原子秒为单位，其时间原点定于1980年1月6日UTC 0时。GPST比IAT慢19s，而它和UTC时间的差异为整数秒，并且这个差值会随着时间的增加而积累（2009年，GPST和UTC相差15s）。GPST时间格式由从GPST原点开始的周数和周内秒数组成。例如2009年7月9日13点08分36秒（转成时分秒格式的GPST）用GPST表示就是第1539周392916秒。参考资料[5]介绍了GPST和UTC的转换方法。

### （4）本地时间<sup>[6]</sup>

本地时间基于UTC。它将全球分为24个时区，每一时区之中心为相隔15度经线，每一国家都处于一个或以上的时区内。第一时区的中心位于格林尼治子午线（简称子午线）。该时区以西的地方慢一小时或以上，而东面则较其快。本地时间表达方法遵循ISO 8601，其格式为“年月日T时分秒Z（或者时区标识）”。例如，20131030T093000Z，表示2013年10月30日09点30分0秒，Z表示标准时间。北京时间，就是20131030T093000+08，其中“+08”表示东八区。

提示 以上是本书和时间系统相关的知识。这部分内容原本非常复杂，还涉及较多天文方面的概念。在此，建议读者先掌握本节所述内容。

## 4. 卫星轨道相关知识

本节将介绍卫星轨道等方面的知识。首先是卫星运行所遵循的开普勒三定律。

### （1）开普勒三定律

卫星围绕地球运行时将遵循开普勒三定律。图9-6所示为开普勒第一和第二定律的示意图。

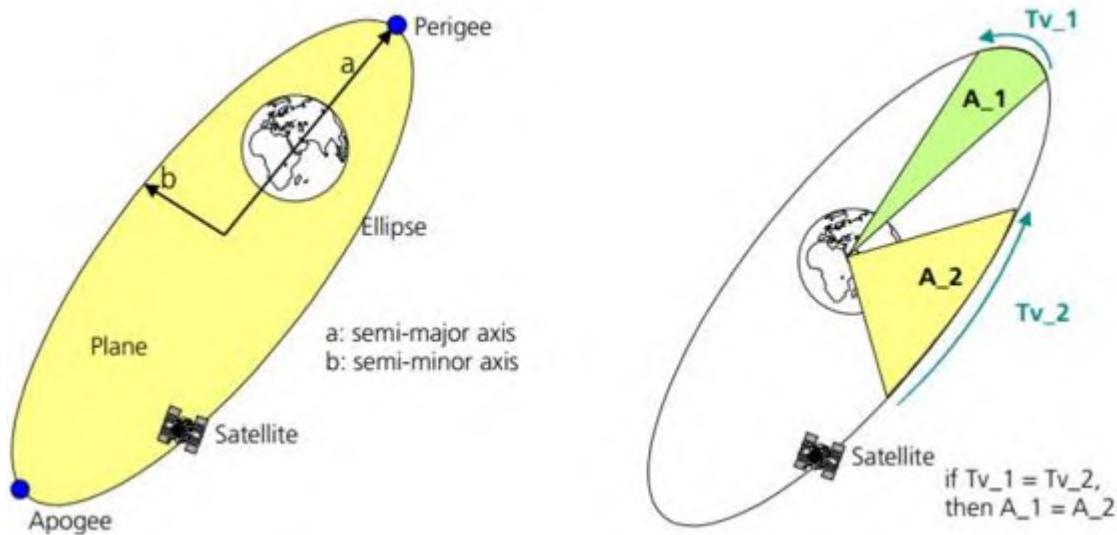


图9-6 开普勒第一和第二定律

左图所示为开普勒第一定律。图中的Perigee为近地点，Apogee为远地点。根据开普勒第一定律<sup>③</sup>，卫星将围绕地球做椭圆运动，地球为该椭圆两个焦点中的一个。

右图所示为开普勒第二定律。根据开普勒第二定律，在相同的时间内，卫星运行时所扫过的区域的面积相同。即如果图中的时间段T<sub>v\_1</sub>等于时间段T<sub>v\_2</sub>，面积A<sub>1</sub>等于面积A<sub>2</sub>。

根据开普勒第三定律可知，围绕地球椭圆轨道运行的卫星，其椭圆轨道半长轴的立方与运行周期的平方之比为常量。第三定律可用公式二表达。

[公式二]

$$k = \frac{a^3}{T^2}$$

开普勒第三定律中，a为半长轴，T为卫星运行周期，k为常量，取值为 $\frac{GM}{4\pi^2}$ 。其中，M为地球的质量，G为万有引力常数。

开普勒三定律主要用来计算卫星运行位置等相关参数，例如第三定律常用来计算卫星的轨道高度。这部分内容见参考资料[7]。

## (2) 卫星轨道及星历

卫星轨道虽然涉及很多空间科学方面的知识，但对于本书来说，我们只需掌握卫星运行轨道的几个重要参数和概念即可。图9-7展示了卫星运行轨道及相关参数。

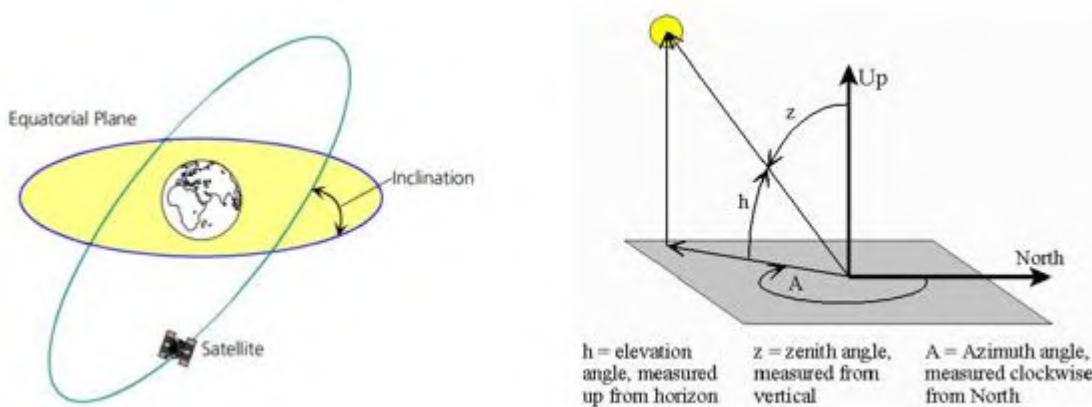


图9-7 卫星运行轨道

左图中，Equatorial Plane为赤道平面，卫星轨道本身是一个椭圆轨道，它和赤道平面有一个夹角。这个夹角叫轨道倾角（图中的 Inclination）。右图中，假设观察者站在坐标原点观察左上角的卫星，则 $h$ 代表仰角（Elevation angle）， $z$ 代表天顶角（Zenith angle），而正北方向离卫星投影点的顺时针角度 $A$ 为方位角（Azimuth angle）。

**提示** 上述参数是卫星运行轨道中几个非常重要的参数，不过，读者现在只需要记住它们的定义即可。

根据轨道倾角、运行周期等参数，人们将卫星轨道分为如图9-8所示的几大类。

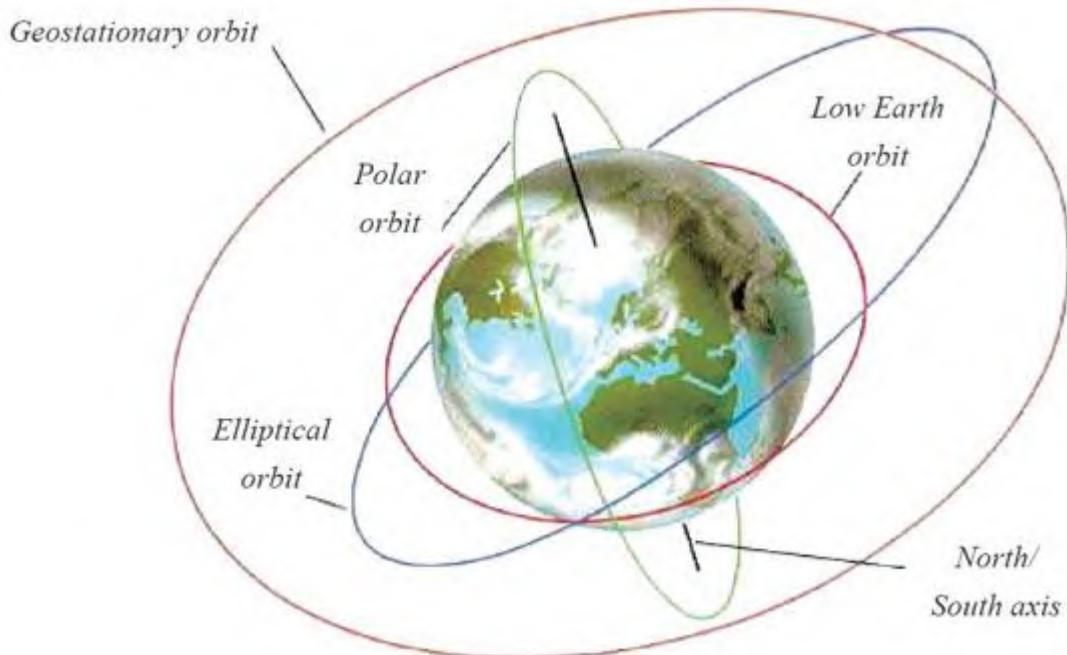


图9-8 卫星轨道分类

1) 地球同步轨道 (Geosynchronous Earth Orbit, GEO) : 特点是其轨道高度距离地面大约35786km, 卫星运行周期等于地球自转周期 (23小时56分4秒), 卫星运行方向和地球自转方向一致。最后, 轨道是圆形 (即偏心率为0)。根据轨道倾角的不同, 地球同步轨道还可细分为静止同步轨道、倾斜同步轨道和极地同步轨道。这三者的特点如下文所述。

- 静止同步轨道 (Geostationary Satellite Orbit, GSO) : 如果轨道面与地球赤道面重合 (即轨道倾角为0), 则这种轨道叫静止同步轨道。该轨道的特点是: 从地面观察者看到该轨道上的卫星始终位于某一位置, 似乎保持静止不动。利用该轨道上的3颗卫星就可以实现除南北极很小一部分地区外的全球通信。
- 倾斜同步轨道 (Inclined Geostationary Orbit, IGSO) : 如果轨道倾角大于0并小于90度, 这种轨道叫倾斜同步轨道。
- 极地同步轨道 (Polar Earth Orbit, PEO) : 如果轨道倾角等于90度, 称为极地同步轨道。运行在这种轨道上的卫星能到达南北极区上

空，所以那些需要在全球范围内进行观测和应用的气象卫星等多采用这种轨道。

2) 中地球轨道 (Medium Earth Orbit, MEO) : 也叫中圆轨道，距离地面10000km，卫星运转周期在2至12小时之间。运行在该轨道上的卫星大部分是导航卫星，例如GPS导航卫星有一部分运行在该轨道上。

3) 低地球轨道 (Low Earth Orbit, LEO) : 也叫近地轨道或低地轨道，距离地面大约1000km。由于近地轨道离地面较近，绝大多数对地观测卫星、测地卫星、空间站都采用近地轨道。

4) 高椭圆轨道 : 是一种具有较低近地点和极高远地点的椭圆轨道，其远地点高度大于静止卫星的高度 (36000km)。根据开普勒定律，卫星在远地点附近区域的运行速度较慢，因此这种极度拉长的轨道的特点是卫星到达和离开远地点的过程很长，而经过近地点的过程极短。这使得卫星对远地点下方的地面区域的覆盖时间可以超过12小时。具有大倾斜角度的高椭圆轨道卫星可以覆盖地球的极地地区，所以对于像俄罗斯这样的高纬度国家而言，高椭圆轨道比同步轨道更有实际作用。

以上是卫星运行轨道的几个重要参数<sup>④</sup>，除此之外，还有两个重要概念需要读者了解。

- 星历表 (Ephemeris) : 本来用来记录天体特定时刻的位置的。而在GNSS中，星历表则记录了卫星的一些运行参数，它使得我们通过星历表就可以计算出任意时刻的导航卫星的位置和速度。下文我们将见到在GPS中，星历表包含了非常详细的卫星轨道和位置信息，所以其数据量较大，传输时间较长。为了克服这个问题，人们设计了星历表的简化集，即历书。

- 历书 (Almanac) : 包含卫星的位置等相关信息，不过它是星历数据的简化集，其精度较低。所以，历书数据量较小，传输时间较短。

**提示** 星历和历书对于GPS定位计算来说至关重要。本章后文将介绍二者所包含的参数信息。

至此，本书所涉及的与卫星导航原理相关的知识介绍就告一段落，这些内容对于讲解本章知识点来说已经足够。但本节所述内容仅仅是卫

星导航全部知识的一小部分，有志从事卫星导航工作的读者还需要进一步花费时间来学习相关的专业知识。

- ① 偏心率也叫扁率，对应的英文名为flattening。
- ② 时间系统相关的知识非常复杂，参考资料[4]介绍得最为简练。
- ③ 开普勒三大定律本来描述的是行星在宇宙空间绕太阳公转所遵循的定律。不过导航卫星围绕地球运行也遵循此定律。所以笔者直接以导航卫星和地球为对象来介绍开普勒三大定律。
- ④ 卫星运行轨道的分类总结由笔者提炼并整理从网上搜索到的相关内容而来。

## 9.2.2 GPS系统组成及原理

本节将主要介绍和GPS相关的基础知识，先来看GPS系统的组成<sup>[8]</sup>。如图9-9所示。

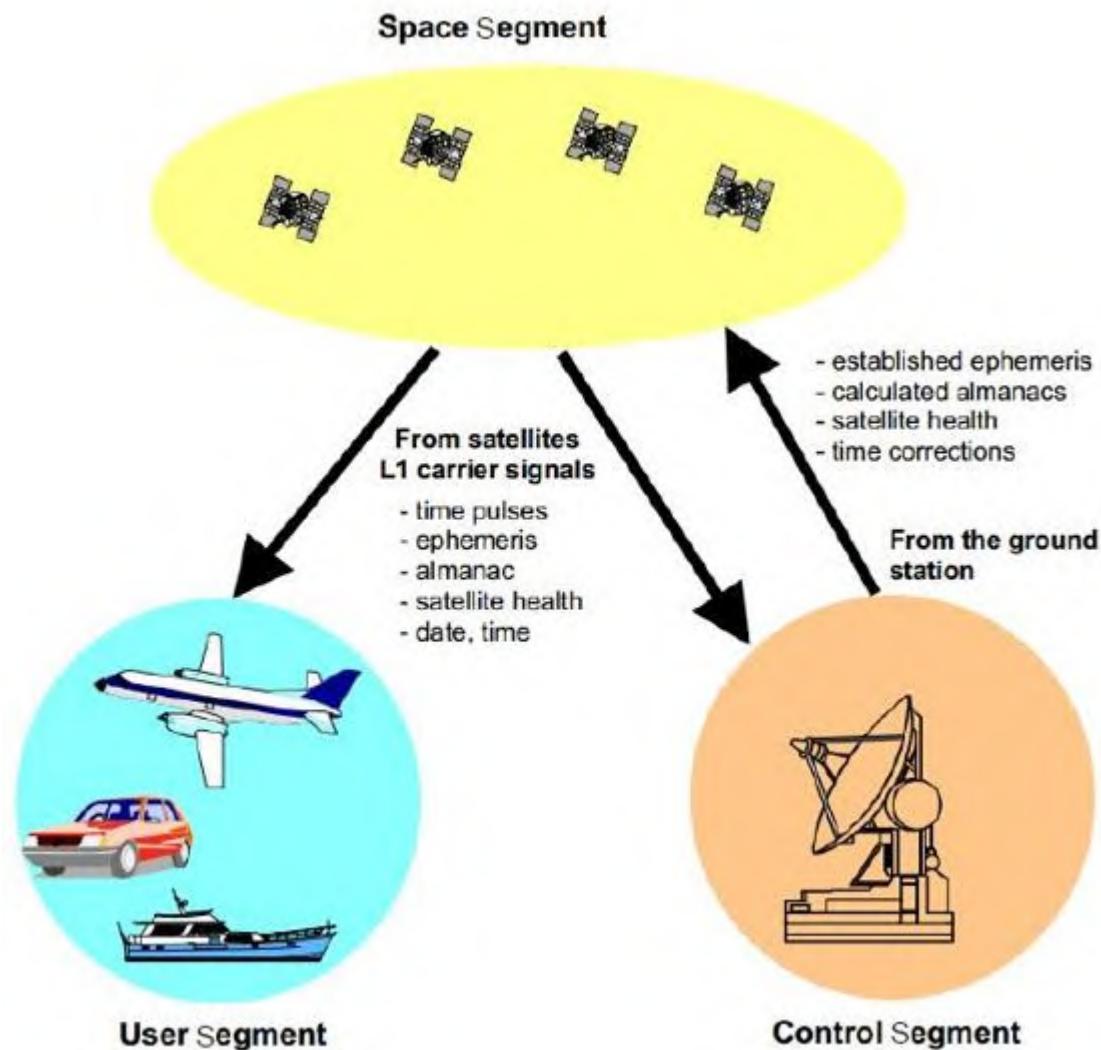


图9-9 GPS系统组成

由图9-9可知，GPS包含如下三个段。

- 空间段（Space Segment, SS）：空间段由GPS卫星组成。

- 控制段（Control Segment, CS）：控制段用来控制和监视GPS的运行。控制段包括一个主控站（Master Control Station，位于美国科罗拉多州）、数个监控站（Monitoring Station）、地面控制站（Ground Control Station）以及地面天线（Ground Antenna）。图9-10为目前GPS系统的CS站点分布图。

- 用户段（User Segment, US）：用户段主要是GPS的使用者。GPS中，用户被分为民用用户（Civilian Users）和军用用户（Military Users）两大类。其中，军用用户需要得到相关部门的授权才能获取更高精度的GPS数据。

GPS这三个段将借助GPS规定的通信频段以及数据封装格式进行通信。其中，空间段和控制段能双向通信，而用户段只能从空间段获取数据。

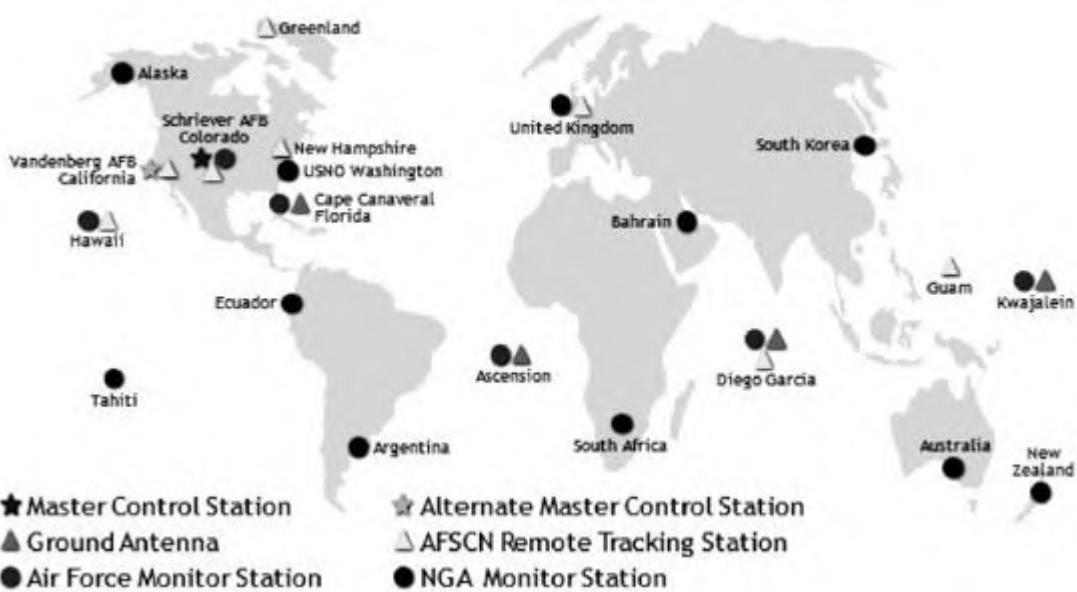


图9-10 CS站点分布图

说明 GPS提供两种类型的服务，分别是标准定位服务（Standard Positioning Service, SPS）和精密定位服务（Precision Positioning Service, PPS）。其中，SPS主要面向全世界的民用用户，而PPS主要面向美国及其盟国的军事部门以及民用的特许用户。

下面介绍GPS空间段以及GPS通信及数据包方面的知识。

## 1. GPS空间段

GPS空间段的建设历经了30多年的时间。表9-1展示了这期间GPS卫星更新换代的几次重要事件。

表 9-1 GPS 空间段建设大事记<sup>[9]</sup>

计划名	卫星发射时间段	卫星特点	其他
Block I	1978 ~ 1985	第一代 GPS 卫星，主要为了验证 GPS 的可行性	由当时的 Rockwell International 公司（现在的波音公司）制造。卫星设计寿命为 4.5 年。一共发射了 11 次，失败 1 次。目前这一代的 GPS 卫星已全部退役
Block II	1989 ~ 1990	第二代 GPS 卫星，在第一代 GPS 卫星基础上进行了一些改进，能满足 GPS 系统的要求	由 Rockwell International 公司发射，共 9 颗。设计寿命 7.5 年，现在已经全部退役
Block IIA	1990 ~ 1997	Block II 卫星的升级版，A 代表 Advanced	由 Rockwell International 公司发射，共 19 颗卫星（编号从 SVN-22 到 SVN-40）。到 2013 年 10 月，仅剩下 8 颗该类型的 GPS 卫星还在运行
Block IIR	1997 ~ 2004	R 代表 Replenishment，用于替代 II 和 IIA 类型的卫星。它支持 GPS 星载时钟监控（on-board clock monitoring）	由 Lockheed Martin 公司制造，共 13 颗卫星（编号为 SVN-47、SVN-51、SVN-54、SVN-56、SVN-59 直到 SVN-61）。目前有 12 颗该类型的卫星在运行

(续)

计划名	卫星发射时间段	卫星特点	其他
Block IIR (M)	2005 ~ 2009	M 代表 Modernized，该类型的卫星增加了一种新的民用及军用 GPS 信号	由 Lockheed Martin 公司制造，共 8 颗卫星，分别是 SVN-48、SVN-49、SVN-50、SVN-52、SVN-53、SVN-55、SVN-57 和 SVN-58。目前只有 SVN-49 号卫星不能使用
Block IIF	2010 ~ 现在	F 代表 Follow-on，这类卫星使用铷和铯做原子钟，精度提高到误差为每天 80 亿分之一秒。同时，GPS 信号强度、精确度也得到了提高	由波音公司制造，共 12 颗卫星（编号从 SVN-62 到 SVN-73）。到 2013 年 7 月 21 日为止，一共有 5 颗 IIF 型卫星处于运行状态
Block III	2013 ~ 现在	最新型的 GPS 卫星，还处于设计建设阶段	由 Lockheed Martin 公司制造，编号从 SVN-74 开始。设计寿命 15 年，支持 DASS (Distress Alerting Satellite System，一种搜救系统)

参考资料[10]总结了GPS空间段建造历史以及GPS卫星发射计划。

目前为止，GPS空间段由32颗GPS卫星<sup>①</sup>（卫星的英文名为 Satellite，也称为Space Vehicle，简写为SV）组成，这些卫星分布在6个轨道上，每个轨道与地球赤道面的倾角为55度。GPS卫星轨道高

度为20180千米，卫星在轨道上的运行周期大约为12小时。不过，由于地球的自转，人们在地面上观测GPS卫星，在23小时56分左右会回到最初的观测位置。图9-11所示为GPS卫星轨道分布图。

由于每颗GPS卫星的信号只能覆盖地球表面的一部分，所以GPS空间段在设计时就保证任何时候，地球表面任何地方都能被至少4颗GPS卫星信号覆盖。图9-12为某时刻从地面观测到的GPS卫星的位置分布图。

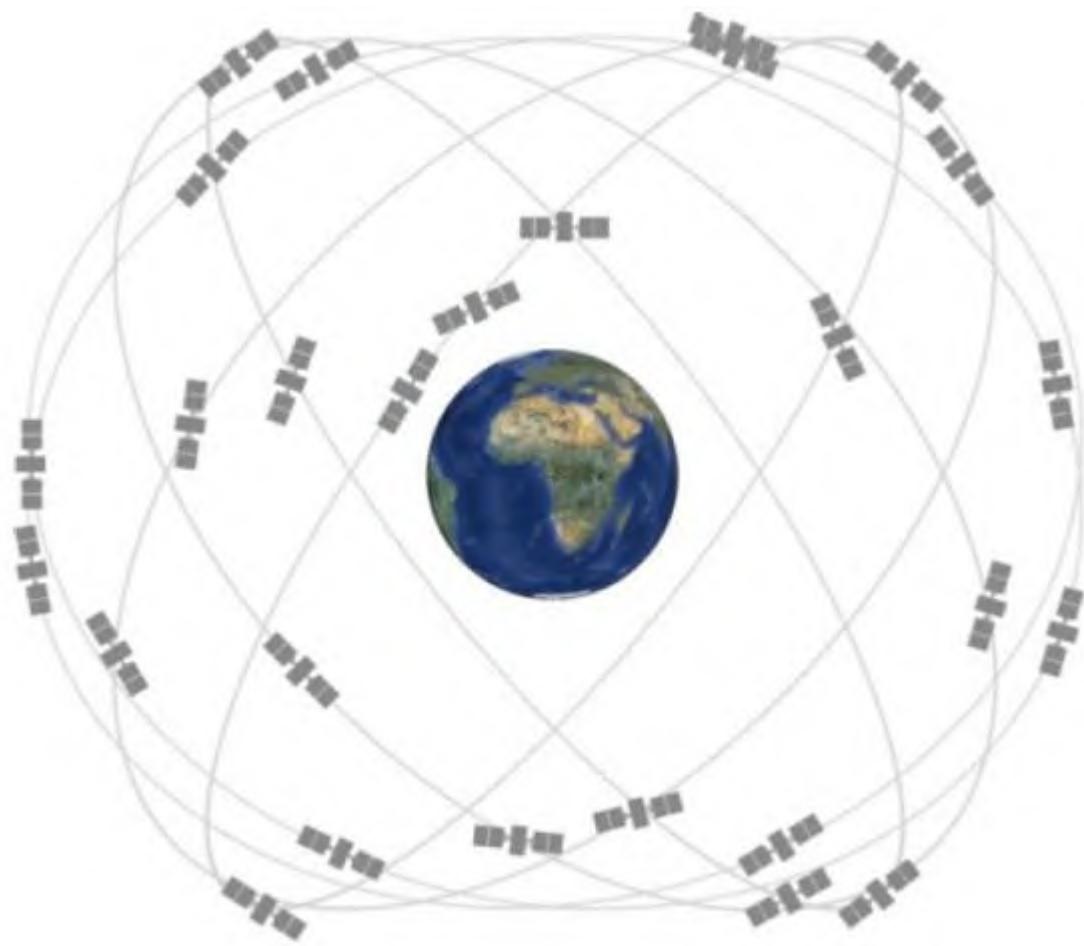


图9-11 GPS卫星轨道分布

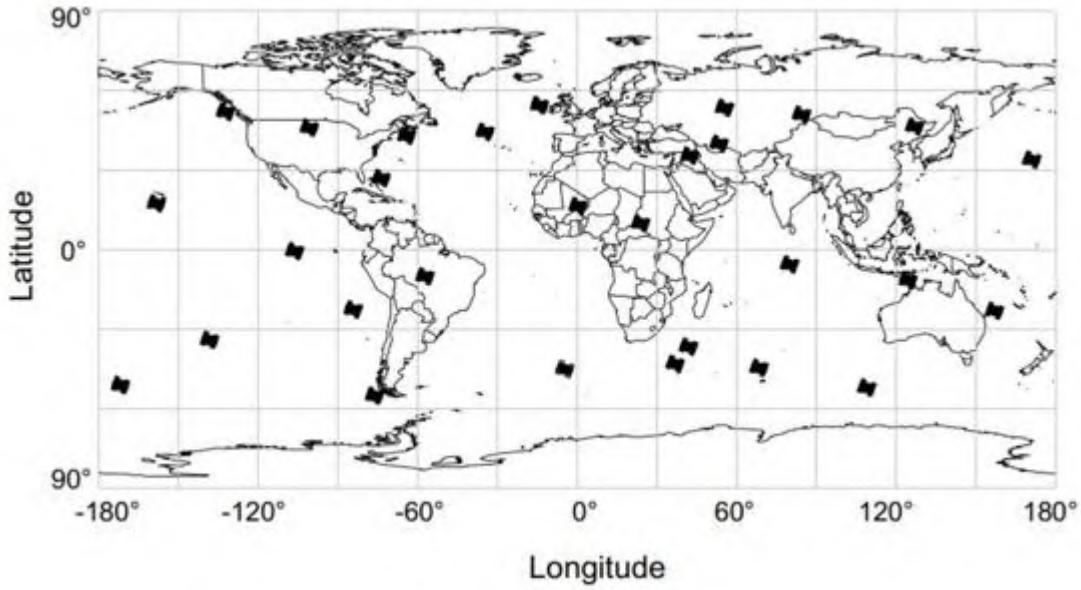


图9-12 2001年4月14日UTC时间12点整GPS卫星分布图<sup>[11]</sup>

提示 为什么要确保至少4颗卫星的信号能覆盖到地球表面任意地方呢？根据前面介绍的测距原理可知，要计算接收器的位置即（x，y，z）坐标值就需要3颗卫星，而由于接收器时钟和卫星时钟的不同步，所以还需要至少一颗卫星用来计算信号传输时间。综上，GPS定位需要至少4颗卫星参与。

在此推荐使用GpsPredict软件获取和展示GPS卫星轨道及相关信息。图9-13所示为该软件运行时的界面。

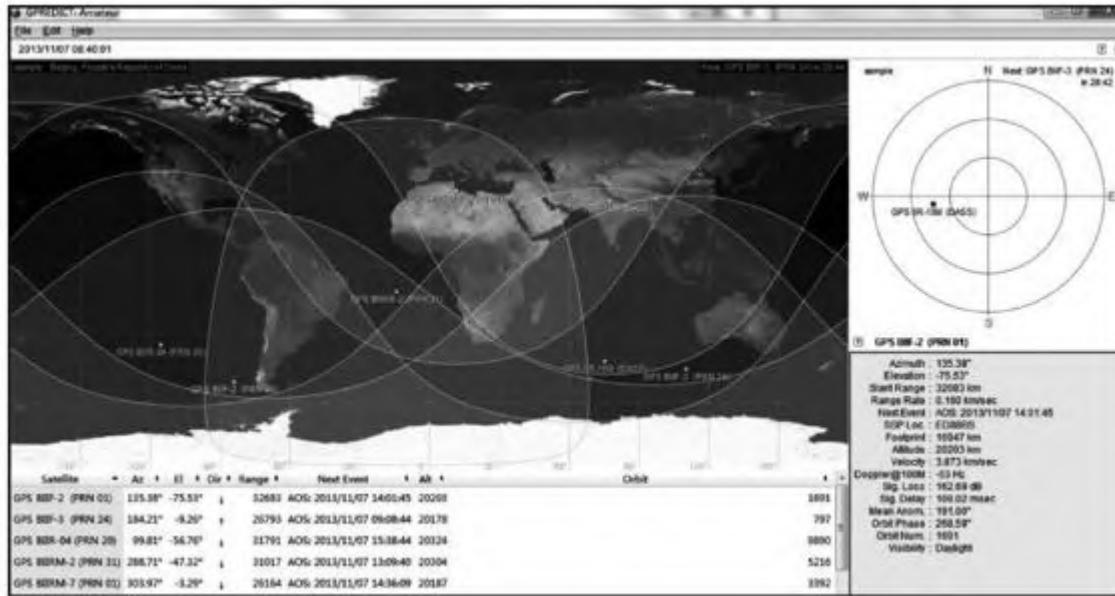


图9-13 GpsPredict运行界面

提示 GpsPredict的软件下载地址为  
<http://sourceforge.net/projects/gpredict/files/>。

## 2. GPS通信频段

GPS卫星和地面监控站以及接收器使用无线电波进行通信，GPS一共使用了三个频段的无线电波来传输数据，如图9-14<sup>[11]</sup> 所示。

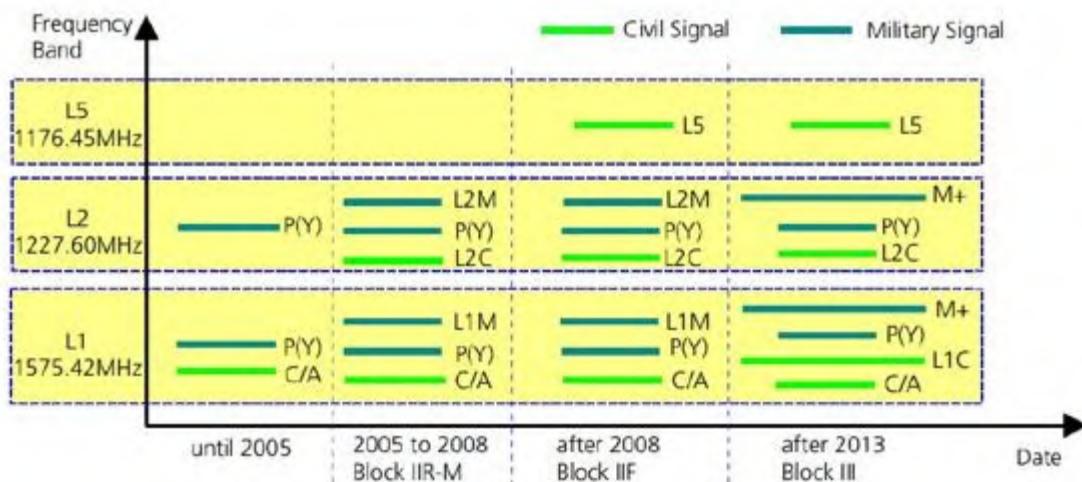


图9-14 GPS卫星通信频段

GPS一共使用三种频段的无线电波，由于它们都位于无线电频谱的L频段，所以它们分别被称为L1（中心频率1575.42MHz）、L2（中心频率1227.60MHz）和L5（中心频率1176.45MHz）。

在2005年之前，GPS卫星使用L2和L1频段的无线电波。其中，L1频段传输两种GPS信号，一个是民用的C/A码（全称是Coarse/Acquisition Code），它代表粗捕获码数据。另外一个是军用的P(Y)码，它代表精测数据（P代表Precise，Y代表数据是加密的）。L2频段仅传输P(Y)码，即仅供军用。下文还将详细GPS信号方面的知识。

IIR(M)型号的GPS卫星在L2频段上增加了一个名为L2C（C为Civil的意思）的GPS信号。L2C信号可以和C/A信号共同使用（即所谓的双频）以减少大气电离层<sup>②</sup>的影响从而提高定位精度（其精度甚至能超过军用级定位的精度，详情见参考资料[12]）。另外，L1和L2频段上新增了针对军用用户的L1M和L2M信号，它们均采用BOC（Binary Offset Code）方法进行调制和解调，可显著增强军用信号的抗干扰能力。

IIF卫星能在L5上发射民用GPS信号，这类信号主要为航空安全服务，它具有更高的功率，更大的带宽和更稳定的服务。详情见参考资料[13]。

在L1频段，III型卫星将支持一种名叫L1C的新GPS信号。L1C信号可增强GPS系统和其他GNSS系统（如中国的北斗导航系统也将广播L1C信号）之间的交互性（interoperability）。

图9-15总结了各类型GPS卫星所支持的通信频段信息。

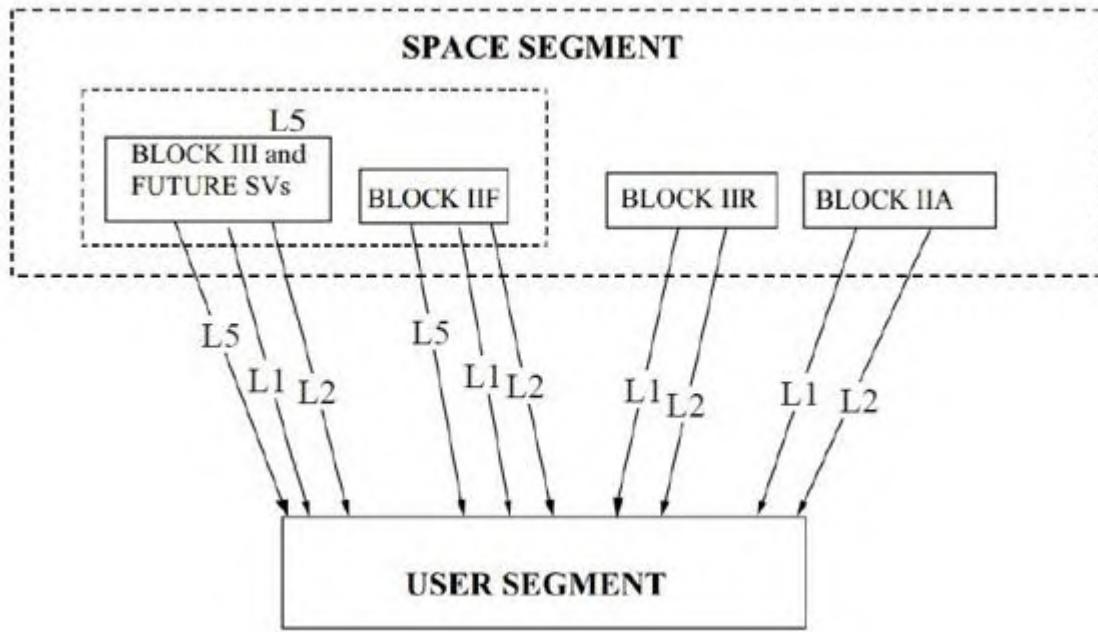
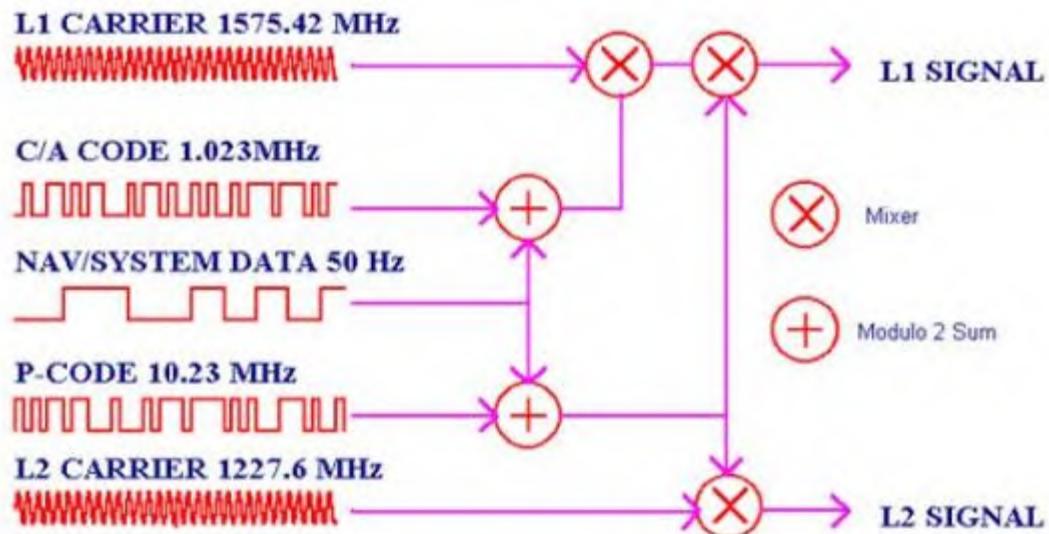


图9-15 GPS各类型卫星所支持的通信频段<sup>[14]</sup>

了解了GPS卫星通信频段的知识后，下面让我们把注意力放到GPS卫星通过这些频段所传输的数据上来，即和GPS信号相关知识点。

### 3. GPS信号<sup>[11]</sup>

GPS信号将借助上一节所述的GPS卫星通信频段进行无线电传输，它由三部分组成，如图9-16所示。



## 图9-16 GPS信号的组成

由图9-16可知，GPS信号包含主要三个组成部分。

1) 载波：分别是L1和L2（注意，本书不讨论L5的情况，感兴趣的读者可阅读参考资料[14]），其中心频率分别是1575.42MHz和1227.60MHz。

2) 测距码（Ranging Code）：用来测量卫星和接收器之间距离的一种信号。测距码其实是一种经过精心设计的伪随机噪声<sup>③</sup>（Pseudo-Random Noise, PRN）。

GPS有C/A码和P码两种测距码。

- C/A码（粗捕获码），频率为1.023MHz，周期为1ms，码长为1023，码元的宽度为293.05m，测距精度为2m到3m。
- P码（精捕获码），频率为10.23MHz，是和粗捕获码对应的测距码，其周期为7天，码长为 $6.1871 \times 10^{12}$ ，码元周期0.097752微秒，相应码元宽度为29.3m，测距精度为0.3m。P码供军事应用，故可以对它进行密。加密后的P码称为“Y码”。

3) 导航电文（Navigation Data, 也叫D码）：在定位计算时，除了测距码外还需要卫星的一些信息，例如星历、时间等。这些数据封装在GPS导航电文中，其传输频率为50比特每秒（即50Hz）。导航电文的详情见下节。

C/A码仅在L1频段上发送，而P码同时在L1和L2频段发送，根据前面介绍的双频知识，接收器可通过接收L1和L2频段的P码以消除大气电离层造成的延时影响从而进一步提高定位精度。

注意 在数字通信中，一个数字脉冲称为一个码元。一个周期中码元的个数称为码字的长度，简称为码长，常用n表示。

C/A码的码元宽度为293.05m，这是通过以下公式得来。

光速 (300000\*1000m)

码长 (1023) \* 周期 (1ms\*1000)

其测距精度是如何计算出来的呢？接收器在工作时会生成一个C/A码，这个C/A码将和某个卫星发送的C/A码进行匹配。匹配时涉及码相位数字信号处理方面的工作，理想情况下其最高精度能达到码元宽度的1%，所以C/A码的测距精度为 $293.05 \times 1\%$ （约3m）。

关于GPS信号方面的进一步知识，请读者阅读参考资料[15]。下面来看看GPS导航电文的内容。

#### 4. GPS导航电文

##### (1) 数据格式及内容

GPS导航电文 (Navigation Message) 有其特定的格式，如图9-17<sup>[11]</sup>所示。

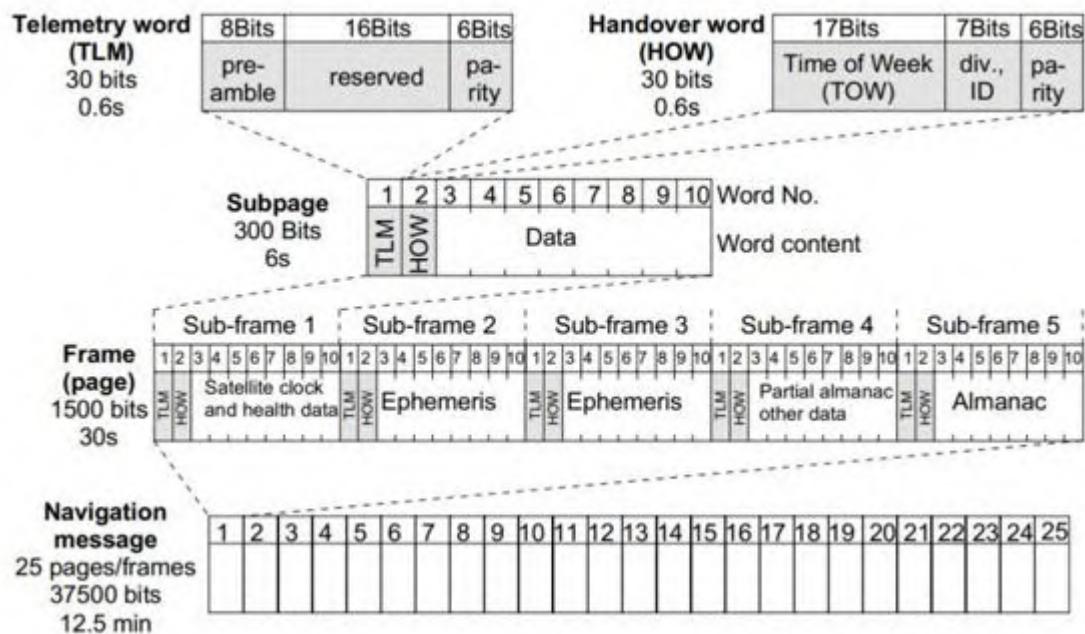


图9-17 GPS导航电文格式

如图9-17所示，GPS导航电文的基本单位是帧（Frame，也叫Page）。一帧包含1500比特。导航电文的传输速率是每秒50比特，故传输完整的一帧数据需30s。

每一帧中的1500比特又被平均分配，每300比特组成一个子帧（Sub-Frame），所以一帧包含5个子帧。每一个子帧又由10个字码（word）组成，每一个字码包含30位数据。子帧的第一个字码叫TLM（Telemetry Word，遥测码），第二个字码叫HOW（Handover Word，转换字）。下文将详细介绍TLM和HOW的组成。

一个完整的GPS导航电文由25帧组成，共37500比特，故全部传输完它们共需12.5分钟。

表9-2为读者总结了GPS导航电文25帧所包含的数据。

表 9-2 导航电文帧数据内容<sup>[16]</sup>

子 帧	帧	数 据
1	1 ~ 25	发送卫星的 GPS 时间等信息
2, 3	1 ~ 25	发送卫星的星历数据
	1, 6, 11, 16, 21	保留
	2, 3, 4, 5, 7, 8, 9, 10	SV-25 到 SV-32 的历书数据
	12, 19, 20, 22, 23, 24	保留
4	13	NMCT (Navigation Message Correction Table, 导航电文改正表)
	14, 15	保留给系统使用
	17	特殊消息 (Special Message)
	18	电离层参数和 UTC 数据
	25	32 颗卫星的配置情况 (如 Anti-Spoofing 反欺骗是否开启) 以及 SV-25 到 SV-32 的卫星健康状况
5	1 ~ 24	SV-1 到 SV-24 的历书数据
	25	SV-1 到 SV-24 的健康状况及历书参考时间等

由表9-2可知，所有的25帧数据中，其子帧1到子帧3的内容相同。它们都用来描述信号发射卫星的一些信息。此处特别提醒读者，子帧1~3包含的是某颗卫星自己的GPS时间和星历数据。所以，对地面接收器来说，某颗卫星的数据每隔30s（每一帧传输的时间为30s，而每一帧的前三个子帧都包含了该卫星最新的信息）就可以得到更新。

特别注意 一个GPS卫星所发送的导航电文包括自己的星历数据以及其他卫星的历书数据。

现在来看TLM和HOW的内容，如图9-18<sup>[16]</sup> 所示。

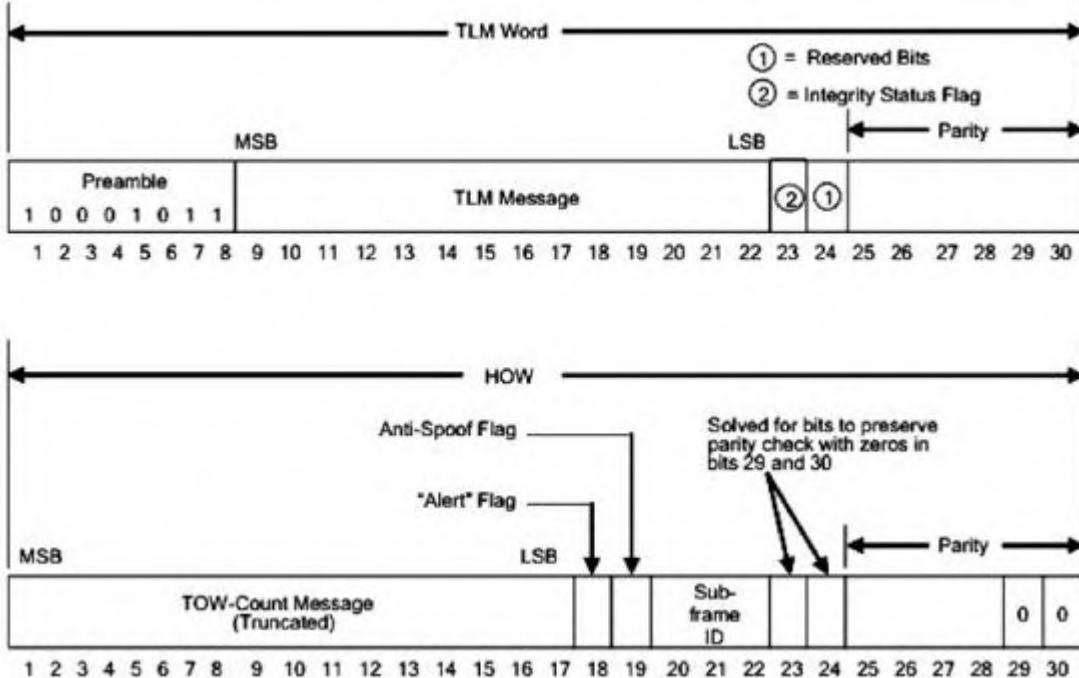


图9-18 TLM和HOW的构成

图9-18中，TLM前8位（Preamble，也叫前导码）由用于同步的二进制数10001011开始。其第9~22位为TLM消息，供PSS用户及控制段和空间段使用。第23位为完整性状态标志（Integrity Status Flag，ISF），第24位保留，最后六位是奇偶校验码。

HOW前17位用于传输星期时间（Time of the Week，TOW），第18位为警告（Alert）标志，该值为1时将提醒SPS用户测量精度较差。第19位为反欺骗（Anti-Spoof，A-S）标志，该值为1表示A-S功能开启。20到22位为子帧的ID（一个帧中包含五个子帧，子帧的ID从1开始编号）。最后几位用于奇偶校验。

## (2) 星历和历书

星历和历书的内容如表9-3所示。

表 9-3 星历和历书所包含的参数<sup>②</sup>

星 历			历 书		
参数名称	位 长	含 义	参数名称	位 长	含 义
$t_{oe}$	16	Reference Time Ephemeris	$t_{ea}$	8	Reference Time Almanac
e	32	Eccentricity (偏心率)	e	16	Eccentricity (偏心率)

(续)

星 历			历 书		
参数名称	位 长	含 义	参数名称	位 长	含 义
$\Omega_0$	32	Longitude of Ascending Node of Orbit Plane at Weekly Epoch	$\Omega_0$	24	Longitude of Ascending Node of Orbit Plane at Weekly Epoch
$\sqrt{A}$	32	Square Root of the Semi-Major Axis	$\sqrt{A}$	24	Square Root of the Semi-Major Axis
$\omega$	32	Argument of Perigee	$\omega$	24	Argument of Perigee
$M_0$	32	Mean Anomaly at Reference Time	$M_0$	24	Mean Anomaly at Reference Time
$\Delta n$	16	Mean Motion Difference From Computed Value	$a_{\text{ff}}$	11	用于定位计算的参数
$C_{rc}$	16	Amplitude of the Cosine Harmonic Correction Term to the Orbit Radius	$a_{\text{ff}}$	11	用于定位计算的参数

(一) 注意, 表9-2中并未列出全部的星历和历书数据项, 感兴趣的读者请参考[16]。

注意, 表9-3中仅包含了星历和历书全部参数项的一部分。完整的星历和历书参数定义见参考资料[16]的Table 20-III和Table20-VI。另外, 从上表中读者也会发现, 对于同样的参数而言, 其在历书数据中的精度要比它在星历数据中的精度低(即参数的位长较短)。

提示 为了避免翻译不准带来的误解, 表9-3中的参数含义说明直接使用了其在官方文档中的英文说明。参考资料[16]也有数学公式描述这些参数的作用。

## 5. 定位计算相关知识

### (1) 定位计算原理<sup>[17]</sup>

本节将介绍GPS定位计算相关的知识。在“测距原理介绍”一节中我们曾提到说要计算三维坐标系中接收器的位置需要三颗GPS卫星, 而为了解决接收器和GPS卫星时钟的不同步问题, 则需要第四颗GPS卫星参与计算。图9-19展示了定位计算的原理图。

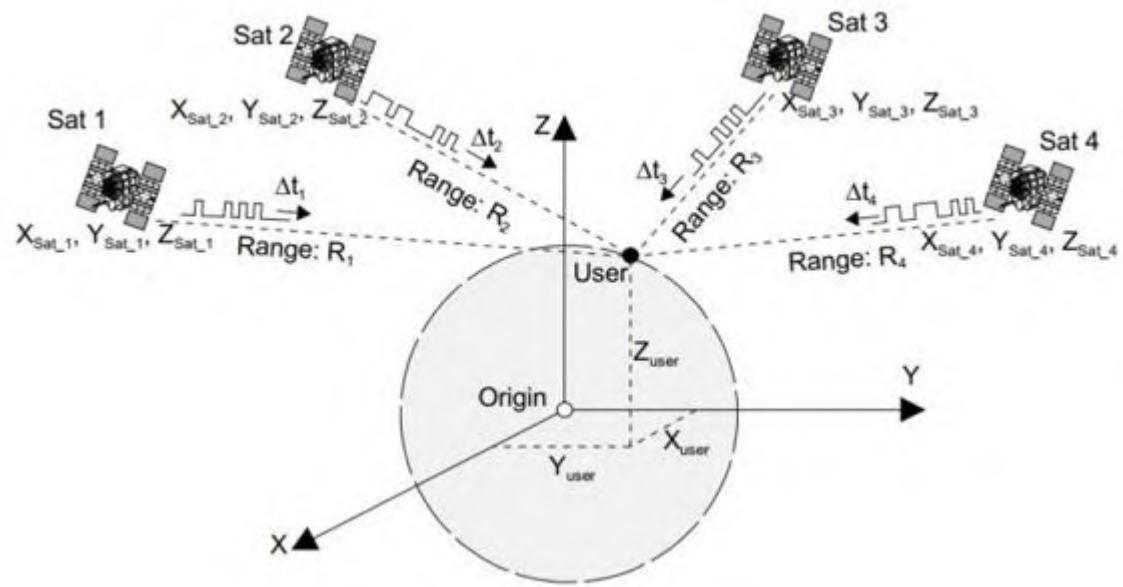


图9-19 GPS定位计算原理

图9-19中，接收器的位置由  $(X_{user}, Y_{user}, Z_{user})$  表示。

GPS卫星的位置由  $(X_{Sat\_i}, Y_{Sat\_i}, Z_{Sat\_i})$  唯一确定（注意，接收器可根据卫星的星历等参数信息将卫星在ECI坐标系的值转换成ECEF坐标系的值）。GPS卫星发送的GPS信号到达接收者所用的传输时间由  $\Delta t_i$  表示。这样，每个GPS卫星到接收器的距离就可以计算出来。图9-19中，该距离由  $R_i$  表示。

现在来考虑GPS卫星与接收器的时间误差问题。借助高精度的原子时钟以及地面控制站的监控与修正，可以认为GPS卫星之间的时钟是同步。这样，GPS卫星和地面接收器的时间误差就可以用一个参数来表示了。

基于上述内容，得到公式三。

[公式三]

$$\Delta t_{measured} = \Delta t + \Delta t_0$$

$$PSR = \Delta t_{measured} \cdot c = (\Delta t + \Delta t_0) \cdot c$$

$$PSR = R + \Delta t_0 \cdot c$$

$\Delta t_0$  为接收器的时间误差， $\Delta t$ 为真实的信号传输时间， $\Delta t_{\text{measured}}$  为接收器的GPS信号传输时间。PSR为伪距（pseudorange）。R为GPS卫星到接收器的距离。

显然，在三维笛卡尔坐标系中，R的值可由下面的公式计算得到。

$$R = \sqrt{(X_{\text{Sat}} - X_{\text{user}})^2 + (Y_{\text{Sat}} - Y_{\text{user}})^2 + (Z_{\text{Sat}} - Z_{\text{user}})^2}$$

最终，我们可得到一组方程式，见公式四。

[公式四]

$$PSR_i = \sqrt{(X_{\text{Sat}_i} - X_{\text{user}})^2 + (Y_{\text{Sat}_i} - Y_{\text{user}})^2 + (Z_{\text{Sat}_i} - Z_{\text{user}})^2 + \Delta t_0}$$

以图9-19为例，上述公式的i从1~4。

如何计算上述方程组呢？一种常用的方法是通过泰勒级数将其线性化，然后再借助偏微分方程求解。图9-20展示了该方法的原理。

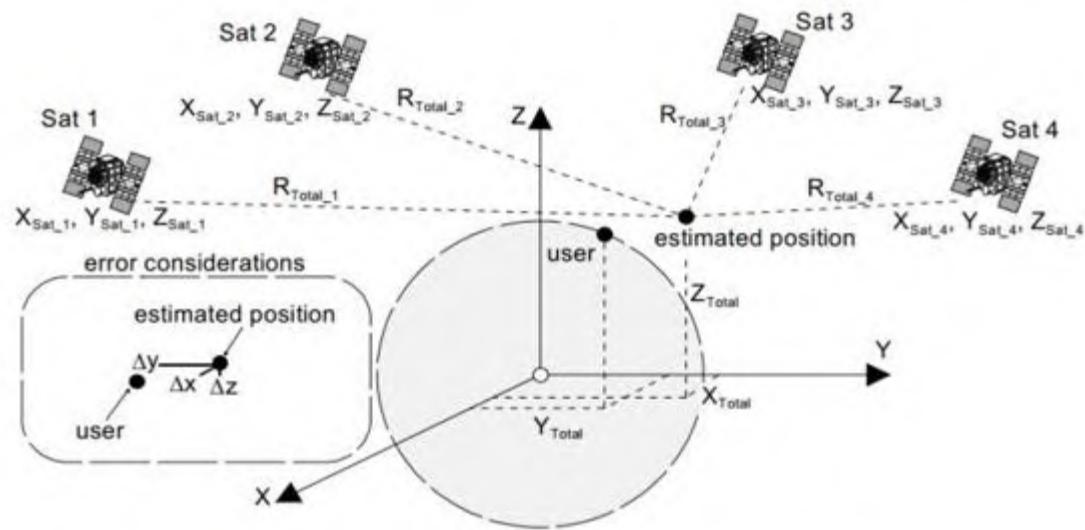


图9-20 公式四求解原理

如图9-20所示，一个新的位置点被引入，该位置点叫估算位置点（Estimated Position）。GPS卫星离该位置点的距离由 $R_{Total\_i}$  表示。

估算位置点离接收器真实的位置之间有一段距离，二者坐标的差别由公式五表达。

[公式五]

$$X_{user} = X_{Total} + \Delta x$$

$$Y_{user} = Y_{Total} + \Delta y$$

$$Z_{user} = Z_{Total} + \Delta z$$

$$R_{Total\_i} = \sqrt{(X_{Sat\_i} - X_{Total})^2 + (Y_{Sat\_i} - Y_{Total})^2 + (Z_{Sat\_i} - Z_{Total})^2}$$

$X_{user}$  为接收器的X坐标， $X_{Total}$  为估算点的X坐标，二者的差值为 $\Delta x$ 。

经过一系列的公式替换和变量求偏导，可得到图9-21所示的矩阵。

$$\begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \\ \Delta t_0 \end{bmatrix} = \begin{bmatrix} \frac{X_{Total} - X_{Sat\_1}}{R_{Total\_1}} & \frac{X_{Total} - Y_{Sat\_1}}{R_{Total\_1}} & \frac{Z_{Total} - Z_{Sat\_1}}{R_{Total\_1}} & c \\ \frac{X_{Total} - X_{Sat\_2}}{R_{Total\_2}} & \frac{X_{Total} - Y_{Sat\_2}}{R_{Total\_2}} & \frac{Z_{Total} - Z_{Sat\_2}}{R_{Total\_2}} & c \\ \frac{X_{Total} - X_{Sat\_3}}{R_{Total\_3}} & \frac{X_{Total} - Y_{Sat\_3}}{R_{Total\_3}} & \frac{Z_{Total} - Z_{Sat\_3}}{R_{Total\_3}} & c \\ \frac{X_{Total} - X_{Sat\_4}}{R_{Total\_4}} & \frac{X_{Total} - Y_{Sat\_4}}{R_{Total\_4}} & \frac{Z_{Total} - Z_{Sat\_4}}{R_{Total\_4}} & c \end{bmatrix}^{-1} \cdot \begin{bmatrix} PSR_1 - R_{Total\_1} \\ PSR_2 - R_{Total\_2} \\ PSR_3 - R_{Total\_3} \\ PSR_4 - R_{Total\_4} \end{bmatrix}$$

图9-21 GPS定位矩阵计算公式

接收器将利用图9-21所示的公式进行迭代计算，直到图左边的 $\Delta x$ 等参数值小于期望误差（如0.1m）为止。然后，接收器的位置可通过公式五和根据图9-21中得到的 $\Delta x$ 等值计算出来。

注意 除了计算接收器的坐标位置外，GPS还能计算出接收器的移动速度，这是基于多普勒效应来实现的。以GPS为例，多普勒效应就是当GPS卫星与接收器之间存在相对运动时，接收器一端收到的GPS信号的频率和GPS卫星实际发送的信号的频率并不相同，二者之差称为多普勒频移。由于GPS卫星的速度可根据其导航电文中的信息推算出来，故接收器根据多普勒频移的相关公式就很容易计算出自己的移动速度了。关于GPS测速方面的知识，可阅读参考资料[18]。

## (2) DOP介绍<sup>[17]</sup>

上一节介绍了GPS定位计算的原理。在真实环境中，GPS定位计算中还存在某些误差，这些误差的原因大体由如下几个部分组成。

- 卫星时钟：虽然卫星时钟的精度已经很高了，但由于光速的值很大，这就造成时间上10ns的偏差都会造成距离上3m的误差。
- 卫星本身的轨道位置：卫星在轨道上的位置精度在5m左右。
- 光速：GPS信号从太空中的卫星到地面接收器传输时其速度不是固定值，而是会受到电离层和对流层的影响。
- 接收器的时钟：接收器的时钟和卫星时钟不同步，这也会造成相应的误差。
- GPS信号的多路径效应：GPS信号传输过程中常会因为建筑物或其他反射物发生反射。显然，这些反射信号的传输时间比没有反射的信号的传输时间要长，这就给接收器测距时造成一定的误差。

参与定位计算的GPS卫星的空间分布也会对最终计算结果有较大影响。本节重点介绍它。

如上，本节重点关注GPS卫星空间分布情况对定位计算的影响。在GPS系统中，因卫星的空间分布造成的测距误差可用DOP (Dilution Of Precision，精度衰减因子) 等一组值来描述，这一组值如下。

- GDOP (Geometric-DOP，几何精度衰减因子)：描述卫星空间分布情况对位置计算和时间测量的影响。

- PDOP (Positional-DOP, 位置精度衰减因子)：描述卫星空间分布情况对位置计算的影响。
- HDOP (Horizontal-DOP, 水平精度衰减因子)：描述卫星空间分布情况对水平位置（二维空间）位置计算的影响。
- VDOP (Vertical-DOP, 垂直精度衰减因子)：描述卫星空间分布情况对高度计算的影响。
- TDOP (Time-DOP, 时间精度衰减因子)：描述卫星空间分布情况对时间测量的影响。

从上述各项DOP的描述可知，卫星空间分布的情况将影响定位计算的精度，这是为什么呢？来看图9-22的DOP原理。

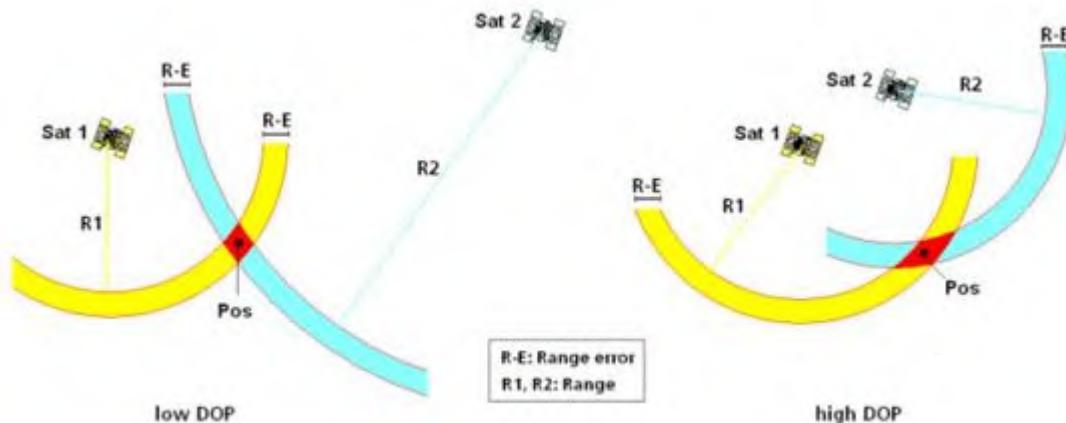


图9-22 DOP原理

左图中，Sat 1和Sat 2这两颗卫星相距较远，而右图中Sat 1和Sat 2两颗卫星相距较近。

计算接收器Pos的位置时，Sat 1和Sat 2的测距都会存在一定的测距误差。每个卫星测距所造成的误差由图中的圆环表示。根据9.2.1节对测距原理的介绍，Pos的位置位于两个圆环的重合区域之中。

很明显，左图中Pos所在的重合区域面积较小，而右图中Pos所在的重合区域面积较大。这说明左图的测距误差比右图的测距误差小。所以，左图卫星分布区域情况对应的DOP值较小，而右图的DOP较大。

**提示** 根据DOP的取值情况，定位测量的质量可划分为4个级别<sup>[15]</sup>。质量非常高：DOP值为1~3。质量高：DOP值为4~5。质量一般：DOP值为6时。质量差：DOP值大于6时。

关于DOP更详细的信息，可进一步阅读参考资料[17]。

### (3) 首次定位时间

首次定位时间（Time To First Fix, TTFF）是衡量GPS接收设备性能的一个重要指标，它描述的是GPS接收器需要花费多长时间来捕获GPS卫星信号直到计算出自己的位置。目前，TTFF因启动模式<sup>[19]</sup>不同而有所区别，这些启动模式分别如下。

- 冷启动模式（也叫出厂模式）：在这种模式下，GPS接收器没有保存有效星历、时间及位置等信息，所以它需要从周围可搜索到的GPS卫星那接收信号并获取用于定位的数据（星历、历书、时间等信息）。前面曾介绍，传输一个完整的GPS导航电文需要12.5分钟。在这种模式下，TTFF至少为12.5分钟。
- 暖启动模式：在这种模式下，GPS接收器保存有历书数据（不超过180天）、旧时间信息（不超过20秒）及旧位置信息（不超过100千米范围），但星历数据失效（超过4小时）。如此，在计算位置时，GPS接收机需要从GPS卫星那接收星历数据（读者还记得吗，导航电文的25个帧中第2、3子帧包含发送卫星的星历数据，由于每一帧发送时间为30秒，所以星历数据每隔30秒就会更新一次）。所以，在这种模式下，TTFF至少为30秒。
- 热启动模式：在这种模式下，GPS接收机具有有效星历数据、时间及位置等信息，这样，GPS接收器就无须解码GPS导航电文中的星历数据，它只要利用GPS信号进行测距计算就可以了。热启动模式下，TTFF速度很快，能做到10秒以内。

显然，减少TTFF对提升用户的使用体验有极大的帮助。根据上述内容可知，TTFF的瓶颈主要在星历、历书数据等信息的获取上。为了解决此问题，人们设计了Assisted GPS（辅助GPS）方法。AGPS使得GPS接收机能通过移动通信网络（如2G/3G等，传输速度远超GPS卫星信号的传输速度）下载星历数据等信息，从而加快首次定位时间。关于AGPS的内容，请读者阅读9.2.2节。

**提示** 如果确实需要接收和解析GPS卫星信号，多通道接收方法可用来同时接收多个卫星的信号从而提升TTFF。关于这一点见参考资料[20]。

## 6. NMEA-0183和GPX

本节将介绍和GPS相关的两种数据文件格式，先来看NMEA-0183。

### (1) NMEA-0183<sup>[21]</sup>

NMEA-0183是美国国家海洋电子协会(National Marine Electronics Association, NMEA)为海用电子设备制定的标准格式。GPS接收机可按照该标准定义的格式输出诸如定位时间、纬度、经度、高度、定位所用卫星数、DOP值等很多信息。

**提示** NMEA-0183的输出内容由ASCII字符组成。简单点说，可以用文本软件来查看和修改NMEA数据。

NMEA文件的内容由一条一条的语句组成，每一条语句都有对应的类型，表9-4列举了其中一些常用语句的类型以及它们所包含的数据信息。

表 9-4 NMEA 的语句类型

类 型	全 称	所含数据信息
GGA	GPS Fixed Data	包含时间和位置信息
GLL	Geographic Position-Latitude/Longitude	包含经度、纬度、UTC时间等信息
GSA	GNSS DOP and Active Satellites	包含DOP以及活跃卫星的信息
GSV	GNSS Satellites in View	包含可见卫星的一些信息
MSS	MSK Received Signal	包含GPS卫星信息方面的一些信息

下面我们来看一个NMEA语句示例。

#### [NMEA语句示例]

```
$GPGGA, 161229.487, 3723.2475, N, 12158.3416, W, 1, 07, 1.0, 9.0,  
M, 18.0, M, 50, 0000*18<CR><LF>
```

上面这条NMEA语句中各项信息的含义如表9-5所示。

表 9-5 NMEA 示例解释

取 值	含 义	取 值	含 义
\$GPGGA	NMEA 中，任 何 语 句 都 以 \$ 开 头。 GPGGA 表示该语句类型为 GGA	9.0, M	平 均 海 平 面 高 度 (Mean Sea Level Altitude)。M 表示单位为米
161229.487	代 表 UTC 时 间 (格 式 为 hhmmss.sss)	18.0, M	大 地 水 准 面 高 (Geoid Separation)。M 表示单位为米
3723.2475, N	数 字 代 表 纬 度 (格 式 为 ddmm.mmmm)，字母 N 代 表 北 纬 (S 代 表 南 纬)	50	该 选 项 只 有 在 使用 DGPS 时 才 有 意 义。如 果 不 使用 DGPS，则 该 选 项 为 空
12158.3416, W	数 字 代 表 经 度 (格 式 为 ddmmm.mmmm)，字母 W 代 表 西 经 (E 代 表 东 经)	0000	代 表 DGPS 参 考 站 编 号
1	定 位 指 示 器 (Position Fix Indicator)，其 取 值 为 0 表 示 此 次 定 位 无 效，1 为 使用 GPS SPS 服 务，定 位 有 效，2 表 示 使用 DGPS SPS 模 式，定 位 有 效	*18	这 条 语 句 的 校 验 和
07	表 示 使 用 了 哪 个 卫 星。卫 星 编 号 从 0 ~ 12	<CR><LF>	NMEA 语 句 以 回 车 换 行 符 结 尾
1.0	H D O P，水 平 精 度 衰 减 因 子		

NMEA的介绍就到此为止。更详细的信息请阅读参考资料[21]。

## (2) GPX<sup>[22]</sup>

和NMEA不同，GPX（GPS eXchange Format）将GPS数据封装在XML文件中，所以它遵循XML相关的语法。GPX比较简单，通过一个例子来介绍。

### [GPX示例]

```

<?xml version="1.0" ?>
<!-- GPX文件中，gpx标签是根节点-- &gt;
&lt;gpx version="1.0" creator="ExpertGPS 1.1 -
http://www.topografix.com"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns="http://www.topografix.com/GPX/1/0"
      xsi:schemaLocation="http://www.topografix.com/GPX/1/0
      http://www.topografix.com/GPX/1/0/gpx.xsd"&gt;
<!-- time标签代表该文件的创建时间--&gt;
&lt;time&gt;2002-02-27T17:18:33Z&lt;/time&gt;
<!-- bounds标签用来描述该文件中所有经度和纬度的最大及最小值 --&gt;
&lt;bounds minlat="42.401051" minlon="-71.126602"
maxlat="42.468655"
      maxlon="-71.102973" /&gt;
<!-- wpt为waypoint的简称，代表某个位置点--&gt;
&lt;wpt lat="42.438878" lon="-71.119277"&gt;
    &lt;!-- ele描述该点的海拔，单位为米 --&gt;
    &lt;ele&gt;44.586548&lt;/ele&gt;
</pre>

```

```

<!-- wpt下的time标签表示创建或修改整个wpt标签的时间，为UTC时间 -->
<time>2001-11-28T21:05:28Z</time>
<name>5066</name><!--name表示该wpt的GPS名 (GPS name of
this wpt) -->
<sym>Crossing</sym><!--sym表示该wpt的GPS符号名 (Symbol
name) >
</wpt>
<!-- 其他的wpt元素 -->
<!-- rte代表一条路径，它按顺序记录了这条路径中的关键位置点-->
<rte>
    <name>BELLEVUE</name>  <!-- rte的name标签表示路径名-->
    <number>1</number> <!-- number表示路径编号-->
    <!-- rtept: 其实就是wpt，只不过在rte标签中叫rtept-->
    <rtept lat="42.430950" lon="-71.107628">
        <ele>23.469600</ele>
        <time>2001-06-02T00:18:15Z</time>
        <name>BELLEVUE</name>
        <cmt>BELLEVUE</cmt>  <!-- cmt是comment的缩写，其作用类似于备注说明-->
        <sym>Parking Area</sym>
    </rtept>
    <!-- 其他的rtept元素 -->
</rte>
</gpx>

```

上述例子展示了GPX文件格式中的一些主要构成部分。GPX中有三个比较重要的概念。

- Waypoint : 路点，由<wpt>标签标示，代表一个感兴趣的点或者地图上的某个点。wpt英文解释为“wpt represents awaypoint, point of interest, or named feature on amap”。
- Route : 路径，由<rte>标签标示。路径由一组有序的路点构成。rte的英文解释为“rte represents route—an ordered list of waypoints representing a series of turn points leading to a destination”。注意，在Route中，wpt是转向点（turn points）。
- Track : 轨迹，由<trk>标签标示。用于记录某人从源地址到目标地址所走过的那些路点。Track的英文解释为“represents a track—an ordered list of points describing a path”。

Track和Route的区别很微妙，二者关系如图9-23所示。标有WP字样的点为Waypoint。源地址的路点为WP0297，目标地址的路点为WP0307。

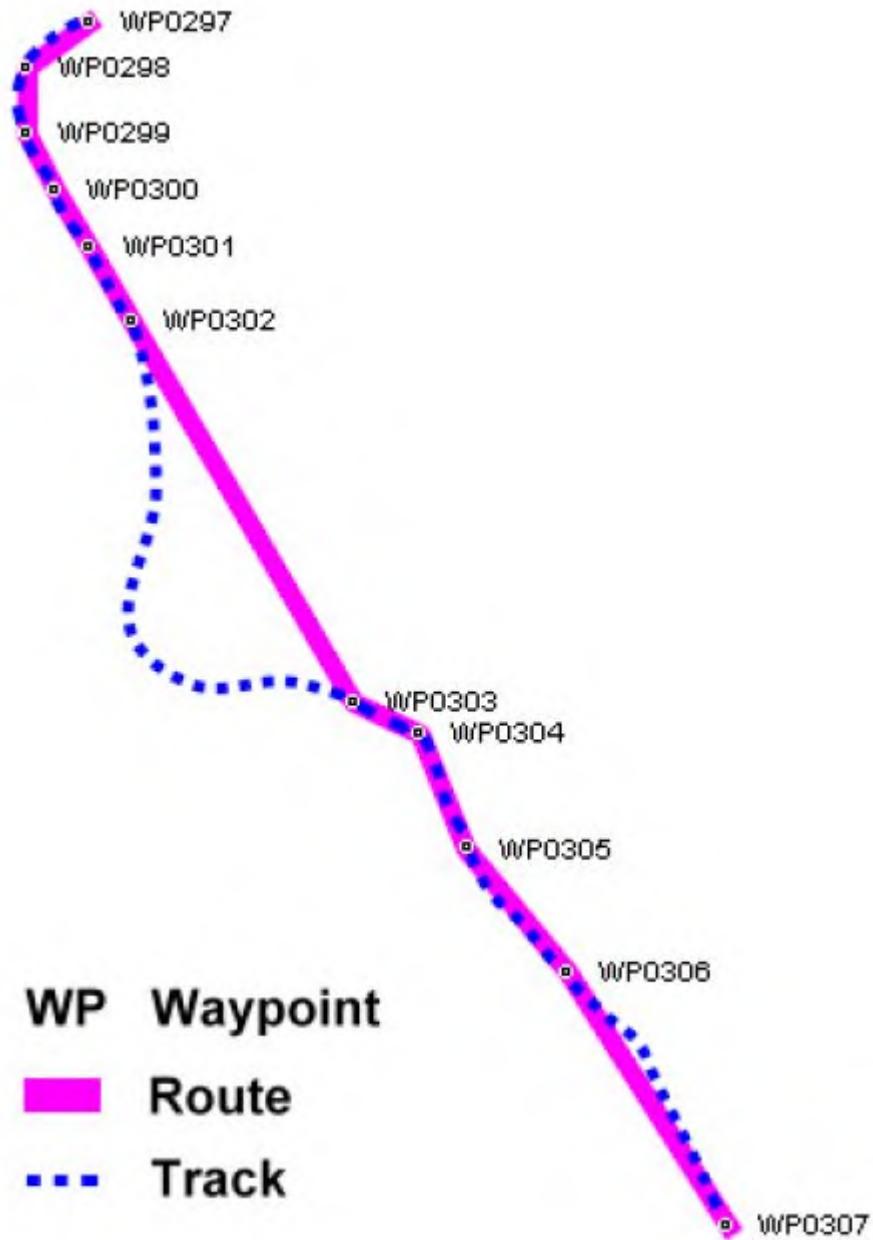


图9-23 Track和Route

图中的实线表示路径，而由路点组成的那条曲线就是轨迹。轨迹记录了某人在某个时间段从源到目标所经过的那些WP。所以在GPX文件的轨迹标签内所列出的WP同时还会记录此人经过该点时的时间信息。

和轨迹略微不同，路径表示从源点达到目标点之间的一些关键点。

**提示** 参考资料[23]描述的Track和Route的区别似乎和GPX官网所给示例（本节所分析的GPX实例就来自于GPX官网）不同。其中，路径点和是否有人在某时刻途经过它没有关系，路径点天然就存在，所以不含时间戳信息（时间信息描述的就是人们在什么时刻达到过此处）。但是本节的GPX实例中路径点却包含了时间戳信息，并且官方对路径中路点的描述是“turn point”，其英文解释是“a point at which there is a change in direction or motion”。

除了GPX外，还有一种名为KML的文件格式被谷歌地球和谷歌手机地图等软件使用。KML（Keyhole Markup Language）最初由Keyhole公司开发，是一种基于XML标准的格式。KML可描述点、线、图像等多种地理信息。Android的DDMS也支持KML。

**注意** 请读者自行研究KML的知识，相关资料见参考资料[24]。

## 7. GPS增强系统

为了提升GPS定位的精确度和易用性，人们还设计了一些GPS增强系统，这些增强系统大体可分为如下几种。

- DGPS （Differential Global Positioning System，差分GPS），用于提高GPS的定位精度。
- SBAS （Satellite-Based Augmentation System，星基增强系统），用于提高GPS定位精度以及可靠性。
- AGPS （Assisted GPS，辅助GPS），通过从移动网络下载星历等数据以提升GPS定位速度。
- HSGPS （High Sensitivity GPS，高精度GPS），用于提升GPS接收器的灵敏度。

下面介绍DGPS、SBAS以及AGPS的内容。

**注意** 关于HSGPS见参考资料[25]。

(1) DGPS和SBAS<sup>[25]</sup>

在介绍DOP时曾提到过GPS定位计算时的一些误差，而DGPS以及SBAS的目标就是减少这些误差所带来的影响。SBAS和DGPS有一定关联，所以我们先来看DGPS，其工作原理如下。

- 为了减少（或者修正）定位计算的误差，人们事先把GPS接收机放在位置已精确测定的点上，这些点叫基站。基站的接收机通过接收GPS卫星信号，测得并计算出它们到卫星的伪距，将伪距和已知的精确距离相比较，求得该点在GPS系统中的伪距测量误差。
- 然后这些基站再将这些误差作为修正值以标准数据格式通过播发台向周围空间播发。
- 在基站附近的DGPS用户一方面接收GPS卫星信号进行测距，同时它接收来自基站的误差修正信息，并以此来修正定位结果，从而提高定位精度。

DGPS用户离基站多远才算附近呢？下面有两个参考距离<sup>[15]</sup>。

- 如果使用伪距差分定位（Code Differential Positioning）技术，则DGPS和基站的距离最好在200千米以内。
- 如果使用载波相位差分定位（Carrier-Phase Differential Positioning）技术，则DGPS和基站的距离最好在20千米内。

如上所述，基站负责将修正数据以标准格式向周围空间播发。为此，人们也制订了一些协议来规范化这一工作。这些规范以及它们的优缺点如表9-6所示。

表 9-6 DGPS 修正数据传输方法介绍

播发系统名	播发频率	优 点	缺 点	数据格式
Long and medium wave broadcasters	100 ~ 600KHz	覆盖范围广，能达到 1000 千米	数据传输速率慢	RTCM SC104 <sup>②</sup>
Maritime radio beacon	283 ~ 315KHz	覆盖范围广，能达到 1000 千米	数据传输速率慢	RTCM SC104
Aviation radio beacon	255 ~ 415KHz	覆盖范围广，能达到 1000 千米	数据传输速率慢	RTCM SC104
Short wave broadcaster	3 ~ 30 MHz	覆盖范围广	覆盖范围广，信号质量容易受外界环境干扰	RTCM SC104
VHF, FM	30 ~ 300MHz	数据传输速率快，改造现有基站也很容易	覆盖范围受限	RTCM SC104
GSM、GPRS 等 移 动 网络	450, 900, 1800MHz	改造现有基站比较容易	覆盖范围受限，数据同步也有一定问题	RTCM SC104

(续)

播发系统名	播发频率	优 点	缺 点	数据格式
GEO satellite system	1.2 ~ 1.5 GHz	覆盖面广	成本较高	RTCM SC104 RTCA DO-229C <sup>②</sup>

(一) RTCM (Radio Technical Commission for Maritime, 国际海运事业无线电技术委员会)。SC 104 (Special Committee 104) 定义了 DGPS 修正数据的格式。

(二) RTCA (Radio Technical Commission for Aeronautics, 航空无线电技术委员会), DO-229C 是为它指定的一个标准。

由上表可知, 修正数据的格式主要分为 RTCM SC104 和 RTCA DO-229C 两种。当修正数据用卫星发送时, 这种系统就叫 SBAS。SBAS 使用的协议格式为 RTCA DO-229C。

当然, SBAS 的功能远不止简单地播发修正数据, 它还能监测 GPS 或其他 GNSS 卫星的情况以加强信号的可靠性和安全性。图 9-24 展示了目前几个已投入使用或在建的 SBAS 系统以及它们的覆盖范围。

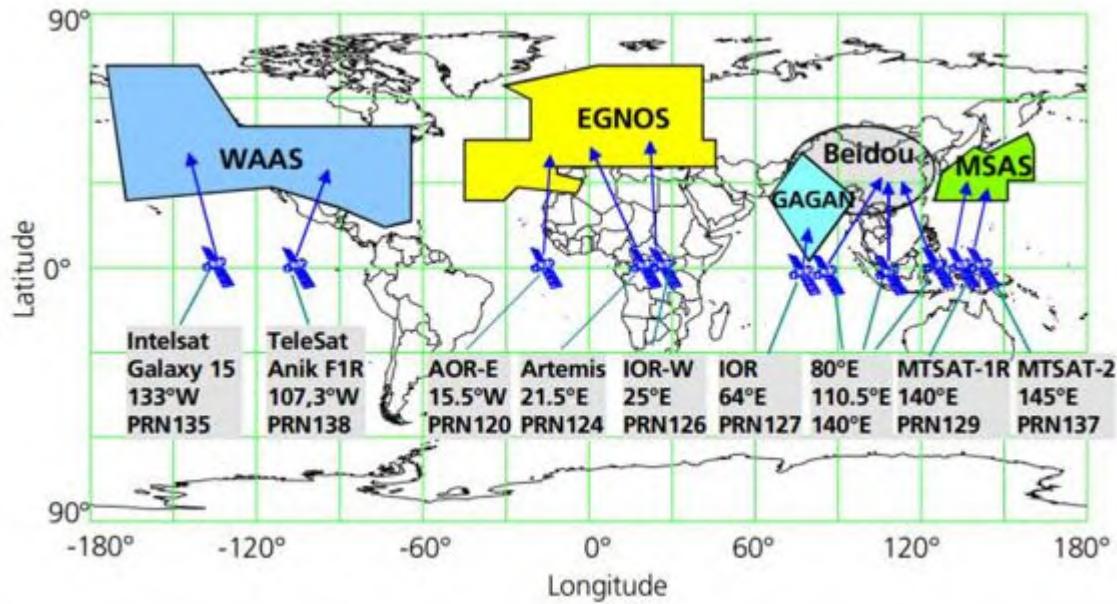


图9-24 SBAS系统

图9-24中几个主要的SBAS从左至右分别如下。

- WAAS: 美国建造的广域增强系统 (Wide Area Augmentation System)。
- EGNOS: 欧盟建造的欧洲静地导航覆盖服务 (European Geostationary Navigation Overlay Service)。
- GAGAN: 印度建造的地球同步轨道增强导航系统 (GPS And GEO Augmented Navigation)。
- Beidou: 中国建造的北斗导航系统。
- MSAS: 日本建造的多功能卫星增强系统 (Multifunctional Satellite Augmentation System)。

虽然每个单独的SBAS只能覆盖一定的范围，但通过RTCA DO-229C协议，SBAS之间的数据能够保证兼容性。

## (2) AGPS

AGPS的作用很简单，就是在没有有效星历数据等定位计算所需信息的情况下（或者在GPS信号不好甚至没有GPS信号的环境中），使得GPS接收器能通过别的方式获取所需信息以加快定位速度。AGPS的原理如图9-25所示。

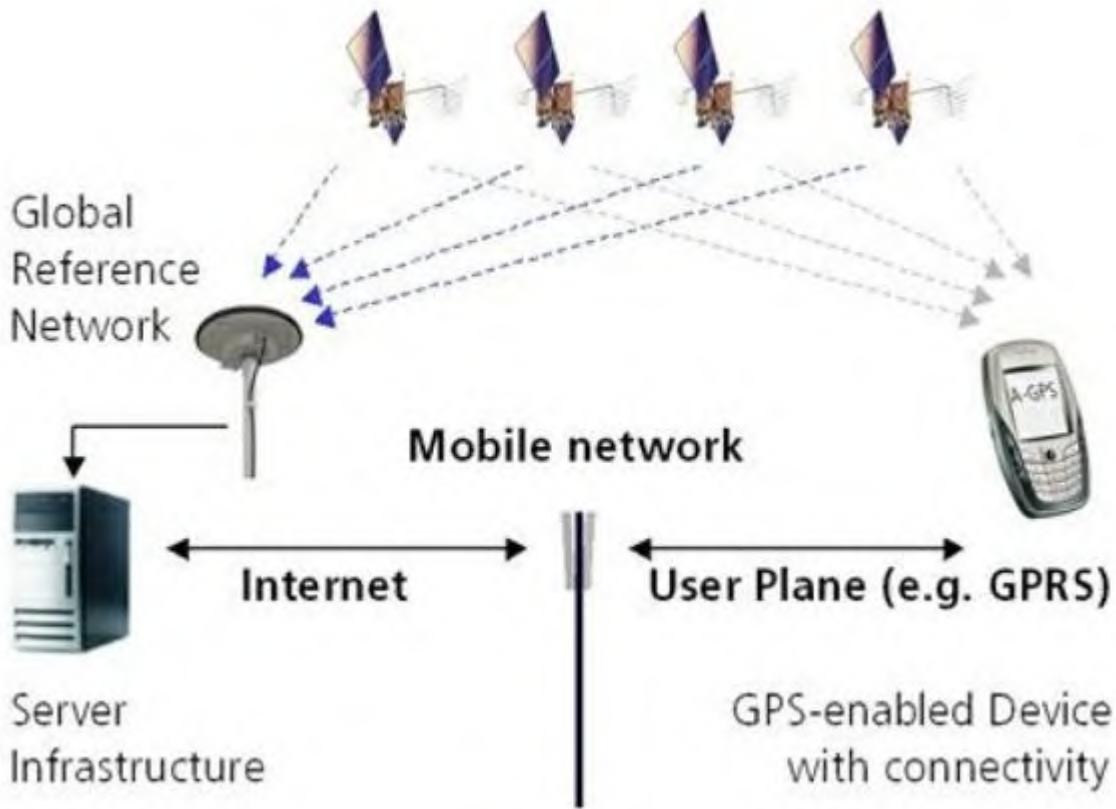


图9-25 AGPS原理

如图9-25所示，AGPS是一个比较复杂的系统，它首先需要在全球建造一个用于收集GPS卫星数据的参考网络（Global Reference Network）。目前比较知名的参考网络由IGS（International GNSS Service）组织建造，它在全球80多个国家设有300多个GPS卫星跟踪站。IGS的官方网站（<http://igscb.jpl.nasa.gov/>）提供GLONASS和GPS的卫星数据下载服务。

AGPS将收集到的这些数据（一般称为辅助数据，Aiding Data）保存在服务器上并对外界提供下载。例如图9-25中，支持AGPS的手机就可以通过移动通信网络发起相关请求以获得这些辅助数据从而加快自己的定位速度。

以上是对AGPS原理非常简略的介绍。在AGPS的实现细节中，还有如下几个比较重要的知识点。首先是User Plane和Control Plane的概念，这两个词源自移动通信领域。

- Control Plane用于在手机和基站间传输控制信息，例如手机收到一个来电信号就是通过Control Plane传输的。
- User Plane主要用来传送数据包（例如TCP/IP、HTTP等）。我们用的3G或GPRS上网就是User Plane。显然，手机通过User Plane来获取Aiding Data这种方式更具通用性和扩展性，例如它能通过TCP或UDP等协议来传输数据。关于CP和UP的区别，见参考资料[26]。

手机如何利用AGPS来进行定位呢？AGPS可分为MSB（Mobile Station Based）和MSA（Mobile Station Assisted）两种运行模式。

- MSB模式下，手机从AGPS位置服务器上下载辅助数据，然后手机再结合GPS卫星信号进行定位计算。这种方式需要手机下载辅助数据，同时它还需要利用其自身的CPU、内存等资源进行最终的定位计算。MSB模式的优点是网络负担小且定位延时小，适合短时间内的连续定位。
- MSA模式下，手机接收并解调GPS卫星信号，然后将这些伪距信息传给AGPS位置服务器。AGPS位置服务器根据手机所发送的数据、自己所保存的卫星数据以及其他一些信息（例如手机当时通信的基站位置）计算出手机所在位置，然后将该信息返回给手机。MSA模式下，手机无须使用自己的CPU等资源来进行定位计算。MSA的优点是对终端的性能要求低，但其定位的延时大，不适合高速行驶情况下的定位。

**提示** 如果接收器只使用GPS卫星信号进行定位，这种工作模式称为Standalone（也叫Autonomous）模式。MSA或MSB都需要手机接收GPS卫星信号。对于无法接收GPS卫星信号的地区（如办公室等）该怎么办呢？这就需要借助其他方法了，例如Cell-ID定位方法，其原理很简单，就是通过获取目标手机所在的蜂窝小区ID来确定其所在的位置。

AGPS涉及的内容非常多，为了更规范地为用户提供AGPS服务，OMA（Open Mobile Alliance，开放移动联盟）制定了一整套服务和标准，这套服务和标准统称为OMA-SUPL（Secure User Plane Location）。了解OMA-SUPL的工作流程非常有助于理解本章下文对Android平台中AGPS相关的代码分析。

① 根据参考资料[10]，这32颗卫星中的31颗处于运行状态，另外一颗不可用。美国一共发射了64颗卫星，未来还有卫星更新计划。美国GPS官方对卫星的说明见参考资料[9]。

② GPS双频接收机可以同时接收两个不同频率的载波信号。它将利用不同频率的电磁波在大气电离层传输时所造成的延迟时间不一致的原理来减少电离层带来的定位误差。

③ PRN的特点是看起来像随机的噪声，但又不是真正的噪声，它是一种比较复杂的数字编码。

### 9.2.3 OMA-SUPL协议[27][28]

OMA-SUPL包含一套非常复杂的协议，它综合了移动通信领域现有的一些标准和协议（如3GPP相关协议、WAP等），其目的是充分利用移动网络的相关特性以为用户提供更好的位置服务。OMA-SUPL目前最新版本是3.0，表9-7列举了OMA-SUPL各版本的特点。

表 9-7 OMA-SUPL 各版本特点

版本号	承载网络支持	运行模式	定位协议	定位方法
SUPL-1.0	GSM/GPRS/EDGE WCDMA/TD-SCDMA CDMA/CDMA2000	GSM/WCDMA/TD-SCDMA/ 以及 CDMA/CDMA-2000 中 的代理模式 <sup>②</sup> CDMA/CDMA-2000 的非代 理模式	RRLP/ RRC/ TIA-801	A-GPS (Based 或 Assisted) / Autonomous GPS/Enhanced Cell/Sector ID 等
SUPL-2.0	新增 LTE/HSPA+WLAN/ WiMAX/I-WiMAX		新增 LPP	新增 A-GNSS (Based 或 Assisted)/Autonomous GNSS/ OTDOA over LTE
SUPL-3.0	新增 固定宽带 (如 DSL) /WLAN	只支持代理模式	只支持 LPP、 TIA-801 和 LPPe， 而 RRLP 和 RRC 不再使用	新增 SET Based Enhanced Cell/Sector ID/High Accuracy A-GNSS/Sensor 等

(一) 关于代理模式和非代理模式的区别，请参考9.2.3节SUPL架构。

表9-7中的缩写词含义如下。

- RRLP (Radio Resource LCS Protocol) 是一种协议，LCS是 Location Services的缩写。
- RRC (Radio Resource Control) 是一种协议。
- LPP (LTE Positioning Protocol) 是基于LTE的定位协议。
- LPPe (OMA LPP Extensions) 是LPP扩展协议。
- TIA (Telecommunications Industry Association) 是美国电信工业协会。
- OTDOA (Observed Time Difference of Arrival) 是一种移动定位技术。

提示 OMA-SUPL涉及很多来自移动通信领域的概念和词汇。由于篇幅问题，笔者不打算对它们进行深入介绍，感兴趣的读者可自行研究。

## 1. SUPL架构

OMA-SUPL是一个比较复杂的系统，图9-26所示为它的架构。

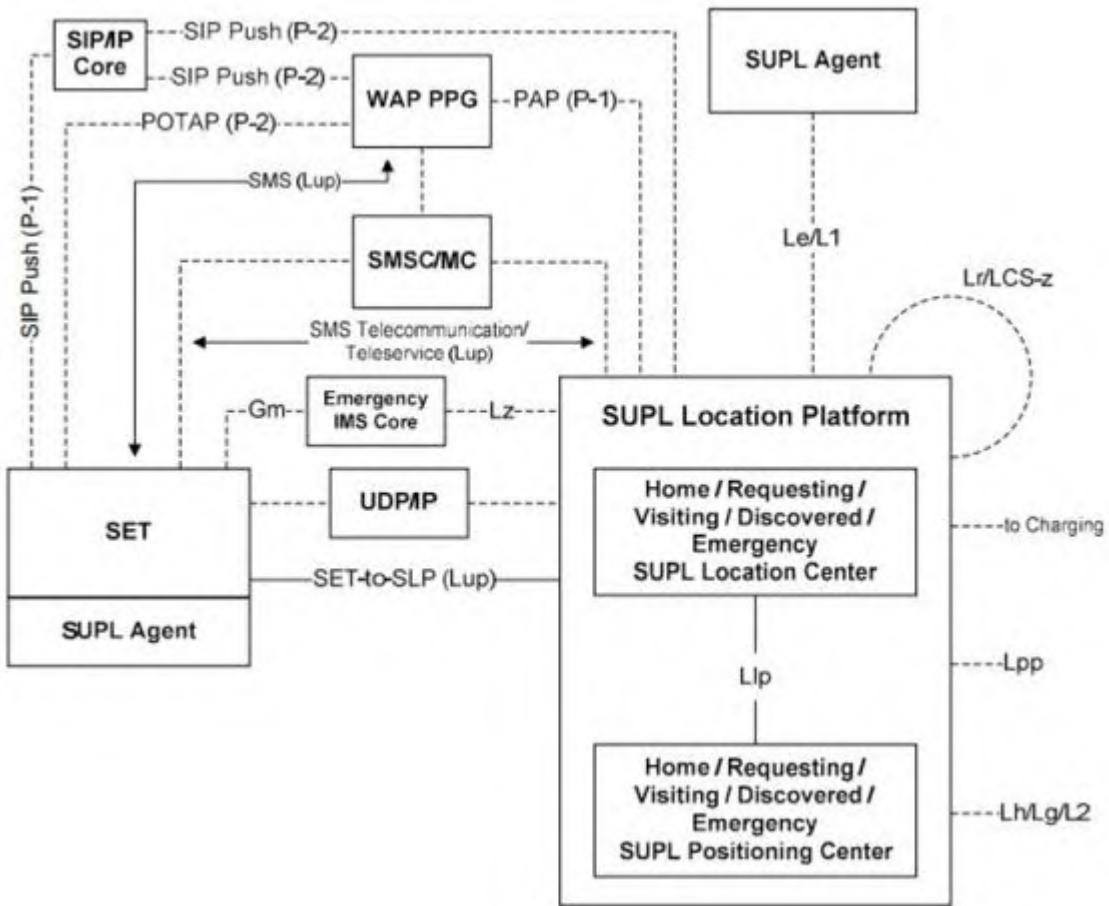


图9-26 SUPL架构

图9-26中主要包含三个部分。

- 左下方的SET代表AGPS服务的客户端，例如我们的Android智能手机。在规范中，SET全称是SUPL Enabled Terminal（终端）。
- 右下方的SLP<sub>①</sub>（SUPL Location Platform）包含两个重要组成部分，一个是SLC（SUPL Location Center），其作用是和SET交互，例如处理来自SET的请求；另外一个是SPC（SUPL Positioning

Center），其作用是进行定位计算。如果SET直接和SPC交互，则称为非代理工作模式。相反，如果SET借助SLC与SPC交互的话，则称为代理模式<sup>[29]</sup>。SUPL 3.0版协议只支持代理模式，故SET将只能和SLC交互。

- 右上方的SUPL Agent ②，是一个需要获取位置信息的应用程序。SUPL Agent可以运行在SET中（如图中左下方的SET和SUPL Agent），也可以运行在SUPL Network ③ 中，如图中右上方单独绘制的SUPL Agent。不论哪种情况，定位请求只能由SUPL Agent发起。如果SUPL Agent在SET中，这种请求方式叫SET Initiated请求（终端始发定位请求）[29]。如果SUPL Agent位于SUPL Network中，则这种请求方式叫Network Initiated请求（网络始发定位请求）。

除了上述三个部分外，图9-26中的连线用于表示它们之间交互所使用的协议等信息。

对于网络始发定位请求而言，SLP需要通知目标SET参与定位工作（而在终端始发定位请求中，请求的发起者与SET在一个设备上），这个流程也叫SUPL INIT。SUPL INIT支持的协议很多，例如通过SIP、WAP、SMS等，或者直接利用UDP、TCP等。在使用SIP、WAP或SMS等协议时还需要借助移动通信领域中现有的组件（如SMS需要先通过短信中心SMS Center来处理），所以图中也绘制了这些必要的组件以及这些组件和SLP交互的协议，如SMSC、SIP/IP Core、WAP PRG (Wireless Application Protocol Push Proxy Gateway)、PAP (Push Access Protocol)、POTAP (Push Over The Air Protocol) 等。SET和SLP交互的流程由ULP (User Location Protocol，下节将详细介绍它) 描述。

在SLP中，SLC和SPC交互的协议叫ILP (Internal Location Protocol)。

OMA-SUPL还为SET、SLC和SPC定义了一组Function来描述它们应该具有的功能，表9-8列举了这些Function的名称和功能。

表 9-8 OMA-SUPL 功能定义

Function 名	SLC	SPC	SET	功能说明
SPF (SUPL Privacy Function)	可选		可选	用来确保定位用户的隐私得到保护。位置信息就属于个人隐私
SIF (SUPL Initiation Function)	可选		可选	Network Initiated 请求方式有一个问题需要解决，即网络端的 SUPL Agent 发起定位请求时，它需要一种方式通知目标 SET 参与这个处理流程。SIF 的功能就是以某种方式通知 SET 参与定位请求处理。目前支持的方式有 OMA Push、SMS、UDP 或 SIP Push。以 SMS 为例，系统将发送一条特殊的数据短信给目标 SET，从而目标 SET 就知道自己该参与某个定位请求处理了
SSF (SUPL Security Function)	可选		可选	处理认证、授权、数据安全等方面的工作
SRSF (SUPL Roaming Support Functions)	可选			处理漫游等方面的工作
SCF (SUPL Charging Function)	可选			处理计费等方面的工作
SSMF (SUPL Service Management Function)	可选			主要用来管理 SET 的位置，例如存储、修改、获取 SET 的位置等
SSPF (SUPL SET Provisioning Function)			可选	用来配置 SET
STF (SUPL Triggering Function)	可选		可选	<ul style="list-style-type: none"> <li>SET 可设置一个间隔时间（例如 5 秒），从而实现每隔 5 秒就从 SLP 那获取位置信息</li> <li>SET 可以条件触发的方式从 SLP 那获取位置信息。这些条件包括 SET 进入或离开某个区域等</li> </ul>
SSDF (SUPL SLP Discovery Function)	可选		可选	使 SET 能发现周围的其他 SLP，例如 Emergency-SLP。注意，在同一个区域，不同运营商可能都部署有 SLP
SADF (SUPL Assistance Delivery Function)		可选		选择、生成或分发定位辅助数据
SRRF (SUPL Reference Retrieval Function)		可选	可选	从参考网络那获取定位辅助数据
SPCF (SUPL Positioning Calculation Function)		可选	可选	根据相关数据计算最终的位置

SUPL 的内容非常多，不过和本章后文代码分析相关的内容大多集中在 ULP 协议即工作流程上。下面来看看 ULP。

## 2. ULP 介绍

ULP 主要描述 SET 和 SLP 之间该如何交互以完成定位请求。根据上一节对 SUPL Agent 的介绍，ULP 的使用分为两大类 [④](#)。

- SUPL Agent 位于 SET 中，由于定位请求只能由 SUPL Agent 发起，所以在 ULP 中，它被称为 SET Initiated 定位请求。其典型的使用案例就是在 Android 手机中打开导航类应用，这将触发手机发起一次定位请求。

- SUPL Agent位于SUPL Network中，这种情况称为Network Initiated 定位请求。例如，某些网络服务需要跟踪SET的位置，就会使用这种方式。不过，笔者目前没有找到与之相关的典型使用案例，有知晓的读者不妨与大家分享相关知识。

这两大类ULP应用场景对应的工作流程各不相同，我们先来看最常见的SET Initiated请求的工作流程。

### (1) SET Initiated ULP工作流程

图9-27描述了SET Initiated ULP的工作流程。

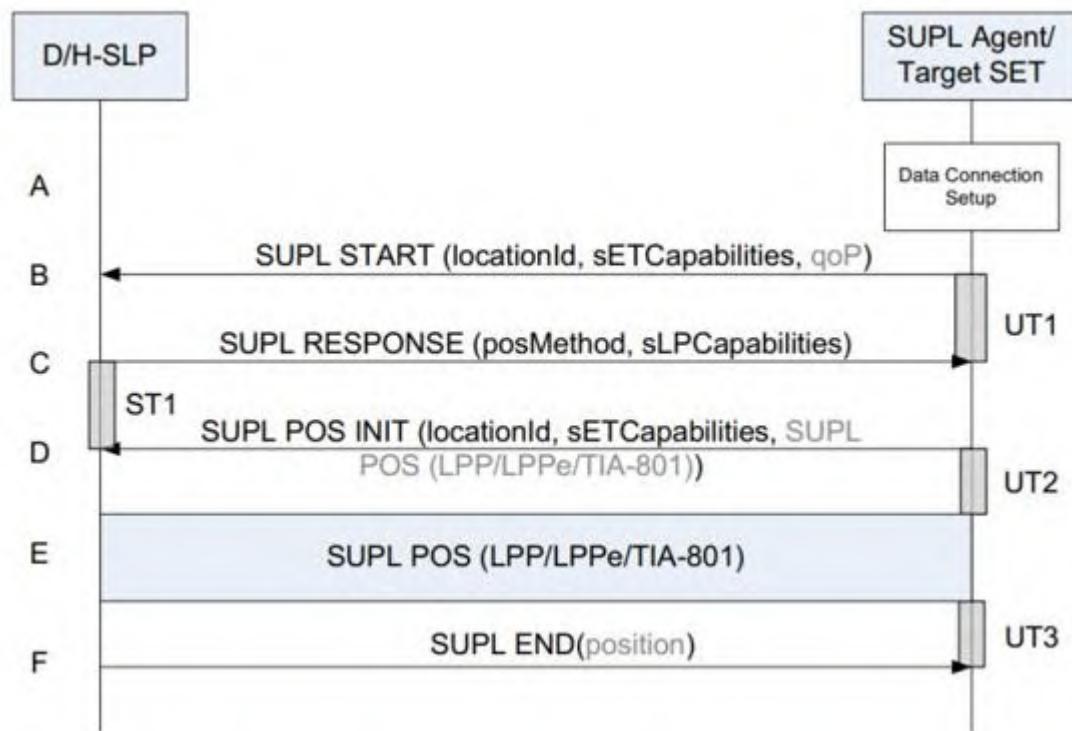


图9-27 SET Initiated ULP流程

1) SET首先和SLP建立数据链接。为了保证数据的安全性，这个链接需要基于TLS（Transport Layer Security，传输层安全）。图中的D/H-SLP为Discovered/Home-SLP的缩写，H-SLP即SET所在运营商所建立的SLP，而D-SLP为SET搜索到的SLP。

2) SET发送SUPL START命令给SLP，该命令携带了一些参数，包括locationId（如果使用移动通信网络，则该参数包括基站的Cell

Info。如果使用Wi-Fi，则该参数包括AP的信息）、  
SETCapabllities（SET的能力，如支持的定位数据封装协议、支持的  
定位方法等，详情可参考表9-7）。

3) SLP回复SUPL RESPONSE命令给SET。RESPONSE命令包含了SLP支持的  
定位方法（由posMethod表示），以及SLP支持的定位能力（由  
sLPCapabilities描述）。

4) SET发送SUPL POS INIT命令给SLP，该命令包含了SET的初始位置等  
信息。

5) 接着，SET和SLP通过一个或多个SUPL POS消息来计算位置。根据  
AGPS使用的模式（MSB或MSA），位置的计算方法也不尽相同。

6) 当位置计算完毕后，SLP发送SUPL END命令给SET，二者随后断开  
TLS链接。

**提示** 关于ULP各消息所包含的参数信息，请读者自行阅读参考资料  
[28]。

## (2) Network Initiated ULP工作流程

图9-28所示为Network Initiated (NI) ULP的工作流程图。

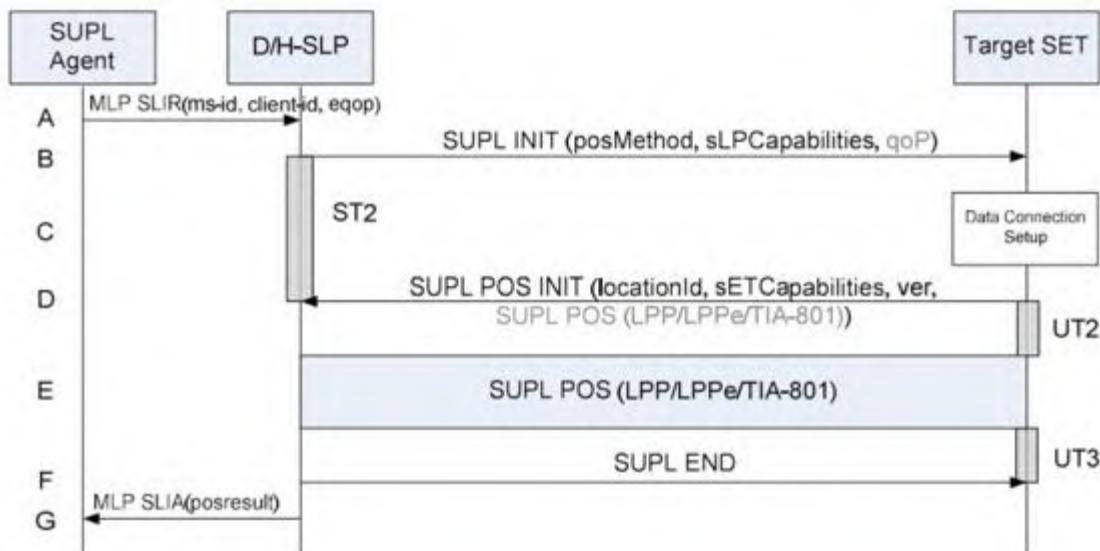


图9-28 Network Initiated ULP流程

图9-28中，由于SUPL Agent位于SUPL Network，所以它和SLP的交互遵守MLP（Mobile Location Protocol）。SLP收到SUPL Agent的SLIR（Standard Location Immediate Request）请求后，它将发送SUPL INIT命令给SET。此处需强调，如果SET和SLP此时还没有建立数据链接，SUPL INIT将通过OMA Push消息或数据短信等方式发送给SET，SET收到SUPL INIT命令后将和SLP建立数据链接。

此后，SLP和SET之间的交互与图9-27类似。SLP最终通过SLIA（Standard Location Immediate Answer）将定位信息发送给SUPL Agent。

**提示** 此处不再详述ULP的细节，请读者自行阅读参考资料[28]。

至此，本章所涉及的GPS相关基础知识就全部介绍完毕。相信读者能感觉到这些内容背后的专业知识是多么庞大和复杂。在此，希望立志成为GPS专家的读者继续保持谦虚的态度，认真学习，争取为中国的北斗导航系统添砖加瓦。

**提示** 本节所述内容能覆盖Android GPS相关模块（不含驱动及芯片底层模块）代码中80%左右的背景知识。

**①** OMA-SUPL中有大量的缩写词汇，请读者阅读时务必注意它们的全称。

**②** 规范中的定义是"A Software and/or hardware entity accessing the SUPL enabler in order to obtain location information"。

**③** 读者可将其理解为一个帮助定位的系统，规范中的定义是"Access network which facilitates the location determination functionality and provides the SUPL bearer"。

**④** 此处讨论仅针对OMA SUPL ULP规范中的Immediate Service。

## 9.3 Android中的位置管理

### 9.3.1 LocationManager架构

GPS的根本目的是为使用者提供位置相关的信息。Android系统设计了一个以LocationManagerService为核心的位置管理架构提供相关的位置服务。

图9-29所示的Android平台LocationManager架构按顺时针可分为四部分。

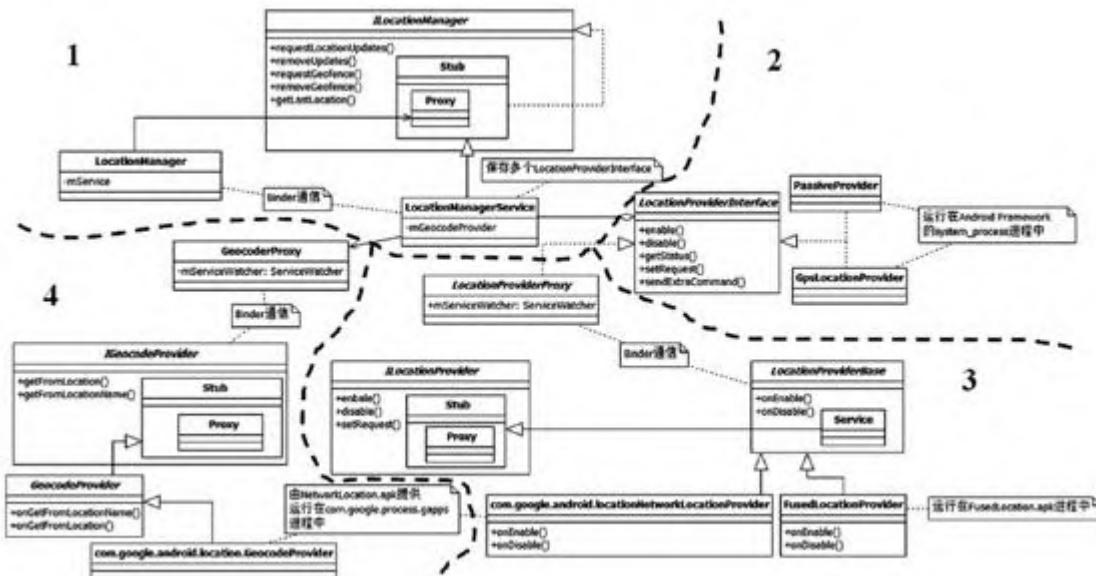


图9-29 Android平台中LocationManager架构

第一部分为LocationManagerService（简称LMS）和其客户端LocationManager（简称LM）。LMS和Android Java Framework中其他Service一样由SystemServer创建并运行在system\_process进程中<sup>①</sup>。LMS内部将统一管理Android平台中能提供位置服务的相关模块，而LM为那些需要使用位置服务的应用程序服务。LM和LMS之间通过Binder进行交互。下节所示的示例应用程序将介绍LM的用法。

Android平台中能提供位置服务的相关模块统称为Location Provider（位置提供者，LP）。位置提供者必须实现 LocationProviderInterface 接口。这些接口对应的对象实例由LMS来创建和管理。在所有这些位置提供者中，Android Framework实现了其中的PassiveProvider和GpsLocationProvider。这两个LP由LMS创建并运行在system\_process进程中。下文介绍LMS时还会详细介绍 PassiveProvider和GpsLP。

除了使用GPS定位外，系统还支持网络定位（Network Location）方法来获取位置信息。这种方法大致的工作原理是，某地区的移动通信基站（Cell Tower）或无线网络AP的位置信息都已事先获取并保存在相关服务提供商的服务器上。当手机使用网络定位时，它首先向服务器查询自己所连接或搜索到的基站位置或AP的位置，然后根据信号的强度推算自己的大致位置。相比GPS定位而言，网络定位速度快，耗电少，适用于室内和室外，但精度较GPS差。Android原生代码并不提供 Network Location Provider相关的功能，它一般由第三方应用厂商提供，例如Google的GMS（Google Mobile Service）包中有一个 NetworkLocation.apk就提供了该功能，而国内上市的手机则使用百度公司提供的NetworkLocation\_Baidu.apk。由于它们运行在应用程序所在的进程中，所以系统定义了ILocationProviderProxy接口使LMS能管理这些由应用程序提供的位置服务。这些应用的位置服务需要实现 LocationProviderBase抽象类。相关类结构如图中区域3所示。

区域3中的FusedLocationProvider是一个比较有意思的LP。它本身不能提供位置信息，其内部将综合GpsLP和NetworkLP的位置信息，然后向使用者提供最符合使用者需求的数据。即它能根据使用者对电源消耗、精度两方面的要求以选择GpsLP或/和NetworkLP作为真实的LP。同时，FusedLP能选择GpsLP或NetworkLP提供的位置信息中最好的那一个返回给使用者。简单点说，FusedLP出现之前，一个比较完善的LP客户端需要同时操作和管理GpsLP和NetworkLP，而有了FusedLP后，客户端只需要使用它即可，其余事情由FusedLP内部来管理。注意，FusedLP也由应用程序提供，它运行在FusedLocationProvider.apk所在的进程中。

除了提供位置信息外，系统（借助第三方应用提供）还支持位置信息和地址信息相互转换，即得到某个地址（如国家、市区、街道名等）的位置信息（如经纬度信息），或者根据位置信息得到其对应的地址

信息。由于地址和位置信息的映射关系一般也由第三方应用提供，所以LMS利用GeocodeProxy和第三方应用中实现IGeocodeProvider的对象交互。相关类结构如图中区域4所示。

**提示** FusedLocationProvider的代码非常简单，感兴趣的读者可自行研究。可参考SDK中关于这方面的讨论，其位置为  
<https://developer.android.com/guide/topics/location/strategies.html>。

了解Android平台中LM的架构后，笔者从以下两个方面详细介绍Android中的位置管理模块。

- 通过一个示例展示如何利用LocationManager功能来获取自己的位置信息以及地址信息。
- 介绍LMS相关的模块及工作原理。这些模块包括LocationManagerService、GpsLocationProvider、GPS HAL层相关控制接口等。

**①** 关于system\_process，读者可阅读《深入理解Android：卷II》第3章。

### 9.3.2 LocationManager应用示例

本节所使用的示例运行后的界面如图9-30所示。

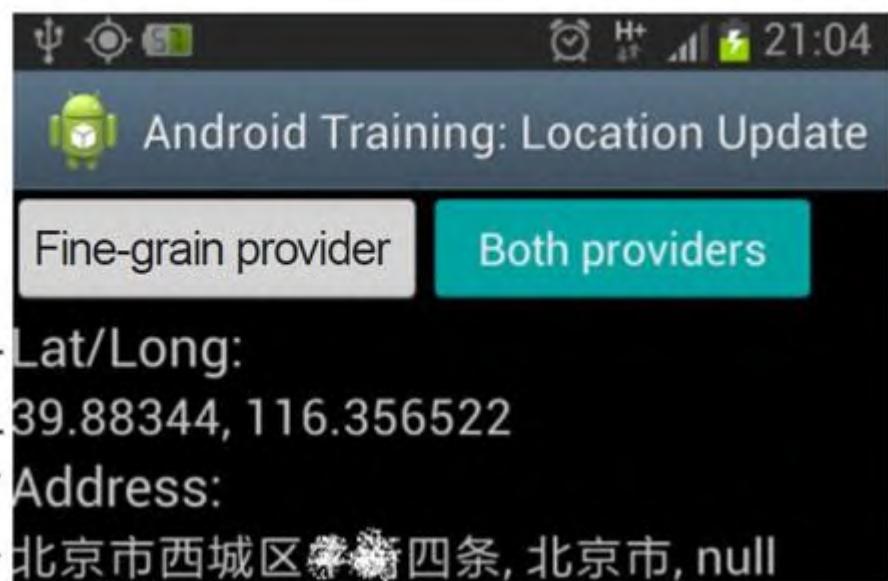


图9-30 示例运行效果

- 左边按钮为“Fine-grain provider”，表示通过GpsLP来获取位置信息。
- 右边按钮为“Both Providers”，表示同时使用GpsLP和NetworkLP来获取位置信息。
- 按钮下方的“Lat/Long”表示当前设备的位置信息（经纬度值），而“Address”表示根据该位置信息得到的地址信息。

示例非常简单，所有内容都集中在LocationActivity.java文件中。先来看onCreate函数，代码如下所示。

[-->LocationActivity.java: : onCreate]

```
public void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.main);
```

```

    .....// UI 等初始化
    // Geocoder: 只有Android 2.3版本以后系统才支持该功能
    // 同时还需要判断是否存在GeocoderProvider
    mGeocoderAvailable =
        Build.VERSION.SDK_INT >=
    Build.VERSION_CODES.GINGERBREAD &&
        Geocoder.isPresent();

    mHandler = new Handler() {.....// Handler的作用是更新图9-30中
的位置和地址信息};

    // 客户端必须要获取LocationManager来和LMS交互
    mLocationManager = (LocationManager)
getSystemService(Context.LOCATION_SERVICE);
}

```

接着，LocationActivity在其onResume中将调用setup函数完成进一步的初始化工作，其代码如下所示。

[-->LocationActivity.java: : setup]

```

private void setup() {
    Location gpsLocation = null;    Location networkLocation =
null;
    mLocationManager.removeUpdates(listener);
    mL.LatLng.setText(R.string.unknown);
    mAddress.setText(R.string.unknown);
    if (mUseFine) { // mUseFine对应为图9-30中的“Fine-grain
Provider”按钮
        .....
        // requestUpdatesFromProvider为关键函数，下文将详细分析
        gpsLocation = requestUpdatesFromProvider(
            LocationManager.GPS_PROVIDER, // 该参数为字符
串，值为“gps”
            R.string.not_support_gps);
        if (gpsLocation != null) updateUILocation(gpsLocation);
    } else if (mUseBoth) { // mUseBoth对应为图9-30中的“Both
Providers”按钮
        gpsLocation = requestUpdatesFromProvider(
            LocationManager.GPS_PROVIDER,
            R.string.not_support_gps);
        networkLocation = requestUpdatesFromProvider(
            LocationManager.NETWORK_PROVIDER, // 该参数
值为“network”
            R.string.not_support_network);
    }
}

```

```
    }
}
```

看setup中的关键函数requestUpdatesFromProvider代码如下所示。

[-->LocationActivity.java: : requestUpdatesFromProvider]

```
private Location requestUpdatesFromProvider(final String provider,
                                              final int errorResId) {
    Location location = null;
    // 判断由provider指定的LP是否启用。
    if (mLocationManager.isProviderEnabled(provider)) {
        /*
         * 调用LocationManager的requestLocationUpdates函数，该函数用于
         * 注册一个回调函数以
         * 接收位置变化信息，其各个参数的作用如下。
         * provider: 用于指明使用哪个LP，目前可取参数有“gps”（对应为
         * GpsLP）、“network”（对应
         * 为NetworkLP）、“passive”（对应为PassiveProvider）。
         * TEN_SECONDS: 用于指明多少毫秒更新一次位置数据，本例中使用的值为
         * 10秒。
         * TEN_METERS: 用于指明位置变化多少时更新一次数据，本例中使用的值为
         * 10米。
         * listener: 类型为LocationListener，当位置发生变化时，其
         * onLocationChanged函数将被调用。
         */
        mLocationManager.requestLocationUpdates(provider,
                                                TEN_SECONDS,
                                                TEN_METERS, listener);
        // getLastKnownLocation用于获取LP上一次保存的位置信息数据
        // Android平台中，位置信息用Location类表示
        location =
mLocationManager.getLastKnownLocation(provider);
    }
    return location;
}
```

requestLocationUpdates是LM中非常重要的函数，请读者务必把握其用法。

当位置信息发生变化后，LocationListener的onChange函数将被调用。本例中使用的LocationListener相关代码如下所示。

[-->LocationActivity.java: : LocationListener]

```
private final LocationListener listener = new
LocationListener() {
    public void onLocationChanged(Location location) {
        updateUIlocation(location); // 下文将分析它
    }
    // 当用户在设置中启用或禁止相关LocationProvider后，下面这两个函数将
    // 被调用
    public void onProviderEnabled(String provider) { }
    public void onProviderDisabled(String provider) { }
    // 当LocationProvider的状态发生变化时，下面这个函数将被调用
    public void onStatusChanged(String provider, int status,
Bundle extras) {}
};
```

updateUIlocation函数代码如下所示。

[-->LocationActivity.java: : updateUIlocation]

```
private void updateUIlocation(Location location) {
    // updateUIlocation的参数location代表对应LP得到的位置信息
    // 示例程序将根据该信息来更新图9-30中“Lat/Long”的值
    Message.obtain(mHandler, UPDATE_LATLNG,
        location.getLatitude() + ", " +
    location.getLongitude()).sendToTarget();

    // doReverseGeocoding根据位置信息来获取地址信息
    if (mGeocoderAvailable) doReverseGeocoding(location);
}
```

doReverseGeocoding用于根据位置信息来获取地址信息，由于该工作往往需要通过网络来查询，所以doReverseGeocoding内部将创建一个AsyncTask用来完成此工作。我们直接来看AsyncTask的代码。

[-->LocationActivity.java: : ReverseGeocodingTask]

```
private class ReverseGeocodingTask extends AsyncTask<Location,
Void, Void> {
    Context mContext;
```

```
.....
protected Void doInBackground(Location... params) {
    // 创建一个Geocoder对象，它可用于处理地址信息和位置信息的转换
    Geocoder geocoder = new Geocoder(mContext,
    Locale.getDefault());
    Location loc = params[0];
    List<Address> addresses = null;
    try {
        // getFromLocation用于根据位置信息来查询对应的地址信息
        addresses =
geocoder.getFromLocation(loc.getLatitude(), loc.getLongitude(),
1);
    } .....
    if (addresses != null && addresses.size() > 0) {
        Address address = addresses.get(0);
        String addressText = String.format("%s, %s, %s",
                address.getMaxAddressLineIndex() > 0 ?
address.getAddressLine(0) : "",
address.getLocality(), address.getCountryName());
        // 更新图9-30中的“Address”信息
        Message.obtain(mHandler, UPDATE_ADDRESS,
addressText).sendToTarget();
    }
    return null;
}
}
```

通过上述示例可以发现，Android平台中使用LM非常简单，其主要工作如下。

- 1) 先创建一个LocationManager对象，用于和LMS交互。
- 2) 然后调用requestLocationUpdates以设置一个回调接口对象LocationListener，同时还需要指明使用哪个LP<sub>①</sub>。
- 3) 当LP更新相关信息后，LocationListener对应的函数将被调用，应用程序在这些回调函数中做相关处理即可。
- 4) 如果应用程序需要在位置和地址信息做转换，则使用Geocoder类提供的函数即可。

虽然LM比较简单，但它提供的都是一些很基本的功能，如果想实现一些诸如显示地图信息这样的功能，LM就无能为力了。为了实现一些更复杂的位置相关的功能，Google提供了更高级的API来帮助开发者。关于这一部分内容，建议读者阅读参考资料[30]。

① 根据审稿专家的意见，有些应用会只传进定位的条件（精度），而不去指定使用哪个LP。在不带GPS功能的平台，若应用只指定从GPS中获取定位数据，则它将得不到位置信息。

### 9.3.3 LocationManager系统模块

根据前文对Android平台中位置管理相关模块的介绍可知，LocationManagerService（LMS）是系统模块的核心，我们先来看它的初始化。

#### 1. LMS初始化

系统启动时，LMS将由SystemServer创建，其初始化函数为systemReady，代码如下所示。

```
[-->LocationManagerService.java: : systemReady]

public void systemReady() {
    // 创建一个工作线程，其线程函数为LocationManagerService中的run函数
    Thread thread = new Thread(null, this, THREAD_NAME);
    thread.start();
}
// 直接来看LMS的run函数
public void run() {
    Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
    Looper.prepare();
    // 创建一个LocationWorkerHandler，其主要工作为将LP投递的位置信息返回给客户端
    mLocationHandler = new LocationWorkerHandler();
    init(); // 重要的初始化函数
    Looper.loop();
}
```

LMS的主要初始化工作都集中在init中，该函数的代码如下所示。

```
[-->LocationManagerService.java: : init]

private void init() {
    .....// 创建Wakelock等工作。关于Wakelock的工作原理，可阅读《深入理解Android：卷II》第5章
    // 系统有一个黑白名单用于禁止使用某些特定的LP。在黑白名单中，LP由其对应的Java包名指定
    mBlacklist = new LocationBlacklist(mContext,
        mLocationHandler);
```

```

mBlacklist.init();
/*
LocationFudger很有意思，Android平台提供粗细两种精度的位置信息。其中，粗精度的位置信息由下面这个
LocationFudger根据细精度的位置信息进行一定数学模糊处理后得到。粗精度默认的最大值为2000米，最小值为200米。厂商可在Settings.db数据库secure表的“locationCoarseAccuracy”设定。
本章不讨论LocationFudger的内容，请感兴趣的读者自行研究。
*/
mLocationFudger = new LocationFudger(mContext,
mLocationHandler);
synchronized (mLock) {
    loadProvidersLocked(); // 关键函数：创建或加载系统中的LP，下节将详细分析它
}
/*
GeofenceManager为地理围栏管理对象。Android平台中其作用如下。
客户端可以设置一个地理围栏，该地理围栏的范围是一个以客户端设置的某个点（指定其经/纬度）为中心
的圆，圆的半径也由客户端设置。当设备进入或离开该地理围栏所覆盖的圆时，LMS将通过相关回调函数通知客户端。目前Android 4.2中还没有在SDK中公开地理围栏相关的API。有需要使用该功能
的程序可通过Java放射机制调用LocationManager的addGeofence函数。
GeofenceManager比较简单，感兴趣的读者也可自行阅读。
*/
mGeofenceManager = new GeofenceManager(mContext,
mBlacklist);
.....// 监听APK安装\卸载等广播事件以及监听用户切换事件
/*
根据Settings中的设置情况来开启或禁止某个LP。在此函数中，各LP实现的
LocationProviderInterface接口的enable/disable函数将被调用。
*/
updateProvidersLocked();
}

```

init函数的内容比较多，本节重点关注其中的loadProvidersLocked函数。

**提示** 读者学完本章后不妨研究LocationFudger和GeofenceManager，其内容比较有意思。

### (1) loadProvidersLocked流程

loadProvidersLocked用于创建及加载系统中所有的LocationProvider，其代码如下所示。

[-->LocationManagerService.java: : loadProvidersLocked]

```
private void loadProvidersLocked() {  
    /*  
     * 先创建PassiveProvider。该LP名称中的Passive(译为“被动”)所对应的场景  
     * 比较有意思，此  
     */
```

处举一个例子。假设应用程序A使用GpsLP。GpsLP检测到位置更新后将通知应用程序A。应用程序B

如果使用PassiveProvider，当GpsLP更新位置后，它也会触发PassiveProvider以通知应用程

序B。也就是说，PassiveProvider自己并不能更新位置信息，而是靠其他LP来触发位置更新的。

特别注意，PassiveProvider的位置更新是由LMS接收到其他LP的位置更新通知后主动调用

PassiveProvider的updateLocation函数来完成的。

目前，PassiveProvider被SystemServer中的TwilightService使用，TwilightService

将根据位置信息来计算当前时间是白天还是夜晚（Twilight有“黎明”之意）

[①](#)

。

另外，GpsLP也会使用PassiveProvider来接收NetworkLP的位置信息。

PassiveProvider非常简单，请读者在学完本章基础后再自行研究它。

```
*/  
PassiveProvider passiveProvider = new PassiveProvider(this);  
// LMS将保存所有的LP  
addProviderLocked(passiveProvider);  
// PassiveProvider永远处于启用状态。mEnabledProviders用于保存那些  
被启用的LP  
mEnabledProviders.add(passiveProvider.getName());  
mPassiveProvider = passiveProvider;  
  
if (GpsLocationProvider.isSupported()) {  
    // 创建GpsLP实例，我们将用两节来介绍
```

```
    GpsLocationProvider gpsProvider = new
GpsLocationProvider(mContext, this);
    mGpsStatusProvider =
gpsProvider.getGpsStatusProvider();
    mNetInitiatedListener =
gpsProvider.getNetInitiatedListener();
        addProviderLocked(gpsProvider); // 保存GpsLP
        // GpsLP属于真实的位置提供者，所以把它单独保存在mRealProvider
中
        mRealProviders.put(LocationManager.GPS_PROVIDER,
gpsProvider);
    }

Resources resources = mContext.getResources();
ArrayList<String> providerPackageNames = new
ArrayList<String>();
/*
    config_locationProviderPackageNames存储了第三方LP的Java包名。
Android原生代码中该值
    只有一个，为“com.android.location.fused”。FusedLP对应的源码路径为
frameworks/base/
    packages/FusedLocation。
*/
    String[] pkgs = resources.getStringArray(
com.android.internal.R.array.config_locationProviderPackageNames
);
    if (pkgs != null)
providerPackageNames.addAll(Arrays.asList(pkgs));
/*
```

加载应用程序实现的LP服务时，LMS将检查它们的签名信息以及版本信息。笔者研究了这部分代码，

感觉其目的可能是想提供一种数据一致性的保护机制，举个例子。

笔者的Galaxy Note 2中默认安装的是百度提供的

NetworkLocation\_baidu.apk，

笔者安装Google的NetworkLocation.apk时，由于二者签名不一致（根据LMS相关代码工的作原理可

知，百度的NetworkLP属于config\_locationProviderPackageNames指定的，其签名信息将被保存。

后续再安装的LP将检查其签名是否和之前保存的签名信息是否一致。也就是说，后续的LP必须使用

NetworkLocation\_Baidu或FusedLP相同的签名才能被LMS加载。而FusedLP属于Android原生

应用，一般由手机厂商提供，第三方应用程序不太可能拿到其签名），根据上述信息可知，只有百度旗

下或得到它授权的第三方LP才能在笔者的Galaxy Note 2中安装和使用。

签名检查这部分代码在下文介绍的LocationProviderProxy中也有，读者仅了解其目的即可。

ensureFallbackFusedProviderPresentLocked用来检查FusedLP的签名和版本信息。

\*/

```
ensureFallbackFusedProviderPresentLocked(providerPackageNames);
```

```
/*
加载NetworkLP，其中参数如下。
NETWORK_PROVIDER:类型为字符串，值为“network”。
NETWORK_LOCATION_SERVICE_ACTION:类型为字符串，值为
“com.android.location.service.v2.NetworkLocationProvider”，
它代表应用程序所实现的LP服务名，其作用见下节关于
LocationProviderProxy的介绍。
```

注意，和Android 4.1比起来，Android 4.2 LMS相关的代码变化较大，  
下文将介绍LocationProviderProxy的工作流程。

\*/

```
LocationProviderProxy networkProvider =
LocationProviderProxy.createAndBind(
    mContext, LocationManager.NETWORK_PROVIDER,
    NETWORK_LOCATION_SERVICE_ACTION,
    providerPackageNames, mLocationHandler,
mCurrentUserId);
if (networkProvider != null) {
    // NetworkLP也属于真实的LP
    mRealProviders.put(LocationManager.NETWORK_PROVIDER,
networkProvider);
    // 应用程序实现的LP保存在mProxyProviders变量中
    mProxyProviders.add(networkProvider);
    addProviderLocked(networkProvider);
}
.....
// 加载FusedLP
LocationProviderProxy fusedLocationProvider =
LocationProviderProxy.createAndBind(
    mContext, LocationManager.FUSED_PROVIDER,
    FUSED_LOCATION_SERVICE_ACTION,
    providerPackageNames, mLocationHandler,
mCurrentUserId);
if (fusedLocationProvider != null) {
    addProviderLocked(fusedLocationProvider);
    mProxyProviders.add(fusedLocationProvider);
    // FusedLP默认处于启用的状态
```

```

mEnabledProviders.add(fusedLocationProvider.getName());
        mRealProviders.put(LocationManager.FUSED_PROVIDER,
fusedLocationProvider);
    } .....
    // 创建Geocoder。GeocoderProxy的工作流程和LocationProviderProxy
类似
    mGeocodeProvider = GeocoderProxy.createAndBind(mContext,
providerPackageNames,
        mCurrentUserId);
    .....
}

```

在LMS的初始化函数中，`loadProvidersLocked`用于创建和加载系统中所有的LP如下。

- `PassiveProvider`: 提供被动式的位置数据更新服务，其位置数据来源于其他的LP。
- `GpsLocationProvider`: 由LMS创建并加载，运行在LMS所在的进程`system_process`中，属于系统提供的LP服务。
- `NetworkLocationProvider`: 该LP服务由应用程序提供。
- `FusedLocationProvider`: 由FusedLocation.apk服务，属于系统提供的应用程序。其内部将使用其他的LP。
- `GeocodeProvider`: 由第三方应用程序提供。一般和NetworkLP位于同一个应用程序中。

对于本章来说，GpsLP是重中之重。不过在介绍它之前，我们先来看看LMS是如何与位于应用进程中的其他LP交互的。

## (2) LocationProviderProxy介绍

由`loadProvidersLocked`的代码可知，LMS通过`LocationProviderProxy`（简称LPPProxy）来加载应用进程中的LP。相关函数是LPPProxy的`createAndBind`，代码如下所示。

[-->`LocationProviderProxy.java`: : `createAndBind`]

```

public static LocationProviderProxy createAndBind(Context
context, String name, String action,

```

```
    List<String> initialPackageNames, Handler handler, int
userId) {
    // 创建一个LPPProxy对象
    LocationProviderProxy proxy = new
    LocationProviderProxy(context, name, action,
        initialPackageNames, handler, userId);
    if (proxy.bind()) return proxy;
    else return null;
}
```

LPPProxy的createAndBind中有两个关键函数，分别是LPPProxy的构造函数以及bind。我们先来看其构造函数，代码如下所示。

[-->LocationProviderProxy.java: : LocationProviderProxy]

```
private LocationProviderProxy(Context context, String name,
String action,
        List<String> initialPackageNames, Handler handler,
int userId) {
    mContext = context;
    mName = name;
    /*
        ServiceWatcher是LPPProxy中最重要的对象。在Android LM架构中，应用程序实现的LP服务都
        通过Android四大组件中的Service提供。ServiceWatcher就是LPPProxy中用来连接和监视应用程序
        实现的LP服务的。
    */
    mServiceWatcher = new ServiceWatcher(mContext, TAG, action,
initialPackageNames,
        mNewServiceWork, handler, userId);
}
```

在createAndBind函数的最后，LPPProxy将调用bind函数，而这个bind将触发ServiceWatcher的start被调用，我们直接来看它。

[-->ServiceWatcher.java: : start]

```
public boolean start() {
    synchronized (mLock) {
        // bindBestPackageLocked见下文解释
        if (!bindBestPackageLocked(null)) return false;
    }
    // 监听应用程序安装、卸载、更新等广播事件
    mPackageMonitor.register(mContext, null, UserHandle.ALL,
```

```
true);
    return true;
}
```

bindBestPackageLocked的工作包括如下。

- 检查目标应用程序的签名。
- 根据createAndBind第三个参数查找该目标应用程序实现的Service（Android四大组件之一）。以NetworkLP为例，目标应用程序必须提供一个名为“com.android.location.service.v2.NetworkLocationProvider”的服务。
- 然后绑定到该服务上，并获取一个类型为ILocationProvider接口的实例。通过该实例，LPPProxy可与位于应用程序中的LP服务交互。

bindBestPackageLocked中最重要的函数是bindToPackageLocked，其代码如下所示。

```
[-->ServiceWatcher.java: : bindToPackageLocked]

private void bindToPackageLocked(String packageName, int
version) {
    unbindLocked();
    Intent intent = new Intent(mAction);
    intent.setPackage(packageName);
    mPackageName = packageName;
    mVersion = version;
    // 绑定到应用程序的LP Service
    mContext.bindService(intent, this, Context.BIND_AUTO_CREATE
|
        Context.BIND_NOT_FOREGROUND | Context.BIND_NOT_VISIBLE,
mCurrentUserId);
}
```

绑定成功后，ServiceWathcer的onServiceConnected函数将被调用。

```
[-->ServiceWatcher.java: : onServiceConnected]
```

```
public void onServiceConnected(ComponentName name, IBinder
binder) {
    synchronized (mLock) {
```

```

        String packageName = name.getPackageName();
        if (packageName.equals(mPackageName)) {
            mBinder = binder;
            if (mHandler != null && mNewServiceWork != null)
                mHandler.post(mNewServiceWork); //  

mNewServiceWork由LPPProxy提供
        } .....
    }
}

```

mNewServiceWork是一个Runnable对象，它由LPPProxy提供，其代码如下所示。

[-->LocationProviderProxy.java: : Runnable]

```

private Runnable mNewServiceWork = new Runnable() {
    public void run() {
        boolean enabled;
        ProviderProperties properties = null;
        ProviderRequest request;
        WorkSource source; ILocationProvider service;
        synchronized (mLock) {
            enabled = mEnabled; request = mRequest;
            source = mWorksource;
            // 返回ILocationProvider接口实例，它可和应用程序中LP交
互
            service = getService();
        }
        try {
            /*
             * 获取LP的属性信息，这些属性统一封装在类型为
ProviderProperties的对象中，LP的属性
             * 由ProviderProperties的成员变量表示，这些变量如下。
             * mRequiresNetwork: 是否需要使用网络。
             * mRequiresSatellite: 是否需要使用卫星。
             * mRequiresCell: 是否需要使用基站。
             * mHasMonetaryCost: 是否需要计费。
             * mSupportsAltitude: 是否能提供海拔高度信息。
             * mSupportsSpeed: 是否提供速度信息。
             * mSupportsBearing: 是否支持方位信息。
             * mPowerRequirement: 耗电量级别，可取值有Criteria类定义
的HIGH、MEDIUM和LOW三种级别。
             * mAccuracy: 精度级别，可取值有Criteria类定义的COARSE、
FINE、HIGH、LOW四种级别。
            */
        }
    }
}

```

```

        properties = service.getProperties();
        .....
        if (enabled) {
            service.enable(); // 启动这个LP
            if (request != null) { // 如果客户端有请求的话,
则将该请求发送给LP
                service.setRequest(request, source);
            }
        }
    } catch .....  

    synchronized (mLock) {
        mProperties = properties;
    }
}
};


```

至此，LMS的初始化工作暂告一段落，下面来看看  
GpsLocationProvider的创建。

### (3) GpsLocationProvider初始化

GpsLocationProvider类本身有一段初始化代码，如下所示。

[-->GpsLocationProvider.java: : static语句]

```
// GpsLP定义了一些native函数，此处的class_init_native将初始化相关JNI方法。后文介绍
static { class_init_native(); }
```

接着来看GpsLP的创建，其代码如下所示。

[-->GpsLocationProvider.java: : GpsLocationProvider]

```
public GpsLocationProvider(Context context, ILocationManager
ilocationManager) {
    mContext = context;
    /*
```

NTP为Network Time Protocol之意，它是一种用来同步计算机时间的协议。该时间的源是UTC。

在下面这行代码中，GpsLP将创建一个NtpTrustedTime对象，该对象将采用SNTP(Simple NTP)协议

来和指定NTP服务器通信以获取准确的时间。Android平台中，NTP服务器可在两个地方设置。

1) 在系统资源文件中设置，由字符串config\_ntpServer表示，默认值

为“2.android.pool.ntp.org”。

请求处理的超时时间由整型参数config\_ntpTimeout控制，默认值为20000ms。

2) 也可在Settings的数据库中设置，对应的控制选项为“ntp\_server”和“ntp\_timeout”。

NtpTrustedTime优先使用Settings设置的信息。

NtpTrustedTime比较简单，对NTP感兴趣的读者见参考资料[31]。

\*/

```
mNtpTime = NtpTrustedTime.getInstance(context);  
mILocationManager = ilocationManager;  
// GpsNetInitiatedHandler和ULP Network Initiated请求有关  
// 主要处理来自GPS HAL层通知的NI (Network Initiated) 事件  
mNIHandler = new GpsNetInitiatedHandler(context);
```

```
mLocation.setExtras(mLocationExtras);
```

```
.....
```

```
mConnMgr = (ConnectivityManager) context.getSystemService(  
    Context.CONNECTIVITY_SERVICE);
```

```
mProperties = new Properties();
```

```
try {
```

```
/*
```

读取GPS配置文件，PROPERTIES\_FILE的位置为“/etc/gps.conf”。

笔者的Galaxy Note中，该文件的内容如下所示，其中“#”后的内容为笔者添加的注释。

```
NTP_SERVER=north-america.pool.ntp.org #指定NTP_SERVER
```

#下面这几个参数指定AGPS LTO (Long Term Orbits) 数据的下载地址。LTO存储了GPS卫星

#的星历数据。从下面这些地址中的“4day”来看，这些数据的有效期为4天。AGPS使用时需要

#先从服务器上下载一些辅助数据。如果周围没有网络的情况下该怎么办呢？LTO就好比离线地图，

#当周围没有网络时，终端可以利用事先保存的LTO数据来初始化GPS的定位。当然，LTO有时

#效限制，一般是4天。

```
XTRA_SERVER_1=http://gllto.glpals.com/4day/glo/v2/latest/lto2.dat
```

```
XTRA_SERVER_2=http://gllto.glpals.com/4day/glo/v2/latest/lto2.dat
```

```
XTRA_SERVER_3=http://gllto.glpals.com/4day/glo/v2/latest/lto2.dat
```

```
SUPL_HOST=supl.google.com #指定SUPL的主机和端口
```

```
SUPL_PORT=7276
关于LTO，见参考资料[32]。
*/
File file = new File(PROPERTIES_FILE);
FileInputStream stream = new FileInputStream(file);
mProperties.load(stream); // 解析该配置文件
stream.close();
// 获取配置文件中设置的SUPL主机地址以及端口号
mSuplServerHost = mProperties.getProperty("SUPL_HOST");
String portString =
mProperties.getProperty("SUPL_PORT");
if (mSuplServerHost != null && portString != null) {
    mSuplServerPort = Integer.parseInt(portString);
    .....
}
// C2K是CDMA2000的缩写。C2K_HOST和C2K_PORT主要用于GPS模块的
测试
// 对用户来说，这两个参数没有太大的意义
```

[②](#)

```
mC2KServerHost = mProperties.getProperty("C2K_HOST");
portString = mProperties.getProperty("C2K_PORT");
if (mC2KServerHost != null && portString != null) {
    mC2KServerPort = Integer.parseInt(portString);
    .....
}
}.....
mHandler = new ProviderHandler();
// SUPL的初始化可以由两种特殊的短信触发，下文将简单介绍
listenForBroadcasts这个函数
listenForBroadcasts();
mHandler.post(new Runnable() {
    public void run() {
        LocationManager locManager =
            LocationManager)
mContext.getSystemService(Context.LOCATION_SERVICE);
        // 接收来自NetworkLP的位置更新通知
        // 当GpsLP收到来自NetworkLP的位置信息后，将把它们传给GPS
```

HAL层去处理

```
locManager.requestLocationUpdates(LocationManager.PASSIVE_PROVIDER,
        0, 0, new NetworkLocationListener(),
        mHandler.getLooper());
    }
}
}
```

下面来看上述代码中提到的listenForBroadcasts函数，其内容如下所示。

[-->GpsLocationProvider.java: : listenForBroadcasts]

```
private void listenForBroadcasts() {
    IntentFilter intentFilter = new IntentFilter();
    /*
     * SUPL INIT流程可由一条特殊的数据短信（Data Message）触发。注意，数据短信和我们接触最多的
     * 文本短信（Text Message）不同。下面这个IntentFilter将接收发往
     * 127.0.0.1:7275的数据短信。
     * 7275为OMA-SUPL使用的端口号，关于OMA-SUPL端口号见可参考资料
     * [28]。
     * 关于Android中数据短信的收发，见参考资料[33]。
     */
    intentFilter.addAction(Intents.DATA_SMS_RECEIVED_ACTION);
    intentFilter.addDataScheme("sms");
    intentFilter.addDataAuthority("localhost", "7275");
    mContext.registerReceiver(mBroadcastReciever,
        intentFilter, null, mHandler);

    // SUPL INIT也可由WAP推送短信触发，该短信包含的数据类型为MIME中的
    // "application/vnd.omaloc-supl-init"
    intentFilter = new IntentFilter();

    intentFilter.addAction(Intents.WAP_PUSH_RECEIVED_ACTION);
    try {
        intentFilter.addDataType("application/vnd.omaloc-
supl-init");
    } .....
    mContext.registerReceiver(mBroadcastReciever,
        intentFilter, null, mHandler);
```

```

// 监听ALARM事件和网络事件 (CONNECTIVITY_ACTION)
intentFilter = new IntentFilter();
intentFilter.addAction(ALARM_WAKEUP);
intentFilter.addAction(ALARM_TIMEOUT);
// 监听网络事件, 下文介绍AGPS时还会讨论它

intentFilter.addAction(ConnectivityManager.CONNECTIVITY_ACTION);
mContext.registerReceiver(mBroadcastReciever,
intentFilter, null, mHandler);
}

```

当GpsLP收到指定的数据短信或WAP推送短信后, checkSmsSuplInit或checkWapSuplInit函数将被调用。这两个函数的功能比较简单, 就是将短信的内容传递到GPS HAL层, 来看看它们的代码。

[-->GpsLocationProvider.java: : checkSmsSuplInit和checkWapSuplInit]

```

private void checkSmsSuplInit(Intent intent) {
    SmsMessage[] messages =
Intents.getMessagesFromIntent(intent);
    for (int i=0; i <messages.length; i++) {
        byte[] supl_init = messages[i].getUserData();
        native_agps_ni_message(supl_init,supl_init.length);
    }
}
private void checkWapSuplInit(Intent intent) {
    byte[] supl_init = (byte[]) intent.getExtra("data");
    native_agps_ni_message(supl_init,supl_init.length);
}

```

至此, LMS的初始化流程就算介绍完毕。LMS本身还包括其他一些功能, 例如通知客户端位置更新等。这部分内容都比较简单, 读者完全可自行学习并掌握它们。

下面来看GpsLP的工作流程。

## 2. GpsLP工作流程分析

GpsLP整体结构如图9-31所示。GpsLP分为Java、JNI层、HAL层以及内核层。其中, JNI层和HAL层都属于Native层。

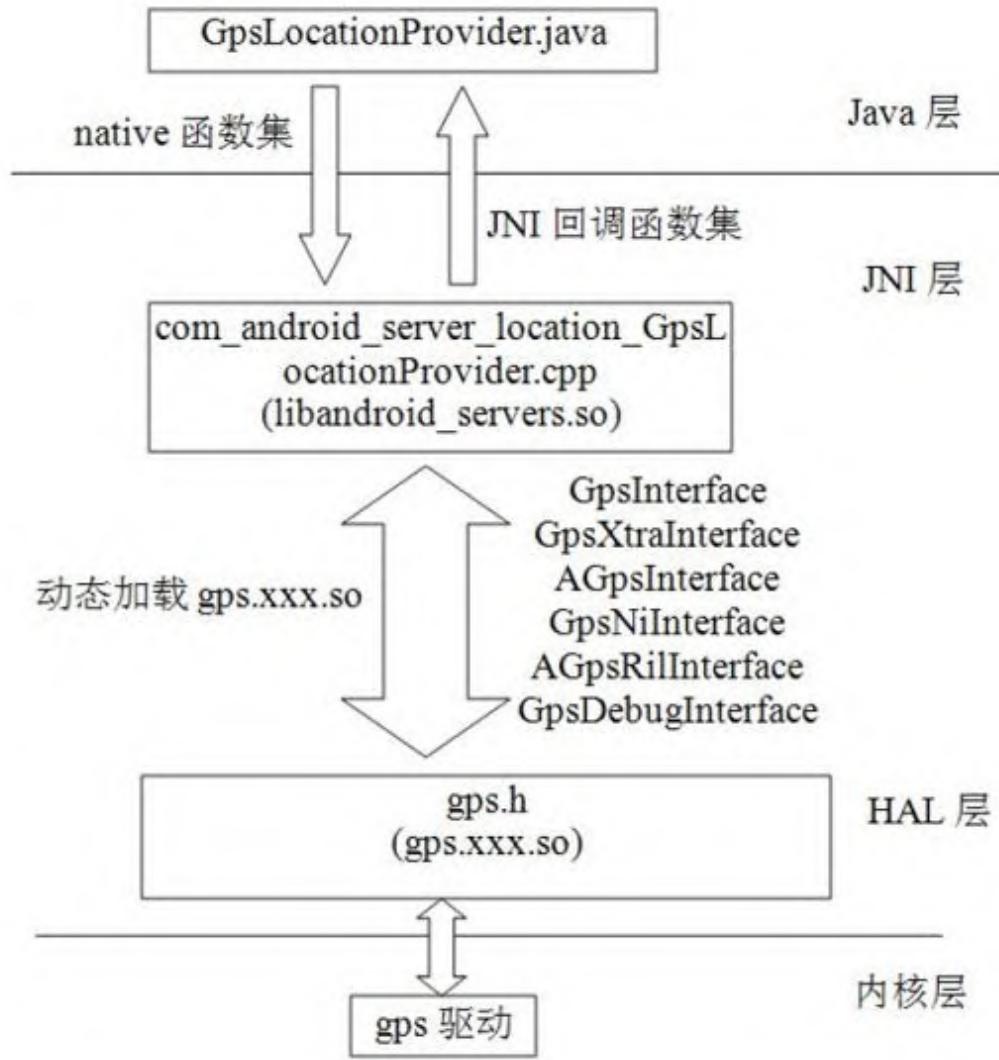


图9-31 GpsLP架构

- Java层主要文件是GpsLocationProvider. java。它通过native函数集与JNI层模块通信。本章后文将介绍native函数集的内容。
- JNI层包括一个核心动态库，即libandr-oid\_server. so。其中，和GpsLP相关的JNI函数位于com\_android\_server\_loca-tion\_GpsLocationProvider. cpp中。JNI层通过JNI回调函数集将GPS信息返回给Java层。本章后文将介绍JNI回调函数集的内容。
- HAL层也包含一个核心动态库，其命名规范为”gps. xxx. so”。其中，“xxx”为手机所使用的硬件平台名。以笔者的Galaxy Note 2为例，对应的GPS HAL层动态库文件名为gps. exynos4. so。Android系统

为GPS HAL层和GPS JNI层双写通信定义了多个接口函数集（如GpsInterface等，它们都定义在gps.h文件中），下文介绍。

- 最后，GPS HAL层将和内核层中的gps驱动交互。

这里要特别对GPS HAL层进行说明。从AOSP源码来看，几大厂商都没有公开其GPS HAL层的实现，AOSP中GPS HAL模块的默认实现也没有可参考价值。同时，网络上能找到的关于GPS HAL模块（从HAL层到驱动层）的框架或设计资料非常少。基于上述考虑，本节对GPS HAL层介绍的内容将集中在GpsInterface等JNI与HAL层交互接口上。

**注意** 高通公司在其开源的QRD（高通参考设计）代码中提供了高通平台GPS HAL层模块的实现，但高通平台HAL层实现采用的是C/S架构。GPS HAL模块仅仅将上层请求转成QMI（QC MSM Interface）消息并发送给相关服务去处理。同样，由于缺乏相关文档，笔者很难搞清楚高通平台上GPS模块的具体工作流程。

下面我们将从Java层开始介绍GpsLP的工作流程。首先是GPS的启动流程。

### （1）启动GPS

Android平台中，GPS的开启和关闭主要在设置程序中控制，相关控制界面如图9-32所示。



图9-32 定位服务设置界面

当用户单击图9-32中三个选择框时，以下函数将被触发。

[-->LocationSettings.java: : onPreferenceTreeClick]

```
public boolean onPreferenceTreeClick(PreferenceScreen  
preferenceScreen,  
        Preference preference) {  
    final ContentResolver cr = getContentResolver();  
    if (preference == mNetwork) { // 对应图9-32中“使用无线网络”选项  
        Settings.Secure.setLocationProviderEnabled(cr,  
                LocationManager.NETWORK_PROVIDER,  
                mNetwork.isChecked());  
    } else if (preference == mGps) { // 对应图9-32中“使用GPS卫星”选  
   项  
        boolean enabled = mGps.isChecked();  
        Settings.Secure.setLocationProviderEnabled(cr,  
                LocationManager.GPS_PROVIDER, enabled);  
        if (mAssistedGps != null) {  
            mAssistedGps.setEnabled(enabled);  
        }  
    }  
}
```

```

        } else if (preference == mAssistedGps) { // 对应图9-32中“使用辅助性GPS”选项
            // 注意，国内某些运营商定制的手机没有该选项。此处将直接修改
            Settings
                // 数据库中secure表中“assisted_gps_enabled”字段值
                Settings.Global.putInt(cr,
                    Settings.Global.ASSISTED_GPS_ENABLED,
                    mAssistedGps.isChecked() ? 1 : 0);
        }
        return true;
    }
}

```

上述代码中的相关操作将修改settings数据库中location\_providers\_allowed字段。当“GPS卫星”和“无线网络”两项都选中时，location\_providers\_allowed字段取值将变成“network, gps”。显然，LMS只要通过ContentObserver<sup>③</sup>监听该字段的变化就可快速响应用户的设置。LMS中，设置对该字段监听的相关代码如下所示。

[-->LocationManagerService.java: : init]

```

mContext.getContentResolver().registerContentObserver(
    Settings.Secure.getUriFor(Settings.Secure.LOCATION_PROVIDERS_ALL
        OWNED), true,
    new ContentObserver(mLocationHandler) {
        public void onChange(boolean selfChange) {
            synchronized (mLock) {
                updateProvidersLocked(); // 最终调用的处理函数是
                updateProvidersLocked
            }
        }
    }, UserHandle.USER_ALL);

```

updateProvidersLocked除了根据设置的情况调用对应LP的enable或disable函数外，还需要通知LP的监听者（即9.3.1节应用示例中介绍的LocationListener对象）。根据LP的启动或禁止情况，LocationListener的onProviderEnabled/onProviderDisabled将被调用。

现在，重点介绍GpsLP的enable函数。该函数内部将发送ENABLE\_MSG消息，而该消息最终将调用GpsLP的handleEnable进行处理。该函数的代码如下所示。

[-->GpsLocationProvider.java: : handleEnable]

```
private void handleEnable() {
    synchronized (mLock) {
        if (mEnabled) return;
        mEnabled = true;
    }
    boolean enabled = native_init(); // 初始化GPS HAL层模块
    if (enabled) {
        mSupportsXtra = native_supports_xtra(); // 判断GPS模块是否支持xtra
        if (mSuplServerHost != null) { // 设置SUPL服务器地址和端口
            native_set_agps_server(AGPS_TYPE_SUPL,
mSuplServerHost, mSuplServerPort);
        }
        if (mC2KServerHost != null) {
            native_set_agps_server(AGPS_TYPE_C2K,
mC2KServerHost, mC2KServerPort);
        }
    } .....// 处理disabled的情况
```

在handleEnable函数中，GpsLP主要通过调用native函数集中的几个函数来初始化底层GPS模块。这些函数将在介绍GPS HAL层时再来详细介绍。

当GPS模块启动成功后，GPS HAL层将通过JNI回调函数通知GpsLP底层GPS引擎的工作能力，这个回调函数是setEngineCapabilities，其代码如下所示。

[-->GpsLocationProvider.java: : setEngineCapabilities]

```
private void setEngineCapabilities(int capabilities) {
    mEngineCapabilities = capabilities; // capabilities代表GPS引擎的工作能力，详情见下文解释
    if (!hasCapability(GPS_CAPABILITY_ON_DEMAND_TIME) &&
!mPeriodicTimeInjection) {
        mPeriodicTimeInjection = true;
        requestUtcTime(); // 输入UTC时间信息
    }
}
```

目前，Android平台中定义的GPS引擎工作能力取值有如下几种。

- GPS\_CAPABILITY\_SCHEDULING：如果设置，则GPS模块在工作周期内（即GPS开启之后，关闭之前这一段时间）能定时通知位置信息，例如每10秒通知一次位置信息。如果GPS模块不支持该功能，表明GPS模块在每个工作周期内只能通知一次位置信息。对于这种GPS，GpsLP将通过不断启动和关闭GPS模块来模拟实现GPS\_CAPABILITY\_SCHEDULING的功能。下文代码中将介绍模拟实现方面的内容。
- GPS\_CAPABILITY\_MSB：如果设置，则GPS模块支持Mobile Service Based AGPS。
- GPS\_CAPABILITY\_MSA：如果设置，则GPS模块支持Mobile Service Assisted AGPS。
- GPS\_CAPABILITY\_SINGLE\_SHOT：如果设置，表明GPS模块支持单次定位。与单次定位相对应的是连续定位。
- GPS\_CAPABILITY\_ON\_DEMAND\_TIME：GPS在工作周期内可能需要UTC时间信息。如果设置此能力，GPS模块在需要UTC时间信息时将主动通过相关回调函数（即代码中的requestUtcTime函数，JNI层也会直接调用它）从GpsLP那获取UTC时间。如果没有设置它，则GpsLP将每隔24小时获取UTC时间并输入给GPS模块。

**提示** 简单来说，GPS\_CAPABILITY\_SCHEDULING表示GPS模块支持连续定位，即GPS模块会不断更新位置信息。而GPS\_CAPABILITY\_SINGLE\_SHOT表示GPS模块支持单次定位，即只GPS模块只会通知一次位置信息。单次定位功能适用于那些无需连续获取位置信息的应用程序。这样，GPS通知完位置信息后即可停止工作以节省电力。另外，LocationManager中有一个requestSingleUpdate函数，其功能和单次定位类似。但由于不是所有GPS模块都支持单次定位，所以代码中并没有利用GPS\_CAPABILITY\_SINGLE\_SHOT标志。

现在，假设GPS启动成功，接下来的工作就是启动GPS导航功能。

## （2）启动GPS导航

当客户端调用LocationManager的requestLocationUpdates并设置使用GpsLP（可参考9.3.1节中的示例代码）后，GpsLP的setRequest函数将被调用，该函数的代码如下所示。

[-->GpsLocationProvider.java: : setRequest]

```
public void setRequest(ProviderRequest request, WorkSource source) {
    // 发送SET_REQUEST消息
    // 注意，GpsLP将把客户端发送的ProviderRequest转换成GpsLP使用的
    GpsRequest
    sendMessage(SET_REQUEST, 0, new GpsRequest(request, source));
}
```

SET\_REQUEST消息将由handleSetRequest处理，其代码如下所示。

[-->GpsLocationProvider.java: : handleSetRequest]

```
private void handleSetRequest(ProviderRequest request,
WorkSource source) {
    if (request.reportLocation) { // 该变量用于判断客户端是否要求接收位
置更新信息
        .....// 监视客户端的用电情况。略去相关内容
        mFixInterval = (int) request.interval; // 客户端设置的位
置更新间隔
        if (mFixInterval != request.interval)
            mFixInterval = Integer.MAX_VALUE;

        // mStarted变量用于判断导航是否已经开启
        if (mStarted &&
hasCapability(GPS_CAPABILITY_SCHEDULING)) {
            /*
             * 如果GPS模块支持定时通知位置信息，则设置其运行模式为
GPS_POSITION_RECURRENCE_PERIODIC，
             * 同时还需要将间隔时间传递给GPS模块。
            */
            if (!native_set_position_mode(mPositionMode,
GPS_POSITION_RECURRENCE_PERIODIC,
                mFixInterval, 0, 0))
                Log.e(TAG, "set_position_mode failed in
setMinTime()");
            } else if (!mStarted) { // 如果之前没有启动导航，则此处启动
它
                startNavigating(); // 下文分析
            }
        } else {
            updateClientUids(new int[0]); // 计算客户端耗电情况
            stopNavigating(); // 停止导航
            mAlarmManager.cancel(mWakeupIntent);
        }
    }
}
```

```

        mAlarmManager.cancel(mTimeoutIntent);
    }
}

```

假设之前没有启动导航，根据上述代码可知，startNavigating将被调用，相关代码如下所示。

[-->GpsLocationProvider.java: : startNavigating]

```

private void startNavigating() {
    if (!mStarted) {
        mTimeToFirstFix = 0;    mLastFixTime = 0;
        mStarted = true;
        mPositionMode = GPS_POSITION_MODE_STANDALONE;
        // 默认工作模式为GPS，即不使用AGPS
        if (Settings.Global.getInt(mContext.getContentResolver(),
                Settings.Global.ASSISTED_GPS_ENABLED, 1) != 0) {
            if (hasCapability(GPS_CAPABILITY_MSB))
                // 如AGPS启用，则设置工作模式为AGPS，并且采用MSB
                mPositionMode = GPS_POSITION_MODE_MS_BASED;
        }
        // 如果GPS模块不支持定时通知位置信息，则interval取值为1秒
        int interval = (hasCapability(GPS_CAPABILITY_SCHEDULING) ?
mFixInterval : 1000);
        if (!native_set_position_mode(mPositionMode,
                GPS_POSITION_RECURRENCE_PERIODIC,
                interval, 0, 0)) {.....// 设置定位模式失败处理}
        if (!native_start()) {.....// 导航启动失败处理}
        // 清空GPS卫星信息
        updateStatus(LocationProvider.TEMPORARILY_UNAVAILABLE, 0);
        mFixRequestTime = System.currentTimeMillis();
        // 如果GPS模块不支持定时通知位置信息
        if (!hasCapability(GPS_CAPABILITY_SCHEDULING)) {
            /*
             NO_FIX_TIMEOUT为60秒。对于那些不支持定时通知的GPS模块来说，如果客户端要求的
             更新间隔大于60秒，并且在这之间没有收到GPS的位置通知（这表明GPS还没定位自己
             的位置），则此处会设置一个超时控制以停止GPS导航。此处的代码逻辑需要结合GpsLP
             中正常的超时管理逻辑来理解。
            */
            if (mFixInterval >= NO_FIX_TIMEOUT)

```

```

        mAlarmManager.set(AlarmManager.ELAPSED_REALTIME_WAKEUP,
                           SystemClock.elapsedRealtime() +
                           NO_FIX_TIMEOUT, mTimeoutIntent);
                }
            }
}

```

此处介绍GpsLP如何处理那些不能定时通知位置信息的GPS引擎。GpsLP将注册两个定时触发Intent用于启用和关闭GPS。

- ALARM\_TIMEOUT: 在该Intent的处理中, GpsLP将停止导航。
- ALARM\_WAKEUP: 在该Intent的处理中, GpsLP将启用导航。

### (3) 位置信息通知处理

当GPS模块更新位置时, GPS JNI层将调用GpsLP的reportLocation函数, 其代码如下所示。

```
[-->GpsLocationProvider.java: : reportLocation]

private void reportLocation(int flags, double latitude, double
longitude, double altitude,
                           float speed, float bearing, float accuracy, long
timestamp) {
    synchronized (mLocation) {
        mLocationFlags = flags; // 标志信息
        // 此次通知的位置消息是否有经纬度信息
        if ((flags & LOCATION_HAS_LAT_LONG) ==
LOCATION_HAS_LAT_LONG) {
            mLocation.setLatitude(latitude); // 设置经纬度及时间戳
            mLocation.setLongitude(longitude);
            mLocation.setTime(timestamp);

        mLocation.setElapsedRealtimeNanos(SystemClock.elapsedRealtimeNan
os());
    }
    // 检查是否有海拔信息
    if ((flags & LOCATION_HAS_ALTITUDE) ==
LOCATION_HAS_ALTITUDE)
        mLocation.setAltitude(altitude);
    else
        mLocation.removeAltitude();
    // 检查是否有速度信息
    if ((flags & LOCATION_HAS_SPEED) == LOCATION_HAS_SPEED)

```

```

        mLocation.setSpeed(speed);
    else mLocation.removeSpeed();
    // 检查是否有方位信息
    if ((flags & LOCATION_HAS_BEARING) == LOCATION_HAS_BEARING)
        mLocation.setBearing(bearing);
    else mLocation.removeBearing();
    // 是否有精度信息(以米为单位)
    if ((flags & LOCATION_HAS_ACCURACY) ==
LOCATION_HAS_ACCURACY)
        mLocation.setAccuracy(accuracy);
    else mLocation.removeAccuracy();

    mLocation.setExtras(mLocationExtras);

    try {
        // mILocationManager指向LMS, 它会把此处的位置信息通知给客户端
        mILocationManager.reportLocation(mLocation, false);
    } .....
} // synchronized (mLocation) 代码段结束

mLastFixTime = System.currentTimeMillis();
// 计算首次定位时间, 即TTFF
if (mTimeToFirstFix == 0 && (flags & LOCATION_HAS_LAT_LONG) ==
LOCATION_HAS_LAT_LONG) {
    mTimeToFirstFix = (int)(mLastFixTime -
mFixRequestTime);
    synchronized (mListeners) {
        int size = mListeners.size();
        for (int i = 0; i < size; i++) {
            Listener listener = mListeners.get(i);
            try {
                /*
                 GpsLP还支持另外一种类型的监听者, 即
GpsStatusListener,
主要用来通知GPS卫星信息。客户端通过LocationManager
的
addGpsStatusListener来注册监听对象。
此处将调用GpsStatusListener的onFirstFix函
数。
*/
            }
            listener.mListener.onFirstFix(mTimeToFirstFix);
        } .....
    }
}

```

```

/*
    .....
    对于不支持GPS_CAPABILITY_SCHEDULING的GPS模块来说，当GpsLP收到一次位置通知事件后，  

    它将先暂停GPS导航，然后等到超时时间到达后，再启用导航。
*/
if (mStarted && mStatus != LocationProvider.AVAILABLE) {
    // 和startNavigating函数最后一段代码相对应，取消在那里设置的超时  

    监控
    if (!hasCapability(GPS_CAPABILITY_SCHEDULING) &&
        mFixInterval < NO_FIX_TIMEOUT)
        mAlarmManager.cancel(mTimeoutIntent); // 取消超时控制

    Intent intent = new
    Intent(LocationManager.GPS_CHANGE_ACTION);
    intent.putExtra(LocationManager.EXTRA_GPS_ENABLED, true);
    mContext.sendBroadcastAsUser(intent, UserHandle.ALL);
    updateStatus(LocationProvider.AVAILABLE, mSvCount);
}
// 对于不支持GPS_CAPABILITY_SCHEDULING功能的GPS模块，GpsLP将软件模  

拟连续定位功能
if (!hasCapability(GPS_CAPABILITY_SCHEDULING) && mStarted &&
    mFixInterval > GPS_POLLING_THRESHOLD_INTERVAL)
    hibernate(); // 停止导航。该函数中将设置ALARM_WAKEUP定时任务以  

重新启用导航
}

```

GpsLP处理位置通知的相关代码主要就在上面介绍的reportLocation中。GpsLP最终会将位置信息告诉LMS，而LMS的handleLocationChanged函数还有许多工作要接着开展。这部分内容请读者自行阅读。

#### (4) reportStatus和reportSvStatus

除了通知位置信息外，GPS模块还会通过reportStatus向GpsLP反馈GPS模块的工作状态，该函数如下所示。

[-->GpsLocationProvider.java: : reportStatus]

```

private void reportStatus(int status) {
    synchronized (mListeners) {
        boolean wasNavigating = mNavigating;
        // 根据状态来更新mNavigating和mEngineOn这两个变量
        switch (status) {

```

```

        // 下面这两个状态分别表示GPS导航开始和结束
        case GPS_STATUS_SESSION_BEGIN: {.....}
        case GPS_STATUS_SESSION_END: {.....}
        // 下面这两个状态分别表示GPS引擎开启和关闭
        case GPS_STATUS_ENGINE_ON: {.....}
        case GPS_STATUS_ENGINE_OFF:{.....}
    }

    if (wasNavigating != mNavigating) {
        int size = mListeners.size();
        for (int i = 0; i < size; i++) {
            Listener listener = mListeners.get(i);
            try {// 调用GpsListener的onGpsStarted函数或
onGpsStopped函数
                if (mNavigating)
                    listener.onGpsStarted();
                else listener.onGpsStopped();
            } .....
        }
        // 发送广播
        Intent intent = new
Intent(LocationManager.GPS_ENABLED_CHANGE_ACTION);

        intent.putExtra(LocationManager.EXTRA_GPS_ENABLED, mNavigating);
        mContext.sendBroadcastAsUser(intent,
UserHandle.ALL);
    }
}

```

reportStatus比较简单，读者了解即可。另外，GPS模块也会通过reportSvStatus返回卫星的信息，这个函数比较有意思，我们来看看。

[-->GpsLocationProvider.java: : reportSvStatus]

```

private void reportSvStatus() {
/*
    下面这个函数用于从GPS HAL层读取卫星的状态，其返回值为卫星的个数。
    该函数每个参数的类型都是一个int数组，其中，除最后一个参数对应的int数组
元素个数为3外，
    而其他参数的int数组元素个数为MAX_SVS（值为32），每个参数的作用如下。
    mSvs: 用于存储卫星的编号。
    mSnrs: 用于存储卫星的SNR（信号噪声比）。
    mSvElevations: 存储卫星的高度信息。

```

mSvAzimuths：存储卫星的方位信息。

mSvMasks：该数组包含三个整型值，每个整型值字长32位，分别对应32颗卫星。

EPHEMERIS\_MASK（值为0）：该变量的每一位代表GPS模块是否获取了该卫星的ephemeris数据。

ALMANAC\_MASK（值为1）：该变量的每一位代表GPS模块是否获取了该卫星的almanac数据。

USED\_FOR\_FIX\_MASK（值为2）：该变量的每一位代表该卫星是否参与了上一次的定位信息计算。

```
/*
int svCount = native_read_sv_status(mSvs, mSnrs,
                                     mSvElevations, mSvAzimuths, mSvMasks);
synchronized (mListeners) {
    int size = mListeners.size();
    for (int i = 0; i < size; i++) {
        Listener listener = mListeners.get(i);
        try {// 通知客户端
            listener.mListener.onSvStatusChanged(svCount, mSvs, mSnrs,
                                                 mSvElevations, mSvAzimuths,
                                                 mSvMasks[EPHEMERIS_MASK],
                                                 mSvMasks[ALMANAC_MASK],
                                                 mSvMasks[USED_FOR_FIX_MASK]);
        } .....
    }
    updateStatus(mStatus,
    Integer.bitCount(mSvMasks[USED_FOR_FIX_MASK]));
    .....
}
```

只要监听这些信息，读者就能实现类似GpsTestPlus应用的效果，如图9-33所示。

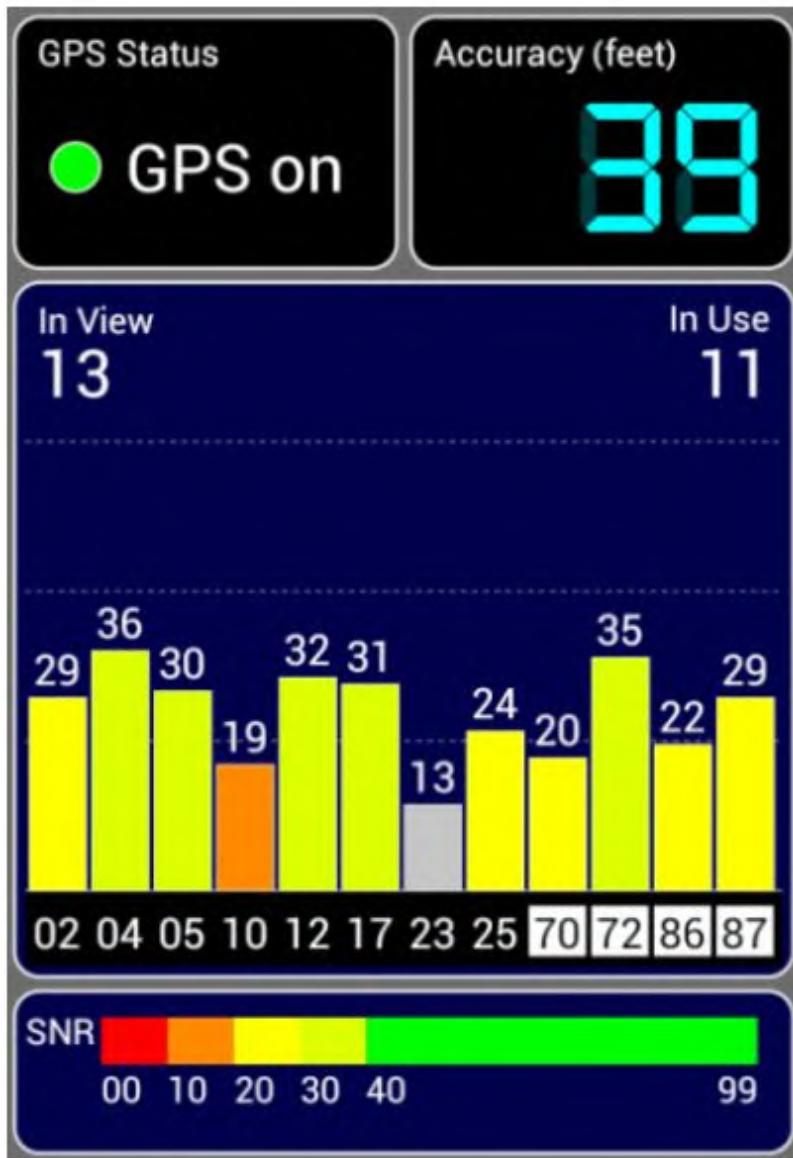


图9-33 GpsTestPlus应用效果

提示 GpsTestPlus是一个比较常用的GPS测试软件。笔者曾经反编译过GpsTestPlus，其中和LM相关的函数调用比较简单，感兴趣的读者可尝试研究其代码。

本节主要介绍了GpsLP的主要工作流程，与之相关的知识点如下。

- GpsLP启动GPS、启动导航、位置信息处理、状态和卫星信息通知等几个主要工作的代码分析。从流程上来说，这部分代码难度非常小，笔者觉得大多数读者都能学会。

- GPS工作能力等重要标志及相关处理逻辑。
- GPS Java层调用的native函数。下文还将详细介绍这些函数的作用。

### 3. AGPS工作流程分析

Android平台中，AGPS的处理逻辑也集中在GpsLocationProvider. java中，本节就来介绍GpsLP中AGPS的相关代码。

#### (1) 处理网络事件

AGPS需要使用移动网络，所以早在GpsLP初始时，GpsLP就注册了对CONNECTIVITY\_ACTION广播的监听。该广播事件由ConnectivityService发送，并且属于Sticky的广播。根据《深入理解Android：卷II》6.4.1节“registerReceiver分析之二”的介绍可知，对于Sticky广播的接收者而言，系统会立即调度一次广播发送流程以将当前的网络事件传递给监听者。也就是说，GpsLP的onReceive函数将很快被触发。该函数的代码如下所示。

[-->GpsLocationProvider. java: : onReceive]

```
public void onReceive(Context context, Intent intent) {
    String action = intent.getAction();
    .....
    // 我们只关注CONNECTIVITY_ACTION事件
} else if
(action.equals(ConnectivityManager.CONNECTIVITY_ACTION)) {
    int networkState;
    // 获取网络状态
    if
(intent.getBooleanExtra(ConnectivityManager.EXTRA_NO_CONNECTIVITY, false))
        networkState =
LocationProvider.TEMPORARILY_UNAVAILABLE;
    else networkState = LocationProvider.AVAILABLE;
    // 获取网络事件的相关信息。NetworkInfo表示此次事件主角，它描述了哪个网络发生了什么事件
    NetworkInfo info =
intent.getParcelableExtra(ConnectivityManager.EXTRA_NETWORK_INFO);
    ConnectivityManager connManager = (ConnectivityManager)
```

```

mContext.getSystemService(Context.CONNECTIVITY_SERVICE);
    info = connManager.getNetworkInfo(info.getType());
    // 更新网络状态，其内部将发送UPDATE_NETWORK_STATE消息
    updateNetworkState(networkState, info);
}
}

```

UPDATE\_NETWORK\_STATE消息最终由handleUpdateNetworkState处理，此函数的代码如下所示。

[-->GpsLocationProvider.java: : handleUpdateNetworkState]

```

private void handleUpdateNetworkState(int state, NetworkInfo
info) {
    mNetworkAvailable = (state == LocationProvider.AVAILABLE);

    if (info != null) {
        // 判断系统是否允许使用移动数据
        boolean dataEnabled =
Settings.Global.getInt(mContext.getContentResolver(),
                    Settings.Global.MOBILE_DATA, 1) ==
1;
        // 假设移动数据功能已经启用，故networkAvailable为true
        boolean networkAvailable = info.isAvailable() &&
dataEnabled;
        // 获取APN，APN（Access Point Number）用于确定使用哪种方式连接到
        // 网络
        String defaultApn = getSelectedApn();
        if (defaultApn == null) defaultApn = "dummy-apn";
        // 将这些信息传递到GPS HAL层
        native_update_network_state(info.isConnected(),
info.getType(),
                                info.isRoaming(),
networkAvailable,
                                info.getExtraInfo(),
defaultApn);
    }
    /*
        mAgsDataConnectionState初始状态为
        AGPS_DATA_CONNECTION_CLOSED。该值在reportAGpsStatus
        中被修改。下面if这段代码逻辑需要结合reportAGpsStatus来综合理解。初
        次进来，if条件不满足。
    */
    if (info != null && info.getType() ==
ConnectivityManager.TYPE_MOBILE_SUPPL

```

```

    && mAGpsDataConnectionState ==
AGPS_DATA_CONNECTION_OPENING) {
/*
注意，这段代码只有在mAGpsDataConnectionState状态为
AGPS_DATA_CONNECTION_OPENING
才起作用。
*/
String apnName = info.getExtraInfo();
if (mNetworkAvailable) {
    if (apnName == null) apnName = "dummy-apn";
    mAGpsApn = apnName;
    if (mAGpsDataConnectionIpAddr != 0xffffffff) {
        boolean route_result;
        // requestRouteToHost:将SUPL的数据都路由到指定的主机地址
        上
        route_result =
mConnMgr.requestRouteToHost(ConnectivityManager.TYPE_MOBILE_SUPL
,
                           mAGpsDataConnectionIpAddr);
    }
    native_agps_data_conn_open(apnName); // 打开数据通道
    mAGpsDataConnectionState = AGPS_DATA_CONNECTION_OPEN;
}
// 如果当前有可用网络，则GpsLP将获取NTP时间以及下载xtra数据
if (mNetworkAvailable) {
    if (mInjectNtpTimePending == STATE_PENDING_NETWORK)
        sendMessage(INJECT_NTP_TIME, 0, null); // 触发
handleInjectNtpTime函数
    if (mDownloadXtraDataPending == STATE_PENDING_NETWORK)
        sendMessage(DOWNLOAD_XTRA_DATA, 0, null); // 触发
handleDownloadXtraData函数
}
}

```

handleUpdateNetworkState还算比较简单，其主要工作如下。

- 如果GPS模块要求开启AGPS数据下载（这部分逻辑下节再介绍，即mAGpsDataConnectionState的值为AGPS\_DATA\_CONNECTION\_OPENING），则handleUpdateNetwork将开展相关操作。
- 如果网络启用并且GpsLP之前没有获取过NTP时间以及下载过Xtra数据，GpsLP将通过INJECT\_NTP\_TIME和DOWNLOAD\_XTRA\_DATA两个消息获取NTP时间以及下载Xtra数据。

GPS模块什么时候会要求开启AGPS数据下载呢？相关函数集中在reportAGpsStatus中。

## (2) reportAGpsStatus分析

GPS模块将通过reportAGpsStatus和GpsLP交互，该函数代码如下所示。

```
[-->GpsLocationProvider.java: : reportAGpsStatus]

private void reportAGpsStatus(int type, int status, int ipaddr)
{
    switch (status) {
        case GPS_REQUEST_AGPS_DATA_CONN:// GPS模块要求启用数据链接
            /*
             此处，设置mAGpsDataConnectionState为
             AGPS_DATA_CONNECTION_OPENING,
             该状态值表示数据链接处于开启过程中。
            */
            mAGpsDataConnectionState = AGPS_DATA_CONNECTION_OPENING;
            /*
             要求使用移动数据。startUsingNetworkFeature是
             ConnectivityManager中一个比较重要的函
             数，其第一个参数表示此处操作的网络是移动网络，第二个参数表示使用
             SUPL相关的功能。
            */
            int result = mConnMgr.startUsingNetworkFeature(
                ConnectivityManager.TYPE_MOBILE,
                Phone.FEATURE_ENABLE_SUPL);
            mAGpsDataConnectionIpAddr = ipaddr; // 保存地址
            // APN_ALREADY_ACTIVE表示startUsingNetworkFeature指定的网
            络及功能已经启用
            if (result == PhoneConstants.APN_ALREADY_ACTIVE) {
                if (mAGpsApn != null) { // mAGpsApn在onReceive函数中被
                    设置
                    if (mAGpsDataConnectionIpAddr != 0xffffffff) {
                        boolean route_result;
                        route_result = mConnMgr.requestRouteToHost(
                            ConnectivityManager.TYPE_MOBILE_SUPL, mAGpsDataConnectionIpAddr);
                    }
                    native_agps_data_conn_open(mAGpsApn); // 调用
                    native函数表示AGPS数据链接
                    // 开启AGPS数据链接开启成功
                }
            }
        }
    }
}
```

```

        mAfpsDataConnectionState =
AGPS_DATA_CONNECTION_OPEN;

    } else if (result == PhoneConstants.APN_REQUEST_STARTED)
{
    // APN_REQUEST_STARTED表示startUsingNetworkFeature设置的
    // 请求已经发送给相关模块处理
    } else.....
break;
// 下面这个值表示AGPS不再需要使用数据链接
case GPS_RELEASE_AGPS_DATA_CONN:{.....}
break;
// 下面这个值表示GPS模块中AGPS数据链接初始化完毕
case GPS_AGPS_DATA_CONNECTED: {.....}
break;
// 下面这个值表示GPS模块中AGPS数据链接完毕
case GPS_AGPS_DATA_CONN_DONE: {.....}
break;
// 下面这个值表示GPS模块中AGPS数据链接失败
case GPS_AGPS_DATA_CONN_FAILED: {.....}
break;
}
}

```

reportAGpsStatus主要对两个状态进行处理，即GPS\_REQUEST\_AGPS\_DATA\_CONN和GPS\_RELEASE\_AGPS\_DATA\_CONN。

总之，当网络准备好后，GpsLP将调用native\_agps\_data\_conn\_open启用GPS模块中AGPS数据链接功能。

### (3) 下载Xtra数据

最后，看看Xtra数据的下载处理，这部分功能由handleDownloadXtraData实现，代码如下所示。

[-->GpsLocationProvider.java: : handleDownloadXtraData]

```

private void handleDownloadXtraData() {
if (mDownloadXtraDataPending == STATE_DOWNLOADING) return;
.....
mDownloadXtraDataPending = STATE_DOWNLOADING;
mWakeLock.acquire();
// 利用后台线程下载数据
AsyncTask.THREAD_POOL_EXECUTOR.execute(new Runnable() {

```

```

    public void run() {
        GpsXtraDownloader xtraDownloader = new
        GpsXtraDownloader(
                mContext, mProperties);
        // 下载LTO数据
        byte[] data = xtraDownloader.downloadXtraData();
        // 将这些数据传递给GPS HAL层
        if (data != null) native_inject_xtra_data(data,
        data.length);
        // 发现XTtra数据下载完毕通知
        sendMessage(DOWNLOAD_XTRA_DATA_FINISHED, 0, null);
        .....
        mWakeLock.release();
    }
}
}
}

```

GpsLP中的AGPS流程也比较简单。GpsLP只要监控网络事件以及接收来自GPS HAL层的AGPS状态通知即可顺利完成相关工作。

结合前面对GpsLP工作流程的分析可知，Android平台中，Java层的GPS模块和Native层（包括JNI和HAL层）GPS模块交互非常多，下面将重点分析GPS Native层方面的知识。

#### 4. GPS JNI与HAL层

在GpsLocationProvider初始化一节曾提到，GpsLocationProvider类有一个静态的初始化代码，在那段代码中，class\_init\_native函数将初始化JNI层相关模块。马上来看它。

##### (1) JNI与HAL层初始化

class\_init\_native函数对应的JNI函数如下所示。

[-->com\_android\_server\_location\_GpsLocationProvider.cpp: :  
android\_location\_GpsLocationProvider\_class\_init\_native]

```

static void
android_location_GpsLocationProvider_class_init_native(JNIEnv*
env, jclass clazz) {
    int err;
    hw_module_t* module;
    // 获取JNI回调函数集对应的Java MethodId

```

```

// 如果不熟悉JNI, 请阅读《深入理解Android: 卷 I》第2章
method_reportLocation = env->GetMethodID(clazz,
"reportLocation", "(IDDDFFFJ)V");
.....// JNI回调函数集的其他函数, 下文会详细介绍它们
method_setEngineCapabilities = env->GetMethodID(clazz,
"setEngineCapabilities", "(I)V");
.....
// 加载GPS HAL层模块, GPS_HARDWARE_MODULE_ID的值为"gps"
err = hw_get_module(GPS_HARDWARE_MODULE_ID, (hw_module_t const**) &module);
if (err == 0) {
    hw_device_t* device;
    // 打开GPS HAL层模块
    err = module->methods->open(module,
GPS_HARDWARE_MODULE_ID, &device);
    if (err == 0) {
        gps_device_t* gps_device = (gps_device_t *) device;
        // 获取GPS HAL层最主要的交互接口GpsInterface
        // 它是JNI层和HAL层的重要互通通道
        sGpsInterface = gps_device-
>get_gps_interface(gps_device);
    }
}
// GPS HAL模块对外还提供了几个重要交互接口
if (sGpsInterface) {
    sGpsXtraInterface = // 用来和GPS HAL层中xtra模块交互的接口
        (const GpsXtraInterface*) sGpsInterface-
>get_extension(GPS_XTRA_INTERFACE);
    sAGpsInterface = // 用来和GPS HAL层AGPS模块交互的接口
        (const AGpsInterface*) sGpsInterface-
>get_extension(AGPS_INTERFACE);
    sGpsNiInterface = // 用来和GPS HAL层NI (Network
Initiated) 模块交互的接口
        (const GpsNiInterface*) sGpsInterface-
>get_extension(GPS_NI_INTERFACE);
    sGpsDebugInterface = // 用来调试的接口
        (const GpsDebugInterface*) sGpsInterface-
>get_extension(GPS_DEBUG_INTERFACE);
    sAGpsRilInterface = // 用来和GPS HAL层AGPS及RIL相关的接口
        (const AGpsRilInterface*) sGpsInterface-
>get_extension(AGPS_RIL_INTERFACE);
}
}

```

GpsLP启用后, native\_init函数将被调用, 它对应的JNI函数代码如下所示。

```
[-->com_android_server_location_GpsLocationProvider.cpp: :
android_location_GpsLocationProvider_init]

static jboolean
android_location_GpsLocationProvider_init(JNIEnv* env, jobject
obj)
{
    // this must be set before calling into the HAL library
    if (!mCallbacksObj)
        mCallbacksObj = env->NewGlobalRef(obj);
    // 向GPS HAL层设置回调函数接口
    // 当GPS HAL层有情况需要通知JNI层时，这些回调函数将被调用
    if (!sGpsInterface || sGpsInterface->init(&sGpsCallbacks) != 0)
        return false;
    // 设置其他接口的回调函数
    if (sGpsXtraInterface && sGpsXtraInterface-
>init(&sGpsXtraCallbacks) != 0)
        sGpsXtraInterface = NULL;
    if (sAGpsInterface) sAGpsInterface->init(&sAGpsCallbacks);
    if (sGpsNiInterface) sGpsNiInterface-
>init(&sGpsNiCallbacks);
    if (sAGpsRilInterface) sAGpsRilInterface-
>init(&sAGpsRilCallbacks);
    return true;
}
```

通过上述代码可知，Android平台中通过定义多个交互接口实现了GPS Java层、JNI层以及HAL层的交互问题。显然，理解这些接口的作用将有助于我们学习Android平台中GPS模块的实现。下面介绍Java层与JNI层的交互接口。

## (2) Java层与JNI层交互接口函数

先来看GPS Java层调用的JNI函数，它们的作用如下所示。

[-->GpsLocationProvider.java]

```
// 初始化JNI层相关的类型
native void class_init_native();
// 判断系统是否支持GPS
native boolean native_is_supported();
// 初始化GPS HAL层模块
native boolean native_init();
```

```

// 清理GPS HAL层模块所分配的资源
native void native_cleanup();
/*
    设置GPS模块工作模式，其各个参数取值含义如下。
    mode: GPS工作模式，目前可取值有GPS_POSITION_MODE_STANDALONE（值为
0，仅GPS工作），
                GPS_POSITION_MODE_MS_BASED（值为1，AGPS MSB模式）、
GPS_POSITION_MODE_MS_ASSISTED
                （值为2，AGPS MSA模式）。
    recurrence: 位置更新模式，目前可取值有
GPS_POSITION_RECURRENCE_PERIODIC（值为0，连续
定位）、GPS_POSITION_RECURRENCE_SINGLE（值为1，单次定位）。
    min_interval: 最短位置更新时间，单位为毫秒。
    preferred_accuracy: 期望的位置更新精度，单位为米。
    preferred_time: 期望的TTFF时间，单位为毫秒。
*/
native boolean native_set_position_mode(int mode,
                                         int recurrence, int min_interval,
                                         int preferred_accuracy, int preferred_time);
// 下面这两个函数用于启动和关闭导航
native boolean native_start();
native boolean native_stop();
// 删掉AGPS辅助数据，其flags参数的取值请读者阅读
GpsLocationProvider.java的deleteAidingData函数
native void native_delete_aiding_data(int flags);
// 读取卫星信息，该函数在“reportStatus和reportSvStatus介绍”中学过
native int native_read_sv_status(int[] svs, float[] snrs,
                                 float[] elevations, float[] azimuths, int[] masks);
// 读取NMEA数据
native int native_read_nmea(byte[] buffer, int bufferSize);
// 输入位置信息，在GpsLP中，该位置信息由NetworkLP提供
native void native_inject_location(double latitude, double
longitude, float accuracy);

/*
输入NTP时间。
time: 为NtpTimeTrustedTime从网络中获取到的NTP时间。
timeReference: 为设备从开机到一次NTP请求处理成功后的所耗费的时间，由
SystemClock的
elapsedRealtime函数返回。
uncertainty: 准确度。在Android系统中，该值为NTP请求发送和接收往返时间的一般。详情可参考
SntpClient.java文件。
*/
native void native_inject_time(long time, long timeReference,

```

```

int uncertainty);
// GPS模块是否支持XTRA数据
native boolean native_supports_xtra();
// 输入XTRA数据，即LTO数据
native void native_inject_xtra_data(byte[] data, int length);
// 用于调试，获取GPS模块内部状态
native String native_get_internal_state();

// 打开AGPS数据下载通道，参数apn指明了所要使用的APN
native void native_agps_data_conn_open(String apn);
// 关闭AGPS数据下载通道
native void native_agps_data_conn_closed();
// GpsLP处理AGPS相关事宜失败时候调用下面这个函数
native void native_agps_data_conn_failed();
// 将来自数据短信或WAP推送短信得到的信息传递给GPS模块
native void native_agps_ni_message(byte [] msg, int length);
/*
设置AGPS服务端地址，其中type取值有两种。
AGPS_TYPE_SUPL: 值为1，代表SUPL服务器。
AGPS_TYPE_C2K: 值为2，代表C2K服务器。
*/
native void native_set_agps_server(int type, String hostname,
int port);

/*
当GPS模块需要使用APGS时，会调用reportNiNotification函数（详情见下文）以
通知用户。
用户处理完后，将通过下面这个函数告知GPS模块处理结果。注意，这部分内容涉及
OMA-SUPL相关知识，
请读者阅读参考资料[28]。该函数的参数如下。
notificationId: 通知id，代表GPS HAL层的某一个处理请求。
userResponse有三种取值，分别是
GPS_NI_RESPONSE_ACCEPT:值为0，代表用户允许相关操作。
GPS_NI_RESPONSE_DENY:值为1，用户拒绝相关操作。
GPS_NI_RESPONSE_NORESP:值为2，代表用户无回应。
NI和GpsNetInitiatedHandler有关，读者可自行研究。
*/
native void native_send_ni_response(int notificationId, int
userResponse);
/*
设置AGPS参考位置信息，其各参数解释如下。
type:取值可为AGPS_REF_LOCATION_TYPE_GSM_CELLID（值为1，代表GMS网络的
cell id）、
AGPS_REF_LOCATION_TYPE_UMTS_CELLID（值为2，代表CDMA网络的cell
id）、

```

AGPS\_REG\_LOCATION\_TYPE\_MAC (值为3, 代表MAC地址)。

mcc: Mobile Country Code (移动国家码), 由3位数字组成, 唯一地识别移动用户所属的国家, 中国为460。

mnc: Mobile Network Code (移动网络码), 用于识别移动用户所归属的移动网络。中国移动TD系统使用00,

中国联通GSM系统使用01, 中国移动GSM系统使用02, 中国电信CDMA系统使用03。

lac: Location Area Code (位置区码)。移动通信中, 为了确定终端台的位置, 每个移动网络的覆盖

区都被划分成许多位置区, 位置区码(LAC)则用于标识不同的位置区。

cid: cell id (基站编号)。

\*/

```
native void native_agps_set_ref_location_cellid(int type, int  
mcc, int mnc,  
                int lac, int cid);
```

/\*

设置终端与移动网络相关的一些参数信息, 其type参数决定了setid参数的取值。

type可取值如下。

AGPS\_SETID\_TYPE\_NONE:值为0, 无意义。

AGPS\_SETID\_TYPE\_IMSI:值为1, 代表IMSI (international mobiles  
subscriber identity,

国际移动用户号码标识)。IMSI信息存储在SIM卡上。

AGPS\_SETID\_TYPE\_MSISDN:值为2, 代表Mobile Subscriber ISDN (用户号码), 即手机号码。

\*/

```
native void native_agps_set_id(int type, String setid);
```

/\*

通知GPS HAL层系统当前网络的状态, 其各参数解释如下。

connected: 网络是否连接。

type: 网络类型, 可取值请参考ConnectivityManager中各网络类型的定义情况。常用的有TYPE\_MOBILE、

TYPE\_WIFI等。

roaming: 是否处于漫游状态。

available: 网络是否可用。

extraInfo: 附加信息。

defaultAPN: 默认APN。

\*/

```
native void native_update_network_state(boolean connected, int  
type,  
                boolean roaming, boolean available, String  
extraInfo, String defaultAPN);
```

现在来看看JNI回调Java层的函数, 如下所示。

[-->GpsLocationProvider.java]

```
// GPS模块汇报位置信息
void reportLocation(int flags, double latitude,
                     double longitude, double altitude,
                     float speed, float bearing, float accuracy, long
timestamp):
// GPS模块通知GPS工作状态
void reportStatus(int status):
// GPS模块通知卫星状态
void reportSvStatus():
// GPS模块通知AGPS状态
void reportAGpsStatus(int type, int status, int ipaddr):
// GPS模块通知NMEA信息
void reportNmea(long timestamp):
// GPS模块通知GpsLP自己的能力
void setEngineCapabilities(int capabilities):
// GPS模块要求下载XTRA数据
void xtraDownloadRequest():
// GPS模块通知Network Initiated通知，其各参数解释如下
void reportNiNotification(
    int notificationId, // GPS HAL层分配的通知ID
    int niType,          // NI类型，可取值请参考gps.h的
GpsNiType的定义
    int notifyFlags,     // 标志信息，可取值请参考gps.h的
GpsNiNotifyFlags定义
    int timeout,          // 等待用户处理的超时时间
    // 当超时发生后，系统采用的默认处理结果。其取值和
native_send_ni_respons中第二个参数一样
    int defaultResponse,
    String requestorId, // 代表请求者的ID
    String text,          // 通知信息
    int requestorIdEncoding, // requestorId的编码格式，
参考gps.h GpsNiEncodingType的定义
    int textEncoding, // text的编码格式
    String extras       // 附加信息
):
// 要求获取参考位置信息，flags参数目前没有使用
void requestRefLocation(int flags):
// 要求设置移动网络相关信息，flags参数表示要获取什么样的信息
// 可取值同native_agps_set_id的type参数一致
void requestSetID(int flags):
// 要求获取UTC时间
void requestUtcTime():
```

接下来看GPS JNI与HAL层的交互接口。主要介绍JNI调用HAL层的接口。

### (3) GpsInterface及其他交互接口

GpsInterface是GPS JNI与HAL层交互的主要接口。在Android平台中，该接口定义在一个同名的结构体中，其内容如下所示。

[-->gps.h: : GpsInterface结构体]

```
typedef struct {
    size_t           size; // GpsInterface结构体的长度
    // 初始化GPS模块。其参数为GPS模块所需的回调函数
    int   (*init)( GpsCallbacks* callbacks );
    // 下面这两个函数用于开启或关闭导航
    int   (*start)( void );
    int   (*stop)( void );
    // 清理GPS模块所分配的资源
    void  (*cleanup)( void );
    // 输入UTC时间，参数解释同native_inject_time
    int   (*inject_time)( GpsUtcTime time, int64_t
timeReference, int uncertainty );
    // 输入位置信息，参数解释同native_inject_location
    int   (*inject_location)( double latitude, double longitude,
float accuracy );
    // 删除赋值信息，其参数解释同native_delete_aiding_data
    void  (*delete_aiding_data)( GpsAidingData flags );
    // 设置GPS模块的工作模式，其参数解释同native_set_position_mode
    int   (*set_position_mode)( GpsPositionMode mode,
                                GpsPositionRecurrence recurrence,
                                uint32_t min_interval, uint32_t preferred_accuracy,
                                uint32_t preferred_time );
    // 获取GPS模块实现的扩展接口
    // AGPS扩展接口对应的name为“agps”、xtra扩展接口对应的name为“xtra”
    const void* (*get_extension)( const char* name );
} GpsInterface;
```

比较GpsInterface和Java层定义的native函数，发现二者结合非常紧密。实际上，JNI实现的那些native函数最终都会把请求通过HAL层接口交给GPS模块去处理。

再来看gps.h定义的GpsXtraInterface接口，相关内容封装在同名的结构体中，如下所示。

[-->gps.h: : GpsXtraInterface结构体]

```
typedef struct {
    size_t          size;
    // 初始化GPS中的xtra相关模块
    int (*init)( GpsXtraCallbacks* callbacks );
    // 输入xtra数据, 其参数解释同native_inject_xtra_data
    int (*inject_xtra_data)( char* data, int length );
} GpsXtraInterface;
```

接下来看AGpsInterface结构体，代码如下所示。

[-->gps.h: : AGpsInterface]

```
typedef struct {
    size_t          size;
    // 初始化AGPS模块
    void (*init)( AGpsCallbacks* callbacks );
    // 打开AGPS数据连接, 其参数解释同native_data_conn_open
    int (*data_conn_open)( const char* apn );
    // 关闭AGPS数据连接, 其参数解释同native_data_conn_close
    int (*data_conn_closed)();
    // AGPS数据连接操作失败, 同native_data_conn_fail
    int (*data_conn_failed)();
    // 设置AGPS服务器地址等相关信息, 参数解释同native_agps_set_server
    int (*set_server)( AGpsType type, const char* hostname, int
port );
} AGpsInterface;
```

最后来看GpsNiInterface和AGpsRilInterface接口。GpsNiInterface的代码如下所示。

[-->gps.h: : GpsNiInterface]

```
typedef struct
{
    size_t          size;
    // 初始化NI模块
    void (*init)( GpsNiCallbacks *callbacks );
    // 发送NI回复, 请参数同native_send_ni_response
    void (*respond)( int notif_id, GpsUserResponseType
user_response );
} GpsNiInterface;
```

AGpsRilInterface的代码如下所示。

[-->gps.h: : AGpsRilInterface]

```
typedef struct {
    size_t          size;
    // 初始化AGPS Ril相关的处理模块
    void (*init)(AGpsRilCallbacks* callbacks);
    // 设置参考位置信息，其第一个参数类型为AGpsRefLocation
    // 该结构体的成员与native_agps_set_ref_location_cellid函数的参数一一对应
    void (*set_ref_location)(const AGpsRefLocation
    *agps_reflocation,
                           size_t sz_struct);
    // 设置AGPS移动网络ID信息，其参数解释同native_agps_set_id
    void (*set_set_id)(AGpsSetIDType type, const char* setid);

    // 设置NI消息，其参数解释同native_agps_ni_message
    void (*ni_message)(uint8_t *msg, size_t len);
/* 注意，下面这两个函数的参数合起来就是native_update_network_state的
参数。Java层调用
一次native_update_network_state将触发下面这两个函数被调用更新移动
网络状态。
*/
    void (*update_network_state)(int connected, int type,
                                int roaming, const char* extra_info);
    // 设置网络连接状态
    void (*update_network_availability)(int available, const
char* apn);
} AGpsRilInterface;
```

本节对Android平台中LocationManagerService及相关模块进行了介绍，尤其对本章的核心GpsLocationProvider及GPS各层次及之间的交互接口进行了重点讲解。

和本书前面介绍的WifiService、WifiP2pService以及NfcService比起来，LMS不复杂，这其中的主要原因包括以下。

- LMS提供的服务本身比较简单，它的核心功能就是提供位置信息。
- GpsLP通过合理的分层接口设计使得GPS HAL层之上的代码能够不受底层硬件的影响。

另外，希望读者在学习完本章后，对以下内容开展进一步的学习。

- 研究PassiveProvider和FusedLocationProvider的内容。掌握LMS如何与位于应用进程的LP进行交互。
- 学习LocationFudger的内容，掌握如何模糊位置信息。
- 学习GeofenceManager的内容。

最后，读者可尝试反编译NetworkLocation.apk，掌握NetworkLP以及Geocoder的实现原理<sup>④</sup>。

① 笔者在Android源码中经常能发现类似TwilightService这样在SDK中暂时还没有踪迹的模块。这些模块预示了Android未来版本中可能提供的一些新的功能。关于Twilight，Google Play上有一个比较有意思的APK，读者不妨安装试试，地址为  
<https://play.google.com/store/apps/details?id=com.urbandroid.lux>。

② 关于C2K的解释，笔者在此对高通公司资深研发经理毛晓冬的大力支持表示感谢。

③ 读者可参考《深入理解Android：卷II》第7章。

④ 出于对版权的考虑，不能在书中对NetworkLocation.apk反编译的结果开展详细讨论。如果时机合适，笔者将在博客上对它的实现进行介绍。

## 9.4 本章总结和参考资料说明

### 9.4.1 本章总结

本章内容分为两大块。

- 首先对本章所用到的GPS基础知识进行了介绍，这部分内容比较广，难度不大，读者可轻松掌握。
- 然后对Android平台中的位置管理模块进行了详细介绍。从代码上看，这些模块的难度都不大。

从整体情况来看，GPS是全书难度最大的一章，其中一个主要原因是GPS所涉及的背景知识非常庞杂，而且历经几十年的发展，相关的知识更新速度也非常快。

目前，与GNSS相关的LBS还在高速发展，谁能在如此激烈的竞争中拔得头筹呢？当然是有技术积淀，并能灵活快速应对市场需求的公司和组织。在此，希望读者不要满足于掌握本章甚至本书的内容，而应该把目标放得更远一些，只有跟上技术发展的脚步，才能在激烈竞争中处于有利的位置。

## 9.4.2 参考资料说明

### 1. 概述

[1] [http://baike.baidu.com/subview/152851/5072513.htm  
fromId=152851&from=rdtself](http://baike.baidu.com/subview/152851/5072513.htm?fromId=152851&from=rdtself)

说明：百度百科对LBS的介绍，读者可了解其“产生背景”一节的内容。

### 2. GPS基础知识

GPS基础知识这一节主要的参考资料是《GPS Essentials of Satellite Navigation Compendium》一书。对软件工程师而言，它是笔者找到的关于GPS知识最全面和最通俗易懂的一本书。该书电子版可参考<http://www.docin.com/p-67411894.html>。

### 3. 坐标系

[2] 《GPS Essentials of Satellite Navigation Compendium》第2章“Coordinate systems”

说明：对GPS坐标系相关知识的介绍，读者可先略过投影相关的内容。

[3] [http://principles.ou.edu/earth\\_figure\\_gravity/geoid/](http://principles.ou.edu/earth_figure_gravity/geoid/)

说明：对GPS高度和海拔高度的解释。

### 4. 时间系统

[4] 《GPS基本原理及其Matlab仿真》3.2节“GPS时间系统”

说明：时间系统是本章最难讲解的部分了，读者可在本章基础上做深入学习。

[5] <http://www.doc88.com/p-241652413475.html>  
<http://www.doc88.com/p-241652413475.html>

说明：GPS与民用时间的转换。

[6] <http://www.hko.gov.hk/gts/time/basicterms-localtimec.htm>

说明：中国香港天文台官方网站对本地时间的介绍。

## 5. 卫星轨道等知识

[7] 《GPS Essentials of Satellite Navigation Compendium》第3章“Foundations of satellite technology”

## 6. GPS基础知识

[8] <http://www.gps.gov/systems/gps/>

说明：GPS系统的美国官方网站，内容非常详细和生动，建议读者仔细阅读。

[9] <http://www.gps.gov/systems/gps/space/>

说明：GPS官方网站对GPS空间段建设历史的描述。

[10]

[http://en.wikipedia.org/wiki/List\\_of\\_GPS\\_satellite\\_launches](http://en.wikipedia.org/wiki/List_of_GPS_satellite_launches)

说明：维基百科对GPS卫星发射规划的介绍。

[11] 《GPS Essentials of Satellite Navigation Compendium》第4章“GNSS technology: the GPS example”

## 7. GPS通信频段

[12]

<http://www.gps.gov/systems/gps/modernization/civilsignals/#L2C>

[13]

<http://www.gps.gov/systems/gps/modernization/civilsignals/#L5>

说明：GPS官方网站对L2C和L5频段的介绍。

#### [14] Interface Specification GPS 705C

说明：该资料是GPS官方规范，用于定义空间段和地面控制段以及用户段通过L5频段交互的接口。如果不需要对L5做进一步了解，可略过。GPS官方规范的下载地址为<http://www.gps.gov/technical/>。

### 8. GPS信号

#### [15] 《Understanding the GPS: Introduction to the GPS》第二部分“Basic Signal Structure And Error”

说明：这本书是《GPS Essentials of Satellite Navigation Compendium》一书的升级参考书籍，讲解得非常透彻，覆盖面也比较广，建议读者深入阅读。

### 9. GPS导航电文

#### [16] Interface Specification GPS 200G

说明：该资料是GPS官方规范，用于定义空间段和地面控制段以及用户段通过L1及L2频段交互的接口。这应该是GPS的核心规范了，建议读者详细阅读。其中，附录II、III和IV详细介绍了GPS卫星发送导航电文的组成及相关参数。

### 10. 定位计算相关知识

#### [17] 《GPS Essentials of Satellite Navigation Compendium》第6章“Calculating position”

#### [18] <http://www.doc88.com/p-797227641283.html>

说明：“GPS测速精度研究及应用”，一篇硕士学位论文，读者权当参考。

#### [19]

<http://wenku.baidu.com/view/3541a8a0b0717fd5360cdcc4.html>

说明：中国移动A-GPS终端技术规范。

[20] <http://www.gpsinformation.org/dale/why12.htm>

说明：该资料对多channel功能可提升GPS接收器定位速度的原因进行了介绍。

## 11. NMEA-0183和GPX

[21] <http://www.doc88.com/p-992198901288.html>

说明：NMEA Reference Manual. NMEA规范文档，感兴趣的读者可详细阅读。

[22] <http://www.topografix.com/gpx.asp>

说明：GPX官方网站。GPS格式比较简单，读者可轻松学会。

[23] [http://en.wikipedia.org/wiki/GPS\\_eXchange\\_Format](http://en.wikipedia.org/wiki/GPS_eXchange_Format)

说明：维基百科对GPX的介绍。

[24] <https://developers.google.com/kml/documentation/>

说明：Google关于KML的介绍，上面有一些初学者入门指南资料。

## 12. GPS增强系统

[25] 《GPS Essentials of Satellite Navigation Compendium》第7章”Improved GPS: DGPS, SBAS, A-GPS and HSGPS”

[26]

[http://www.tutorialspoint.com/lte/lte\\_radio\\_protocol\\_architecture.htm](http://www.tutorialspoint.com/lte/lte_radio_protocol_architecture.htm)

说明：Control Plane和User Plane的区别。

## 13. OMA-SUPL介绍

[27] OMA Secure User Plane Location Architecture 3.0版

说明：OMA-SUPL官方技术文档，它对SUPL整个架构和相关功能进行了定义，读者务必认真阅读。

[28] OMA User Plane Location Protocol 3.0版

说明：ULP协议文档，建议读者认真阅读。注意，OMA-SUPL相关协议的最高版本为3.0，读者可从官方网站

[http://technical.openmobilealliance.org/Technical/release\\_program/supl\\_V3\\_0.aspx](http://technical.openmobilealliance.org/Technical/release_program/supl_V3_0.aspx)下载。

[29]

<http://wenku.baidu.com/view/cc6b150703d8ce2f006623e2.html>

说明：“基于SUPL的移动定位系统的研究和设计”，一篇硕士学位论文，对移动定位技术介绍得非常全面，建议读者认真阅读。

#### 14. LocationManager应用示例

[30] <https://developer.android.com/google/play-services/location.html>

说明：Android SDK中关于Google Play Service中Location API的官方介绍，建议那些对开发高级LBS相关应用程序感兴趣的读者阅读它。

#### 15. NTP相关

[31] [http://en.wikipedia.org/wiki/Network\\_Time\\_Protocol](http://en.wikipedia.org/wiki/Network_Time_Protocol)

说明：维基百科关于NTP协议的介绍。

#### 16. LTO相关

[32] <http://www.broadcom.com/products/GPS/Location-Based-Services/LTO-AGPS>

说明：博通公司关于LTO的介绍，建议读者仔细阅读。

## 17. 数据短信收发

[33] <http://blog.fordemobile.com/2012/09/use-sms-to-send-and-receive-raw-data.html>

# 读累了记得休息一会儿哦~

公众号：古德猫宁李

- 电子书搜索下载
- 书单分享
- 书友学习交流

网站：[沉金书屋 https://www.chenjin5.com](https://www.chenjin5.com)

- 电子书搜索下载
- 电子书打包资源分享
- 学习资源分享

# 附录

[-->笔者发给吴劲良先生的邮件]

吴兄：

我有个问题想和你深层次讨论，就是关于这本书的定位。先说说我的看法。

1) 对于初学者（就是完全没有Wi-Fi、NFC、GPS经验的人），这本书肯定是入门书，但是它的难度比普通意义上认为的入门书难。

2) 对于中级学者，这些人定位在1~2年或者有过实际修改bug的经验，但是缺乏全局理解的人，这本书也合适。不过，可能有部分内容对他们来说比较简单。另外，关于NFC和GPS的知识，从统计情况来看，NFC和GPS的问题非常少。从面试情况来看，对NFC芯片datasheet的了解（GPS应该是没有这方面公开的资料）也很重要。不过本书没有考虑NFC、GPS以及Wi-Fi HAL层的内容。一方面我感觉Wi-Fi驱动层和协议结合非常紧密，有点钻研精神的读者在本书基础上，再有一些驱动经验就可以搞定。而NFC HAL层未来发展趋势可能会和wpa\_supplicant一样，即不会出现NXP、Broadcom这样太过特定的内容。GPS一般不太可能让外人看到驱动的代码。我专门看过QC开源项目CODEAurora<sup>①</sup>中GPS的HAL代码，它是C/S架构的，只有Client端内容，而且都是简单地发些命令，然后接收回复，没有核心技术。

3) 对于高级学者，经验和理论知识都比较到位的人，本书可以当做参考书来看，不过内容相对会显得浅显。

一方面，昨天和Eva沟通后，觉得本书没有太多实际经验，确实如此。我自己定位这本书还是想打通整个知识面，实际经验的话，需要理论联系实践。现在很多工程师只有实践，没有理论，或者理论关注较少。另一方面，如果专门讲实践，这种书反而价值不高，因为可操作性太低。它不像网管类的书籍，一步一步跟着做就行了。

这是我对本书定位的一些看法，吴兄，你能否从一线工程师，培养新人等多方面讲下你的感受？不足之处也提出来哈。

最后，写完这本书后，我感觉在Wi-Fi、NFC和GPS这几块，核心都是芯片厂商做好了，我们唯一可做的就是改改bug，攒足实战经验，似乎可发挥的地方非常少（NFC CE模式还有很多发挥空间，尤其是安全交付解决方案之类的）。吴兄，对这个问题又怎么看呢？

诚挚欢迎吴兄的金玉良言！

邓凡平

[-->吴劲良先生的回复]

邓兄：

正如邓兄提到，本书对不同知识深度的学者而言，会有各自的收获，可引初学者入门，可给中级学者问题分析的线索，可给高级学者一个知识思索的机会（对比自己的理解和补充知识），本书能起抛砖引玉的作用，不同读者从中收获多少还得看个人，多思考的读者还可以从书中学习到邓兄分析问题的思路，反思如何提升自己的搜索技巧。

本书以理论分析为主，没有具体问题的解答，但是我觉得够了。这不是一本Q&A的书，Wi-Fi、NFC、GPS这三大部分，Android涉及的主干支知识都有，读者可以选择性地深入分析，每个人对知识点的追求都不一样，很难满足所有人的需求。就个人而言，我会对Android Wi-Fi的休眠策略、Location的网络定位感兴趣，跟实际工作遇到的问题相关。

“NFC和GPS问题非常少”，这会跟功能模块是否被广泛使用和应用的广度有关，被使用多了可能会暴露问题多些，应用场景多也会促使功能的开发，自然会引出新问题。GPS HAL的代码各厂家都不提供，Broadcom、MTK、RDA只是提供so，有可能是涉及核心技术，估计是一些Command的实现，GPS一般是UART接口，UART只负责上层与模组的数据通信。

对于负责无线模组的新人，我对他们工作的安排是：先做功能的验证测试，让他从测试中加深对功能点的理解，知道哪些点是容易出问

题；然后会给一些已经调试好的模组让其单独调试，目的是熟悉调试一个模块需要做哪些工作；最后会渐渐地让其承担一些实际任务。学习的安排是：学习NL802.11，USB、SDIO、UART和I<sup>2</sup>C等模组接口驱动的分析，然后会从内核开始学习，如：Wi-Fi driver->netd->wpa\_supplicant->HAL->framework，Android会安排一些核心知识点的学习，主要是理清工作的机制。最终是希望新人在头脑中有一幅Android网络结构图，并能将其画出来。

由于需要先确保相关的外围模组能配合主控使用，这也决定平常无线工作的重点会在模组的移植调试上，涉及的内核驱动的调试较多，现在Android做得越来越完善，大问题很少，小问题还是有，但解决起来还好（Android 4.4上Wi-Fi目前测试出原生代码有几个bug，较严重的一个是在关闭Wi-Fi时没关闭supplicant创建的socket，每次打开Wi-Fi时又创建，socket打开个数累积超过65536时，后续操作将失败）。

无线模块Wi-Fi、BT、NFC和GPS，核心的技术是在芯片厂，而且是在芯片设计中，driver的编写只是其中的很小一部分，即使是相对复杂的Wi-Fi driver，投入两三个人，花两个月的时间把driver写出来完全没问题。这个我也认同发挥的地方很少，除非是从应用角度去开发新的功能或做一些功能创新。但从工作的角度看，要把这些无线模块支持好，也很容易，调一款新的Wi-Fi就像在弄一个小系统，需要把系统调稳，没有bug并可以达到量产的标准，往往耗上一两个月的时间。虽然发挥的地方少，但当前看来这方面技术人员的需求还是挺大的。

吴劲良

① <https://www.codeaurora.org/>，此处可下载高通参考设计（QC Reference Design）的代码。