# 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

# 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. 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

2、执行命令 ./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

## 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. ndroid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

# 实现修改Android系统的开机动画

Android系统中，与开机动画相关的文件都放在一个叫bootanimation.zip的压缩包文件内，bootanimation.zip随system.img一起发布。需要放到system/media目录中.如果我们要查找系统自带的bootanimation.zip文件，可以将system目录还原，在其下的media目录下存在一个bootanimation.zip文件，替换该文件，我们就能完成开机动画的修改。bootanimation.zip文件解压看看里面的具体包含的内容：首先包含一个desc.txt文件，还会包含若干类似part0，part1的目录，其中desc.txt文件是必须的，part0、part1等目录至少要存在一个。part0、part1目录中存放的是图像文件，诸如001.png、002.png类似的命名有规律的图像文件。Android系统中读取这些静态图像，并按一定的显示规律，和频率产生动画效果，desc.txt文件就是用来描述加载规律和频率信息的文件。

为了使Android拥有Root权限，需要在代码中执行su命令。在执行su命令的过程中会创建一个新的拥有root权限的进程，通过该进程进行的任何操作都是在root权限下进行的

//执行su命令，并创建一个新进程（Process对象）

Process process = Runtime.getRuntime().exec("su");

//获取新进程的OutputStream对象，可以通过该对象发出要执行的命令

OutputStream os = process.getOutputStream();

//获取新进程的InputStream对象，可以通过该对象获取命令执行后返回的数据

InputStream is = process.getInputStream();

......

最后还要实现设备的重启，在Android系统中，要实现重启有两种方式：

1. 执行reboot命令.执行reboot命令需要root权限。也就是说，只要拥有了root权限，任何应用程序都可以重启Android设备

2. 调用PowerManager.reboot命令。该种重启方式只有System用户能使用，自由system用户才允许设置android.permission.REBOOT权限

PowerManager pm = (PowerManager)getSystemService(Context.POWER\_SERVICE);

pm.reboot("change boot animation");

最后执行重启设备命令需要在清单配置文件中配置android.permission.REBOOT权限

# [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。

## 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。

## 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



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是如何启动Launcher。

ActivityManagerService服务是从其中main()启动的，在main()中主要完成一些初始化动作，然后在systemReady()中调用resumeTopActivityLocked()，这个方法属于ActivityStack类，在这个方法中启动launcher主要是利用方法startHomeActivityLocked()这个方法属于ActivityManagerService类。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，具体相应的Activity由mMainStack中的

startActivityLocked()启动相应的Activity。具体流程见下图：



# 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.

## 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

# Android系统设置

系统设置是Android系统中非常重要的系统应用，也是整个android系统的控制中枢，关于设备的硬件，状态，软件，安全等都需要通过系统设置进行控制和查看。比如wifi状态，网络连接状。特别的，系统设置并不同于大多数的其他系统应用，系统设置不仅拥有platform签名（即系统签名），而且属于内核应用，所以系统设置要比非内核应用的系统应用有更大的权限。系统签名方法见第1章。

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

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

# Api

## **禁止4大组件**

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

又或者是开机引导程序。

### 禁止组件的API：

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

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

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

可用状态：COMPONENT\_ENABLED\_STATE\_ENABLED

默认状态：COMPONENT\_ENABLED\_STATE\_DEFAULT

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

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

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

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

### 获取组件的状态

public int getComponentEnabledSetting(ComponentName componentName)

## 参考

Pk

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

# 普通应用知识

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

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

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

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

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

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

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

# REF

[系统应用源代码分析2的所有章节笔记](http://blog.csdn.net/sdjzyuxinburen/article/details/50645193)

# QA

* 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