关键字：Android系统 ROM，系统app，系统应用，分布

# Todo

蓝牙：**visibility IS NULL OR visibility == '0')何时出发**

Android7.0新特性Shortcut：<https://blog.csdn.net/zhangyalong_android/article/details/69257114>

压力测试的UI；com.cghs.stresstest/.StressTestActivity

# 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

kpad-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

# 固件包app分析

## /system/priv-app

目录

原生：

DefaultContainerService：应用服务，负责实现APK等相关资源文件在内部或外部存储器上的存储工作。

**DownloadProvider**：下载管理器:通知ui

ExternalStorageProvider

FusedLocation

InputDevices:只有一个广播接收器，没有任何逻辑处理，有键盘映射

ManagedProvisioning

MediaProvider: 媒体数据库的

MmsService: 发送彩信时，终端将建立短连接, 容易看出MmsService是运行在Phone进程中的

MusicFX: 音频控制面板

OneTimeInitializer: 首次启动时用于安装谷歌应用的，删除之后双清会卡在开机向导,无ui

ProxyHandler

**Settings**

SettingsProvider

SharedStorageBackup: 共享备份存储

Shell

StressTest

**SystemUI**

TeleService: http://blog.csdn.net/yihongyuelan/article/details/19930861

Telecom

TelephonyProvider

VpnDialogs

**WallpaperCropper**

预装：

GoogleInput

Google\_pdf\_viewer

## /system/app

BasicDreams

Browser

CaptivePortalLogin

CertInstaller

**Crystal\_Sky\_Launcher**

DJIService

DocumentsUI

DownloadProviderUi

Exchange2：邮件

GPSRecoder：自定义的，cpp代码

Galaxy4：壁纸？

Gallery2

HTMLViewer

HoloSpiralWallpaper：很丑的动态壁纸

KeyChain：http://blog.csdn.net/innost/article/details/44199503

LatinIME

LiveWallpapers

LiveWallpapersPicker

MediaFloat：？

NoiseField？

OpenWnn:另外的输入法

PacProcessor: 链接电脑的时候用的～但是告诉你中国连接电脑用不到PacProcessor.apk！

PackageInstaller

PhaseBeam：动态壁纸：光束 PhaseBeam

PhotoTable：照片屏幕保护程序

PicoTts: 这个是Android原生就有的语音引擎，中文语音似乎google暂时还没有,我们有必要加么

Provision:开机向导

Rk4kVideoPlayer

RkExplorer

**SystemSetup**

**SystemUpgrade**

UserDictionaryProvider

VisualizationWallpapers

WinStartService？

Webview?

/system/vendor

## /system/framework

## app详解

### DefaultContainerService

源码路径：frameworks/base/packages/

一个类无UI

|  |
| --- |
| //创建DefaultContainerService类的ComponentName对象  static final ComponentName DEFAULT\_CONTAINER\_COMPONENT = new ComponentName(DEFAULT\_CONTAINER\_PACKAGE,  "com.android.defcontainer.DefaultContainerService");  private boolean connectToService() {  //设置Intent  Intent service = new Intent().setComponent(DEFAULT\_CONTAINER\_COMPONENT);  Process.setThreadPriority(Process.THREAD\_PRIORITY\_DEFAULT);  //通过bindServiceAsUser来启动DefaultContainerService服务  if (mContext.bindServiceAsUser(service, mDefContainerConn,Context.BIND\_AUTO\_CREATE, UserHandle.OWNER)) {  mBound = true;  return true;  }  Process.setThreadPriority(Process.THREAD\_PRIORITY\_BACKGROUND);  return false;  } |

我们安装一个应用的时候还需要另外一个apk来提供服务，这个apk就是DefaultContainerService.apk

### DownloadProvider

Download的源码编译分为两个部分，一个是DownloadProvider.apk, 一个是DownloadProviderUi.apk.

这两个apk的源码分别位于

packages/providers/DownloadProvider/ui/src

packages/providers/DownloadProvider/src

<http://blog.csdn.net/chaoy1116/article/details/22384841>

### ExternalStorageProvider

frameworks/base/packages/ExternalStorageProvider

root@zs600b:/ # logcat | grep externalstorage

I/ActivityManagerService( 486): Killing 21621:com.android.externalstorage/u0a5 (adj 15): empty for 16356s

无UI，就是Provider而已

<http://blog.csdn.net/snail201211/article/details/77979988>

### FusedLocation

frameworks/base/packages/FusedLocation

融合位置，

<https://github.com/kesenhoo/android-training-course-in-chinese/blob/master/location/retrieve-current.md>

### ManagedProvisioning

配置管理器？用来干嘛的？

<http://blog.csdn.net/Liu1314you/article/details/52028823、>

工作手机的安全模式？

### MediaProvider

媒体数据库的封装类，代码量比较大（四千多行），功能比较复杂，但总的来说就是创建数据库，对外提供URI以实现对数据库的增删改查功能；

<http://gqdy365.iteye.com/blog/2150883>

### **MmsService**

Android中彩信相关的应用为MmsService，我们看看它AndroidManifest.xml中的部分片段：

<application android:label="MmsService"

android:process="com.android.phone"

android:usesCleartextTraffic="true">

<service android:name=".MmsService"

android:enabled="true"

android:exported="true"/>

</application>

容易看出MmsService是运行在Phone进程中的。在这篇博客中，我们不深入研究彩信服务的启动和收发彩信的过程，主要看看彩信如何建立和释放短连接。

### wallpapercropper

frameworks/base/packages/

### DocumentsUI

frameworks/base/packages/

小米继承了迅雷的下载引擎

<http://www.cnblogs.com/zqlxtt/p/4451949.html>

Android中Download由三个部分组成：

1.DocumentsUI -----> /[frameworks](http://androidxref.com/5.1.0_r1/xref/frameworks/)/[base](http://androidxref.com/5.1.0_r1/xref/frameworks/base/)/[packages](http://androidxref.com/5.1.0_r1/xref/frameworks/base/packages/)/[DocumentsUI](http://androidxref.com/5.1.0_r1/xref/frameworks/base/packages/DocumentsUI/)/

2.DownloadManager ---->/[frameworks](http://androidxref.com/5.1.0_r1/xref/frameworks/)/[base](http://androidxref.com/5.1.0_r1/xref/frameworks/base/)/[core](http://androidxref.com/5.1.0_r1/xref/frameworks/base/core/)/[java](http://androidxref.com/5.1.0_r1/xref/frameworks/base/core/java/)/[android](http://androidxref.com/5.1.0_r1/xref/frameworks/base/core/java/android/)/[app](http://androidxref.com/5.1.0_r1/xref/frameworks/base/core/java/android/app/)/

3.DownloadProvider ---->/[packages](http://androidxref.com/5.1.0_r1/xref/packages/)/[providers](http://androidxref.com/5.1.0_r1/xref/packages/providers/)/[DownloadProvider](http://androidxref.com/5.1.0_r1/xref/packages/providers/DownloadProvider/)/

下图中用MVC的分层将这三部分做了划分：



其中DocumentsUI作为视图层（V）负责展示Download信息

DownloadManager和DownloadProvder的一部分作为控制层（C）负责下载的逻辑控制

DownloadProvder的另一部分则作为数据层（M）负责数据的存储

### HTMLViewer

shell@kpad:/ $ dumpsys activity | grep mFo

mFocusedActivity: ActivityRecord{2ec50016 u0 com.android.htmlviewer/.HTMLViewerActivity t1}

详解app

### Browser

<http://blog.csdn.net/weirdo_c/article/details/50716848>

### CaptivePortalLogin

### CertInstaller

/packages/apps

在Android中安装数字签名，被调用

### KeyChain

<http://blog.csdn.net/innost/article/details/44199503>

数字签名

### LatinIME

/packages/inputmethods/LatinIME/

com.android.settings/.inputmethod.InputMethodAndSubtypeEnablerActivity

com.android.inputmethod.latin/.settings.SettingsActivity

<http://blog.csdn.net/asasasdasfaf/article/details/53744675>

### PicoTts

<http://www.manew.com/blog-166890-7593.html>

## com.android.captiveportallogin

com.android.captiveportallogin.CaptivePortalLoginActivity

<action android:name="android.net.conn.CAPTIVE\_PORTAL"/>

## 名词解释

### portal认证

就是连接wifi成功后,有些需要去一个登录页面,登录后才能正常使用wifi。在现在商店比如肯德基,万科,万达等很多外场,都有这样的认证方式

### android的原生portal认证

## 独立出portal认证到浏览器

将这部分功能独立出来,放到自己的浏览器中或者apk中,改怎么做呢?有两种方式

1、接收系统的intent 就是已经有系统的通知了,当用户点击通知的时候弹出自己apk中的acivity这里面需要在activity中加入如下属性,才会弹出用户选择框。

<action android:name="android.net.conn.CAPTIVE\_PORTAL" />

2、不依赖系统自己做检测,自己弹出通知流程如下

用一个接收器接收网络变化广播,然后启动个服务,在服务中做如下判断,如果连接不上就弹出通知,后面就和方案1一样了

## 定制部分

**DownloadProvider**

**Settings**

**SystemUI**

**WallpaperCropper**

**文本查看器**

**LatinIME：输入法太丑了。。**

Tts？

## REF

[解密:Android设置默认程序](http://droidyue.com/blog/2014/07/13/set-preferred-application-in-android-chinese-edition/index.html)

# 系统应用常见属性

## coreApp="true"：

关于coreApp=true的说明，在manifest中增加该属性，其实并不是代表该APP具有系统权限，**而是把该类app归类为核心APP，核心app其实也是最小android framework系统。**那么核心APP的作用是什么呢？在Android3.0之后，Android就增加了加密机制，当系统开机时检测到系统加密，他就把核心APP全部启动，并显示UI提供用户输入密码，密码正确后才会启动完整系统。

## original-package

[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="" />

标签中package属性用于设置应 用程序的进程名，即在运行时使用ddms查看到的进程名。  
标签用以设置应用 源码包名，即Java文件所在的源码程序包层次，android工程中真实的源代码层次结构。  
中package属性若与的android:name值相同，配置组建时android:name属性值 可以使用”.ClassName”形式。  
使用标签后，在中的android:name属性需要写完整的包名，”.ClassName”形式无效。

注意:**标签中package属性只是告诉系统应用的进程名；因此进程名（Manifest中package属性的值）与的值可以不一样**。

若某些客户不想改变他们的apk，但是需要启动我们的apk，这样就需要把我们的报名+类名改成客户apk指定的名字，若你又不想修改源码的代码结构，此时orgin-package就发挥重大用途了；添加origin-package，修改packagename，改完之后，只需要替换R文件的路径即可（先通过AS或eclipse导入新的R，然后ctrl+H替换整个project的R文件即可）

## requiredForAllUsers&supportsRtl

是否需要默认面对所有用户，因为android支持多用户系统切换

此应用支持从右向左布局,此属性为解决入阿拉伯文，希伯来语等从右向左的阅读习惯

## android:allowBackup="false"

第三方应用开发者需要在应用的 AndroidManifest.xml 文件中配置 allowBackup 标志(默认为 true)来设置应用数据是否能能够被备份或恢复。当这个标志被设置为true时应用程序数据可以在手机未获取 ROOT   
的情况下通过adb调试工具来备份和恢复，这就允许恶意攻击者在接触用户手机的情况下在短时间内启动手机 USB 调试功能来窃取那些能够受到   
AllowBackup 漏洞影响的应用的数据，造成用户隐私泄露甚至财产损失。

## usesCleartextTraffic：

API23引入。是否允许网络使用明文传输。默认值为true。

<http://www.chinagdg.com/thread-32313-1-1.html>

## defaultToDeviceProtectedStorage/directBootAware

**defaultToDeviceProtectedStorage:**属性仅适用于系统应用， 属性用于将默认的应用存储位置重定向到 DE 存储空间（而非 CE 存储空间）。使用此标记的系统应用必须要仔细审核存储在默认位置的所有数据，并将敏感数据的路径更改为使用 CE 存储空间。使用此选项的设备制造商应仔细检查要存储的数据，以确保其中不含任何个人信息。

directBootAware 属性则适用于所有应用。将相应应用中的所有组件均标记为加密感知型组件

https://blog.csdn.net/cigogo/article/details/78932974

## sharedUserId

fdsg

android:sharedUserId="android.uid.system"

android:sharedUserId="android.uid.shared"

android:sharedUserId="android.media

**android.uid.system**

settings/launcher/setup/

**android.uid.systemui**

**systemui**

# Home 启动原理

|  |
| --- |
| <**intent-filter android:priority="10"**>  <**action android:name="android.intent.action.MAIN"**/>  <**category android:name="android.intent.category.LAUNCHER"**/>  <**category android:name="android.intent.category.HOME"**/>  <**category android:name="android.intent.category.DEFAULT"**/>  <**category android:name="android.intent.category.SETUP\_WIZARD"**/> </**intent-filter**> |

Priority越大优先级越高，务必把程序放在system/pri-app下，或者曾经就不会有问题

# com.andorid.[provision](http://www.cnblogs.com/mythou/p/3425570.html)

Android 初始化设置向导

源码：packages/apps/ Provision

## 概述

Provision是一个系统初始化引导程序==设置向导，原生Provision只做了一件事，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，具体差异见下文

设计思想：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。

（Provision.apk改变）。mProvisioningObserver 即是用来监听设备归属状态的变化，以禁用或启用某些功能. 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. 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。

### 恢复出厂设置

settings get global device\_provisioned

settings put global device\_provisioned 0

and5

settings get global user\_setup\_complete

settings put global user\_setup\_complete 0

and7

settings get global user\_setup\_complete

settings put global user\_setup\_complete 0

settings get secure user\_setup\_complete

settings put global user\_setup\_complete 0

secure

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？

#### 实用命令

settings get global user\_setup\_complete  
 settings put global user\_setup\_complete 0  
  
 settings get secure user\_setup\_complete  
 settings put global user\_setup\_complete 0  
  
pm enable com.kpad.setup/com.kpad.setup.activity.SetupIndexActivity  
 cat /data/system/users/0/package-restrictions.xml  
--------------------------------------  
##启动则关闭  
am start -n com.kpad.setup/com.kpad.setup.activity.SetupIndexActivity  
##启动则关闭  
am start -n com.kpad.setup/com.kpad.setup.activity.SetupTermsActivity  
###ok  
am start -n com.kpad.setup/com.kpad.setup.activity.SetupWelcomeActivity  
##SetupWifiActivity

## Settings.Secure.USER\_SETUP\_COMPLETE

在framework中搜索

，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。

### Core

DevicePolicyManager

DevicePolicyManagerService

PhoneFallbackEventHandler

### Services

#### PhoneWindowManager

isUserSetupComplete

##### powerMultiPressAction：toggling theater mode

MULTI\_PRESS\_POWER\_THEATER\_MODE

##### launchAssistAction

##### Shortcut

if (isUserSetupComplete()) {

shortcutService.notifyShortcutKeyPressed(shortcutCode);

}

##### startActivityAsUser\*\*\*\*

那么setup是怎么调用系统设置的setup呢

private void startActivityAsUser(Intent intent, UserHandle handle) {

if (isUserSetupComplete()) {

mContext.startActivityAsUser(intent, handle);

} else {

Slog.i(TAG, "Not starting activity because user setup is in progress: " + intent);

}

}

##### navigation bar is disabled

private final Runnable mHiddenNavPanic = new Runnable() {

@Override

public void run() {

synchronized (mWindowManagerFuncs.getWindowManagerLock()) {

if (!isUserSetupComplete()) {

// Swipe-up for navigation bar is disabled during setup

return;

}

private void requestTransientBars(WindowState swipeTarget) {

synchronized (mWindowManagerFuncs.getWindowManagerLock()) {

if (!isUserSetupComplete()) {

// Swipe-up for navigation bar is disabled during setup

return;

}

##### goHome()禁止

f

if (!isUserSetupComplete()) {

Slog.i(TAG, "Not going home because user setup is in progress.");

return false;

}

##### updateSystemBarsLw

// update navigation bar

boolean oldImmersiveMode = isImmersiveMode(oldVis);

boolean newImmersiveMode = isImmersiveMode(vis);

if (win != null && oldImmersiveMode != newImmersiveMode) {

final String pkg = win.getOwningPackage();

mImmersiveModeConfirmation.immersiveModeChangedLw(pkg, newImmersiveMode,

isUserSetupComplete(), isNavBarEmpty(win.getSystemUiVisibility()));

}

#### TaskRecord

#### PreloadAppsInstaller

#### GestureLauncherService

#### MediaSessionService

#### NightDisplayService

### Systemui

#### Statusbar

##### PhoneStatusBar禁用qs

###### mUserSetupObserver

mUserSetup = userSetup;

// ensure quick settings is disabled until the current user makes it through the setup wizard

private boolean mUserSetup = false;

private ContentObserver mUserSetupObserver = new ContentObserver(new Handler()) {

@Override

public void onChange(boolean selfChange) {

final boolean userSetup = 0 != Settings.Secure.getIntForUser(

mContext.getContentResolver(),

Settings.Secure.USER\_SETUP\_COMPLETE,

0 /\*default \*/,

mCurrentUserId);

if (MULTIUSER\_DEBUG) Log.d(TAG, String.format("User setup changed: " +

"selfChange=%s userSetup=%s mUserSetup=%s",

selfChange, userSetup, mUserSetup));

if (userSetup != mUserSetup) {

mUserSetup = userSetup;

if (!mUserSetup && mStatusBarView != null)

animateCollapseQuickSettings();

if (mKeyguardBottomArea != null) {

mKeyguardBottomArea.setUserSetupComplete(mUserSetup);

}

if (mNetworkController != null) {

mNetworkController.setUserSetupComplete(mUserSetup);

}

}

if (mIconPolicy != null) {

mIconPolicy.setCurrentUserSetup(mUserSetup);

}

}

};

###### updateQsExpansionEnabled

/\*\*

\* Disable QS if device not provisioned.

\* If the user switcher is simple then disable QS during setup because

\* the user intends to use the lock screen user switcher, QS in not needed.

\*/

private void updateQsExpansionEnabled() {

mNotificationPanel.setQsExpansionEnabled(isDeviceProvisioned()

&& (mUserSetup || mUserSwitcherController == null

|| !mUserSwitcherController.isSimpleUserSwitcher())

&& ((mDisabled2 & StatusBarManager.DISABLE2\_QUICK\_SETTINGS) == 0)

&& !ONLY\_CORE\_APPS);

}

##### ZenModeControllerImpl

#### Keyboard

KeyboardUI

#### Recents

Recents

## Settings.Global.DEVICE\_PROVISIONED

## 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>

# WallpaperPicker

packages\apps\WallpaperPicker\src\com\android\wallpaperpicker

https://blog.csdn.net/A771642/article/details/77948878

# 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

# 蓝牙app

## 对外接口BluetoothOppReceiver

Fasdf

<**receiver  
 android:process="@string/process"  
 android:exported="true"  
 android:name=".opp.BluetoothOppReceiver"  
 android:enabled="@bool/profile\_supported\_opp"**>  
 <**intent-filter**>  
 <**action android:name="android.bluetooth.adapter.action.STATE\_CHANGED"** />  
 *<!--action android:name="android.intent.action.BOOT\_COMPLETED" /-->* <**action android:name="android.btopp.intent.action.OPEN\_RECEIVED\_FILES"** />  
 </**intent-filter**>  
</**receiver**>

可以看到实在opp文件夹下的BluetoothOppReceiver中进行处理的

else if (action.equals(Constants.ACTION\_OPEN\_RECEIVED\_FILES)) {

if (V) Log.v(TAG, "Received ACTION\_OPEN\_RECEIVED\_FILES.");

Intent in = new Intent(context, BluetoothOppTransferHistory.class);

in.setFlags(Intent.FLAG\_ACTIVITY\_NEW\_TASK | Intent.FLAG\_ACTIVITY\_CLEAR\_TOP);

in.putExtra("direction", BluetoothShare.DIRECTION\_INBOUND);

in.putExtra(Constants.EXTRA\_SHOW\_ALL\_FILES, true);

context.startActivity(in);

跳转到BluetoothTransferHistory，在跳转时对intent做了一些处理，首先是设置了一些flags，说明一下

## BluetoothOppNotification

**/android/packages/apps/Bluetooth/**

com.android.bluetooth/.opp.BluetoothOppIncomingFileConfirmActivity

蓝牙分为两类

1.经典蓝牙协议：比如iphone或者测试房的蓝牙音箱，这种协议是更新链接状态的，搞吞吐量设置

2.BLE协议：基于蓝牙4.0，只有在传输的时候才链接，低功耗设计，用于物联网

因此

连接android手机如果是走BLE协议，就没有无法显示链接状态

## 查看传输历史

Sdf

BluetoothOppTransferHistory

2011年就加入这个功能了

selection = selection + **" AND (visibility IS NULL OR visibility == '0')"**;

关键代码都在onCreate中，可以看出来通过调用managedQuery方法按指定的条件查询指定的uri，获取到cursor后传给adapter，并将adapter与listview绑定显示数据。

首先是对传输列表数据源的获取----managedQuery

public final Cursor managedQuery(Uri uri, String[] projection, String selection,

String sortOrder) {

Cursor c = getContentResolver().query(uri, projection, selection, null, sortOrder);

if (c != null) {

startManagingCursor(c);

}

return c;

}

managedQuery方法定义在Activity中，可以看到首先获取到ContentResolver对象，然后调用query方法进行查询指定uri的数据。有几点需要注意，通过该方法获取到的cursor无需去调用close方法将其关闭，因为activity会在合适的时候将其关闭。但是有一点，如果你的cursor对象调用了stopManagingCursor方法时，必须手动去调用cursor.close方法将其关闭，因为此时，activity不会自动去关闭

需要传入四个参数

* uri   : the uri of the content provider to query      所要查询的contentProvider的uri
* projection  :  projection list of columns to return  查询到后所要返回的cursor的列名，如果想要全部返回可以将参数置为null
* selection  :  selectio SQL WHERE clause            查询条件
* sortOrder  : sortorder SQL ORDER BY clause    按某种条件排序

所查询的uri为

/\*\*

\* The content:// URI for the data table in the provider

\*/

public static final Uri CONTENT\_URI = Uri.parse("content://com.android.bluetooth.opp/btopp");

想要读取该uri下的数据需要在Androidmanifest.xml配置文件中添加如下权限

<uses-permission android:name="android.permission.ACCESS\_BLUETOOTH\_SHARE" />

查询到后传给adapter进行加载view

举一个例子，在item上显示远程蓝牙name的话可以使用如下代码

tv = (TextView)view.findViewById(R.id.targetdevice);

//获取到本地蓝牙适配器

BluetoothAdapter adapter = BluetoothAdapter.getDefaultAdapter();

//获取到索引

int destinationColumnId = cursor.getColumnIndexOrThrow(BluetoothShare.DESTINATION);

//获取到本条数据中所对应的远程设备

BluetoothDevice remoteDevice = adapter.getRemoteDevice(cursor

.getString(destinationColumnId));

//获取到name

String deviceName = BluetoothOppManager.getInstance(context).getDeviceName(remoteDevice);

tv.setText(deviceName);

显示接收到的文件源码其实实现起来不难，就是借助contentResolver来读取uri的数据并显示出来，那么数据必须要通过contentprovider的方式保存，但是接收到的文件时保存在哪个contentprovider？？如何进行保存？会保存哪些信息？有哪些列？接下来就进行分析

首先根据uri---com.android.bluetooth.opp去查找清单配置文件

<provider android:name=".opp.BluetoothOppProvider"

android:authorities="com.android.bluetooth.opp"

android:exported="true"

android:process="@string/process">

<path-permission

android:pathPrefix="/btopp"

android:permission="android.permission.ACCESS\_BLUETOOTH\_SHARE" />

</provider>

可以发现存储所用的provider为BluetoothOppProvider.java，正好借此机会分析下provider的用法。

## 存储接收到的文件的ContentProvider

Df

该类位于\android\packages\apps\Bluetooth\src\com\android\bluetooth\opp文件夹下，直接继承与ContentProvider。希望是在对contentProvider有一定的了解后阅读如下分析

provider属于组件，需要再清单配置文件中进行配置

<provider android:name=".opp.BluetoothOppProvider"

android:authorities="com.android.bluetooth.opp"

android:exported="true"

android:process="@string/process">

<path-permission

android:pathPrefix="/btopp"

android:permission="android.permission.ACCESS\_BLUETOOTH\_SHARE" />

</provider>

自定义一个provider需要添加如下属性authorities：域名，如果需要访问权限，就规定所需要的访问权限

java代码中的处理如下

首先对于BluetoothOppProvider的uri进行解析

定义一个urimatcher对象，以供应用对uri进行访问解析数据

/\*\* URI matcher used to recognize URIs sent by applications \*/

private static final UriMatcher sURIMatcher = new UriMatcher(UriMatcher.NO\_MATCH);

接下来为urimatcher对象注册路径，在源码中只注册了两个

/\*\* URI matcher constant for the URI of the entire share list 返回整个数据列表\*/

private static final int SHARES = 1;

/\*\* URI matcher constant for the URI of an individual share 返回一条记录\*/

private static final int SHARES\_ID = 2;

static {

sURIMatcher.addURI("com.android.bluetooth.opp", "btopp", SHARES);

sURIMatcher.addURI("com.android.bluetooth.opp", "btopp/#", SHARES\_ID);

}

紧接着看到源码中创建了数据库，也就是说contentprovider将数据存储到数据库

数据库的name为：btop.db

/\*\* Database filename \*/

private static final String DB\_NAME = "btopp.db";

所创建的列包括14个，如下所示

private void createTable(SQLiteDatabase db) {

try {

db.execSQL("CREATE TABLE " + DB\_TABLE + "(" + BluetoothShare.\_ID

+ " INTEGER PRIMARY KEY AUTOINCREMENT," + BluetoothShare.URI + " TEXT, "

+ BluetoothShare.FILENAME\_HINT + " TEXT, " + BluetoothShare.\_DATA + " TEXT, "

+ BluetoothShare.MIMETYPE + " TEXT, " + BluetoothShare.DIRECTION + " INTEGER, "

+ BluetoothShare.DESTINATION + " TEXT, " + BluetoothShare.VISIBILITY

+ " INTEGER, " + BluetoothShare.USER\_CONFIRMATION + " INTEGER, "

+ BluetoothShare.STATUS + " INTEGER, " + BluetoothShare.TOTAL\_BYTES

+ " INTEGER, " + BluetoothShare.CURRENT\_BYTES + " INTEGER, "

+ BluetoothShare.TIMESTAMP + " INTEGER," + Constants.MEDIA\_SCANNED

+ " INTEGER); ");

} catch (SQLException ex) {

Log.e(TAG, "couldn't create table in downloads database");

throw ex;

}

}

介绍一个小知识，在数据库类的oncreate方法中可以直接并且只调用createTable方法，但是在更新数据库时需要先将数据库删除然后再调用createTable创建，删除数据库方法如下

private void dropTable(SQLiteDatabase db) {

try {

db.execSQL("DROP TABLE IF EXISTS " + DB\_TABLE);

} catch (SQLException ex) {

Log.e(TAG, "couldn't drop table in downloads database");

throw ex;

}

}

数据库实在provider的oncreate方法中进行创建的

然后就会provider对数据库进行增删改查操作--

## REF

Sdf

https://cloud.tencent.com/developer/article/1028790

### 挑起逻辑

**if** (action.equals(Constants.***ACTION\_OPEN\_OUTBOUND\_TRANSFER***)) {  
 **if** (***V***) Log.*v*(***TAG***, **"Received ACTION\_OPEN\_OUTBOUND\_TRANSFER."**);  
  
 Intent in = **new** Intent(context, BluetoothOppTransferHistory.**class**);  
 in.setFlags(Intent.***FLAG\_ACTIVITY\_NEW\_TASK*** | Intent.***FLAG\_ACTIVITY\_CLEAR\_TOP***);  
 in.putExtra(**"direction"**, BluetoothShare.***DIRECTION\_OUTBOUND***);  
 context.startActivity(in);  
} **else if** (action.equals(Constants.***ACTION\_OPEN\_INBOUND\_TRANSFER***)) {  
 **if** (***V***) Log.*v*(***TAG***, **"Received ACTION\_OPEN\_INBOUND\_TRANSFER."**);  
  
 Intent in = **new** Intent(context, BluetoothOppTransferHistory.**class**);  
 in.setFlags(Intent.***FLAG\_ACTIVITY\_NEW\_TASK*** | Intent.***FLAG\_ACTIVITY\_CLEAR\_TOP***);  
 in.putExtra(**"direction"**, BluetoothShare.***DIRECTION\_INBOUND***);  
 context.startActivity(in);  
} **else if** (action.equals(Constants.***ACTION\_OPEN\_RECEIVED\_FILES***)) {  
 **if** (***V***) Log.*v*(***TAG***, **"Received ACTION\_OPEN\_RECEIVED\_FILES."**);  
  
 Intent in = **new** Intent(context, BluetoothOppTransferHistory.**class**);  
 in.setFlags(Intent.***FLAG\_ACTIVITY\_NEW\_TASK*** | Intent.***FLAG\_ACTIVITY\_CLEAR\_TOP***);  
 in.putExtra(**"direction"**, BluetoothShare.***DIRECTION\_INBOUND***);  
 in.putExtra(Constants.***EXTRA\_SHOW\_ALL\_FILES***, **true**);  
 context.startActivity(in);

*/\*\* the intent extra to show all received files in the transfer history \*/***public static final** String ***EXTRA\_SHOW\_ALL\_FILES*** = **"android.btopp.intent.extra.SHOW\_ALL"**;

**mShowAllIncoming** = getIntent().getBooleanExtra(  
 Constants.***EXTRA\_SHOW\_ALL\_FILES***, **false**);

**int** dir = getIntent().getIntExtra(**"direction"**, 0);

# 优化

## 无LAUNCHER编译配置

<**action android:name="android.intent.action.MAIN"**/>  
<**category android:name="android.intent.category.LAUNCHER"**/>

## 模块裁剪

Rom精简 http://club.huawei.com/thread-4643339-1-1.html

<http://tcgz.leanote.com/post/Android%E7%B3%BB%E7%BB%9F%E8%87%AA%E5%B8%A6app%E5%8A%9F%E8%83%BD%E5%8F%8A%E5%88%A0%E5%87%8F%E8%AF%B4%E6%98%8E>

## 性能

# 普通应用知识

* 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)