# 前言

分析系统应用，需要按照步骤来进行分析，一般需要注意的几点主要有：

最好把源码单独提取出来，并编译运行，这样哪里不清楚就改哪里，不够这里可能有系统api无法识别的问题，需要做系统签名。

* 寻找一个程序的入口，Java，C，C＃都是从main函数开始执行的，分析源码程序时也要先照一个入口
* 先弄清楚程序实现的主要功能是什么
* 关注流程和逻辑，不要过于关注细节。要学会抓大放小。

编译系统app：批量删除product属性的string资源, 未加特殊说明，都是用的android5.1.1（SDK22）

# Android系统应用

## 什么是Android系统应用

Android系统应用具有如下几个特点，这些特点都是普通的Android应用不具备的。

* 嵌入到Android ROM中，通过普通的方法无法卸载这些程序。
* 拥有更高的权限。例如，可以实现静默安装，不会出现权限提示对话框，由于Google Play是Android系统应用，所以即使在没有root权限的情况下，Google Play也可以读写/data/app目录1，所以只需要将APK文件复制到这个目录即可安装程序，根本就不需要通过PackageInstaller2进行安装。
* 可以调用Android SDK的私有API，这些API在ADT开发环境下无法调用。

注：安装apk的本质就是拷贝文件到**/data/app或者其子目录**，1）在root的条件下C:\Users\key.guan>adb push A.apk /data/app 然后reboot就完成了A.apk的安装了；2）adb install –r A.apk

## 为什么要研究Android系统应用

有很多直接与Android应用交互的接口都是在这些系统应用中实现的。了解系统应用，意味着对Android系统最上层的API有一个非常透彻的了解。所有的Android系统应用的源代码都在packages目录。其中packages/app是最核心的目录，所有内嵌的APK程序都在该目录中。app目录告诉我们一切Android系统的窗口（Activity）。研究Android系统应用的源代码好处多多：

* 了解Android系统中有哪些窗口、Content Provider、Service和Broadcast可以与之交互。
* 充分掌握很多高级应用的使用方法，例如，OTA升级是如何实现的。
* 对实现原理比较感兴趣的读者可以通过这些源代码了解像Launcher2、短信管理等应用的内容构造。
* 对于想编写可以完成系统级操作的应用的读者，可以学会如何将Android应用嵌入到ROM，升级为Android系统应用，进而可以做任何自己想做的事。
* 对于那些有极客情结的读者，完全定制自己的ROM是最令人振奋的梦想，而Android系统应用将是实现这一梦想最重要的基石。

## 如何编写Android系统应用

Android系统应用和普通的Android应用基本相同，都是主要用Java语言编写的APK程序。不过前者与后者有如下两点不同。

* 签名不同。Android系统应用使用的是系统签名，或称为平台（Platform）签名，而普通的Android应用使用的是一般的签名。
* 可以访问的API不同。在Android SDK中有很多API（类、接口、方法等）都声明为hide1。这类API不允许在普通的Android应用中访问，而只有经过Platform签名的Android系统应用才能使用这些API。

Android源代码中包含了4个签名，前面使用的platform就是其中之一。这些签名文件中在如下的目录中。

<Android源代码根目录>/build/target/product/security，含义。

* testkey：普通的Key，默认情况下使用。如果Android.mk文件不设置LOCAL\_CERTIFICATE 变量，就使用该签名。
* **platform** ：使用该签名的系统应用可以**执行平台的核心功能**。
* shared ：使用该签名的系统应用可以和home/contacts进程共享数据。
* media ：使用该签名的系统应用将成为media/download 系统的一部分。

LOCAL\_CERTIFICATE 变量可以指定上述4个值，例如，如果值为platform，会使用security目录的platform.pk8文件对APK文件进行签名。如果值为media，会使用media.pk8文件对APK文件进行签名。

我们开发的apk需要用到系统权限，需要在AndroidManifest.xml中添加共享系统进程属性：

android:sharedUserId="android.uid.system"

android:sharedUserId="android.uid.shared"

android:sharedUserId="android.media

apk的签名就需要是系统签名(platform、shared或media)才能正常使用。否则报INSTALL\_FAILED\_SHARED\_USER\_INCOMPATIBLE错，如何编译和运行Android系统应用呢？

### 源码命令行签名（Android.mk）

这种方式比较麻烦，你需要有编译过的源码环境，并按如下步骤：

1、拷贝App源码到Android源码的packages/apps/目录或者子目录下，且App源码是普通(Eclipse)格式的

2、配置Android.mk，在其中添加LOCAL\_CERTIFICATE := platform 或 shared 或 media

3、使用mm编译App，生成的apk即系统签名

Android.mk是Android源代码专用的编译文件，相当于GCC的Makefile文件，内容只需要在apps目录中找一个Android系统应用，如PackageInstaller，将该程序中Android.mk文件复制一份，然后进行适当修改即可。

下面先看一下FirstSystemApp程序的Android.mk文件的内容。

src/ch06/FirstSystemApp/Android.mk

LOCAL\_PATH:= $(call my-dir)

include $(CLEAR\_VARS)

LOCAL\_MODULE\_TAGS := optional

LOCAL\_SRC\_FILES :=$(call all-java-files-under, src)

LOCAL\_STATIC\_JAVA\_LIBRARIES += android-support-v4

#　将编译生成FirstSystemApp.apk文件

**LOCAL\_PACKAGE\_NAME := FirstSystemApp**

**LOCAL\_CERTIFICATE := platform**

include $(BUILD\_PACKAGE)

Android.mk文件的大多数内容都是标准的写法，只有下面两行代码需要了解一下。

LOCAL\_PACKAGE\_NAME := FirstSystemApp

LOCAL\_CERTIFICATE := platform

其中第一行需要根据不同的Android系统应用进行修改。系统会根据LOCAL\_PACKAGE\_NAME变量的值生成APK文件，例如，本例该变量的值是FirstSystemApp，所以会在out/target/product/ maguro/system/app目录生成FirstSystemApp.apk文件， adb push命令上传到/system/app目录即可完成安装。

LOCAL\_CERTIFICATE变量表示签名类型，系统应用通常设为platform，表示Platform签名。

### Java命令行签名

这种方式签名简单，App可以在Eclipse或Android Studio下编译，然后给apk重新签名即可。但调试比较痛苦，即使写成脚本，也需要重复一样的操作，相关文件：

../build/target/product/security/platform.x509.pem

../build/target/product/security/platform.pk8

../out/host/linux-x86/framework/signapk.jar （源码路径: ../build/tools/signapk）

步骤

1、将相关文件及源apk文件置于同一路径下  
2、检查源apk包，去掉META-INF/CERT.SF 和 META-INF/CERT.RSA 文件（可以忽略这一步）  
3、执行签名命令java -jar signapk.jar platform.x509.pem platform.pk8 old.apk new.apk即可

### keytool-importkeypair集成系统签名

让Android Studio集成系统签名，需要用到一个工具keytool-importkeypair，这个工具的作用是将系统签名的相关信息导入到已有的签名文件里。可从[这里](https://github.com/getfatday/keytool-importkeypair)下载。工具的使用方法参考[使用keytool工具](http://czj4451.iteye.com/blog/1487684)这篇文章。最好使用mac 或Linux系统下的AS生成，因为keytool-importkeypair工具是执行在类**Unix**内核的系统上的，不要整成**win**格式的文本了，在window上会有些问题（在git的sh执行，需要修改命令tmpdir=`mkdir -p "/tmp/${scriptname}.XXXX"`）,步骤

1. 在AS下生成demo.jks签名文件：菜单栏->build->Generate Signed APK..->

证书查看：keytool -list -v -keystore SystemSignDemo2.jks -storepass 123456

1. 执行命令编译出demo.jks ：linux下需要为脚本文件添加可执行权限

./keytool-importkeypair -k demo.jks -p 123456 -pk8 platform.pk8 -cert platform.x509.pem -alias demo

# demo.jks : 签名文件

# 123456 : 签名文件密码

# platform.pk8、platform.x509.pem : 系统签名文件

# demo : 签名文件别名

3.builde.gradle配置demo.jks

在android区域下(与defaultConfig同级)添加配置：

signingConfigs {

release {

storeFile file("../signature/demo.jks")

storePassword '123456'

keyAlias 'demo'

keyPassword '123456'

}

debug {

storeFile file("../signature/demo.jks")

storePassword '123456'

keyAlias 'demo'

keyPassword '123456'

}

}

## 系统应用环境

**背景**  
目前我们对settings和systemui ui定制多，靠mk方式，没有ide的高效功能（比如：自动import，layout实时预览，代码自动补全，代码语法检查，断点调试）。因此我们需要用android stdio这个ide来提高ui定制效率

**目标（拟）**  
1.as编译：在as环境下，可以用户窗口正常编译，充分利用ide的高效功能，(+提取系统class.jar，系统签名)  
2.兼容多种编译方式：支持mk和as模块编译，支持系统全量编译  
3.便捷的git操作，团队协作，合理gitignore，  
4.可移植性强：便于从android7.1直接迁移到android8.0

**同行现状**

芯片公司：MTK/RK/intel, ui定制非常少，一般用source insight

手机厂商：hw/小米/酷派， 目前一般都用AS，各种导入包/依赖/AAPT编译问题，如何导入和解决相关的编译问题/网上参考资料较少。

举例：比如系统设置在android7.0又多依赖了settingslib，然后product字段/资源重载机制as的aapt又不支持，

问题：

1.系统设置/SystemUI都是在AS环境开发的么？

2.原生系统设置是基于mk，eclipse架构来编译的，如何迁移到AS下呢？

期望效果，兼容多种编译方式：支持mk和as模块编译，支持系统源码环境全量（mmm）编译；可移植性强，便于从android7.1直接迁移到android8.0（比如系统设置在android7.0比Android5又多依赖了settingslib，Android8还没开始研究，不知道有什么坑）

3.重复资源如何处理？

举例1：字符串资源字段如何兼容product字段  
<string name="launcher\_index\_application" product="product1">Applications1</string>  
<string name="launcher\_index\_application" product="product2">Applications2</string>

编译报错  
Error: Found item String/launcher\_index\_application more than one time

举例2：

资源字段如何解决资源重载问题？AAPT的overlap参数在AS不能识别，配置无效

4.是否有专门的SDK团队在做系统应用的封装了原生的SDK支持，大概需要多少人力

**KPAD**

基本按照同行的【手机厂商】的流程在做，

【系统设置】目前完成度20%

【文件管理器】

专人团队搞SDK，很高技术含量和时间， AS环境，进行framework进行重新封，相对大的企业都是很细分的，我在MTK也只是只做telephony framework，装自己的SDK: 原理就是把@hide接口封装能够被使用，同时兼容不同Android版本主要解决屏蔽不同android 版本的系统差异，android5.0/7.0./8.0系统差异都很大（android7有了）

当前状态：其实你之前的那个class.jar,jar和sdk也差不多了吧。具体到每个系统应用，得根据实际原生代码情况来解决吧，比如系统设置在android7.0又多依赖了settingslib，然后product字段/资源重载机制as的aapt又不支持

dpad-ui

UI部分在AS调试好，剩下的在source insight弄

一般手机厂商，

所以，他们的laucher systemUI settings都是能用AS的:

Source Insight

在android7.0，日后可能是android8.0

系统应用开发规范化调研：

系统设置（/SystemUI）其他公司如何开发的？用eclipse，还是AS。编译用的mmm还是as

系统设置：RK用eclipse看代码，用make编apk。酷派/用

文件管理器:系统签名，AS环境

+风险点：AAPT\_FLAG资源覆盖问题怎么解决（AS未必支持）

解决办法1：修改gradle，字段是否可以支持

解决办法2：用定制后的aapt去支持AS，未经实操

### 全新应用

应用签名：利用keytool-importkeypair做好系统签名继承。不必依赖系统源码环境

### 原生应用

## 小结

完整的系统签名的app过程

1. 应该是使用平台签名，例如此处的Settings.apk的Android.mk文件中指定的签名平台是platform即系统平台签名，所以在签名的时候会使用系统的签名文件进行签名。platform对应的系统签名文件的位置为：

android源码根目录/build/target/product/security/platform.pk8和android源码根目录/build/target/product/security/platform.x509.pem 两个文件

2. 在应用工程的清单配置文件AndroidManifest.xml文件中指定共享用户ID，并将coreApp属性设置为true

3. android源码中使用mm/mmm命令进行编译，其权限就会与系统设置一致；

4. 将apk文件复制到Android设备的/system/app目录中

# 系统应用环境配置方案

关键字：系统应用 系统设置 android7.1 android stdio settings

## 概述

### 行业现状

芯片公司：MTK/RK/intel, ui定制非常少，并且主要是处理系统底层，一般用source insight等文本工具

手机厂商：hw/小米/酷派， 目前一般都用AS，各种导入包/依赖/AAPT编译问题，如何导入和解决相关的编译问题/网上参考资料较少。一般是专门的sdk团队负责。

迁移到AS环境下有一定的工作量，

举例：比如系统设置在android7.0又多依赖了settingslib，然后product字段/资源重载机制as的aapt又不支持，

### 环境方案对比

直接把文件夹拖入eclipse/android studio，本质和记事本效率其实差别不大

|  |  |  |
| --- | --- | --- |
| 生命周期 | 记事本环境方案 | AS环境方案 |
| 1.编码 | 工具：  vi/sourceinsight/notepad/  文档结构不合理：res和src文档结构不合理，修改资源文件和代码文件非常不方便  语法助手：几乎没有 | 文档结构：  语法助手：自动：import，layout实时预览，代码自动补全，代码语法检查，重构变量/方法等等等等。。 |
| 2.编译 | linux的mm命令  语法问题出错：回到1 | 自动 |
| 3.安装 | com控制台的adb install | 自动 |
| 4.运行 | 手动点击/或者脚本启动 | 自动 |
| 5.调试 | 业务逻辑问题：回到1： 插入Log.d代码，然后2->3->4  Ui问题：回到1->2->3->4->5 | 直接依附进程，断点调试 |
| 小结 |  |  |
| 效率指数 | ：1 | 3-5 |
| 环境配置难度 | ：0  不用任何配置 | 环境配置难度：7d  很复杂，Settings为例，依赖10个左右的jar包，3个其他工程，需要编译系统sdk，消除as的aapt和linux的aapt差异 |
| 系统覆盖率 | 100%，应用层，系统层，底层都ok | 20%，只支持部分apk，framwork/native/kernel/driver无法支持 |

目前我们对settings和systemui ui定制多，采用as符合当前业务和效率的需要

### 代码基本框架

由于android系统更新频繁，应用定制必须考虑兼容性问题，做到强移植性，减少如android7.0迁移到8.0的工作量。

### 方案目标

目前我们对settings和systemui ui定制多，采用as符合当前业务和效率的需要。结合目前的协作和工作环境，并且都是远程登陆到服务器工作。该方案需要实现以下目标。

* [as](http://1.as/)环境：在as环境下，可以正常编译，充分利用ide的高效功能，提取系统class.jar，系统签名
* 兼容多种编译方式：支持mk和as模块编译，支持系统全量编译，尽量减少两种编译方式代码的差异性
* 便捷的git操作：团队协作，合理gitignore，可以从服务器代码git pull到本地编写代码
* 可移植性强：便于从android7.1直接迁移到android8.0

### AS说明

Android Studio 3.1.2

Build #AI-173.4720617, built on April 14, 2018

JRE: 1.8.0\_152-release-1024-b02 amd64

JVM: OpenJDK 64-Bit Server VM by JetBrains s.r.o

Windows 7 6.1

## AS本地源码环境搭建-Settings依赖分析

基本思想：初手直接建立git init->优先保证可以as编译删文件->逐个恢复文件为线索，导入jar包。高手可以直接一步到位，走mk分析为线索

### 查找Settings依赖库(Android.mk)

**[javascript]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79765186) [copy](https://blog.csdn.net/qq1028850792/article/details/79765186)

1. LOCAL\_PATH:= $(call my-dir)
2. include $(CLEAR\_VARS)
4. LOCAL\_SRC\_FILES := \
5. $(call all-logtags-files-under, src)
7. LOCAL\_MODULE := settings-logtags
9. include $(BUILD\_STATIC\_JAVA\_LIBRARY)
11. # Build the Settings APK
12. include $(CLEAR\_VARS)
14. LOCAL\_JAVA\_LIBRARIES := bouncycastle core-oj telephony-common ims-common
15. LOCAL\_STATIC\_JAVA\_LIBRARIES := \
16. android-support-v4 \
17. android-support-v13 \
18. android-support-v7-recyclerview \
19. android-support-v7-preference \
20. android-support-v7-appcompat \
21. android-support-v14-preference \
22. jsr305 \
23. settings-logtags
25. LOCAL\_MODULE\_TAGS := optional
27. LOCAL\_SRC\_FILES := \
28. $(call all-java-files-under, src) \
29. $(call all-java-files-under, src-kpad)
31. LOCAL\_RESOURCE\_DIR := $(LOCAL\_PATH)/res \
32. $(LOCAL\_PATH)/res-kpad \
33. frameworks/support/v7/preference/res \
34. frameworks/support/v14/preference/res \
35. frameworks/support/v7/appcompat/res \
36. frameworks/support/v7/recyclerview/res
38. LOCAL\_PACKAGE\_NAME := Settings
39. LOCAL\_CERTIFICATE := platform
40. LOCAL\_PRIVILEGED\_MODULE := **true**
42. LOCAL\_PROGUARD\_FLAG\_FILES := proguard.flags
44. LOCAL\_AAPT\_FLAGS := --auto-add-overlay \
45. --extra-packages android.support.v7.preference:android.support.v14.preference:android.support.v17.preference:android.support.v7.appcompat:android.support.v7.recyclerview
47. ifneq ($(INCREMENTAL\_BUILDS),)
48. LOCAL\_PROGUARD\_ENABLED := disabled
49. LOCAL\_JACK\_ENABLED := incremental
50. LOCAL\_DX\_FLAGS := --multi-dex
51. LOCAL\_JACK\_FLAGS := --multi-dex **native**
52. endif
54. include frameworks/opt/setupwizard/library/common-full-support.mk
55. include frameworks/base/packages/SettingsLib/common.mk
57. include $(BUILD\_PACKAGE)
59. # Use the following include to make our test apk.
60. ifeq (,$(ONE\_SHOT\_MAKEFILE))
61. include $(call all-makefiles-under,$(LOCAL\_PATH))
62. endif

#### LOCAL\_STATIC\_JAVA\_LIBRARIES

为引用的静态库(静态库是需要编译进apk的)

**[javascript]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79765186) [copy](https://blog.csdn.net/qq1028850792/article/details/79765186)

1. LOCAL\_STATIC\_JAVA\_LIBRARIES := \
2. android-support-v4 \
3. android-support-v13 \
4. android-support-v7-recyclerview \
5. android-support-v7-preference \
6. android-support-v7-appcompat \
7. android-support-v14-preference \
8. jsr305 \
9. settings-logtags

#### LOCAL\_JAVA\_LIBRARIES

为非静态库(非静态库是Android系统自带的库)

**[javascript]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79765186) [copy](https://blog.csdn.net/qq1028850792/article/details/79765186)

1. LOCAL\_JAVA\_LIBRARIES := bouncycastle core-oj telephony-common ims-common

#### Include文件夹

**[javascript]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79765186) [copy](https://blog.csdn.net/qq1028850792/article/details/79765186)

1. include frameworks/opt/setupwizard/library/common-full-support.mk
2. include frameworks/base/packages/SettingsLib/common.mk

其中setupwizard/navigationbar和setupwizard/library/和SettingsLib是包含res的工程库,因此不能导入jar包

#### 小结

从Android.mk文件得出结论是:

Settings依赖上述的三个工程(带有res的工程)

静态库

1. android-support-v4 \
2. android-support-v13 \
3. android-support-v7-recyclerview \
4. android-support-v7-preference \
5. android-support-v7-appcompat \
6. android-support-v14-preference \
7. jsr305 \
8. settings-logtags

动态库:

**[javascript]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79765186) [copy](https://blog.csdn.net/qq1028850792/article/details/79765186)

1. bouncycastle core-oj telephony-common ims-common

### SettingsLib

这个模块比较简单

#### comom

**[javascript]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021) @commom.mk

1. LOCAL\_DEX\_PREOPT := **true**
2. ifeq ($(LOCAL\_USE\_AAPT2),**true**)
3. LOCAL\_STATIC\_ANDROID\_LIBRARIES += \
4. android-support-annotations \
5. android-support-v4 \
6. SettingsLib
7. **else**
8. LOCAL\_RESOURCE\_DIR += $(call my-dir)/res
9. LOCAL\_AAPT\_FLAGS += --auto-add-overlay --extra-packages com.android.settingslib
10. LOCAL\_STATIC\_JAVA\_LIBRARIES += \
11. android-support-annotations \
12. android-support-v4 \
13. SettingsLib
14. endif

#### android.mk

fasdf

**[javascript]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. LOCAL\_PATH := $(call my-dir)
2. include $(CLEAR\_VARS)
3. LOCAL\_USE\_AAPT2 := **true**
4. LOCAL\_MODULE := SettingsLib
5. LOCAL\_DEX\_PREOPT := **false**
6. ANDROID\_COMPILE\_WITH\_JACK := **false**
7. #WITH\_DEXPREOPT := false
8. LOCAL\_SHARED\_ANDROID\_LIBRARIES := \
9. android-support-v4 \
10. android-support-v7-recyclerview \
11. android-support-v7-preference \
12. android-support-v7-appcompat \
13. android-support-v14-preference
14. LOCAL\_RESOURCE\_DIR := $(LOCAL\_PATH)/res
15. LOCAL\_JAR\_EXCLUDE\_FILES := none
16. LOCAL\_SRC\_FILES := $(call all-java-files-under, src)
17. include $(BUILD\_STATIC\_JAVA\_LIBRARY)

dfg

### 查找Setupwizard依赖库(Android.mk)

上文分析到还依赖于setupwizard

1. include frameworks/opt/setupwizard/library/common-full-support.mk

具体文件如下：

**[javascript]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. #
2. # Include this make file to build your application against this module.
3. #
4. # Make sure to include it after you've set all your desired LOCAL variables.
5. # Note that you must explicitly set your LOCAL\_RESOURCE\_DIR before including this file.
6. #
7. # For example:
8. #
9. #   LOCAL\_RESOURCE\_DIR := \
10. #        $(LOCAL\_PATH)/res
11. #
12. #   include frameworks/opt/setupwizard/library/common-full-support.mk
13. #
15. # Check that LOCAL\_RESOURCE\_DIR is defined
16. ifeq (,$(LOCAL\_RESOURCE\_DIR))
17. $(error LOCAL\_RESOURCE\_DIR must be defined)
18. endif
20. # Add --auto-add-overlay flag if not present
21. ifeq (,$(findstring --auto-add-overlay, $(LOCAL\_AAPT\_FLAGS)))
22. LOCAL\_AAPT\_FLAGS += --auto-add-overlay
23. endif
25. # Include setup wizard library, if not already included
26. ifeq (,$(findstring setup-wizard-lib-full-support,$(LOCAL\_STATIC\_JAVA\_LIBRARIES)))
27. LOCAL\_RESOURCE\_DIR += \
28. $(call my-dir)/main/res \
29. $(call my-dir)/eclair-mr1/res \
30. $(call my-dir)/full-support/res
31. LOCAL\_AAPT\_FLAGS += --extra-packages com.android.setupwizardlib
32. LOCAL\_STATIC\_JAVA\_LIBRARIES += setup-wizard-lib-full-support
33. endif
35. ## Include transitive dependencies below
37. # Include support-v7-appcompat, if not already included
38. ifeq (,$(findstring android-support-v7-appcompat,$(LOCAL\_STATIC\_JAVA\_LIBRARIES)))
39. LOCAL\_RESOURCE\_DIR += frameworks/support/v7/appcompat/res
40. LOCAL\_AAPT\_FLAGS += --extra-packages android.support.v7.appcompat
41. LOCAL\_STATIC\_JAVA\_LIBRARIES += android-support-v7-appcompat
42. endif
44. # Include support-v7-recyclerview, if not already included
45. ifeq (,$(findstring android-support-v7-recyclerview,$(LOCAL\_STATIC\_JAVA\_LIBRARIES)))
46. LOCAL\_RESOURCE\_DIR += frameworks/support/v7/recyclerview/res
47. LOCAL\_AAPT\_FLAGS += --extra-packages android.support.v7.recyclerview
48. LOCAL\_STATIC\_JAVA\_LIBRARIES += android-support-v7-recyclerview
49. endif

#### 资源LOCAL\_RESOURCE\_DIR

依赖的资源文件如下

1. LOCAL\_RESOURCE\_DIR += \
2. $(call my-dir)/main/res \
3. $(call my-dir)/eclair-mr1/res \
4. $(call my-dir)/full-support/res
5. = frameworks/support/v7/appcompat/res
6. frameworks/support/v7/recyclerview/res

#### LOCAL\_STATIC\_JAVA\_LIBRARIES

setup-wizard-lib-full-support

android-support-v7-appcompat

android-support-v7-recyclerview

#### setup-wizard-lib-full-support模块分析

setup-wizard-本身又依赖setup-wizard-lib-full-support，这个在android,mk定义了该模块

**[javascript]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. ##
2. # Build the full-support library, which includes RecyclerView and any other support libraries as
3. # they are integrated.
4. #
5. include $(CLEAR\_VARS)
7. LOCAL\_AAPT\_FLAGS := --auto-add-overlay \
8. --extra-packages android.support.v7.appcompat \
9. --extra-packages android.support.v7.recyclerview
10. LOCAL\_MANIFEST\_FILE := main/AndroidManifest.xml
11. LOCAL\_MODULE := setup-wizard-lib-full-support
12. LOCAL\_RESOURCE\_DIR := \
13. $(LOCAL\_PATH)/main/res \
14. $(LOCAL\_PATH)/eclair-mr1/res \
15. $(LOCAL\_PATH)/full-support/res \
16. frameworks/support/v7/appcompat/res \
17. frameworks/support/v7/recyclerview/res
18. LOCAL\_SDK\_VERSION := current
19. LOCAL\_SRC\_FILES := $(call all-java-files-under, main/src eclair-mr1/src full-support/src)
20. LOCAL\_STATIC\_JAVA\_LIBRARIES := \
21. android-support-v4 \
22. android-support-v7-appcompat \
23. android-support-v7-recyclerview
25. include $(BUILD\_STATIC\_JAVA\_LIBRARY)

综上：setup-wizard-lib-full-support的AndroidManifest 是main/AndroidManifest.xml，源码目录为 main/src eclair-mr1/src full-support/src，资源文件为main/res /eclair-mr1/res full-support/res

，依赖jar为android-support-v4/android-support-v7-appcompat/android-support-v7-recyclerview

#### 小结

common-full-support依赖setup-wizard-lib-full-support

### 依赖关系图

## 导入对应的jar和源码

### **导入SettingsLib源码**

源码位置：frameworks/base/packages/SettingsLib/

然后设置jdk1.7

根据import报错,

1.需要framework.jar,通过UserLibraries方式导入framework.jar(同时需要配置优先使用这个库，

framework.jar的路径:

/cm13.0/repo/out/target/common/obj/JAVA\_LIBRARIES/

(2).根据如下报错找到对应源码,然后在/cm13.0/repo/,修改名字即可

报错代码路径: /cm13.0/repo/libcore/luni/src/main/java/libcore/icu

Gradle配置

**[javascript]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79765186) [copy](https://blog.csdn.net/qq1028850792/article/details/79765186)

1. provided files('../framework\_libs/core-libart-classes.jar')

### 导入setupwizard/navigationbar源码

路径: /cm13.0/repo/frameworks/opt/setupwizard/navigationbar

报错解决;设置成Is Library

### 导入setupwizard/library源码

路径: /cm13.0/repo/frameworks/opt/setupwizard/library

缺少定义在library/eclair-mr1/res/values/styles.xml和drawable中的的资源文件,将styles.xml中的资源复制到setupwizardlib的styles.xml中,drawable资源复制到setupwizardlib工程中

其他资源依然是

gesture\_fingerprint\_swipe是空文件，重名为.png

gesture\_ambient\_lift

https://android.googlesource.com/platform/packages/apps/Settings/+/12e4beb%5E%21/

### **Settins同上导入framework.jar**

#### **导入静态库**

out/target/common/obj/JAVA\_LIBRARIES

1. LOCAL\_STATIC\_JAVA\_LIBRARIES := \
2. android-support-v4 \
3. android-support-v13 \
4. android-support-v7-recyclerview \
5. android-support-v7-preference \
6. android-support-v7-appcompat \
7. android-support-v14-preference \
8. jsr305 \

settings-logtags

#### **导入动态库**

**[javascript]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79765186) [copy](https://blog.csdn.net/qq1028850792/article/details/79765186)

1. LOCAL\_JAVA\_LIBRARIES := bouncycastle core-oj telephony-common ims-common

#### 错误: 找不到符号符号: 变量 BEARER\_BITMASK位置: 类 Carriers

#### options.compilerArgs.add('-Xbootclasspath/p:framework\_libs/telephony-common.jar')

多个jar包就用分号隔开。

https://blog.csdn.net/BlueGoDisplay/article/details/52400794

## 编译

### Jdk7

### Execution failed for task ':app:transformClassesWithDesugarForDebug'

#### 解决办法

Open your build.gradle and use below

android {

compileSdkVersion 26

buildToolsVersion "26.0.1"

Then **Clean-Rebuild-Run** .

android {

compileSdkVersion 26

buildToolsVersion "26.0.1"

[https://stackoverflow.com/questions/45833937/execution-failed-for-task-apptransformclasseswithdesugarfordebug-error-on-andro?utm\_medium=organic&utm\_source=google\_rich\_qa&utm\_campaign=google\_rich\_qa](javascript:%20void%200)

### DexArchiveMergerException: r8.errors.CompilationError: Program type already present:android.Manifest

clean your project

then rebuild

<https://stackoverflow.com/questions/47079710/dexarchivemergerexception-with-android-studio-3-0>

provided files(**'../framework\_libs/classes-jarjar.jar'**)

**classes-jarjar.jar 和supprot-v14重合了？**

C:\Users\key.guan\Desktop>adb install -r -d F:\K\1-src\3399\test\dpad-settings\build\outputs\apk\debug\dpad-settings-debug.apk

adb: failed to install F:\K\1-src\3399\test\dpad-settings\build\outputs\apk\debug\dpad-settings-debug.apk: Failure [INSTALL\_PARSE\_FAILED\_MANIFEST\_MALFORMED: Fai

led parse during installPackageLI: /data/app/vmdl360143393.tmp/base.apk (at Binary XML file line #153): <meta-data> requires an android:value or android:resource attribute]

Failure [INSTALL\_PARSE\_FAILED\_MANIFEST\_MALFORMED: Failed parse during installPackageLI: /data/app/vmdl566395868.tmp/base.apk (at Binary XML file line #3421): <meta-data> requires an android:value or android:resource attribute]

INSTALL\_PARSE\_FAILED\_MANIFEST\_MALFORMED meta-data android:resource

|  |  |  |
| --- | --- | --- |
|  | 3437 |  |
|  |  |  |
| 1000 | 3434 |  |
| 1500 | 3432 |  |
| 2000 | 3433 |  |
| 2500 | 3431 |  |
| 2700 | 3434 |  |
| 2800 | 3432 |  |
|  |  |  |
| 2850 | 3437 |  |
| 2900 | 3437 |  |
| 2918 | 3437 |  |
| 3005 | 3437 |  |
|  |  |  |

### <item name="@\*android: not found

<https://www.niwoxuexi.com/blog/android/article/310.html>

因为已经不支持@开头使用android自带的属性，我们只要把@符号删掉就可以了

<https://stackoverflow.com/questions/47033594/android-studio-3-0-error-style-attribute-androidattr-windowenteranimation-n>

'@\*android:attr/regularColor' not found 同理

### <item name="@android:preferenceStyle

<**item name="@android:preferenceStyle"**>@style/Preference</**item**>  
<**item name="@android:editTextPreferenceStyle"**>@style/EditTextPreference</**item**>  
<**item name="@android:dropdownPreferenceStyle"**>@style/Preference.DropDown.Material</**item**>  
<**item name="@android:preferenceFragmentStyle"**>@style/PreferenceFragmentStyle</**item**>

### 错误: 找不到符号符号: 变量 EXTRA\_KEY\_ALERT\_TITLE位置: 接口 Phone

### EventLogTags找不到

#### 问题分析

在指定子目录下查找所有logtags文件

Find all of the logtags files under the named directories.

Meant to be used like:

SRC\_FILES := $(call all-logtags-files-under,src)

查找.logtag文件

$1:指定子目录名称

src/com/android/settings/EventLogTags.logtags

#### 问题解决

暂时用一个空壳吧

**compileDebugJavaWithJavac**

## 安装

### ignatures do not match

Failure [INSTALL\_FAILED\_UPDATE\_INCOMPATIBLE: Package com.android.settings signatures do not match the previously installed version; ignoring!]

Adb Failure [INSTALL\_FAILED\_TEST\_ONLY

com.android.settings signatures

没有系统签名的原因！

### base.apk code is missing

Failure [INSTALL\_FAILED\_INVALID\_APK: Package couldn't be installed in /data/app/com.android.settings-1: Package /data/app/com.android.settings-1/base.apk code is missing]

$ adb shell pm uninstall com.android.settings

DELETE\_FAILED\_INTERNAL\_ERROR

Error while Installing APK

## 运行

### DrawerLayout must be measured with MeasureSpec.EXACTLY

[drawerlayout必须给一个确定的大小不能用wrap\_content](https://blog.csdn.net/mikogodzd/article/details/50164669)

触发方法： 系统设置->关于手机

#### 问题分析

发生在DeviceInfoSettingsActivity，其他Activity不会发生该问题原因在于风格**，但是为何mm方式下不会崩溃？**

**android:layout\_width="match\_parent"  
android:layout\_height="match\_parent"**

1. 设死长宽，修改为1920\*1080可以规避，不会报错，但是异常
2. DeviceInfoSettingsActivity成弹窗风格，**android:theme="@style/Theme.SubSettingsDialogWhenLarge"**

在使用ToolBar+DrawerLayout的时候，总是报如下的错误：

**android:layout\_width="match\_parent"  
android:layout\_height="match\_parent"**

**https://blog.csdn.net/qq\_38859786/article/details/72852950**

#### 问题解决

直接删除修改风格

，但是此时

### SeekBar$OnSeekBarChangeListener)' on a null object reference

为何原生的没有？NPE

### 有两个进程com.android.phone

本来就用的

https://blog.csdn.net/myfriend0/article/details/78591098

### SecurityException: Package com.android.settings is currently frozen!

adb shell am broadcast -a android.intent.action.MASTER\_CLEAR

显示异常。

### Must specify preferenceTheme in theme

### @android:attr/preferenceFragmentStyle' not found

getActivity().getTheme().resolveAttribute(  
 android.support.v7.preference.R.attr.***preferenceTheme***, tv, **true**);

## 新增代码

### kpad定制架构

dpad-core

通信协议

## 待优化

### 目前远程编译很慢

方案1：分析as慢的原因，从源头解决问题

方案2：Git可以拉取到本地D盘么

配置ssh

然后push git review 框架

方案3：制作一个win脚本。每次拷贝文件pull/push脚本吧，

方案4：登陆远程unbunttu界面，安装android stadio？

保留:m目前的样子 编译mk\_settings 脚本

### [Android Studio under Network mound -](https://stackoverflow.com/questions/30151183/android-studio-under-network-mound)

https://stackoverflow.com/questions/30151183/android-studio-under-network-mound

## REF

[使用Android Studio开发系统级应用](https://www.jianshu.com/p/3a4b5f1c09c1)

[**Android.mk文件学习笔记**](https://blog.csdn.net/lzh398651363/article/details/73527245)

[**Android系统源码Settings导入eclipse**](https://blog.csdn.net/qq_25804863/article/details/48669477)

**[Android Studio 导入系统 jar包](https://www.cnblogs.com/bluestorm/p/6744140.html)**

## TASK

[Android AAPT详解](https://www.jianshu.com/p/8d691b6bf8b4)

<https://www.jianshu.com/p/85c8ce13fcad?utm_campaign=maleskine&utm_content=note&utm_medium=seo_notes&utm_source=recommendation>

android6.0 ：<https://gitee.com/lkuan/HelloAndroid/tree/master/AOSPLib/Android-6.0.0_r1/JAVA_LIBRARIES>

# 系统应用规范

SDK:专门团队在整，我们只是做了个精简版的freamework

系统应用 android studio

系统设置 android studio：无

## 使用android系统自带资源

### 系统定义的id

这里我们定义了一个ListView，定义它的id是"@+id/mylist"。实际上，如果没有特别的需求，就可以利用系统定义的id，类似下面的样子。

|  |
| --- |
| <ListView      android:id="@android:id/list"      android:layout\_width="fill\_parent"      android:layout\_height="fill\_parent"/> |

在xml文件中引用系统的id，只需要加上“@android:”前缀即可。如果是在Java代码中使用系统资源，和使用自己的资源基本上是一样的。不同的是，需要使用android.R类来使用系统的资源，而不是使用应用程序指定的R类。这里如果要获取ListView可以使用android.R.id.list来获取。

android.R.id

resource android:id/seekbar is private.

则定义为：<**SeekBar android:id="@\*android:id/seekbar"**

com.android.internal.R.id.***seekbar***

### 系统的图片资源

Hg

假设我们在应用程序中定义了一个menu，xml文件如下。

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <menu xmlns:android="http://schemas.android.com/apk/res/android">      <item          android:id="@+id/menu\_attachment"          android:title="附件"          android:icon="@android:drawable/ic\_menu\_attachment" />  </menu> |

其中代码片段android:icon="@android:drawable/ic\_menu\_attachment"本来是想引用系统中已有的Menu里的“附件”的图标。但是在Build工程以后，发现出现了错误。提示信息如下：

|  |
| --- |
| error: Error: Resource is not public. (at 'icon' with value '@android:drawable/ic\_menu\_attachment'). |

从错误的提示信息大概可以看出，由于该资源没有被公开，所以无法在我们的应用中直接引用。既然这样的话，我们就可以在Android SDK中找到相应的图片资源，直接拷贝到我们的工程目录中，然后使用类似android:icon="@drawable/ic\_menu\_attachment"的代码片段进行引用。

这样做的好处，一个是美工不需要重复的做一份已有的图片了，可以节约不少工时；另一个是能保证我们的应用程序的风格与系统一致。

|  |
| --- |
| 经验分享：  Android中没有公开的资源，在xml中直接引用会报错。除了去找到对应资源并拷贝到我们自己的应用目录下使用以外，我们还可以将引用“@android”改成“@\*android”解决。比如上面引用的附件图标，可以修改成下面的代码。  android:icon="@\*android:drawable/ic\_menu\_attachment"  修改后，再次Build工程，就不会报错了。 |

### 系统的字符串资源

Tg

 假设我们要实现一个Dialog，Dialog上面有“确定”和“取消”按钮。就可以使用下面的代码直接使用Android系统自带的字符串。

|  |
| --- |
| <LinearLayout          android:orientation="horizontal"          android:layout\_width="fill\_parent"          android:layout\_height="wrap\_content">          <Button              android:id="@+id/yes"              android:layout\_width="fill\_parent"              android:layout\_height="wrap\_content"              android:layout\_weight="1.0"              android:text="@android:string/yes"/>          <Button              android:id="@+id/no"              android:layout\_width="fill\_parent"              android:layout\_height="wrap\_content"              android:layout\_weight="1.0"              android:text="@android:string/no"/>      </LinearLayout> |

如果使用系统的字符串，默认就已经支持多语言环境了。如上述代码，直接使用了@android:string/yes和@android:string/no，在简体中文环境下会显示“确定”和“取消”，在英文环境下会显示“OK”和“Cancel”。

### 系统的Style

Dfs

假设布局文件中有一个TextView，用来显示窗口的标题，使用中等大小字体。可以使用下面的代码片段来定义TextView的Style。

|  |
| --- |
| <TextView          android:id="@+id/title"          android:layout\_width="wrap\_content"          android:layout\_height="wrap\_content"          android:textAppearance="?android:attr/textAppearanceMedium" /> |

其中android:textAppearance="?android:attr/textAppearanceMed ium"就是使用系统的style。需要注意的是，使用系统的style，需要在想要使用的资源前面加“?android:”作为前缀，而不是“@android:”。

### 利用系统的颜色定义

Sdfa

除了上述的各种系统资源以外，还可以使用系统定义好的颜色。在项目中最常用的，就是透明色的使用。代码片段如下。

|  |
| --- |
| android:background ="@android:color/transparent" |

|  |
| --- |
| 经验分享：  Android系统本身有很多资源在应用中都可以直接使用，具体的，可以进入android-sdk的相应文件夹中去查看。例如：可以进入$android-sdk$\platforms\android-8\data\res，里面的系统资源就一览无余了。  开发者需要花一些时间去熟悉这些资源，特别是图片资源和各种Style资源，这样在开发过程中，能够想到有相关资源并且直接拿来使用。 |

# [Android 初始化设置向导Provision](http://www.cnblogs.com/mythou/p/3425570.html)

源码路径：packages/apps/ Provision

## 概述

Provision是一个系统初始化引导程序，类似于我们的设置向导，原生的Android里面Provision只做了一件事，就是写入一个 DEVICE\_PROVISIONED标记。不过这个标记作用很大，这个标记只会在系统全新升级（双清）的时候写入**一次**，**代表了Android系统升级准备完成**，指示其他Framework和其他程序，机器已经Provisioned。没有Provisioned的机器，有些功能的表现是不一样的，比如，锁屏程序不会锁屏；对HOME key的处理也不同；电话也是打不进来的等。

Settings.Global.DEVICE\_PROVISIONED设置的变化。这一设置表示此设备是否已经归属于某一个用户。比如当用户打开一个新购买的设备时，初始化设置向导将会引导用户阅读使用条款、设置帐户等一系列的初始化操作。在初始化设置向导完成之前， Settings.Global.DEVICE\_PROVISIONED的值为false，表示这台设备并未归属于某一个用户。当设备并未归属于某以用户时，状态栏会禁用一些功能以避免信息的泄露。

## 源码

### Manifest

|  |
| --- |
| <**application**>  <**activity android:name="DefaultActivity"  android:excludeFromRecents="true"**>  <**intent-filter android:priority="1"**>  <**action android:name="android.intent.action.MAIN"** />  <**category android:name="android.intent.category.HOME"** />  <**category android:name="android.intent.category.DEFAULT"** />  </**intent-filter**>  </**activity**> </**application**> |

从上面代码可以看到两个比较有意思的地方，

第一：Provision配置了category.HOME属性，这是桌面程序的标记，简单点说就是Launcher程序才会配置的标记。

第二：priority=1，配置了优先级，也就是说它的优先级比我们原生的Launcher优先级还要高，它会在Launcher启动前就运行起来。一般android广播分为两种一种是Normal Broadcast（普通广播）：Noraml Broadcast是完全异步的，可以在同一时刻(逻辑上)被所有接收者接收到，消息传递的效率比较高。但缺点是接收者不能将处理结果传递给下一个接收者，并且无法终止Broadcast Intent的传播。一种是Ordered Broadcast(有序广播):Ordered Broadcast的接收者将按预先声明的优先级依次接收Broadcast。如A的级别高于B,B的级别高于C,那么Broadcast先传给A,再传 给B,最后传给C。优先级别声明在元素的android:priority属性中，数越大优先级别越高，取值范围为-1000~1000。

### DefaultActivity

|  |
| --- |
| **public class** DefaultActivity **extends** Activity {  @Override  **protected void** onCreate(Bundle icicle)  {  **super**.onCreate(icicle);   *// Add a persistent setting to allow other apps to know the device has been provisioned.* Settings.Global.*putInt*(getContentResolver(), Settings.Global.***DEVICE\_PROVISIONED***, 1); Settings.Secure.*putInt*(getContentResolver(), Settings.Secure.USER\_SETUP\_COMPLETE, 1);  *// remove this activity from the package manager.* PackageManager pm = getPackageManager();  ComponentName name = **new** ComponentName(**this**, DefaultActivity.**class**);  pm.setComponentEnabledSetting(name, PackageManager.***COMPONENT\_ENABLED\_STATE\_DISABLED***,  PackageManager.***DONT\_KILL\_APP***);   *// terminate the activity.* finish();  } } |

上面就是Provision的全部源码，Provision只有一个Activity，只做了这两件事：

* 设置DEVICE\_PROVISIONED标记
* 禁止Provision自己的Activity组件

第一，DEVICE\_PROVISIONED就是我们上面说的系统升级完标记。这个标记是保存在：/data/data/com.android.providers.settings/database/settings.db数据库中，准确来说是settings.db的secure表里面。

第二，禁止组件，所以这个组件只会运行一次，所以我们如果没有格式化/data目录，这个组件就会被禁止。所以会导致一个问题，就是我们的DEVICE\_PROVISIONED标记不会再次被写入，如果我们第一次DEVICE\_PROVISIONED标记没有被写成功，这样可能导致异常的出现。所以，可以在Provision中禁止组件前加入判断语句确保标记写入成功。

settings get global device\_provisioned

settings get global user\_setup\_complete

device\_proviisioned具体控制哪些功能？

user\_setup\_complete具体控制哪些功能

0,造成recent和home虚拟按键异常；.user\_setup\_complete

恢复出厂设置，是哪个时间段的数据？从user\_setup\_complete为1开始？

Settings put global

settings put global device\_provisioned 0

settings put global user\_setup\_complete 0

adb shell settings put global device\_provisioned 1

adb shell settings put global user\_setup\_complete 1

## 定制

通过Provision，可以定制：

1、加入一些初始设置项的设定，比如时区/时间初始设定，背景数据是否允许，是否允许安装非Android市场上的程序，等不需要用户干预就可以完成的设置。

2、加入UI设计，引导用户一步步完成需要用户参与选择或输入的设置过程，也就是设置向导的工作。

# Laucher

android7.x Launcher3

[workspace和allapps加载流程](http://blog.csdn.net/picasso_l/article/details/70140694)

## ActivityManagerService是如何启动Launcher



在SystemServer中启动PackageManagerService和ActivityManagerService后，PackageManagerService主要负责安装系统的应用程序，并把保存应用程序信息在这里就不多阐述，本文主要讲解。

ActivityManagerService.systemReady()

ActivityStack.resumeTopActivityLocked()，

ActivityManagerService .startHomeActivityLocked()。

而在**startHomeActivityLocked()**中我们首先是读取查询intent中Category类型为HOME的Activity，这个信息是保存在PackageManagerService，我们可以通过方法intent.resolveActivityInfo(mContext.getPackageManager(),STOCK\_PM\_FLAGS)

而intent中ategory类型为HOME，在新版本的android中已经不在用这个方法了，而是用

调用ActivityManagerService中的resolveActivityInfo(),实际上是在resolveActivityInfo()调用了resolveIntent()调用了chooseBestActivity()，由于intent中Category类型为HOME的Activity可能有多个，若优先级(android:priority)相同系统会弹出提示框让用户选择，优先级(android:priority)不同，则会选择启动优先级高的Activity，



## 拖拽分析

拖拽是用户在操作桌面经常用到的操作也是客制化比较多的其中之一。拖拽可以分为以下几类：

1、 主屏幕上的ICON和Wiget

2、 文件夹中的图标

3、 抽屉中的ICON和Wiget

这三种情形的处理流程是相似的，所以我们只对在主屏幕的ICON和Wiget拖拽进行分析。

主要用到的类有：

Launcher.java

Workspace.java

DragController.java

### 架构流程分析

主要的流程可以分为三大步：

1、 点击开始拖拽；

2、 拖拽过程中；

3、 拖到目标位置完成拖拽；

Launcher.java

Workspace.java

DragController.java

拖拽处理都是通过长按开始的，Workspace的长按是在Launcher进行处理

### 流程分析以及流程图

#### 长按开始拖拽

https://blog.csdn.net/YinYueRenZhangWenDa/article/details/52398632

# 安装与卸载应用程序PackageInstaller

源码目录/package/apps/PackageInstaller，其主要的功能就是实现应用的安装和卸载功能。

## **mainfest.xml**

|  |
| --- |
| *<?***xml version="1.0" encoding="utf-8"***?>* <**manifest xmlns:android="http://schemas.android.com/apk/res/android"  package="com.android.packageinstaller"**>   <**original-package android:name="com.android.packageinstaller"** />   <**uses-permission android:name="android.permission.INSTALL\_PACKAGES"** />  <**uses-permission android:name="android.permission.DELETE\_PACKAGES"** />  <**uses-permission android:name="android.permission.CLEAR\_APP\_CACHE"** />  <**uses-permission android:name="android.permission.READ\_PHONE\_STATE"** />  <**uses-permission android:name="android.permission.CLEAR\_APP\_USER\_DATA"** />  <**uses-permission android:name="android.permission.READ\_EXTERNAL\_STORAGE"** />  <**uses-permission android:name="android.permission.MANAGE\_USERS"** />  <**uses-permission android:name="android.permission.GRANT\_REVOKE\_PERMISSIONS"** />  <**uses-permission android:name="android.permission.INTERACT\_ACROSS\_USERS\_FULL"** />  <**uses-permission android:name="android.permission.READ\_INSTALL\_SESSIONS"** />   <**application android:label="@string/app\_name"  android:allowBackup="false"  android:theme="@style/Theme.DialogWhenLarge"  android:supportsRtl="true"**>   <**activity android:name=".PackageInstallerActivity"  android:configChanges="orientation|keyboardHidden|screenSize"  android:excludeFromRecents="true"**>  <**intent-filter**>  <**action android:name="android.intent.action.VIEW"** />  <**action android:name="android.intent.action.INSTALL\_PACKAGE"** />  <**category android:name="android.intent.category.DEFAULT"** />  <**data android:scheme="file"** />  <**data android:mimeType="application/vnd.android.package-archive"** />  </**intent-filter**>  <**intent-filter**>  <**action android:name="android.intent.action.INSTALL\_PACKAGE"** />  <**category android:name="android.intent.category.DEFAULT"** />  <**data android:scheme="file"** />  <**data android:scheme="package"** />  </**intent-filter**>  <**intent-filter**>  <**action android:name="android.content.pm.action.CONFIRM\_PERMISSIONS"** />  <**category android:name="android.intent.category.DEFAULT"** />  </**intent-filter**>  </**activity**>   <**activity android:name=".InstallAppProgress"  android:configChanges="orientation|keyboardHidden|screenSize"  android:exported="false"** />   <**activity android:name=".UninstallerActivity"  android:configChanges="orientation|keyboardHidden|screenSize"  android:excludeFromRecents="true"  android:theme="@style/Theme.AlertDialogActivity"**>  <**intent-filter**>  <**action android:name="android.intent.action.DELETE"** />  <**action android:name="android.intent.action.UNINSTALL\_PACKAGE"** />  <**category android:name="android.intent.category.DEFAULT"** />  <**data android:scheme="package"** />  </**intent-filter**>  </**activity**>   <**activity android:name=".UninstallAppProgress"  android:configChanges="orientation|keyboardHidden|screenSize"  android:exported="false"** />   <**activity android:name=".GrantActivity"  android:configChanges="orientation|keyboardHidden|screenSize"  android:excludeFromRecents="true"  android:theme="@android:style/Theme.DeviceDefault.Dialog.NoActionBar"**>  <**intent-filter**>  <**action android:name="android.content.pm.action.REQUEST\_PERMISSION"** />  <**category android:name="android.intent.category.DEFAULT"** />  </**intent-filter**>  </**activity**>  </**application**> </**manifest**> |

从mainfest.xml文件，可以总结一些信息：

1）共定义了两对四个窗口。其中PackageInstallerActivity和InstallAppProgress用于安装应用程序;UninstallerActivity和UninstallAppProgress用于卸载应用程序.

2）intent-filter包含android.intent.action.MAIN的Activity Action的Activity会在系统程序列表中列出相应的应用图标。PackageInstaller中的Activity并**没有注册MAIN**的Action，图标**不会列在应用程序列表中**

3）PackageInstallerActivity采用隐示调用，包含了两个Intent Filter，也就是支持两种方式开启该PackageInstallerActivity；

|  |
| --- |
| Intent intent = new Intent(Intent.ACTION\_INSTALL\_PACKAGE);  intent.setDataAndType(Uri.fromFile(new File("/sdcard/qq.apk")),"application/vnd.android.package-archive");  startActivity(intent); |
| Intent intent = new Intent(Intent.ACTION\_INSTALL\_PACKAGE);  intent.setData(Uri.fromFile(new File("/sdcard/qq.apk")));  startActivity(intent); |

4）类似的，UninstallerActivity也有两种意图去启动

|  |
| --- |
| Intent intent = new Intent(Intent.ACTION\_DELETE);  intent.setData(Uri.parse("package:com.tencent.mobileqq"));  startActivity(intent); |
| Intent intent = new Intent(Intent.ACTION\_UNINSTALL\_PACKAGE);  intent.setData(Uri.parse("package:com.tencent.mobileqq"));  startActivity(intent); |

## 安装Android应用前的校验

上面我们知道安装应用时会调用PackageInstallerActivity弹出安装界面，在安装时，我们可以留意到，还会将要安装的应用所用到的权限以列表的形式进行展示，应用名称，应用图标等信息也会一一进行展示，所以我们要看看PckageInstallerActivity中是如何获取到这些信息的。在PackageInstallerActivity中主要完成的工作如下：

1).从Intent中获取Package URI，Scheme等信息。   
2).对从Intent对象获取的信息进行校验。主要校验Scheme   
3).根据Scheme的具体值（file或者package）进行相应的处理   
4).获取ApplicationInfo对象，该对象包含与Android应用相关的信息，如应用名称，应用图标，应用权限等   
5).初始化用于显示名称和应用图标的控件   
6).校验当前Android系统是否允许“未知来源”的应用被安装   
7).进行安装前的准备工作，显示校验窗口

### PackageInstallerActivity.onCreate

主要完成的功能：   
1. 从Intent对象获取Package URL、Scheme信息   
2. 校验Scheme，根据Scheme不同的值进行逻辑处理。Scheme为file或者package   
3. 获取ApplicationInfo对象，用于获取应用名称，报名，应用图标等信息

|  |
| --- |
| @Override **protected void** onCreate(Bundle icicle) {  **super**.onCreate(icicle);   **mPm** = getPackageManager();  **mInstaller** = **mPm**.getPackageInstaller();   **final** Intent intent = getIntent();  **if** (PackageInstaller.ACTION\_CONFIRM\_PERMISSIONS.equals(intent.getAction())) {  **final int** sessionId = intent.getIntExtra(PackageInstaller.***EXTRA\_SESSION\_ID***, -1);  **final** PackageInstaller.SessionInfo info = **mInstaller**.getSessionInfo(sessionId);  **if** (info == **null** || !info.sealed || info.resolvedBaseCodePath == **null**) {  Log.*w*(***TAG***, **"Session "** + **mSessionId** + **" in funky state; ignoring"**);  finish();  **return**;  }  **mSessionId** = sessionId;  *//获取待安装Android应用的路径或Package* **mPackageURI** = Uri.*fromFile*(**new** File(info.resolvedBaseCodePath));  **mOriginatingURI** = **null**;  **mReferrerURI** = **null**;  } **else** {  **mSessionId** = -1;  **mPackageURI** = intent.getData();  **mOriginatingURI** = intent.getParcelableExtra(Intent.***EXTRA\_ORIGINATING\_URI***);  **mReferrerURI** = intent.getParcelableExtra(Intent.***EXTRA\_REFERRER***);  }   **boolean** requestFromUnknownSource = isInstallRequestFromUnknownSource(intent);  **mInstallFlowAnalytics** = **new** InstallFlowAnalytics();  **mInstallFlowAnalytics**.setContext(**this**);  **mInstallFlowAnalytics**.setStartTimestampMillis(SystemClock.*elapsedRealtime*());  **mInstallFlowAnalytics**.setInstallsFromUnknownSourcesPermitted(  isInstallingUnknownAppsAllowed());  **mInstallFlowAnalytics**.setInstallRequestFromUnknownSource(requestFromUnknownSource);  **mInstallFlowAnalytics**.setVerifyAppsEnabled(isVerifyAppsEnabled());  **mInstallFlowAnalytics**.setAppVerifierInstalled(isAppVerifierInstalled());  **mInstallFlowAnalytics**.setPackageUri(**mPackageURI**.toString());   **final** String scheme = **mPackageURI**.getScheme();  *//从此处可以看到，scheme只有两个值：file或package，为空呢* **if** (scheme != **null** && !**"file"**.equals(scheme) && !**"package"**.equals(scheme)) {  Log.*w*(***TAG***, **"Unsupported scheme "** + scheme);  setPmResult(PackageManager.INSTALL\_FAILED\_INVALID\_URI);  **mInstallFlowAnalytics**.setFlowFinished(  InstallFlowAnalytics.***RESULT\_FAILED\_UNSUPPORTED\_SCHEME***);  finish();  **return**;  }   **final** PackageUtil.AppSnippet as;  *//scheme是package时* **if** (**"package"**.equals(**mPackageURI**.getScheme())) {  **mInstallFlowAnalytics**.setFileUri(**false**);  **try** {  *//获取与package对应的Android应用的信息,包含应用名称,权限列表，应用图标等信息* **mPkgInfo** = **mPm**.getPackageInfo(**mPackageURI**.getSchemeSpecificPart(),  PackageManager.***GET\_PERMISSIONS*** | PackageManager.***GET\_UNINSTALLED\_PACKAGES***);  } **catch** (NameNotFoundException e) {  }  **if** (**mPkgInfo** == **null**) {  Log.*w*(***TAG***, **"Requested package "** + **mPackageURI**.getScheme()  + **" not available. Discontinuing installation"**);  showDialogInner(***DLG\_PACKAGE\_ERROR***);  setPmResult(PackageManager.INSTALL\_FAILED\_INVALID\_APK);  **mInstallFlowAnalytics**.setPackageInfoObtained();  **mInstallFlowAnalytics**.setFlowFinished(  InstallFlowAnalytics.***RESULT\_FAILED\_PACKAGE\_MISSING***);  **return**;  }  *//创建AppSnippet对象。该对象封装了用于待安装Android应用的标题和图标* as = **new** PackageUtil.AppSnippet(**mPm**.getApplicationLabel(**mPkgInfo**.**applicationInfo**),  **mPm**.getApplicationIcon(**mPkgInfo**.**applicationInfo**));  } **else** {  *//scheme为file的情况，或者scheme为空，及从apk文件安装程序* **mInstallFlowAnalytics**.setFileUri(**true**);  *//获取APK文件的绝对路径* **final** File sourceFile = **new** File(**mPackageURI**.getPath());  *//创建APK文件的分析器* PackageParser.Package parsed = PackageUtil.*getPackageInfo*(sourceFile);   *// Check for parse errors* **if** (parsed == **null**) {  Log.*w*(***TAG***, **"Parse error when parsing manifest. Discontinuing installation"**);  showDialogInner(***DLG\_PACKAGE\_ERROR***);  setPmResult(PackageManager.INSTALL\_FAILED\_INVALID\_APK);  **mInstallFlowAnalytics**.setPackageInfoObtained();  **mInstallFlowAnalytics**.setFlowFinished(  InstallFlowAnalytics.***RESULT\_FAILED\_TO\_GET\_PACKAGE\_INFO***);  **return**;  }  **mPkgInfo** = PackageParser.generatePackageInfo(parsed, **null**,  PackageManager.***GET\_PERMISSIONS***, 0, 0, **null**,  **new** PackageUserState());  **mPkgDigest** = parsed.manifestDigest;  as = PackageUtil.*getAppSnippet*(**this**, **mPkgInfo**.**applicationInfo**, sourceFile);  }  **mInstallFlowAnalytics**.setPackageInfoObtained();    *//set view* setContentView(R.layout.install\_start);  **mInstallConfirm** = findViewById(R.id.install\_confirm\_panel);  **mInstallConfirm**.setVisibility(View.***INVISIBLE***);  PackageUtil.*initSnippetForNewApp*(**this**, as, R.id.app\_snippet);   **mOriginatingUid** = getOriginatingUid(intent);   *// 未知源安装检测* **if** ((requestFromUnknownSource) && (!isInstallingUnknownAppsAllowed())) {  *//ask user to enable setting first* showDialogInner(***DLG\_UNKNOWN\_APPS***);  **mInstallFlowAnalytics**.setFlowFinished(  InstallFlowAnalytics.***RESULT\_BLOCKED\_BY\_UNKNOWN\_SOURCES\_SETTING***);  **return**;  }  *//为安装应用做一些准备工作* initiateInstall(); } |

### startInstallConfirm

该方法的功能是显示一个确认对话框，该对话框会列出请求的权限列表；在startInstallConfirm方法中，涉及到了AppSecurityPermissions组件，另外涉及权限的还有三类：   
- AppSecurityPermissions.WHICH\_PERSONAL 涉及隐私的权限项   
- AppsecurityPermissions.WHICH\_DEVICE 涉及与设备相关的权限项   
- AppSecurityPermissions.WHICH\_NEW 涉及的新增的权限列表项 应用更新时如果有额外的新的权限增加会使用到该项。

|  |
| --- |
| **private void** startInstallConfirm() {  TabHost tabHost = (TabHost)findViewById(android.R.id.***tabhost***);  tabHost.setup();  ViewPager viewPager = (ViewPager)findViewById(R.id.pager);  TabsAdapter adapter = **new** TabsAdapter(**this**, tabHost, viewPager);  adapter.setOnTabChangedListener(**new** TabHost.OnTabChangeListener() {  @Override  **public void** onTabChanged(String tabId) {  **if** (***TAB\_ID\_ALL***.equals(tabId)) {  **mInstallFlowAnalytics**.setAllPermissionsDisplayed(**true**);  } **else if** (***TAB\_ID\_NEW***.equals(tabId)) {  **mInstallFlowAnalytics**.setNewPermissionsDisplayed(**true**);  }  }  });  *//该字段表示否时显示权限列表* **boolean** permVisible = **false**;  **mScrollView** = **null**;  **mOkCanInstall** = **false**;  **int** msg = 0;  **if** (**mPkgInfo** != **null**) {  *//AppSecurityPermissions 是一个组件 封装了一些列处理权限的功能* AppSecurityPermissions perms = **new** AppSecurityPermissions(**this**, **mPkgInfo**);  *//获取与隐私相关的权限数量* **final int** NP = perms.getPermissionCount(AppSecurityPermissions.WHICH\_PERSONAL);  *//获取与设备相关的权限数量* **final int** ND = perms.getPermissionCount(AppSecurityPermissions.WHICH\_DEVICE);  **if** (**mAppInfo** != **null**) {  msg = (**mAppInfo**.**flags** & ApplicationInfo.***FLAG\_SYSTEM***) != 0  ? R.string.install\_confirm\_question\_update\_system  : R.string.install\_confirm\_question\_update;  *//显示权限列表的ScrollView控件* **mScrollView** = **new** CaffeinatedScrollView(**this**);  **mScrollView**.setFillViewport(**true**);  *//当安装的应用已经存在时(更新应用时),获取是否有额外的权限请求 AppSecurityPermissions.WHICH\_NEW* **boolean** newPermissionsFound =  (perms.getPermissionCount(AppSecurityPermissions.WHICH\_NEW) > 0);  **mInstallFlowAnalytics**.setNewPermissionsFound(newPermissionsFound);  **if** (newPermissionsFound) {  permVisible = **true**;  **mScrollView**.addView(perms.getPermissionsView(  AppSecurityPermissions.WHICH\_NEW));  } **else** {  LayoutInflater inflater = (LayoutInflater)getSystemService(  Context.***LAYOUT\_INFLATER\_SERVICE***);  TextView label = (TextView)inflater.inflate(R.layout.label, **null**);  label.setText(R.string.no\_new\_perms);  **mScrollView**.addView(label);  }  adapter.addTab(tabHost.newTabSpec(***TAB\_ID\_NEW***).setIndicator(  getText(R.string.newPerms)), **mScrollView**);  } **else** {  findViewById(R.id.tabscontainer).setVisibility(View.***GONE***);  findViewById(R.id.divider).setVisibility(View.***VISIBLE***);  }  *//将要安装的应用设置的权限请求数量大于0时，将设置的权限列表列出来;* **if** (NP > 0 || ND > 0) {  permVisible = **true**;  LayoutInflater inflater = (LayoutInflater)getSystemService(  Context.***LAYOUT\_INFLATER\_SERVICE***);  View root = inflater.inflate(R.layout.permissions\_list, **null**);  **if** (**mScrollView** == **null**) {  **mScrollView** = (CaffeinatedScrollView)root.findViewById(R.id.scrollview);  }  *//向权限列表控件中添加 私有请求权限控件* **if** (NP > 0) {  ((ViewGroup)root.findViewById(R.id.privacylist)).addView(  perms.getPermissionsView(AppSecurityPermissions.WHICH\_PERSONAL));  } **else** {*//私有请求权限数量为0时，就将控件进行隐藏* root.findViewById(R.id.privacylist).setVisibility(View.***GONE***);  }  **if** (ND > 0) {  ((ViewGroup)root.findViewById(R.id.devicelist)).addView(  perms.getPermissionsView(AppSecurityPermissions.WHICH\_DEVICE));  } **else** {  root.findViewById(R.id.devicelist).setVisibility(View.***GONE***);  }  adapter.addTab(tabHost.newTabSpec(***TAB\_ID\_ALL***).setIndicator(  getText(R.string.allPerms)), root);  }  }  **mInstallFlowAnalytics**.setPermissionsDisplayed(permVisible);  **if** (!permVisible) {  **if** (**mAppInfo** != **null**) {  *// This is an update to an application, but there are no  // permissions at all.* msg = (**mAppInfo**.**flags** & ApplicationInfo.***FLAG\_SYSTEM***) != 0  ? R.string.install\_confirm\_question\_update\_system\_no\_perms  : R.string.install\_confirm\_question\_update\_no\_perms;  } **else** {  *// This is a new application with no permissions.* msg = R.string.install\_confirm\_question\_no\_perms;  }  tabHost.setVisibility(View.***GONE***);  **mInstallFlowAnalytics**.setAllPermissionsDisplayed(**false**);  **mInstallFlowAnalytics**.setNewPermissionsDisplayed(**false**);  findViewById(R.id.filler).setVisibility(View.***VISIBLE***);  findViewById(R.id.divider).setVisibility(View.***GONE***);  **mScrollView** = **null**;  }  **if** (msg != 0) {  ((TextView)findViewById(R.id.install\_confirm\_question)).setText(msg);  }  **mInstallConfirm**.setVisibility(View.***VISIBLE***);  **mOk** = (Button)findViewById(R.id.ok\_button);  **mCancel** = (Button)findViewById(R.id.cancel\_button);  **mOk**.setOnClickListener(**this**);  **mCancel**.setOnClickListener(**this**);  **if** (**mScrollView** == **null**) {  *// There is nothing to scroll view, so the ok button is immediately  // set to install.* **mOk**.setText(R.string.install);  **mOkCanInstall** = **true**;  } **else** {  *//如果设置了权限列表，则当滚动到权限列表末尾时，显示“install”按钮,否则显示“next"下一步* **mScrollView**.setFullScrollAction(**new** Runnable() {  @Override  **public void** run() {  **mOk**.setText(R.string.install);  **mOkCanInstall** = **true**;  }  });  } } |

## 安装应用程序

尽管安装Android应用前的校验工作非常复杂，但这些工作并不是PackageInstaller的主要工作。PackageInstaller的主要任务是安装android应用，而且是静默安装。如果不显示用于校验的窗口，PackageInstaller的安装过程可以让用户完全感觉不到，并且可以是后台异步进行。要想实现静默安装Android应用程序，首先我们应该了解PackageInstaller的安装原理。

1.PackageInstaller的安装原理：当我们浏览了权限列表后，点击安装，会出现一个进度条，提示我们正在安装。如果成功安装了应用，会显示应用已安装的提示信息。我们到PackageInstallerActivity的源码中查看源码是如何实现的,从onclick事件开始分析：

|  |
| --- |
| *//单击“next/ok“按钮浏览权限列表* **if**(v == **mOk**) {  *//已经浏览完所有权限 “next"按钮已经变成了“ok”按钮，已经准备好可以显示安装口* **if** (**mOkCanInstall** || **mScrollView** == **null**) {  **mInstallFlowAnalytics**.setInstallButtonClicked();  **if** (**mSessionId** != -1) {  **mInstaller**.setPermissionsResult(**mSessionId**, **true**);   *// We're only confirming permissions, so we don't really know how the  // story ends; assume success.* **mInstallFlowAnalytics**.setFlowFinishedWithPackageManagerResult(  PackageManager.INSTALL\_SUCCEEDED);  } **else** {  *// Start subactivity to actually install the application  //构造开启安装程序的Intent对象* Intent newIntent = **new** Intent();  newIntent.putExtra(PackageUtil.***INTENT\_ATTR\_APPLICATION\_INFO***,  **mPkgInfo**.**applicationInfo**);  newIntent.setData(**mPackageURI**);  *//将要显示的Activity类界面* newIntent.setClass(**this**, InstallAppProgress.**class**);  newIntent.putExtra(InstallAppProgress.***EXTRA\_MANIFEST\_DIGEST***, **mPkgDigest**);  newIntent.putExtra(  InstallAppProgress.***EXTRA\_INSTALL\_FLOW\_ANALYTICS***, **mInstallFlowAnalytics**);  String installerPackageName = getIntent().getStringExtra(  Intent.***EXTRA\_INSTALLER\_PACKAGE\_NAME***);  **if** (**mOriginatingURI** != **null**) {  newIntent.putExtra(Intent.***EXTRA\_ORIGINATING\_URI***, **mOriginatingURI**);  }  **if** (**mReferrerURI** != **null**) {  newIntent.putExtra(Intent.***EXTRA\_REFERRER***, **mReferrerURI**);  }  **if** (**mOriginatingUid** != VerificationParams.NO\_UID) {  newIntent.putExtra(Intent.EXTRA\_ORIGINATING\_UID, **mOriginatingUid**);  }  **if** (installerPackageName != **null**) {  newIntent.putExtra(Intent.***EXTRA\_INSTALLER\_PACKAGE\_NAME***,  installerPackageName);  }  **if** (getIntent().getBooleanExtra(Intent.***EXTRA\_RETURN\_RESULT***, **false**)) {  newIntent.putExtra(Intent.***EXTRA\_RETURN\_RESULT***, **true**);  newIntent.addFlags(Intent.***FLAG\_ACTIVITY\_FORWARD\_RESULT***);  }  **if**(**localLOGV**) Log.*i*(***TAG***, **"downloaded app uri="**+**mPackageURI**);  startActivity(newIntent);  }  finish(); |

上述方法就是事件驱动型的代码，只是事件的分类逻辑处理并不困难，最重要的就当是构造Intent启动安装窗口对象。

### InstallAppProgress

onCreate方法中主要就是获取在PackageInstallerActivity的点击方法中构造的intent对象，然后获取传递的构造的ApplicationInfo对象，将要安装的应用包的URI，获取将要安装的应用包的Scheme的具体对应的值（file或者package），最后调用初始化视图操作initview。主要分析initView方法如下：

|  |
| --- |
| **public void** initView() {  setContentView(R.layout.op\_progress);  **int** installFlags = 0;  PackageManager pm = getPackageManager();  **try** {  PackageInfo pi = pm.getPackageInfo(**mAppInfo**.**packageName**,   PackageManager.***GET\_UNINSTALLED\_PACKAGES***);  **if**(pi != **null**) {  *//如果应用已安装, 则设置应用的安装模式为更新* installFlags |= PackageManager.INSTALL\_REPLACE\_EXISTING;  }  } **catch** (NameNotFoundException e) {  }  **if**((installFlags & PackageManager.INSTALL\_REPLACE\_EXISTING )!= 0) {  Log.*w*(**TAG**, **"Replacing package:"** + **mAppInfo**.**packageName**);  }   **final** PackageUtil.AppSnippet as;  *//如果scheme为package，则意味着是更新程序* **if** (**"package"**.equals(**mPackageURI**.getScheme())) {  as = **new** PackageUtil.AppSnippet(pm.getApplicationLabel(**mAppInfo**),  pm.getApplicationIcon(**mAppInfo**));  } **else** {  *//否则scheme为file，则通过APK文件的路径获取显示Android应用相关信息的视图* **final** File sourceFile = **new** File(**mPackageURI**.getPath());  as = PackageUtil.*getAppSnippet*(**this**, **mAppInfo**, sourceFile);  }  **mLabel** = as.**label**;  PackageUtil.*initSnippetForNewApp*(**this**, as, R.id.app\_snippet);  **mStatusTextView** = (TextView)findViewById(R.id.center\_text);  **mStatusTextView**.setText(R.string.installing);  **mExplanationTextView** = (TextView) findViewById(R.id.center\_explanation);  **mProgressBar** = (ProgressBar) findViewById(R.id.progress\_bar);  **mProgressBar**.setIndeterminate(**true**);  *// Hide button till progress is being displayed* **mOkPanel** = (View)findViewById(R.id.buttons\_panel);  **mDoneButton** = (Button)findViewById(R.id.done\_button);  **mLaunchButton** = (Button)findViewById(R.id.launch\_button);  **mOkPanel**.setVisibility(View.***INVISIBLE***);   *//获取安装应用包的包名* String installerPackageName = getIntent().getStringExtra(  Intent.***EXTRA\_INSTALLER\_PACKAGE\_NAME***);  Uri originatingURI = getIntent().getParcelableExtra(Intent.***EXTRA\_ORIGINATING\_URI***);  Uri referrer = getIntent().getParcelableExtra(Intent.***EXTRA\_REFERRER***);  *//如上的两个Uri，对于普通的Anroid应用来说为null。* **int** originatingUid = getIntent().getIntExtra(Intent.EXTRA\_ORIGINATING\_UID,  VerificationParams.NO\_UID);  ManifestDigest manifestDigest = getIntent().getParcelableExtra(***EXTRA\_MANIFEST\_DIGEST***);  VerificationParams verificationParams = **new** VerificationParams(**null**, originatingURI,  referrer, originatingUid, manifestDigest);  PackageInstallObserver observer = **new** PackageInstallObserver();   **if** (**"package"**.equals(**mPackageURI**.getScheme())) {  *//scheme为package时，调用更新方法更新程序* **try** {  pm.installExistingPackage(**mAppInfo**.**packageName**);  observer.packageInstalled(**mAppInfo**.**packageName**,  PackageManager.INSTALL\_SUCCEEDED);  } **catch** (PackageManager.NameNotFoundException e) {  observer.packageInstalled(**mAppInfo**.**packageName**,  PackageManager.INSTALL\_FAILED\_INVALID\_APK);  }  } **else** {  *//scheme为file时,调用安装方法* pm.installPackageWithVerificationAndEncryption(**mPackageURI**, observer, installFlags,  installerPackageName, verificationParams, **null**);  } } |

initView方法一如既往的还是获取将要安装程序的一些信息，然后进行做了一些操作，该方法最重要的操作和调用发生在最后，如果scheme为package，则调用pm.installExistingPackage()方法更新应用，反之如果scheme为file，则调用pm.installPackageWithVerificationAndEncryption()方法进行应用的安装。上面说到的两个方法都是PakcageManager中的方法，**两个方法都是静默安装**，在安装的时候不会出现任何的提示。但是由于上述两个方法在PackageManager中是被注释为@hide的，所以，普通的Android应用中无法调用和访问。静默安装是一个异步过程，所以，无论安装成功或者安装失败，都会向用户弹出结果，所以我们在调用方法中看到有个行参是observer，为PackageInstallObserver实例。PackageInstallObserver主要就是处理安装的结果，其中定义了Handler变量mHandler来进行UI操作，用于实现对用户提示。

总结：从技术上来说，**实现静默安装Android应用就是调用PackageManager.installPackageWithVerificationAndEncryption方法即可**。该方法的参数较多，最重要的是需要一个异步安装结果监听器用于处理安装结果。此监听器必须是**IPackageInstallObserver.Stub**的子类。但**是只有安装结果，并没有具体的进度。**

最后还要说明的是：PackageInstaller安装应用程序实现安装应用的功能是需要申请权限：android.permission.INSTALL\_PACKAGES.该权限属于**系统级别的权限**，在普通的应用中无法使用。

## 卸载Android应用

和安装应用时一样，在卸载应用时，PackageInstaller会首先弹出卸载确认提示框，用户点击确认后，再显示卸载进度提示框，最后显示卸载结果.

### showDialogFragment

卸载前的确认框是一个Dialog，在5.1.1源码中为showDialogFragment(new UninstallAlertDialogFragment());

在UninstallAlertFragment中获取要卸载的应用的名字，图标等信息，然后设置title，icon，PositiveButton和NegativeButton事件等。PositiveButton的点击事件会触发UnInstallerActivity中的startUninstallProgress()方法，如下：

### startUninstallProgress

|  |
| --- |
| void startUninstallProgress() {  //构造要卸载的应用的Intent对象  Intent newIntent = new Intent(Intent.*ACTION\_VIEW*);  newIntent.putExtra(Intent.*EXTRA\_USER*, mDialogInfo.user);  newIntent.putExtra(Intent.EXTRA\_UNINSTALL\_ALL\_USERS, mDialogInfo.allUsers);  newIntent.putExtra(PackageInstaller.EXTRA\_CALLBACK, mDialogInfo.callback);  newIntent.putExtra(PackageUtil.*INTENT\_ATTR\_APPLICATION\_INFO*, mDialogInfo.appInfo);  if (getIntent().getBooleanExtra(Intent.*EXTRA\_RETURN\_RESULT*, false)) {  newIntent.putExtra(Intent.*EXTRA\_RETURN\_RESULT*, true);  newIntent.addFlags(Intent.*FLAG\_ACTIVITY\_FORWARD\_RESULT*);  }  newIntent.setClass(this, UninstallAppProgress.class);  startActivity(newIntent); } |

### UninstallAppProgress

展示卸载进度界面,与InstallAppProgress类类似的，onCreate方法中获取了传递的Intent对象中的ApplicationInfo对象以及获取是否删除所有用户数据的标志，重要的卸载方法调用还是在initView方法中，核心调用如下：

|  |
| --- |
| IPackageManager packageManager =  IPackageManager.Stub.asInterface(ServiceManager.getService("package")); PackageDeleteObserver observer = new PackageDeleteObserver(); try {// 静默卸载Android应用   packageManager.deletePackageAsUser(mAppInfo.packageName, observer,  mUser.getIdentifier(),  mAllUsers ? PackageManager.DELETE\_ALL\_USERS : 0); } catch (RemoteException e) {  // Shouldn't happen.  Log.*e*(TAG, "Failed to talk to package manager", e); } |

同样的，deletePackageAsUser是PackageManager类中标注为@hide的方法，在普通的应用中无法访问和调用，需要是系统应用才能调用。另外，PackageDeleteObserver卸载监听器需要继承IPackageDeleteObserver.Stub，这个也是使用了Handler的变量mHandler来处理卸载结果，卸载成功或者卸载失败的UI操作由此mHandler变量来实现。当然和安装应用时一样，需要系统级别的权限申请：android.permission.DELETE\_PACKAGES.

[在android7静默卸载采用方法](https://stackoverflow.com/questions/32473158/packageinstaller-silent-install-and-uninstall-of-apps-by-device-owner-androi/32687988)：

String appPackage = "com.your.app.package";

Intent intent = new Intent(getActivity(), getActivity().getClass());

PendingIntent sender = PendingIntent.getActivity(getActivity(), 0, intent, 0);

PackageInstaller mPackageInstaller = getActivity().getPackageManager().getPackageInstaller();

mPackageInstaller.uninstall(appPackage, sender.getIntentSender());

## 小结

普通的Android应用程序不允许静默安装和静默卸载程序，如果想要实现静默安装或者静默卸载需要将Android应用变成系统应用（进行系统签名）

在技术实现上，静默安装调用的是PackageManager类中的installPackageWithVerificationAndEncryption方法，然后通过一个安装监听器来处理安装结果，Handler变量根据安装结果处理UI显示；静默卸载调用的是PackageManager类中的deletePackageAsUser方法，通过设置一个卸载监听器监听卸载结果，Handler变量根据卸载结果处理UI显示。

* + 静默安装和静默卸载的方法都是标注为@hide的方法，普通应用中无法访问和调用
  + 安装应用需要使用系统级别的android.permission.INSTALL\_PACAGES权限；
  + 卸载应用需要使用系统级别的android.permission.DELETE\_PACKAGES权限。

## REF

# SettingsProvider

Content Provider几乎已经成为了Android系统中两个或多个应用(系统应用和普通应用)之间共享数据库的最直接的方法。

在Android源码中，有两个目录是专门存放Content Provider。

一个是/源码目录/frameworks/providers，允许任何Adroid应用访问，包括：ApplicationProvider、CalendarProvider、ContactsProvider、DownloadProvider、MedialProvider、PartnerBookmarksProvider、TelephonyProvider、TvProvider、UserDictionaryProvider；

另一个是/源码目录/frameworks/base/packages，通常只允许系统级别的Android应用读写数据，**普通Android应用只能读数据**。包含：ExternalStorageProvider、SettingsProvider

具体到Setting Content Provider 是所有系统设置的管理者。在M(Android5.0)版本之前，SettingsProvider中系统设置是存储在settings.db[数据库](http://lib.csdn.net/base/mysql)中；但是在L(Android6.0)之后，SettingsProvider中系统设置改为由xml存储在data分区。

如Wifi状态，蓝牙是否开启，当前设备所处的模式等等，很多很多类似的全局属性控制的属性值。最重要的是global表，该表包含两个字段：name和value.其中name表示系统信息的key，value表示该系统设置项所对应的选项值。数据库结构如下：



## Android M之前的SettingsProvider

进入**/data/data/com.**[**android**](http://lib.csdn.net/base/android)**.providers.settings/databases**

# cd /data/data/com.android.providers.settings/databases

# sqlite3 settings.db

sqlite3 有可能提示找不到，这是由于在当前版本中sqlite3程序没有被编译到system中，这里我们只需要手动push一个sqlite3就好了。Cs果然没有

在使用sqlite3打开settings.db数据库之后，可以使用.help来查看有关sqlite3的一些帮助信息。这里常用的如下所示：



源码

**frameworks/base/packages/SettingsProvider/src/com/Android/providers/settings/DatabaseHelper.**[**Java**](http://lib.csdn.net/base/java)   
**frameworks/base/packages/SettingsProvider/res/values/defaults.xml**

**如何添加属性：**系统属性的设置，我们可以通过修改源码，编译生成SettingsProvider.apk并安装到系统中，就能生效。

### [自定义Android系统默认设置](http://www.cnblogs.com/l2rf/p/5102230.html)

无法在普通app上通过以下方式使用，即使申请了android.permission.WRITE\_SETTINGS等权限也不行。会报SecurityException。

Settings.Global.*putString*(getContentResolver(),  
 **"myKey"**,**"v1"**);  
String v1 = Settings.Global.*getString*(getContentResolver(),  
 **"myKey"**);

### 修改方法默认值方法1

**adb shell settings**,有关其详细用法，可以参照如下提示：

$ adb shell settings get system ringtone

content://media/internal/audio/media/59

$ adb shell settings set system ringtone "content://media/internal/audio/media/60"

Invalid command: set

usage: settings [--user NUM] get namespace key

settings [--user NUM] put namespace key value

settings [--user NUM] delete namespace key

'namespace' is one of {system, secure, global}, case-insensitive

If '--user NUM' is not given, the operations are performed on the owner user.

$ adb shell settings put system ringtone "content://media/internal/audio/media/60"

$ adb shell settings get system ringtone

content://media/internal/audio/media/60

### 修改方法默认值方法2,标准方法

如果要在系统中添加默认设置，需要如下步骤（以带物理键盘的系统中默认打开软键盘为例）：

1、在defaults.xml中为新加的数据定义一个默认值

<integer name="show\_ime\_with\_hard\_keyboard">1</integer>

2、在DatabaseHelper.java中的loadSecureSettings(SQLiteDatabase db)方法中加入新字段的添加代码

loadStringSetting(stmt, Settings.Secure.SHOW\_IME\_WITH\_HARD\_KEYBOARD,

R.string.show\_ime\_with\_hard\_keyboar);

3、重新编译

## Android L 的SettingsProvider

在Android L版本中，Google团队为了优化系统性能、提高安全性等等一些列需求，重构了SettingsProvider模块，在L版本中，不在使用数据库来存储系统设置，而是通过xml将系统设置存储在了**/data/system/user/** 目录下



有关L版本中SettingsProvider的修改更新方法同之前的相同

## 源码

### manifest

|  |
| --- |
| <manifest xmlns:android="http://schemas.android.com/apk/res/android"  package="com.android.providers.settings"  coreApp="true"  android:sharedUserId="android.uid.system">  <application android:allowClearUserData="false"  android:label="@string/app\_label"  android:process="system"  android:backupAgent="SettingsBackupAgent"  android:killAfterRestore="false"  android:icon="@mipmap/ic\_launcher\_settings">    <!-- todo add: android:neverEncrypt="true" -->  <provider android:name="SettingsProvider" android:authorities="settings"  android:multiprocess="false"  android:exported="true"  android:writePermission="android.permission.WRITE\_SETTINGS"  android:singleUser="true"  android:initOrder="100" />  </application>  </manifest> |

通过如上文件内容，我们大概可以得出几个结论：   
1. 拥有system用户权限。标签的coreApp属性是true,且android:sharedUserId=”android.uid.system”意味着SettingProvder与Settings同样拥有system用户的权限，且通过android:process=”system”可以看出，SettingProvider运行在**系统进程中**。   
2. provider标签中android:authorities=”settings”可以知道，访问SettingsProvider的Uri的开头部分一定是“content://settings”;

比如

CONTENT\_URI = Uri.parse("content://settings/global");

3. provider标签中android:writePermission=”android.permission.WRITE\_SETTINGS”可以知道，在执行写操作时需要指定WRITE\_SETTINGS权限。

### Src

在SettingsProvider的src源代码中，共有四个类：   
1. DatabaseHelper类。数据库操作类，实现对setting.db的读写功能，继承自SQLiteOpenHelper类   
2. SettingsBackupAgent类。用于备份和恢复系统设置等功能的类，继承自BackupAgentHelper类   
3. SettingsHelper类。Settings的帮助类   
4. SettingsProvider类。继承自ContentProvider类的最核心的文件   
在这四个类中，最最重要的就是SettingsProvider，其次我关心的可能就是DatabaseHelper类是如何执行数据库语句的。

#### DatabaseHelper

首先，来分析DatabaseHelper类，在DatabaseHelper类中，就是对数据库及表的操作。主要是表的创建，数据库版本更新。onCreate方法中负责创建表，执行表创建语句，然后进行表的初始化，也就是系统的初始设置；onUpgrade方法最长，需要从第一个版本到最新版本一个一个判断，升级。我们看一下具体的初始设置：

|  |
| --- |
| //onCreate中的初始化设置调用  // Load initial volume levels into DB  loadVolumeLevels(db);  // Load inital settings values  loadSettings(db);  //下面是loadSettings方法：  private void loadSettings(SQLiteDatabase db) {  loadSystemSettings(db);  loadSecureSettings(db);  // The global table only exists for the 'owner' user  if (mUserHandle == UserHandle.USER\_OWNER) {  loadGlobalSettings(db);  }  }  //我们选择loadGlobalSettings进行查看，下面分别是设置boolean值，String值，int值  // --- Previously in 'system'  loadBooleanSetting(stmt, Settings.Global.AIRPLANE\_MODE\_ON,  R.bool.def\_airplane\_mode\_on);  loadStringSetting(stmt, Settings.Global.AIRPLANE\_MODE\_RADIOS,  R.string.def\_airplane\_mode\_radios);  loadIntegerSetting(stmt, Settings.Global.WIFI\_SLEEP\_POLICY,  R.integer.def\_wifi\_sleep\_policy); |

通过上面的代码我们看到，最后其实就是引用（或者说是读取）了integer或者bool再或者string中的一些具体的字段，设置到数据库中。我们要想查看这些默认的值，可以到res/values中进行查看。和本地化无关的存放在res/values/default.xml文件中，和本地化有关的（主要指string,涉及语言）则放在相应的语言分包下面；另外，我们还能注意到的是，sqlite不支持boolean值，如果要保存boolean值，是通过integer来保存的。一般1为true,0为false。

#### SettingsProvider

看onCreate（）方法，创建表和各表的索引，然后调用了loadBookmarks(db); loadVolumeLevels(db); loadSettings(db); 三个方法，loadBookmarks(db);是存系统几个常用的应用的intent ( 如，通讯录app，emailApp, 短信app,)，loadVolumeLevels(db); 初始化的是与声音相关的配制， loadSettings(db);方法就是从defaults.xml文件中读取默认值存入相应的表中，验证了之前 的猜测。

loadSettings(db); 里面的三个方法者是从defaults.xml文件中读取默认值存入相应的表中，loadSystemSettings(db)是存入System表中，loadSecureSettings(db)是存入 secure表中。

下面来分析最最核心的SettingsProvider的代码，核心就是insert,delete,update,query四个方法，我们依次看四个方法：

insert方法：insert方法中调用了insertForUser(Uri url, ContentValues initialValues, int desiredUserHandle) 方法。在insert方法中在执行insert方法之前，调用了检查写权限的方法checkWritePermissions，更新db都需要android.Manifest.permission.WRITE\_SECURE\_SETTINGS权限（需要系统签名）和android.Manifest.permission.WRITE\_、SETTINGS。读取则不需要任何权限。

|  |
| --- |
| **private void** checkWritePermissions(SqlArguments args) {  **if** ((TABLE\_SECURE.equals(args.table) || TABLE\_GLOBAL.equals(args.table)) &&  getContext().checkCallingOrSelfPermission(  android.Manifest.permission.WRITE\_SECURE\_SETTINGS) !=  PackageManager.PERMISSION\_GRANTED) {  **throw new** SecurityException(  String.format(**"Permission denial: writing to secure settings requires %1$s"**,  android.Manifest.permission.WRITE\_SECURE\_SETTINGS));  } } |

delete和update方法中也调用了如上的权限检查方法，然后再执行相应的数据库操作方法

query方法：query方法中调用了queryForUser()方法进而执行query查询返回Cursor结果对象。

## settings.db 详解

globl

主意：

正常的

root@zs600b:/system/priv-app # ls -al | grep Set

drwxr-xr-x root root 2017-07-06 11:32 Settings

drwxr-xr-x root root 2017-07-06 11:30 SettingsProvider

异常：

root@zs600b:/system/priv-app # ls -al | grep Set

drwxr-xr-x root root 2017-08-29 21:42 Settings

drwxr-xr-x root root 2017-08-29 21:40 SettingsProvider

-rw-r--r-- root root 126118 2017-08-31 21:35 SettingsProvider.apk

Log：

09-01 10:10:20.509 I/PackageManager( 486): Package com.android.providers.settings at /system/priv-app/SettingsProvider.apk ignored: updated version 22 better than this 22

09-01 10:10:20.509 W/PackageManager( 486): Code path for hidden system pkg : com.android.providers.settings changing from /system/priv-app/SettingsProvider to /system/priv-app/SettingsProvider.apk

09-01 10:10:20.509 W/PackageManager( 486): Failed to parse /system/priv-app/SettingsProvider.apk: null

root@zs600b:/system/priv-app/SettingsProvider/arm # ls -al

-rw-r--r-- root root 254440 2017-08-29 21:40 SettingsProvider.odex

09-01 10:47:21.478 W/art ( 489): Failed to find OatDexFile for DexFile /system/priv-app/SettingsProvider/SettingsProvider.apk ( canonical path /system/priv-app/SettingsProvider/SettingsProvider.apk) with checksum 0x6ade6c23 in OatFile /system/priv-app/SettingsProvider/arm/SettingsProvider.odex

09-01 10:47:21.504 I/dex2oat ( 549): Error opening the directory: /system/lib/plugins

09-01 10:47:21.505 I/dex2oat ( 549): /system/bin/dex2oat --runtime-arg -classpath --runtime-arg --instruction-set=arm --instruction-set-features=div --runtime-arg -Xrelocate --boot-image=/system/framework/boot.art --dex-file=/system/priv-app/SettingsProvider/SettingsProvider.apk --oat-fd=76 --oat-location=/data/dalvik-cache/arm/system@priv-app@SettingsProvider@SettingsProvider.apk@classes.dex --runtime-arg -Xms64m --runtime-arg -Xmx512m

adb install –r

INSTALL\_FAILED\_ALREADY\_EXISTS

means you uninstall before install -r remove purpose

## 实例分析

在framework/base/core/java定义成另一个了

Settings.Secure.putInt(getContext().getContentResolver(), Settings.Secure.LOCATION\_MODE, mode);

## 小结：Get/set方法

### 脚本命令settings

settings get system ringtone

settings get global ringtone

root@gl300k:/ # settings put secure icon\_blacklist ethernet

root@gl300k:/ # settings get secure icon\_blacklist

ethernet

## Ref

[**Android SettingProvider详解**](http://blog.csdn.net/peng_cao/article/details/50887509)

[Settings.Global](https://developer.android.com/reference/android/provider/Settings.Global.html)

# Settings

系统设置是Android系统中非常重要的系统应用，也是整个android系统的控制中枢，关于设备的硬件，状态，软件，安全等都需要通过系统设置进行控制和查看。比如wifi状态，网络连接状。具体的功能是SettingProvider造作数据库实现的。

特别的，系统设置并不同于大多数的其他系统应用，系统设置不仅拥有platform签名（即系统签名），而且属于内核应用，所以系统设置要比非内核应用的系统应用有更大的权限。系统签名方法见第1章。

|  |
| --- |
| <activity android:name="Settings"  android:taskAffinity="com.android.settings"  android:label="@string/settings\_label\_launcher"  android:launchMode="singleTask">  <intent-filter>  <action android:name="android.settings.SETTINGS" />  <category android:name="android.intent.category.DEFAULT" />  </intent-filter>  <meta-data android:name="com.android.settings.PRIMARY\_PROFILE\_CONTROLLED"  android:value="true" /> </activity> |

在5.1.1的Settings源码中,Setting extends SettingActivity.查看SettingActivity发现，在SettingActivity中加载了设置的主界面布局，名为dashboard\_categories，文件目录为项目目录/res/xml/dashboard\_categories文件，布局文件中使用的布局标签为以及列表项,SettingsActivity中加载该布局文件的方法为loadCategoriesFromResource(),该方法存在于buildDashboardCategories(List categories)方法中,其中的行参就是所包含的设置列表项,加载该xml布局的方法就是解析xml解析器，对布局文件进行解析，现将该方法贴在此处，顺便回顾一下解析xml文件的方法：

如上的加载和解析的方法其实是在onResume方法中调用的,在onResume方法中调用了invalidateCategories方法，在invalidateCategories方法中我们会发现其实是采用了回发消息然后处理的消息处理机制，在此方法中发送了MSG\_BUILD\_CATEGORIES消息，在类声明处定义的名为mHandler的Handler中进行处理，随即调用buildDashboardCategories方法.

下面我们重新说回设置界面，上面说到解析和加载xml文件及相关数据展示，下面设计到的就是我们点击每一项时

## 网络

### 数据流量

入口类:DataUsageSummary

Kw: DataUsageSummary 源码

## WIFI选项

Wifi是系统设置中的第一项功能，通过该功能可以进行“关闭/打开WIFI”、扫描可用网络，连接网络等操作，setWifiApEnabled需要系统权限@SystemApi。核心代码如下：

|  |
| --- |
| @Override **public void** onCheckedChanged(CompoundButton switchView, **boolean** isChecked) {  KPADLogUtil.*E*(**TAG**, **"onCheckedChanged,isChecked="** + isChecked);  *//Do nothing if called as a result of a state machine event* setSwitchBarEnabled(**false**);  **if** (**mStateMachineEvent**) {//避免循环调用，造成死循环  **return**;  }  *// Disable tethering if enabling Wifi。关闭热点* **int** wifiApState = **mWifiManager**.getWifiApState();  **if** (isChecked && ((wifiApState == WifiManager.WIFI\_AP\_STATE\_ENABLING) ||  (wifiApState == WifiManager.WIFI\_AP\_STATE\_ENABLED))) {  **mWifiManager**.setWifiApEnabled(**null**, **false**);  }   **if** (!**mWifiManager**.setWifiEnabled(isChecked)) {//打开wifi  *// Error* **mSwitchBar**.setEnabled(**true**);  Toast.*makeText*(**mContext**, **"wifi error"**, Toast.***LENGTH\_SHORT***).show();  } } |

**飞行模式**的初衷是为了在飞机飞行状态下一次性关闭所有可以发出无线信号的设备而加入的功能。这些//设备主要包括3G/2G, Wi-Fi、蓝牙、NFC等设备。

热点(Access Point)和Wi-Fi不能共存

### SoftAp

市面上大多数手机支持SoftAp功能, 系统的Setting程序TetherSettings.java将响应此动作。tether这个词意思是拴绳，拴住的意思，理解起来有点晦涩，这里我的理解是分享的意思，比如 WIFI\_TETHERING(用WIFI分享网路),  USB\_TETHERING（用USB分享网络）……

那么这些网络分享如何满足什么条件才开启呢？疑问

### 从WirelessSettings开始说

在packages/apps/Settings/src/com/android/settings/WirelessSettings.java 的onCreate方法中会去检测系统是否Tether

1. ConnectivityManager cm =
2. (ConnectivityManager) activity.getSystemService(Context.CONNECTIVITY\_SERVICE);
3. **if** (isSecondaryUser || !cm.isTetheringSupported()) {
4. getPreferenceScreen().removePreference(findPreference(KEY\_TETHER\_SETTINGS));
5. } **else** {
6. Preference p = findPreference(KEY\_TETHER\_SETTINGS);
7. p.setTitle(Utils.getTetheringLabel(cm));
8. }
9. protectByRestrictions(KEY\_TETHER\_SETTINGS);

1）调用获取ConnectivityManager实例对象，并调用isTetheringSupported方法来检测是否支持Tether

2）调用getTetheringLabel查看支持哪些网络共享方式，并显示到界面

先看isTetheringSupported方法

1. **public** boolean isTetheringSupported() {
2. **try** {
3. **return** mService.isTetheringSupported();
4. } **catch** (RemoteException e) {
5. **return** **false**;
6. ...

它调用了frameworks/base/services/core/java/com/android/server/ConnectivityService.java 的isTetheringSupported 方法

**[cpp]** [view plain](http://blog.csdn.net/jshazk1989/article/details/17094415) [copy](http://blog.csdn.net/jshazk1989/article/details/17094415)

1. **public** boolean isTetheringSupported() {
2. enforceTetherAccessPermission();
3. **int** defaultVal = (SystemProperties.get("ro.tether.denied").equals("true") ? 0 : 1);
4. boolean tetherEnabledInSettings = (Settings.Global.getInt(mContext.getContentResolver(),
5. Settings.Global.TETHER\_SUPPORTED, defaultVal) != 0);
6. **return** tetherEnabledInSettings && ((mTethering.getTetherableUsbRegexs().length != 0 ||
7. mTethering.getTetherableWifiRegexs().length != 0 ||
8. mTethering.getTetherableBluetoothRegexs().length != 0) &&
9. mTethering.getUpstreamIfaceTypes().length != 0);
10. }

 读取系统ro.tether.denied的属性，返回true则表示关闭tether功能，返回false表示打开tether功能。（有些设备不能显示portable wifi 就是因为此属性返回true,这里需要修改设置为false）

之后会检测是否有配置文件配置了tether，有则使用设置的值，没有查询到配置这使用查询ro.tether.denied属性的默认值。

这里我们假设得到的tetherEnableInSettings为ture，也不意味着就能正确打开Portable Wi-Fi hostpot 界面。因为还需要检测是否支持WIFI网络共享配置

mTethering.getTetherableWifiRegexs().length!=0 检测是否支持网络共享配置，这里Android4.4 与之前的版本有一定区别，但总体不变，大家可以参考其他代码做一些变通

### getTetherableWifiRegexs()方法

frameworks/base/services/java/com/android/server/connectivity/Tethering.java

**[cpp]** [view plain](http://blog.csdn.net/jshazk1989/article/details/17094415) [copy](http://blog.csdn.net/jshazk1989/article/details/17094415)

1. **return** mTetherableWifiRegexs;

直接返回了一个变量。 此变量在frameworks/base/services/java/com/android/server/connectivity/Tethering.java

updateConfiguration()方法中赋值

**[cpp]** [view plain](http://blog.csdn.net/jshazk1989/article/details/17094415) [copy](http://blog.csdn.net/jshazk1989/article/details/17094415)

1. mTetherableWifiRegexs = tetherableWifiRegexs;

而 tetherableWifiRegexs变量赋值也在此方法中

**[cpp]** [view plain](http://blog.csdn.net/jshazk1989/article/details/17094415) [copy](http://blog.csdn.net/jshazk1989/article/details/17094415)

1. String[] tetherableWifiRegexs = mContext.getResources().getStringArray(
2. com.android.internal.R.array.config\_tether\_wifi\_regexs);

这里的config\_tether\_wifi\_regexs数组由配置文件生成

### 配置文件config.xml

frameworks/base/core/res/res/values/config.xml

**[html]** [view plain](http://blog.csdn.net/jshazk1989/article/details/17094415) [copy](http://blog.csdn.net/jshazk1989/article/details/17094415)

1. **<span** style="font-size:14px;"**>**    <!-- List of regexpressions describing the interface (if any) that represent tetherable
2. Wifi interfaces.  If the device doesn't want to support tethering over Wifi this
3. should be empty.  An example would be "softap.\*" --**>**
4. **<string-array** translatable="false" name="config\_tether\_wifi\_regexs"**>**
5. **</string-array>**
6. **</span>**

这里看到需要在此默认的config.xml中的   name="config\_tether\_wifi\_regexs" 子标签下添加用作softap的接口(一般为wlan0 或者eth0 视你的WIFI驱动程序而定)

修改后的标签：

**[html]** [view plain](http://blog.csdn.net/jshazk1989/article/details/17094415) [copy](http://blog.csdn.net/jshazk1989/article/details/17094415)

1. **<span** style="font-size:14px;"**>**    <!-- List of regexpressions describing the interface (if any) that represent tetherable
2. Wifi interfaces.  If the device doesn't want to support tethering over Wifi this
3. should be empty.  An example would be "softap.\*" --**>**
4. **<string-array** translatable="false" name="config\_tether\_wifi\_regexs"**>**
5. **<item>**"wlan0"**</item>**
6. **</string-array>**
7. **</span>**

OK 此处mTethering.getTetherableWifiRegexs().length!=0 返回true， 整个isTetheringSupported 返回真

## 蓝牙

## 流量使用情况

## 恢复出厂设置

最近看恢复出厂的一个问题，以前也查过这方面的流程，所以这里整理一些AP+framework层的流程；

入口：setting-->备份与重置--->恢复出厂设置--->重置手机--->清除全部内容：**自动执行-->手机关机--->开机--->进行恢复出厂的操作--->开机流程**；

需要root权限

App层: sendBroadcast(new Intent("Android.intent.action.MASTER\_CLEAR"));

Adb层: adb shell am broadcast -a android.intent.action.MASTER\_CLEAR

发出了这个广播，后台server会完成后续操作。主意应用程序要有android.permission.MASTER\_CLEAR 这个权限

## 日期和时间

com.android.settings. DateTimeSettings

((AlarmManager) context.getSystemService(Context.ALARM\_SERVICE)).setTime(when);

IAlarmManager mService.setTime(millis);

## 跳转到各个系统设置的方法

// 第一个参为包名，第二个各个设置的类名(可以参考下面，包名不用改变)

ComponentName cm = new ComponentName("com.android.settings",

"com.android.settings.RadioInfo");

intent.setComponent(cm);

intent.setAction("android.intent.action.VIEW");

startActivity(intent);

### 跳转到各个系统设置的列表

com.android.settings.AccessibilitySettings 辅助功能设置

　　com.android.settings.ActivityPicker 选择活动

　　com.android.settings.ApnSettings APN设置

　　com.android.settings.ApplicationSettings 应用程序设置

　　com.android.settings.BandMode 设置GSM/UMTS波段

　　com.android.settings.BatteryInfo 电池信息

　　com.android.settings.DateTimeSettings 日期和坝上旅游网时间设置

　　com.android.settings.DateTimeSettingsSetupWizard 日期和时间设置

　　com.android.settings.DevelopmentSettings 应用程序设置=》开发设置

　　com.android.settings.DeviceAdminSettings 设备管理器

　　com.android.settings.DeviceInfoSettings 关于手机

　　com.android.settings.Display 显示——设置显示字体大小及预览

　　com.android.settings.DisplaySettings 显示设置

　　com.android.settings.DockSettings 底座设置

　　com.android.settings.IccLockSettings SIM卡锁定设置

　　com.android.settings.InstalledAppDetails 语言和键盘设置

　　com.android.settings.LanguageSettings 语言和键盘设置

　　com.android.settings.LocalePicker 选择手机语言

　　com.android.settings.LocalePickerInSetupWizard 选择手机语言

　　com.android.settings.ManageApplications 已下载（安装）软件列表

　　com.android.settings.MasterClear 恢复出厂设置

　　com.android.settings.MediaFormat 格式化手机闪存

　　com.android.settings.PhysicalKeyboardSettings 设置键盘

　　com.android.settings.PrivacySettings 隐私设置

　　com.android.settings.ProxySelector 代理设置

　　com.android.settings.RadioInfo 手机信息

　　com.android.settings.RunningServices 正在运行的程序（服务）

　　com.android.settings.SecuritySettings 位置和安全设置

　　com.android.settings.Settings 系统设置

　　com.android.settings.SettingsSafetyLegalActivity 安全信息

　　com.android.settings.SoundSettings 声音设置

　　com.android.settings.TestingSettings 测试——显示手机信息、电池信息、使用情况统计、Wifi information、服务信息

　　com.android.settings.TetherSettings 绑定与便携式热点

　　com.android.settings.TextToSpeechSettings 文字转语音设置

　　com.android.settings.UsageStats 使用情况统计

　　com.android.settings.UserDictionarySettings 用户词典

　　com.android.settings.VoiceInputOutputSettings 语音输入与输出设置

　　com.android.settings.WirelessSettings 无线和网络设置

参考自：<http://blog.csdn.net/aikongmeng/article/details/44748841>

## And7 开发者选项DevelopmentSettings

# Api

## **禁止4大组件**

为什么要关闭组件？ 在用到组件时，有时候我们可能暂时性的不使用组件，但又不想把组件kill掉，比如创建了一个broadcastReceiver广播监听器，用来想监听第一次开机启动后获得系统的许多相关信息，并保存在文件中，这样以后每次开机启动就不需要再去启动该服务了，也就是说如果没有把receiver关闭掉，就算是不做数据处理，但程序却还一直在后台运行会消耗电量和内存，这时候就需要把这个receiver给关闭掉。

又或者是开机引导程序。

### 禁止组件的API：

void setComponentEnabledSetting (ComponentName componentName, int newState, int flags)

* componentName：组件名称
* newState：组件新的状态，可以设置三个值，分别是如下：

不可用状态：COMPONENT\_ENABLED\_STATE\_DISABLED

可用状态：COMPONENT\_ENABLED\_STATE\_ENABLED

默认状态：COMPONENT\_ENABLED\_STATE\_DEFAULT

* flags:行为标签，值可以是DONT\_KILL\_APP或者0。 0说明杀死包含该组件的app

禁止组件功能后，系统的package信息会记录下来，有关系统的package信息都是保存在/data/system/中package.Xml中，如下：

|  |
| --- |
| <**package name="com.android.provision"   codePath="/system/app/Provision.apk"   nativeLibraryPath="/data/data/com.android.provision/lib"  flags="1" ft="11b7e237e00" it="11b7e237e00"  ut="11b7e237e00" version="15" userId="10005"**>  <**sigs count="1"**>  <**cert index="1"** />  </**sigs**>  <**disabled-components**>  <**item name="com.android.provision.DefaultActivity"** />  </**disabled-components**> </**package**> |

上面绿色部分就是记录了package里面**provision**被禁止的组件信息。禁止之后升级app安装会提示MainActivity} does not exist.如果更新，应用程序列表也搜索不到，只能通过adb uninstall卸载，再重装的方式。

### 获取组件的状态

public int getComponentEnabledSetting(ComponentName componentName)

## 参考

Pk

[**Android 禁用和开启四大组件之setComponentEnabledSetting方法**](http://blog.csdn.net/daiqiquan/article/details/40615115)

# 普通应用知识

[AndroidManifest 中original-package标签](http://blog.csdn.net/ccc20134/article/details/50540800)

<manifest  package="com.android.launcher" >

<original-package android:name="com.android.launcher2" />

这里package="com.android.launcher"，产生的R.[**Java**](http://lib.csdn.net/base/java)就会在com.android.launcher中。  
<original-package android:name="com.android.launcher2" /> 这个地方表示，源码包是com.android.launcher2。

所以在代码中引用的R.java必须是import com.android.launcher.R;应用运行包名是com.android.launcher。

代码的包名是com.android.launcher2。即在项目中需要修改包名时，可以设置 ：<original-package android:name="" />

* 第8章　系统设置（一）
* 第9章　系统设置（二）
* 第10章　系统设置内容提供者（SettingsProvider）
* 第11章　电话与联系人
* 第12章　短信和彩信管理
* 第13章　AndroidHome应用：Launcher2（一）
* 第14章　AndroidHome应用：Launcher2（二）
* 第15章　近场通信（NFC）的实现原理

# REF

[应用程序的安装与卸载笔记](http://blog.csdn.net/sdjzyuxinburen/article/details/50645193)

[如何控制android系统中NavigationBar 的显示与隐藏](http://blog.csdn.net/a2758963/article/details/42675773)

[Android5.1 -Recents分析](http://blog.csdn.net/u013656135/article/details/49686425)

# TASK

* Provisioned标志研究限定哪些程序？我们的为啥是文件方式判断？
* 系统应该发有序广播给我Ordered Broadcast！
* Launcher setComponentEnabledSetting，为啥我的设置不生效？
* 缓存目录结构，seetings app学习
* [Android 7.0 SystemUI(3)--RecentsActivity](http://blog.csdn.net/kebelzc24/article/details/53765379)
* Aidl在服务中的设计

AccessibilityManager：https://www.jianshu.com/p/4cd8c109cdfb

UserManager分析

etActivityInfo(cn, PackageManager.***GET\_META\_DATA***)

搭建一个远程的android编译环境，让大家都可以0成本编译？

<https://osdn.net/people/developermap.php>

专门解小