关键字：Android系统 ROM，系统app，系统应用，分布

# 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

# 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**

# 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团队负责搭建，主要编译framework各种jar，暴露hide方法。

迁移到AS环境下有一定的工作量：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，

### 开发环境方案对比

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

**[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)

fdg

### 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

### 关于support.jar

Fg

*/\* implementation files('framework-libs/android-support-design.jar')  
 implementation files('framework-libs/android-support-v4.jar')  
 implementation files('framework-libs/android-support-v7-appcompat.jar')  
 implementation files('framework-libs/android-support-v7-recyclerview.jar')  
 implementation files('framework-libs/android-support-v13.jar')\*/  
  
 //for we need res ,must* compile **'com.android.support:design:25.4.0'** compile **'com.android.support:support-v4:25.4.0'** compile **'com.android.support:appcompat-v7:25.4.0'** compile **'com.android.support:recyclerview-v7:25.4.0'** compile **'com.android.support:support-v13:25.4.0'**

### **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. framework方法缺失：需要framework.jar,通过Use Libraries方式导入framework.jar(同时需要配置优先使用这个库， （framework.jar通过系统make模块编译方式得到）。

2.根据报错libcore/icu：找到对应源码,导入core-libart-classes.jar （也是通过系统make模块编译方式得到），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

源码路径: /frameworks/opt/setupwizard/navigationbar

并设置成Is Library

### 导入setupwizard/library源码

源码路径: frameworks/opt/setupwizard/library

### **Settings**

#### **导入framework.jar**

同上

#### **导入静态库**

android-support直接在AS下用远程仓库获得，注意版本号和系统保持一致

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 \

jsr305.jar通过系统make得到，

settings-logtags

#### **导入动态库**

一下库都需要通过系统编译得到，源码目录不做展开了

**[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

telephony-common.jar主要是基带相关的包，导入包含了hide的jar，否则会如下报错：

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

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

## 编译

这里描述AS编译过程产生的问题

### AS下的特殊配置

#### Found item String more than one time

#### gesture\_fingerprint\_swipe是空文件异常

这个是手势定义，都是空文件，另外还有[gesture\_ambient\_lift](https://android.googlesource.com/platform/packages/apps/Settings/+/12e4beb%5E%21/)等

在make环境ok，但是as不行。

解决办法：重名为.png

#### 如何导入多个framework下jar包

比如要添加framework.jar 和core.jar。在项目的build.gradle 最后添加上面这段代码，options.compilerArgs.add （），在括号中填写真实的路径，有多个jar包就用分号隔开。

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

1. allprojects {
2. repositories {
3. jcenter()
4. }
5. gradle.projectsEvaluated {
6. tasks.withType(JavaCompile) {
7. options.compilerArgs **<<** "-Xlint:unchecked" **<<** "-Xlint:deprecation"
8. options.compilerArgs.add('-Xbootclasspath/p:framework\_libs/telephony-common.jar;framework\_libs/core.jar')
9. }
10. }
11. }

#### android自带的属性规范

<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

在color.xml把@\*改为@可以干活？systemui

### Jdk7格式

Android7采用的java1.8，AS肯定是可以用1.7的，这个我们需要改过来

### buildToolsVersion 异常

报错：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** [.MORE](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)

### 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？**

### <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**>

### EventLogTags找不到

Make 编译这个class吧

## 安装

### PARSE\_FAILED\_MANIFEST\_MALFORMED

可以编译出apk了，但是此时无法安装

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 |  |
|  |  |  |

### signatures do not match

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

Adb Failure [INSTALL\_FAILED\_TEST\_ONLY

分析：没有系统签名！

解决：系统签名

### base.apk code is missing

问题：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]

分析：AS的编译机制，系统可能PMS对于apk的解析处于一个错误的状态

解决：$ adb shell pm uninstall com.android.settings

DELETE\_FAILED\_INTERNAL\_ERROR

Error while Installing APK

重启系统就好！

## 运行

### DrawerLayout 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

### 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**);

## 系统应用基本架构

### 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>

# SystemUI AS环境搭建



## 模块分析

LOCAL\_PATH:= $(call my-dir)  
  
include $(CLEAR\_VARS)  
  
LOCAL\_MODULE := SystemUI-proto-tags  
  
LOCAL\_SRC\_FILES := $(call all-proto-files-under,src) \  
 src/com/android/systemui/EventLogTags.logtags  
  
LOCAL\_PROTOC\_OPTIMIZE\_TYPE := nano  
LOCAL\_PROTO\_JAVA\_OUTPUT\_PARAMS := optional\_field\_style=accessors  
  
include $(BUILD\_STATIC\_JAVA\_LIBRARY)

$(call all-Iaidl-files-under, src)

librockchip\_pip\_jni

代码安装？

frameworks/support/v7/recyclerview/res \

frameworks/support/v17/leanback/res

## 系统导出

framework-protos \  
SystemUI-proto-tags

Keyguard \

android-support-v14-preference \  
android-support-v17-leanback \

等

## 编译

aka com.android.settings:color/transparent) not found.

**@android:style/Theme.DeviceDefault.Light**

error: resource android:color/quaternary\_device\_default\_settings is private.

导入Keyguaard的res，运行去重复的string

手动删除字符串

JDK配置

compileOptions {  
 sourceCompatibility JavaVersion.***VERSION\_1\_8*** targetCompatibility JavaVersion.***VERSION\_1\_8***}

K[eyguard开始被编译为一个jar包，被SystemUI静态导入](http://www.cnblogs.com/cascle/p/7053688.html)

[Keyguard分析](http://www.cnblogs.com/cascle/p/7053688.html)

[以SystemUI为例，如果需要在SystemUI中引入第三方ja](http://www.voidcn.com/article/p-pcdsvywq-vc.html)

[Android系统源码framework SystemUI导入eclipse编译](https://blog.csdn.net/qq_25804863/article/details/48669667)

AS直接支持MK研究！需要构建环境，更麻烦吧

### 错误: 程序包com.google.protobuf.nano不存在

com.android.framework.protobuf.nano

<https://stackoverflow.com/questions/22247951/android-protobuf-nano-documentation>

### 错误: 程序包IRecentsSystemUserCallbacks不存在

sourceSets {  
 main {  
 aidl.srcDirs = [**'SystemUI/src'**]  
 res.srcDirs = [**'SystemUI/res'**, **'SystemUI/dji/main/res'**, **'SystemUI/dji/test/res'**,**'Keyguard-res'**,**"src/main/res"**]  
 java.srcDirs = [**'SystemUI/src'**, **'SystemUI/dji/main/java'**,**'src/main/java'**,**'SystemUI/dji/test/java'**,**"src/main/java"**]  
 manifest.srcFile **'SystemUI/AndroidManifest.xml'** }  
}

http://www.cnblogs.com/bluestorm/p/6826622.html

### Attr/frameColo

### r more than one time

## 运行

### AutoReinflateContainer

android.view.InflateException: Binary XML file line #72: Binary XML file line #42: Error inflating class com.android.systemui.AutoReinflateContainer

Caused by: android.view.InflateException: Binary XML file line #42: Error inflating class com.android.systemui.AutoReinflateContainer

Toolbar.setNavigationIcon on a null object

### android.content.res.Resources$NotFoundExceptio

# 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

# [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

# DocumentsUI

资源管理器

## 配置

dependencies {  
 compile **'com.google.guava:guava:23.0-android'**

## 编译

**tools:replace="android:label"**

error: style attribute 'attr/actionBarWidgetTheme (aka com.android.DocumentsUI:attr/actionBarWidgetTheme)' not found.

Message{kind=ERROR, text=error: style attribute 'attr/actionBarWidgetTheme

## 安装

### [INSTALL\_FAILED\_CONFLICTING\_PROVIDER

adb shell pm install -t -r "/data/local/tmp/com.android.DocumentsUI"

Failure [INSTALL\_FAILED\_CONFLICTING\_PROVIDER: Package couldn't be installed in /data/app/com.android.DocumentsUI-1: Can't install because provider name com.android.documentsui.recents (in package com.android.DocumentsUI) is already used by com.android.documentsui]

### Failure [INSTALL\_FAILED\_TEST\_ONLY:

# Gallery2

## 导入库

### Guva

https://github.com/google/guava/wiki/Release23

### Android 视频编辑解析库MP4Parser

使用此库可以实现多个音频视频的追加合并，将视频的音频和视频进行分离替换等，同时还支持了字幕的添加，使用时候需要注意音视频格式。主要功能有：

* 将音频视频混合到一个mp4文件中
* 将编码格式相同的录音进行拼接合并: Different resolutions和Different frame-rates是不能进行拼接合并操作的。
* 添加或者修改元数据
* 通过省略帧缩短录音

[实例参考](https://www.jianshu.com/p/c87ada9b0f65)，仓库版本列表[**mvnrepository**](https://mvnrepository.com/artifact/com.googlecode.mp4parser/isoparser)

系统源码提示 参考：[GIT](https://github.com/sannies/mp4parser/issues/277)，但是已经找不到对应的版本了。

导入方案1【失败】：远程依赖，**1.1.22，1.0.6，***1.0-RC-27*

*// compile 'com.googlecode.mp4parser:isoparser:1.1.21'* compile **'org.mp4parser:streaming:1.9.27'** compile **'org.mp4parser:muxer:1.9.27'** compile **'org.mp4parser:isoparser:1.9.27'**

导入方案2【pass】：只好源码依赖之！！！！！！！！！！

[**Android 视频编辑解析库MP4Parser**](https://www.jianshu.com/p/c87ada9b0f65)

[**mvnrepository**](https://mvnrepository.com/artifact/com.googlecode.mp4parser/isoparser)

[Android(java方法）上实现mp4的分割和拼接](http://www.cnblogs.com/android-blogs/p/5711171.html)

#### E/BitmapUtils: createVideoThumbnail

java.lang.reflect.InvocationTargetException

at java.lang.reflect.Method.invoke(Native Method)

at com.android.gallery3d.common.BitmapUtils.createVideoThumbnail(BitmapUtils.java:199)

### XMPCore

配置源码路径就好了

### RenderScript

Renderscript是Android操作系统上的一套API。它基于异构计算思想（指使用不同类型指令集和体系架构的计算单元组成系统的计算方式），专门用于密集型计算，尤其是图像处理、计算机图形、计算机视觉。允许开发者以较少的代码实现功能复杂且性能优越的应用程序。

RenderScript可在设备上所有可用的处理器上并行执行，例如多核CPU、GPU、DSP，所以开发者可以专心写处理算法，而不需要关心调度和负载平衡的问题。

本文通过写一个简单的例子（[源码链接](https://link.jianshu.com/?t=https://github.com/kevin-nazgul/AndroidMisc)）来解释怎么使用RenderScript，例子中将彩色的图片转换为黑白的，为了测试RenderScript的性能，加上手指滑动的效果，如下图：



#### 工程配置

RenderScript的api从Android 3.0（Api level 11）开始系统自带了，在低于3.0的机器上，可以通过support包来使用RenderScript。但是还是推荐都使用support的RenderScript，因为RenderScript可能会有bug，使用support包能够及时得到更新。本文均使用support包中的Api。

要使用RenderScript，除了要导入一个jar包外，还需要复制一些so文件，如果使用gradle的话，就方便一些，只需要在build.gradle文件中增加两句：

defaultConfig {

minSdkVersion 8

targetSdkVersion 16

**renderscriptTargetApi 18**

**renderscriptSupportModeEnabled true** }

jh  **renderscriptTargetApi**：设置生成的字节码的版本，推荐使用最高的API Level，并且设置renderscriptSupportModeEnabled为true。这个选项的合法值从11到最新发布的API Level。如果你的最低SDK的版本和这个值不一样，这个值就会被忽略，在编译的时候，这个值会被设置了最低的SDK版本。

 **renderscriptSupportModeEnabled**：指定在运行的设备不是target version的时候，生成的字节码需要回滚到兼容的版本

#### 脚本语言

为了能在多个平台执行，RenderScript要求开发者使用Rs的脚本语言来实现计算的代码，脚本代码采用了c99语法，所以看起来和C语言很像。代码要求放在.rs文件中，文件需要放在 <project\_root>/src/ 中，脚本包含脚本的入口、函数和变量：

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

1. #pragma version(1)
2. #pragma rs java\_package\_name(com.winomtech.androidmisc.rs)
4. **void** root(**const** uchar4 \*in, uchar4 \*out, uint32\_t x, uint32\_t y) {
5. out->r = out->g = out->b = (in->r + in->g + in->b) / 3;
6. }
8. **void** init() {
9. }

#pragma version(1) 声明脚本中使用的Rs的版本。#pragma rs java\_package\_name 声明脚本生成的Java层的代码所在的包名

*root*（Kernel）函数是rs脚本默认的入口，这种入口函数在rs中称为Kernel函数，默认的Kernel函数名为*root*，且必须返回*void*，并且有以下的参数：

* 指向rs脚本输入输出的内存的指针。在Android 3.2(Api Level 13)或者更早的版本，这两个参数都需要，Android 4.0(Api Level 14)及以后的版本要求一个或两个。

下面的参数是可选的，但是如果你选择使用它们，则都需要提供：

* rs脚本用来实现计算的附加数据的指针，可以是原始类型的指针，也可以是复杂的结构体的指针。
* 附加数据的大小

从Android 4.1 (Api level 16)开始，可以自己定义kernel函数的参数，而不必和前面描述参数及返回值一样。并且可以在同一个脚本中定义多个kernel函数，但是需要在自定义的kernel函数前面增加 \_\_attribute\_\_((kernel))。例如，下面是一个kernel函数，接收两个 uint32\_t 类型的参数，返回一个uchar4：

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

1. uchar4 \_\_attribute\_\_((kernel)) myFun(uint32\_t x, uint32\_t y) {
2. ...
3. }

参数类型uchar4是rs中一类数据类型，称为Vector。Vector通常是普通类型接着2、3、4，比如：float4, int3, double2, ulong4。Vector类型中的成员可以使用多种风格来访问：变量名接着一个点，再接着：

* 字符x、y、z和w
* 字符r、g、b和a
* 字符s或S，随后接着从0开始的下标

uint32\_t x、uint32\_t y两个参数是当前执行到的下标，这两个参数是可选的，最多可以有三维x、y、z，类型必须为uint32\_t。

init函数是一个可选的函数，在kernel函数执行前，init函数会被执行一次，可以在init里面做一些初始化的工作，比如初始化变量。

#### 脚本编译

脚本在执行之前会有两次编译：在android编译的过程中，会发生第一次编译，生成LLVM的字节码；当应用在设备中执行的时候，会发生第二次编译，生成设备的机器码。

#### Java层的接口

因为脚本最后会被编译为机器码，为了方便Java层设置数据和调用接口，rs在android编译的时候，会生成一个java文件，名字为ScriptC\_script\_name，比如Gray.rs文件，会自动生成一个ScriptC\_Gary.java。下面代码是上面rs脚本生成的内容：

生成的代码继承于ScriptC，其中有一些变量和函数名都有\_root的后缀，这个后缀的root其实就是kernel函数的名字，如果一个rs脚本中有多个kernel函数，比如有一个名为test的kernel函数，那么就有一些函数和变量名有\_test后缀。

getKernelID\_root这个是返回root的id，这个id是RenderScript框架所使用的。有时候需要在一个Kernel函数执行完后，再执行另外一个时，就需要使用到它。

forEach\_root函数是用来调用Kernel函数执行的入口，在执行之前，会检查输入和输出是否和脚本中的类型一致，如果不一致直接抛出异常，同时还会检查输入和输出的维度是否一致。

#### 执行

下面代码是缩减后的代码，仅作讲解使用，代码的主要作用是用输入的图片生成一张黑白的图片，并且显示在界面上：

Bitmap mInBitmap = BitmapFactory.decodeResource(getResources(), R.drawable.renderscript\_input); Bitmap mOutBitmap = Bitmap.createBitmap(mInBitmap.getWidth(), mInBitmap.getHeight(), Bitmap.Config.ARGB\_8888); RenderScript mRenderScript = RenderScript.create(getActivity()); ScriptC\_Gray mGrayScript = new ScriptC\_Gray(mRenderScript); Allocation mAllocationIn = Allocation.createFromBitmap(mRenderScript, mInBitmap); Allocation mAllocationOut = Allocation.createFromBitmap(mRenderScript, mOutBitmap); mGrayScript.forEach\_root(mAllocationIn, mAllocationOut); mAllocationOut.copyTo(mOutBitmap); mImageView.setImageBitmap(mOutBitmap);

##### RenderScript

Fg

这个类用来访问RenderScript的上下文，管理RenderScript的初始化、资源、以及生命周期的。在任何其他RS对象使用之前，需要创建一个RenderScript对象，并且需要缓存这个对象，以方便后续使用。当进程完成使用RenderScript的任务的时候，需要调用releaseAllContexts。

通过调用RenderScript.create静态方式，可以得到RenderScript的对象。在Api 23的时候，多次调用create得到的会是同样的对象，所以能够得到同样的上下文，以及同样的配置。在Api 23之前，每次调用create将生成一个新的上下文。

##### ScriptC\_Gray

##### Allocation

fgs应用程序在Android虚拟机上使用RenderScript，但是实际的rs脚本在native层执行，并且需要访问Android虚拟机分配的内存。为此，就需要将虚拟机层分配的内存和RenderScript runtime连接起来，这个过程称为绑定，然后RenderScript runtime就可以方便的使用它需要的内存了，而不需要显式的分配。最终的结果就好像你在C中调用了malloc一样。

为了支持这种内存分配的方式，提供了一组API给Android VM来分配内存，类似于malloc函数的功能。这些类本质是描述应该怎么分配内存以及执行真正的分配。为了更好的理解这些类是怎么工作的，我们看看它们与malloc的关系：

array = (int \*)malloc(sizeof(int)\*10);

这个malloc调用可以分为两个部分：分配的内存单元的大小（sizeof(int)），分配的单元的个数。Android框架提供了类似于这两部分的类，以及类似malloc的类。

Element类代表malloc中的分配单元，例如浮点类型或者一个结构体。Type类对Element和分配的个数进行包装，可以将Type想象成Element的数组。而Allocation类基于给定的Type执行真正的内存分配，以及访问分配的内存。

Type由5个维度组成：X、Y、Z、LOD(level of detail)和Faces(of a cube map)。可以设置X、Y、Z为任意正整数，但是要限制在可用内存的范围内。如果某个维度的值为0，表示没有那个维度，例如：x=10、y=1、z=0表示二维。LOD和Faces用boolean值来表示是否出现了。

Allocation.createFromBitmap是对分配操作的封装，会根据Bitmap的配置来指定对于的Element的值，以及Type的值，并且用Bitmap的宽高来指定x、y的值。

##### forEach\_root

forEach\_root中两个参数分别指定输入和输出，RenderScript会根据输入参数中的维度，遍历整个数组，对每个数值都调用一次Kernel函数，所以rs文件中的in、out、x、y是根据当前遍历的位置的不同而不同。

##### 变量

在前面的rs文件中，只指定了输入和输出，当需要外部传入多个参数时，就需要使用rs的变量了。本文的例子中，除了要指定输入输出，还需要指定从哪个位置开始是需要变灰的，所以我们在rs脚本中增加一个变量：

#pragma version(1)

#pragma rs java\_package\_name(com.winomtech.androidmisc.rs)

int gPos;

...

再编译下android工程，我们再看生成的java代码，会多出来两个方法：

public synchronized void set\_gPos(int v) {

setVar(mExportVarIdx\_gPos, v);

mExportVar\_gPos = v;

}

public int get\_gPos() {

return mExportVar\_gPos;

}

java层就可以通过这两个接口来访问rs中的gPos变量了

#### 总结

RenderScript用起来还算挺简单的，当然也主要是本文实现功能简单。当然RenderScript的作用还是很容易看出来的，可以写一个长循环，然后看cpu使用，在我的手机上面，4个核心可以跑满，而如果使用jni的话，很明显只会在一个核心上面跑。

RenderScript :简单而快速的图像处理，不用复杂的OpenCL就能利用好你手机的GPU计算能力，核心在于写到rs脚本文件

Go

resources.srcDirs = ['gen']

renderscript.srcDirs = ['src']

[RenderScript的基础使用](https://www.jianshu.com/p/b7142cb70b04)

[RenderScript 代码实例](https://www.helplib.com/Java_API_Classes/article_59409)

[How to Use the Renderscript Support Library with Gradle](https://futurestud.io/tutorials/how-to-use-the-renderscript-support-library-with-gradle-based-android-projects)

#### make编译

##### make配置

LOCAL\_SRC\_FILES := \

$(call all-renderscript-files-under, src)

##### 输入



##### 输出

out/target/common/obj/APPS/Gallery2\_intermediates/src/renderscript/



#### AS配置

AS工程依赖



Rs依赖？貌似搞不定呢

java依赖

java.srcDirs = [**'Gallery2/src'**, ,**"src/mp4parser/isoparser/src/main/java"**]

远程仓库依赖？

## gradle编译

### Error: Found item String/no\_storage more than one time

运行脚本：fixGallery2String.py

### filtershow\_state\_button\_background: Error: The file name must end with .xml or .png

命名为.png

### unknown element <permission> found.,

fg

<**permission android:name="com.android.gallery3d.filtershow.permission.WRITE"  
 android:protectionLevel="signature"** />

移动位置

### attribute 'com.android.gallery3d:max\_width' not found.

error: attribute 'com.android.gallery3d:max\_width' not found.

Message{kind=ERROR, text=error: attribute 'com.android.gallery3d:max\_width' not found., sources=[F:\K\1-src\3399\dpad-gallery2\Gallery2\res\layout\filtershow\_activity.xml:75], original message=, tool name=Optional.of(AAPT)}

applicationId **'com.android.gallery3d'**

## 源码编译

### org.apache.http.legacy

<https://blog.csdn.net/zxw136511485/article/details/53185485>

原来是Android 6.0不再支持 Apache HTTP client， 建议使用 HttpURLConnection 代替。如果还想要继续使用 Apache HTTP client 的，请在build.gradle中添加下面的代码

1. android {
2. useLibrary 'org.apache.http.legacy'
3. }

# 优化

## 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)

# 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

protected-broadcast 系统应用自定义广播规范

https://blog.csdn.net/TaylorPotter/article/details/70194248