**PMS自顶向下分析**

# Todo

权限方式来控制第三方安装

扫描不成功删除一些文件

群组权限实现原理

为何是queryIntentActivitiesInternal查的是activity

Release和debug究竟有什么区别，如何加入反调试功能和hook

方案1：SAF权限实现，模仿ES文件管理器，这是android标准，但是也是最耗费人力的方案

方案2：最小化hook权限WRITE\_MEDIA\_STORAGE，go在menifest配置WRITE\_MEDIA\_STORAGE权限，系统hook go为PRIV类型app

原理：WRITE\_MEDIA\_STORAGE权限是一个特权权限，使用该功能的app条件是PRIVILEGED标志（安装目录为system/priv-app）

具体源码为：

frameworks\base\core\res\AndroidManifest.xml

<permission android:name="android.permission.WRITE\_MEDIA\_STORAGE"

android:protectionLevel="signature|privileged" />

参考：https://blog.csdn.net/u013553529/article/details/53167072

方案3：最大化hook权限WRITE\_MEDIA\_STORAGE，go在menifest配置WRITE\_MEDIA\_STORAGE权限，把WRITE\_MEDIA\_STORAGE降级权限dangerous，

WRITE\_EXTERNAL\_STORAGE保持一致（dangerous）

<permission name="android.permission.WRITE\_EXTERNAL\_STORAGE" >

<group gid="media\_rw" />

<group gid="sdcard\_rw" />

ref

https://jingyan.baidu.com/article/cd4c2979ef250b756f6e605a.html

https://stackoverflow.com/questions/35862678/android-external-storage-permission

降级原理：http://www.hhjack.com/android\_permission\_and\_gid/

</permission>

<permission android:name="android.permission.WRITE\_EXTERNAL\_STORAGE"

android:permissionGroup="android.permission-group.STORAGE"

android:label="@string/permlab\_sdcardWrite"

android:description="@string/permdesc\_sdcardWrite"

android:protectionLevel="dangerous" />

<permission android:name="android.permission.WRITE\_MEDIA\_STORAGE"

android:protectionLevel="signature|privileged" />

或者和

如何做PRIVILEGED标志

1.重新系统签名，添加PRIVILEGED标志位，本质上还是在priv-app时生成

2.手动安装到system/priv-app下

为何lican那种修改也是可以的？？

WRITE\_MEDIA\_STORAGE"，把WRITE\_EXTERNAL\_STORAGE强制改为权限组

这样就会降低WRITE\_MEDIA\_STORAGE的权限需求？将为普通权限，然后申请就会有了

其他应用那么怎么办呢，难道也要这么处理

【】Not granting permission android.permission.WRITE\_MEDIA\_STORAGE to package dji.go.v4.debug (protectionLevel=18 flags=0x38183f46)

public static final int PROTECTION\_FLAG\_PRIVILEGED = 0x10;

public static final int PROTECTION\_FLAG\_APPOP = 0x40;

else if ((bp.protectionLevel&PermissionInfo.PROTECTION\_FLAG\_APPOP) == 0) {

// Don't print warning for app op permissions, since it is fine for them

// not to be granted, there is a UI for the user to decide.

if (packageOfInterest == null || packageOfInterest.equals(pkg.packageName)) {

Slog.w(TAG, "Not granting permission " + perm

+ " to package " + pkg.packageName

+ " (protectionLevel=" + bp.protectionLevel

+ " flags=0x" + Integer.toHexString(pkg.applicationInfo.flags)

+ ")");

}

}

W/PackageManager:

if (locationIsPrivileged(disabledPs.codePath)) {

parseFlags |= PackageParser.PARSE\_IS\_PRIVILEGED;

}

Permission is only granted to system apps less... (Ctrl+F1)

Permissions with the protection level signature, privileged or signatureOrSystem are only granted to system apps. If an app is a regular non-system app, it will never be able to use these permissions. Issue id: ProtectedPermissions

dump es的权限

原来是DocumentUI注册的权限？？

<string name="open\_external\_dialog\_root\_request" msgid="8899108702926347720">"要授权<xliff:g id="APPNAME"><b>^1</b></xliff:g>访问您 <xliff:g id="STORAGE"><i>^2</i></xliff:g>上的数据（包括照片和视频）吗？"</string>

com.android.documentsui/.OpenExternalDirectoryActivity

<action android:name="android.os.storage.action.OPEN\_EXTERNAL\_DIRECTORY" />

SAF框架???

安装的时候也会报错：

2019-04-01 16:21:17.596 495-569/? W/PackageManager: Not granting permission android.permission.WRITE\_MEDIA\_STORAGE to package com.estrongs.android.pop (protectionLevel=18 flags=0x3858be44)

android:installLocation="0"

package="com.estrongs.android.pop"

platformBuildVersionCode="27"

platformBuildVersionName="8.1.0">

<uses-sdk

android:minSdkVersion="14"

android:targetSdkVersion="26" />

<uses-permission

android:name="android.permission.WRITE\_EXTERNAL\_STORAGE" />

<uses-permission

android:name="android.permission.WRITE\_MEDIA\_STORAGE" />

切换目录安装其他app会不会有问题

chooseBestActivityl逻辑

Notification\_log.db???

1. xml

appops.xml

package-usage.list

Runtime-permission>xml

<https://blog.csdn.net/QQxiaoqiang1573/article/details/72903237>

<https://blog.csdn.net/asahinokawa/article/details/79115381>

为甚更新

**throw new** PackageManagerException(Log.WARN, **"Package "** + ps.name + **" at "** + scanFile + **" ignored: updated version "** + ps.versionCode  
 + **" better than this "** + pkg.mVersionCode);

原理

https://blog.csdn.net/suifcd/article/details/40823927

横屏幕布局hook

Systemrade()禁用app

签名机制？？？

Hook app名字？？

https://www.jianshu.com/p/8632fdc86009

权限管理

grantRuntimePermissions

Android 6.0 动态权限实战（二 - 终结章 - 高手必看！）------ 关于动态权限的正确使用与理解

<https://blog.csdn.net/jspping/article/details/54288639>

privsion,disable account share

# 基本概念

## 看源码作用

马克思说认识工具，使用工具，并改造世界

**认识**学习代码：理解framework语言，借鉴好的设计模式：比如Settings的抽取,PMS的服务框架

**使用**：更好使用adb，分析常见的安装问题，提高调试效率。为什么安装完，会kill应用程序，为何重启，谁去调用的呢。怎么调用的呢，为何安装时间那么长，我们还是可以操作设备？

**改造**代码：用PMS为逻辑，说明一些平时工作会用到的奇巧淫技，解决一些特殊的需求，特殊通道安装。

## 名词解释

### APK

Android application package ，APK文件格式扩展于JAR格式，而JAR文件格式又是扩展自ZIP格式。所以，可以使用ZIP解压工具，打开APK文件。

一个典型的APK文件内容如下：

├── AndroidManifest.xml **(1)**

├── classes.dex **(2)**

├── lib **(3)**

│ ├── armeabi-v7a

│ │ └── libhello-jni.so

├── META-INF **(4)**

│ ├── CERT.RSA

│ ├── CERT.SF

│ └── MANIFEST.MF

└── resources.arsc **(5)**

<1>：AndroidManifest.xml文件中包含了应用程序的组件，包名，版本等信息。

<2>：classes.dex是应用的可执行代码。

<3>：对于使用了ndk编程的应用来说，APK中还有lib文件夹，这个文件夹下面存放的是适配各个平台的共享库。

<4>：包含应用的签名和清单文件，这部分内容后面还会讲述。

<5>：对于可编译的资源，最后都编译好后，打包进了resources.arsc文件，比如string和style。对于编译好的APK文件，可以使用aapt工具查看资源表。

An Example aapt 用法

* aapt d resources xxx.apk可打印出资源表
* aapt d xmltree xxx.apk res/layout/main.xml 打印布局文件main.xml的内容

### PMS

**即**PackageManagerService,在framework真正执行apk安装的类。AMS

### 开机安装

也叫自动扫描安装，是开机的时候，系统扫描相关路径，完成apk的安装过程

### 手动安装

在使用过程中，用户用adb等命令主动触发的安装

## 源码来源

**Android系统版本：5.1.1，SDK=22，原生**[**源码**](http://androidxref.com/)

## 相关目录/文件

### APK路径

/system/framework/framework-res.apk： 保存的是系统的资源；为SystemUI.apk等提供资源

/system/priv-app ： 系统自带的应用程序；如何将一个应用编译时生成到pri-app中呢？需要在Android.mk中做如下设置即可：LOCAL\_PRIVILEGED\_MODULE := true

/system/app ：系统预装应用程序目录，Browser\ LatinIME\PackageInstaller\RkExplorer\SystemUpgrade

获得adb root权限才能删除

root@zs600c:/system # ls -al | grep app

drwxr-xr-x root root 2017-12-04 10:41 app

drwxr-xr-x root root 2017-12-04 10:39 priv-app

~~/vendor/app 目录是保存设备厂商提供的应用程序。~~

/oem/app

~~扫描" data\app-private"目录，即安装DRM保护的APK文件~~

/data/app ：用户apk安装的目录。安装时把apk文件复制到此目录

### APP数据

/data/data ：存放应用程序的数据

/data/dalvik-cache：将apk中的dex文件安装到dalvik-cache目录下(dex文件是dalvik虚拟机的可执行文件,当然，ART–Android Runtime的可执行文件格式为oat，启用ART时，系统会执行dex文件转换至oat文件)

/data/system/packages.xml：通过它可以看到系统安装的所有软件包.记录了系统的permissions，以及每个apk的name,codePath,flags,ts,version,uesrid，签名和权限等信息，这些信息主要通过apk的AndroidManifest.xml解析获取，解析完apk后将更新信息写入这个文件并保存到flash，下次开机直接从里面读取相关信息添加到内存相关列表中。当有apk升级，安装或删除时会更新这个文件。

### /system/etc/permissions/

读取当前Android设备的硬件特性和设定的相关权限。

## 安装APK

因此我们从文件角度给apk安装下个定义

1. 安装过程：复制APK安装包到data/app目录下，解压并扫描安装包，把dex文件(Dalvik字节码)保存到**dalvik-cache**目录，并data/data目录下创建对应的应用数据目录。
2. 卸载过程：删除安装过程中在上述三个目录下创建的文件及目录。

# 安装方式

## 概述

安装方式分类，分为两大类，自动安装和手动安装

手动安装分为：显示安装(UI安装)，静默安装系统应用商店安装

|  |  |  |
| --- | --- | --- |
| **手动安装** | **弹窗UI安装**  packageinstaller.apk | App自升级  资源管理器、应用宝豌豆荚、浏览器 |
|  | **系统商店安装** | **静默安装，**华为自带应用商店 |
|  | **PM安装** | **静默安装，**AS/adb |
| **自动安装**  **开机安装** | 开机时加载指定系统路径下的APK |  |

一般而言，Android应用安装按照使用工具的不同，可以有如下四种方式：

1. UI安装：第三方应用安装，通过调起packageinstaller完成，比如第三方market的应用宝豌豆荚、文件浏览器（SD卡的双击APK文件安装）、app自升级。有安装界面，系统默认已经安装了一个安装卸载应用的程序，即由packageinstaller.apk应用处理安装及卸载过程的界面
2. 系统APP商店安装：通过各种market应用完成，没有安装界面；
3. PM安装：直接/间接调用系统命令pm，AS/adb
4. 自动扫描安装：系统应用，开机时加载指定系统路径下的APK，没有安装界面；

## ui (packageinstaller)安装

本质上是通过调用系统应用packageinstaller.apk调用PMS完成。

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。

### Manifest.XML

|  |
| --- |
| <**manifest   package="com.android.packageinstaller"**>  <**uses-permission android:name="android.permission.INSTALL\_PACKAGES"** />   <**activity android:name=".PackageInstallerActivity"**>  <**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**> </**activity**>  </**manifest**> |

可以总结一些信息：

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**

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); |
| Adb shell am start -a android.intent.action.VIEW -t application/vnd.android.package-archive -c android.intent.category.DEFAULT -d file:///mnt/sdcard/t.apk |





## 系统商店安装

系统签名，调用pms的方法即可，同pm方法

## pm安装

### pm 基本使用

直接/间接调用系统命令pm，(AS安装)和adb install（最终也是调用pm）

pm.jar的源码和pm脚本位于同一目录下面。

pm命令安装应用，最后调用PMS的installPackageAsUser

实际上走的流程和PackageInstaller差不多。

但是他们也有不同的地方，pm命令可以指定加密解密参数，而PackageInstaller无法指定

通过键入pm我们可以看见他的用法,主要包括以下几个方面

* list => 列出安装的APK/权限组/机器硬件功能(NFC/蓝牙/WIFI)/设备支持库
* path => 查询指定apk的路径
* install => APK安装
* dump => 获取系统中的信息(电量/activity/server)
* clear => 清除APK数据
* hide/unhide => 隐藏与恢复应用(被隐藏应用在应用管理中变得不可见，桌面图标也会消失)
* enable/disable => 禁用和启用应用
* grant/revoke => 赋予/撤销应用权限

|  |
| --- |
| $ adb push \app\build\outputs\apk\app-debug.apk /data/local/tmp/system.launcher  $ adb shell pm install -r "/data/local/tmp/system.launcher"  pkg: /data/local/tmp/system.launcher  Failure [INSTALL\_FAILED\_VERSION\_DOWNGRADE]  $ adb shell pm uninstall system.launcher |

### pm 命令源码分析

其入口函数源文件

frameworks\base\cmds\pm\src\com\android\commands\pm\ pm.java

其中\system\framework\pm.jar 包管理库

包管理脚本 \system\bin\pm 解析

showUsage就是使用方法

|  |
| --- |
| private static void showUsage() {  System.err.println("usage: pm [list|path|install|uninstall]");  System.err.println(" pm list packages [-f]");  System.err.println(" pm list permission-groups");  System.err.println(" pm list permissions [-g] [-f] [-d] [-u] [GROUP]");  System.err.println(" pm list instrumentation [-f] [TARGET-PACKAGE]");  System.err.println(" pm list features");  System.err.println(" pm path PACKAGE");  System.err.println(" pm install [-l] [-r] [-t] [-i INSTALLER\_PACKAGE\_NAME] [-s] [-f] PATH");  System.err.println(" pm uninstall [-k] PACKAGE");  System.err.println(" pm enable PACKAGE\_OR\_COMPONENT");  System.err.println(" pm disable PACKAGE\_OR\_COMPONENT");  System.err.println(" pm setInstallLocation [0/auto] [1/internal] [2/external]");  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*省略\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  } |

中执行函数run(args),这里我们可以看看其中一部分Pm list package 。

**public** **int** run(String[] args) **throws** IOException, RemoteException {

**boolean** validCommand = **false**;

**if** (args.length < 1) {

**return** showUsage();

}

mUm = IUserManager.Stub.asInterface(ServiceManager.getService("user"));

mPm = IPackageManager.Stub.asInterface(ServiceManager.getService("package"));

**if** (mPm == **null**) {

System.err.println(PM\_NOT\_RUNNING\_ERR);

**return** 1;

}

mInstaller = mPm.getPackageInstaller();

mArgs = args;

String op = args[0];

mNextArg = 1;

**if** ("list".equals(op)) {

**return** runList();

}

*//....其他命令*

}

上面的代码通过进程间通信获取的PackageManagaerServer,然后运行具体的指令,下面看看list指令是怎么获取到数据的。

*/\*\**

*\* Execute the list sub-command.*

*\**

*\* pm list [package | packages]*

*\* pm list permission-groups*

*\* pm list permissions*

*\* pm list features*

*\* pm list libraries*

*\* pm list instrumentation*

*\*/*

**private** **int** runList() {

String type = nextArg();

**if** (type == **null**) {

System.err.println("Error: didn't specify type of data to list");

**return** 1;

}

**if** ("package".equals(type) || "packages".equals(type)) {

**return** runListPackages(**false**);

} **else** **if** ("permission-groups".equals(type)) {

**return** runListPermissionGroups();

} **else** **if** ("permissions".equals(type)) {

**return** runListPermissions();

} **else** **if** ("features".equals(type)) {

**return** runListFeatures();

} **else** **if** ("libraries".equals(type)) {

**return** runListLibraries();

} **else** **if** ("instrumentation".equals(type)) {

**return** runListInstrumentation();

} **else** **if** ("users".equals(type)) {

**return** runListUsers();

} **else** {

System.err.println("Error: unknown list type '" + type + "'");

**return** 1;

}

}

这里没有实质性的操作,只是分发list的子指令,下面我们看package指令的实现逻辑。

*/\**

*\* Lists all the installed packages.*

*\*/*

**private** **int** runListPackages(**boolean** showApplicationPackage) {

**int** getFlags = 0;

**boolean** listDisabled = **false**, listEnabled = **false**;

**boolean** listSystem = **false**, listThirdParty = **false**;

**boolean** listInstaller = **false**;

*//A user id constant to indicate the "owner" user of the device*

**int** userId = UserHandle.USER\_OWNER;

**try** {

String opt;

**while** ((opt=nextOption()) != **null**) {

**if** (opt.equals("-l")) {

*// old compat*

} **else** **if** (opt.equals("-lf")) {

showApplicationPackage = **true**;

} **else** **if** (opt.equals("-f")) {

showApplicationPackage = **true**;

} **else** **if** (opt.equals("-d")) {

listDisabled = **true**;

} **else** **if** (opt.equals("-e")) {

listEnabled = **true**;

} **else** **if** (opt.equals("-s")) {

listSystem = **true**;

} **else** **if** (opt.equals("-3")) {

listThirdParty = **true**;

} **else** **if** (opt.equals("-i")) {

listInstaller = **true**;

} **else** **if** (opt.equals("--user")) {

userId = Integer.parseInt(nextArg());

} **else** **if** (opt.equals("-u")) {

getFlags |= PackageManager.GET\_UNINSTALLED\_PACKAGES;

} **else** {

System.err.println("Error: Unknown option: " + opt);

**return** 1;

}

}

} **catch** (RuntimeException ex) {

System.err.println("Error: " + ex.toString());

**return** 1;

}

String filter = nextArg();

**try** {

*//重点关注*

**final** List<PackageInfo> packages = getInstalledPackages(mPm, getFlags, userId);

**int** count = packages.size();

**for** (**int** p = 0 ; p < count ; p++) {

PackageInfo info = packages.get(p);

**if** (filter != **null** && !info.packageName.contains(filter)) {

**continue**;

}

**final** **boolean** isSystem =

(info.applicationInfo.flags&ApplicationInfo.FLAG\_SYSTEM) != 0;

**if** ((!listDisabled || !info.applicationInfo.enabled) &&

(!listEnabled || info.applicationInfo.enabled) &&

(!listSystem || isSystem) &&

(!listThirdParty || !isSystem)) {

System.out.print("package:");

**if** (showApplicationPackage) {

System.out.print(info.applicationInfo.sourceDir);

System.out.print("=");

}

System.out.print(info.packageName);

**if** (listInstaller) {

System.out.print(" installer=");

System.out.print(mPm.getInstallerPackageName(info.packageName));

}

System.out.println();

}

}

**return** 0;

} **catch** (RemoteException e) {

System.err.println(e.toString());

System.err.println(PM\_NOT\_RUNNING\_ERR);

**return** 1;

}

}

上面的代码一方面是添加过滤参数,然后获取所有安装的APK,在根据过滤参数展示对应的APK,而我们最为关心的就是获取所有APK的过程.到底是怎么获取的呢, 我们接着往下看。

@SuppressWarnings("unchecked")

**private** List<PackageInfo> getInstalledPackages(IPackageManager pm, **int** flags, **int** userId)

**throws** RemoteException {

ParceledListSlice<PackageInfo> slice = pm.getInstalledPackages(flags, userId);

**return** slice.getList();

}

到此,我们发现其实获取APK列表的过程是由IPackageManager来实现的

安装时候会调用 runInstall（）方法

|  |
| --- |
| private void runInstall() {  while ((opt=nextOption()) != null) {  if (opt.equals("-l")) {  installFlags |= PackageManager.INSTALL\_FORWARD\_LOCK;  } else **if (opt.equals("-r")) {**  **installFlags |= PackageManager.INSTALL\_REPLACE\_EXISTING;**  }else if (opt.equals("-s")) {  // Override if -s option is specified.  installFlags |= PackageManager.INSTALL\_EXTERNAL;  }  }  PackageInstallObserver obs = new PackageInstallObserver();  try {  **mPm.installPackage(Uri.fromFile(new File(apkFilePath)), obs, installFlags,**  **installerPackageName);**  } |



1. adb install –r A.apk
2. 执行原理？tmp目录作用，INSTALL\_FAILED\_ILLEGITIMATE\_APK各种安装报错原因，cnt的作用，为啥可以。

## 开机安装

系统启动时SystemServer组件启动PMS，会扫描.各个目录下的安装包

## 小结

主要有几种方式，这里先来看看下面两种:

1：系统启动后扫描安装，会调用PackageManagerService的scanPackageLI函数，

2：应用市场安装，应用市场下载后会默认调用PackageManagerService的intallPackage函数，该函数最终也会调用到scanPackageLI，因此只需要分析第二种

# 源码分析-自动安装

APK安装过程：通过PMS，主要完成两件事：1、解析这个应用程序的配置文件 AndroidManifest.xml , 获取它的安装信息,如4大组件； 2、为这个应用程序分配 Linux 用户 ID 和用户组 ID,以便它可以在系统中获得合适的运行权限。

## 如何获取实例

其中的 因为有

mPm = IpackageManager.Stub.asInterface(ServiceManager.getService("package"));

Stub是接口IPackageManage的静态抽象类，asInterface是返回IPackageManager代理的静态方法。因为class PMS extends IPackageManager.Stub所以mPm.installPackage 调用

我们知道在android中，安装应用是由PackageManager来管理的，但是我们发现PackageManager是一个抽象类，他的installPackage方法也没有具体的实现。那在安装过程中是怎么执行的呐？

调用方 查看代码可以知道ApplicationPackageManager是直接继承自PackageManager的，所以最终代码会调用ApplicationPackageManager下的installPackage(Uri packageURI, IPackageInstallObserver observer, int flags,String installerPackageName)，而在installPackage里面又调用了installCommon。

installCommon的实现如下：

private void installCommon(Uri packageURI,

PackageInstallObserver observer, int flags, String installerPackageName,

VerificationParams verificationParams, ContainerEncryptionParams encryptionParams) {

if (!"file".equals(packageURI.getScheme())) {

throw new UnsupportedOperationException("Only file:// URIs are supported");

}

if (encryptionParams != null) {

throw new UnsupportedOperationException("ContainerEncryptionParams not supported");

}

final String originPath = packageURI.getPath();

try {

mPM.installPackage(originPath, observer.getBinder(), flags, installerPackageName,

verificationParams, null);

} catch (RemoteException ignored) {

}

}

6

可以看到在installCommon最终调用了mPm.installPackage那mPm又是什么？可以发现mPM是一个IPackageManager，他在是ApplicationPackageManager的构造函数中传入的，那是什么时候调用构造函数的呐？在ContextImpl中调用getPackageManager时会进行调用，传入的pm在是ActivityThread中获取的。

@Override

public PackageManager getPackageManager() {

if (mPackageManager != null) {

return mPackageManager;

}

IPackageManager pm = ActivityThread.getPackageManager();

if (pm != null) {

// Doesn't matter if we make more than one instance.

return (mPackageManager = new ApplicationPackageManager(this, pm));

}

return null;

}

我们再来看看ActivityThread.getPackageManager()的获取过程，它的代码如下：

public static IPackageManager getPackageManager() {

if (sPackageManager != null) {

//Slog.v("PackageManager", "returning cur default = " + sPackageManager);

return sPackageManager;

}

IBinder b = ServiceManager.getService("package");

//Slog.v("PackageManager", "default service binder = " + b);

sPackageManager = IPackageManager.Stub.asInterface(b);

//Slog.v("PackageManager", "default service = " + sPackageManager);

return sPackageManager;

}

从上面可以看到IPackageManager是进程间通信的客户端， 首先是IPackageManager是通过IPackageManager.aidl文件生成，同时生成了存根类IPackageManager.Stub，代理类：IPackageManager.Stub.Proxy，他是IBinder类型，那远端又是谁呐？远端就是PackageManagerService，PackageManagerService继承自IPackageManager.Stub，因此最终的调用都是通过aidl由PackageManagerService执行。因此我们主要来看看PackageManagerService中的执行过程。

## 设计框架

Android应用没有权限启动linux程序，同样的也无法主动从zygote fork出一个子进程来执行自身代码，那一个app安装后，如何拿到这个app的入口信息？代码文件(dex、so)以及相关资源释放目录的权限如何设置？APP运行时被准许的权限有哪些？这些都是PMS在扫描完一个app后需要确定的。

所以，扫描一个APK, 需要做的事情有：

1） 获取APP暴露的所有组件及相关数据

2） 获取APP声明和准许的权限数据

3） 生成app id，然后基于其生成的user id来作为app本地目录的访问权限控制

4） 释放代码文件，包含dex和so文件

5） 将1和2数据缓存到PMS中，供系统运行时使用。

// 开始安装应用，带LI后缀的函数执行时要带mInstallLock锁

Android的System\_server 进程启动时在启动的过程中，会启动PMS，这个服务负责扫描系统中特定的目录，找到里面的Apk为后缀的文件，然后对这些文件进解析，得到应用程序的相关信息，完成应用程序的安装过程。

不论是cmd安装，还是预装market安装，还是ui安装，最终都会调用到installPackage这个方法入口，本节单独讨论系统是如何执行这一个过程的

## 概述

Android系统在启动时，会把已安装的app重新安装一遍，所谓的“安装”就是遍历各安装目录，解析各app的AndroidManifest.xml，记录它们的安装信息，并为各app分配uid和gid。

PMS初始化过程，分为5个阶段：

1. PMS\_START阶段：

• 创建Settings对象；

• 将6类shareUserId到mSettings；

• 初始化SystemConfig；

• 创建名为“PackageManager”的handler线程mHandlerThread;

• 创建UserManagerService多用户管理服务；

• 通过解析4大目录中的xmL文件构造共享mSharedLibraries；

2. PMS\_SYSTEM\_SCAN\_START阶段：

• mSharedLibraries共享库中的文件执行dexopt操作；

• system/framework目录中满足条件的apk或jar文件执行dexopt操作；

• 扫描系统apk;

3. PMS\_DATA\_SCAN\_START阶段：

• 扫描/data/app目录下的apk;

• 扫描/data/app-private目录下的apk;

4. PMS\_SCAN\_END阶段：

• 将上述信息写回/data/system/packages.xml;

5. PMS\_READY阶段：

• 创建服务PackageInstallerService；

到这里，大致介绍完了整个PMS构造函数的流程，基本上PMS\_SCAN\_END阶段我们apk就算安装完成了，那么接下来我们单独看看其中几个比较重要的模块：

• Settings

• SystemConfig - readPermissions

• scanPackageLI

## PMS启动入口

在SystemServer的startBootstrapServices方法中获得启动pms，通过pms的main方法获得其实例。

[/frameworks/base/services/core/java/com/android/server/pm/PMS.java]

1. **public** **static** PMS main(Context context, Installer installer,
2. **boolean** factoryTest, **boolean** onlyCore) {
3. PMS m = **new** PMS(context, installer, factoryTest, onlyCore);
4. ServiceManager.addService("package", m);
5. **return** m;
6. }

Main方法比较简单，就是实例化了一个pms对象，然后将服务对象注册到ServiceManager中，服务名字为”package”，通过命令adb shell service list列出系统所有注册服务中，可以找到package服务。

C:\Users\key.guan>adb shell service list | findstr package

80 package: [android.content.pm.IPackageManager]

注意：pms比ams晚启动，但比ams提前SystemReady。

PMS是系统启动的时候由SystemServer组件启动的，扫描系统中特定的目录的APk，然后对这些文件进解析，得到应用程序的相关信息。这一过程都在PMS的构造函数完成。

PMS构造函数里面，在每个阶段开始的时候，都会往Eventlog里面打Tag的代码段，记录时间，比如：EventLog.writeEvent(EventLogTags.BOOT\_PROGRESS\_PMS\_START, SystemClock.uptimeMillis());

类似的，总共分为以下5个阶段：

* + 1. BOOT\_PROGRESS\_PMS\_START,
    2. BOOT\_PROGRESS\_PMS\_SYSTEM\_SCAN\_START,
    3. BOOT\_PROGRESS\_PMS\_DATA\_SCAN\_START,
    4. BOOT\_PROGRESS\_PMS\_SCAN\_END,
    5. BOOT\_PROGRESS\_PMS\_READY,

实际的输出日志为：

|  |  |
| --- | --- |
| 1 | root@ag406:/ # logcat -b events -v time | grep boot\_progress\_pms  01-21 16:56:25.485 I/boot\_progress\_pms\_start( 2005): 8270045  01-21 16:56:25.536 I/boot\_progress\_pms\_system\_scan\_start( 2005): 8270096  01-21 16:56:25.798 I/boot\_progress\_pms\_data\_scan\_start( 2005): 8270358  01-21 16:56:25.801 I/boot\_progress\_pms\_scan\_end( 2005): 8270361  01-21 16:56:25.843 I/boot\_progress\_pms\_ready( 2005): 8270403 |

接下来分别说说这几个阶段

## PMS\_START

1. mLazyDexOpt = "eng".equals(SystemProperties.get("ro.build.type"));
2. mMetrics = **new** DisplayMetrics();//DisplayMetrics是一个描述界面显示，尺寸，分辨率，密度的
3. mSettings = **new** Settings(context);
4. mSettings.addSharedUserLPw("android.uid.system", Process.SYSTEM\_UID,
5. ApplicationInfo.FLAG\_SYSTEM, ApplicationInfo.PRIVATE\_FLAG\_PRIVILEGED);
6. mSettings.addSharedUserLPw("android.uid.phone", RADIO\_UID,
7. ApplicationInfo.FLAG\_SYSTEM, ApplicationInfo.PRIVATE\_FLAG\_PRIVILEGED);
8. mSettings.addSharedUserLPw("android.uid.log", LOG\_UID,
9. ApplicationInfo.FLAG\_SYSTEM, ApplicationInfo.PRIVATE\_FLAG\_PRIVILEGED);
10. mSettings.addSharedUserLPw("android.uid.nfc", NFC\_UID,
11. ApplicationInfo.FLAG\_SYSTEM, ApplicationInfo.PRIVATE\_FLAG\_PRIVILEGED);
12. mSettings.addSharedUserLPw("android.uid.bluetooth", BLUETOOTH\_UID,
13. ApplicationInfo.FLAG\_SYSTEM, ApplicationInfo.PRIVATE\_FLAG\_PRIVILEGED);
14. mSettings.addSharedUserLPw("android.uid.shell", SHELL\_UID,
15. ApplicationInfo.FLAG\_SYSTEM, ApplicationInfo.PRIVATE\_FLAG\_PRIVILEGED);
16. String separateProcesses = SystemProperties.get("debug.separate\_processes");

/ 获取debug.separate\_processes属性

// 如果设置了这个属性，那么会强制应用程序组件在自己的进程中运行。

// 一般情况下不会设置这个属性



if (separateProcesses != null && separateProcesses.length() > 0) {

// 所有process都设置这个属性

if ("\*".equals(separateProcesses)) {

mDefParseFlags = PackageParser.PARSE\_IGNORE\_PROCESSES;

mSeparateProcesses = null;

Slog.w(TAG, "Running with debug.separate\_processes: \* (ALL)");

}

// 个别的process设置这个属性

else {

mDefParseFlags = 0;

mSeparateProcesses = separateProcesses.split(",");

Slog.w(TAG, "Running with debug.separate\_processes: "

+ separateProcesses);

}

} else { // 不设置这个属性,一般情况下会走这

mDefParseFlags = 0;

mSeparateProcesses = null;

}

// 获取默认的显示信息，保存到mMetrics

getDefaultDisplayMetrics(context, mMetrics);

// 获取系统配置信息

SystemConfig systemConfig = SystemConfig.getInstance();

mGlobalGids = systemConfig.getGlobalGids();

mSystemPermissions = systemConfig.getSystemPermissions();

mAvailableFeatures = systemConfig.getAvailableFeatures();

1. }

实例化mSettings后，添加system, radio, log, nfc, bluetooth, shell  6种SharedUserSettings到mSettings。

### Settings(context,”/data”)

Settings是Android的全局管理者，用于协助PMS保存所有的安装包信息。Settings这个类包含所有安装后的apk信息，里面保存了一个mPackages映射表，根据apk包名映射对应的apk包信息，比如permissions权限信息 ，name, codePath, mSharedLibraries, restrictions, userid, version等等，这些信息将保存到一个名为 packages的XML文件中，pms服务启动时，如果packages.xml文件存在，那么会先读里面的内容初始化Settings实例，随后packages.xml文件里面的内容会随着apk安装信息的更新而更新。

上面主要就是新建一个Setting对象,然后调用函数addSharedUserLPw(...),在Setting的构造函数中主要就是为上文说的/data/system/packages.xml等文件的创建和赋权限.

Settings类结构如图所示



#### **Settings的构造方法**

Settings类的构造方法如下，主要创建data/system目录下的多个配置文件，例如packages.xml。

[/frameworks/base/services/core/java/com/android/server/pm/Settings.java]

1. Settings(Context context) {
2. **this**(context, Environment.getDataDirectory());
3. }
5. Settings(Context context, File dataDir) {
6. mSystemDir = **new** File(dataDir, "system"); //目录/data/system/
7. mSystemDir.mkdirs();
8. FileUtils.setPermissions(mSystemDir.toString(),
9. FileUtils.S\_IRWXU|FileUtils.S\_IRWXG
10. |FileUtils.S\_IROTH|FileUtils.S\_IXOTH,
11. -1, -1);
12. mSettingsFilename = **new** File(mSystemDir, "packages.xml");
13. //目录/data/system/ packages.xml
14. mBackupSettingsFilename = **new** File(mSystemDir, "packages-backup.xml");
15. mPackageListFilename = **new** File(mSystemDir, "packages.list");
16. FileUtils.setPermissions(mPackageListFilename, 0640, SYSTEM\_UID, PACKAGE\_INFO\_GID);
18. // Deprecated: Needed for migration
19. mStoppedPackagesFilename = **new** File(mSystemDir, "packages-stopped.xml");
20. mBackupStoppedPackagesFilename = **new** File(mSystemDir, "packages-stopped-backup.xml");
21. }

//创建data/system目录

//创建data/system/packages.xml文件

//创建data/system/pacakges-backup.xml文件

//创建data/system/packages.list文件

//创建data/system/packages-stopped.xml文件

//创建data/system/packages-stopped-backup.xml文件

#### 配置文件package.xml

/data/system/packages.xml通过它可以看到系统安装的所有软件包，以及软件包的信息

系统自带的软件能升级（即安装在系统分区system中的包，如电话，短信），可以升级，如果升级/system/app目录中的包，PackageManagerServer.java对此情况进行处理，被升级的包出现package.xml的**updated-package**字段中，新的包信息会写在package字段中，卸载新包后，原包会恢复到package字段中。启动时新的包会优先地被启动

|  |
| --- |
| **<updated-package** name="com.android.providers.settings" codePath="/system/priv-app/SettingsProvider" ft="15e3b5e10b0" it="15e2e387ad8" ut="15e3b5e10b0" version="22" nativeLibraryPath="/system/priv-app/SettingsProvider/lib" primaryCpuAbi="armeabi-v7a" sharedUserId="1000">  <perms />  **</updated-package>** |

### mSettings.addSharedUserLPw

Settings实例化后，调用Settings的addSharedUserLPw方法添加6个系统的sharedUser，保存在Settings的mSharedUsers数组中。下图是SharedUserSettings的类结构，其中SettingBase是SharedUserSetting的基类，基类中包含pkgFlags/pkgPrivateFlags/PermissionsState，另外SettingBase也是PackageSetting的基类。



现在看看函数addSharedUserLPw()的实现。

SharedUserSetting addSharedUserLPw(String name, **int** uid, **int** pkgFlags, **int** pkgPrivateFlags) {

*//ArrayMap<String, SharedUserSetting> mSharedUsers*

SharedUserSetting s = mSharedUsers.get(name);

**if** (s != **null**) {

**if** (s.userId == uid) {

**return** s;

}

PackageManagerService.reportSettingsProblem(Log.ERROR,

"Adding duplicate shared user, keeping first: " + name);

**return** **null**;

}

s = **new** SharedUserSetting(name, pkgFlags, pkgPrivateFlags);

s.userId = uid;

**if** (addUserIdLPw(uid, s, name)) {

mSharedUsers.put(name, s);

**return** s;

}

**return** **null**;

}

该函数主要就是共享UID,例如有的系统应用会有

android:sharedUserId="android.uid.system"

而根据上面的设置就是该APK的UID为Process.SYSTEM\_UID,从而达到共享系统UID的目的.而下面调用的函数addUserIdLPw(...)就是保存该UID和对应的ShareUserSetting 。

Process.java

/\*\*

\* Defines the UID/GID under which system code runs.

\*/

public static final int SYSTEM\_UID = 1000;

在开机PMS初始化的时候，将该name为“android.uid.system”的uid归为1000的system用户ID

### SystemConfig

接着看PackageMangerService的构造函数的下一部分。

#### 功能介绍

SystemConfig systemConfig = SystemConfig.getInstance();

mGlobalGids = systemConfig.getGlobalGids();

mSystemPermissions = systemConfig.getSystemPermissions();

mAvailableFeatures = systemConfig.getAvailableFeatures();

负责读取/etc/permission目录下面的配置文件。这些配置文件中保存的信息有：系统支持的硬件，比如是否支持wifi，gps等；权限映射关系。

描述系统支持的硬件特性的文件，一般满足这样的命名规范：android.hardware.XXX.xml，XXX代表硬件模块名。下面是 samsung manta 的wifi特性文件——android.hardware. wifi.xml的内容：

<permissions>

<feature name="android.hardware.wifi" />

</permissions>

读取出来的feature保存在HashMap中：// were read from the etc/permissions.xml file.

final HashMap<String, FeatureInfo> mAvailableFeatures =

new HashMap<String, FeatureInfo>();

可以查询指定的feature系统是否支持,以及获得所有系统支持的feature.

frameworks/base/core/java/android/content/pm

public abstract FeatureInfo[] getSystemAvailableFeatures();

public abstract boolean hasSystemFeature(String name);

设备目录/etc/permissions下面的特性文件来自于哪里呢？它们实际上是在编译的时候打包到system.image文件中。比如上面的samsung manta 的nfc特性文件就是在manta的device.mk文件中将frameworks/native/data/etc/android.hardware.nfc.xml文件copy到system/etc/permissions/android.hardware.nfc.xml。

/etc/permissions目录下面还有一个非常重要的xml文件——platform.xml，这个文件中记录了Android APP权限与gid，uid的对应关系。这个文件在源码的位置：frameworks/base/data/etc。在这个目录下面还有一个Android.mk文件，负责将platform.xml编译到system镜像中：

LOCAL\_PATH := $(my-dir)

########################

include $(CLEAR\_VARS)

LOCAL\_MODULE := platform.xml

LOCAL\_MODULE\_CLASS := ETC

# This will install the file in /system/etc/permissions

#

LOCAL\_MODULE\_PATH := $(TARGET\_OUT\_ETC)/permissions

LOCAL\_SRC\_FILES := $(LOCAL\_MODULE)

include $(BUILD\_PREBUILT)

上面的例子也给我们提供了另一种参考：如何将配置文件编译到system/ect/permissions中。

下面是platform.xml文件中的部分内容：

<?xml version="1.0" encoding="utf-8"?>

<permissions>

<permission name="android.permission.INTERNET" >

<group gid="inet" />

</permission>

<permission name="android.permission.READ\_LOGS" >

<group gid="log" />

</permission>

<permission name="android.permission.READ\_EXTERNAL\_STORAGE" >

<group gid="sdcard\_r" />

</permission>

<permission name="android.permission.WRITE\_EXTERNAL\_STORAGE" >

<group gid="sdcard\_r" />

<group gid="sdcard\_rw" />

</permission>

</permissions>

platform.xml中主要有三块内容：

* 将APP framework中的权限和底层的gid映射。当APP获得某个权限之后，会获得这个gid所具备的权限。
* 将APP framework的权限赋予某个系统级别的进程。这样这个进程就可以获得操作APP framework资源的。
* jar库文件的映射。APP中通过指定链接的jar库名，通过这层映射关系，可以在链接的找到正确的jar库。

总结：

readPermissions方法读取/ect/permissions目录下的xml文件，并为读取的结果生成相应的数据结构

#### SystemConfig()源码分析

SystemConfig在构造函数里对一些目录进行读取,这些目录包括

~~/system/etc/sysconfig~~

/system/etc/permissions

~~/oem/etc/sysconfig~~

~~/oem/etc/permissions~~

然后解析这些目录下面的文件,这些文件的作用是声明当前设备的功能(NFC/WIFI)等.下面我们看看解析函数

**private** **void** readPermissionsFromXml(File permFile, **boolean** onlyFeatures) {

FileReader permReader = **null**;

**try** {

permReader = **new** FileReader(permFile);

} **catch** (FileNotFoundException e) {

Slog.w(TAG, "Couldn't find or open permissions file " + permFile);

**return**;

}

**final** **boolean** lowRam = ActivityManager.isLowRamDeviceStatic();

**try** {

XmlPullParser parser = Xml.newPullParser();

parser.setInput(permReader);

**int** type;

**while** ((type=parser.next()) != parser.START\_TAG

&& type != parser.END\_DOCUMENT) {

;

}

**if** (type != parser.START\_TAG) {

**throw** **new** XmlPullParserException("No start tag found");

}

*//要求根结点为permissions OR config*

**if** (!parser.getName().equals("permissions") && !parser.getName().equals("config")) {

**throw** **new** XmlPullParserException("Unexpected start tag in " + permFile

+ ": found " + parser.getName() + ", expected 'permissions' or 'config'");

}

*//遍历XML,并将不同的功能保存到不同的ArraySet<>中*

**while** (**true**) {

XmlUtils.nextElement(parser);

**if** (parser.getEventType() == XmlPullParser.END\_DOCUMENT) {

**break**;

}

String name = parser.getName();

**if** ("group".equals(name) && !onlyFeatures) {

String gidStr = parser.getAttributeValue(**null**, "gid");

**if** (gidStr != **null**) {

**int** gid = android.os.Process.getGidForName(gidStr);

mGlobalGids = appendInt(mGlobalGids, gid);

} **else** {

Slog.w(TAG, "<group> without gid in " + permFile + " at "

+ parser.getPositionDescription());

}

XmlUtils.skipCurrentTag(parser);

**continue**;

} **else** **if** ("permission".equals(name) && !onlyFeatures) {

String perm = parser.getAttributeValue(**null**, "name");

**if** (perm == **null**) {

Slog.w(TAG, "<permission> without name in " + permFile + " at "

+ parser.getPositionDescription());

XmlUtils.skipCurrentTag(parser);

**continue**;

}

perm = perm.intern();

readPermission(parser, perm);

} **else** **if** ("assign-permission".equals(name) && !onlyFeatures) {

String perm = parser.getAttributeValue(**null**, "name");

**if** (perm == **null**) {

Slog.w(TAG, "<assign-permission> without name in " + permFile + " at "

+ parser.getPositionDescription());

XmlUtils.skipCurrentTag(parser);

**continue**;

}

String uidStr = parser.getAttributeValue(**null**, "uid");

**if** (uidStr == **null**) {

Slog.w(TAG, "<assign-permission> without uid in " + permFile + " at "

+ parser.getPositionDescription());

XmlUtils.skipCurrentTag(parser);

**continue**;

}

**int** uid = Process.getUidForName(uidStr);

**if** (uid < 0) {

Slog.w(TAG, "<assign-permission> with unknown uid \""

+ uidStr + " in " + permFile + " at "

+ parser.getPositionDescription());

XmlUtils.skipCurrentTag(parser);

**continue**;

}

perm = perm.intern();

ArraySet<String> perms = mSystemPermissions.get(uid);

**if** (perms == **null**) {

perms = **new** ArraySet<String>();

mSystemPermissions.put(uid, perms);

}

perms.add(perm);

XmlUtils.skipCurrentTag(parser);

} **else** **if** ("library".equals(name) && !onlyFeatures) {

String lname = parser.getAttributeValue(**null**, "name");

String lfile = parser.getAttributeValue(**null**, "file");

**if** (lname == **null**) {

Slog.w(TAG, "<library> without name in " + permFile + " at "

+ parser.getPositionDescription());

} **else** **if** (lfile == **null**) {

Slog.w(TAG, "<library> without file in " + permFile + " at "

+ parser.getPositionDescription());

} **else** {

*//Log.i(TAG, "Got library " + lname + " in " + lfile);*

mSharedLibraries.put(lname, lfile);

}

XmlUtils.skipCurrentTag(parser);

**continue**;

} **else** **if** ("feature".equals(name)) {

String fname = parser.getAttributeValue(**null**, "name");

**boolean** allowed;

**if** (!lowRam) {

allowed = **true**;

} **else** {

String notLowRam = parser.getAttributeValue(**null**, "notLowRam");

allowed = !"true".equals(notLowRam);

}

**if** (fname == **null**) {

Slog.w(TAG, "<feature> without name in " + permFile + " at "

+ parser.getPositionDescription());

} **else** **if** (allowed) {

*//Log.i(TAG, "Got feature " + fname);*

FeatureInfo fi = **new** FeatureInfo();

fi.name = fname;

mAvailableFeatures.put(fname, fi);

}

XmlUtils.skipCurrentTag(parser);

**continue**;

} **else** **if** ("unavailable-feature".equals(name)) {

String fname = parser.getAttributeValue(**null**, "name");

**if** (fname == **null**) {

Slog.w(TAG, "<unavailable-feature> without name in " + permFile + " at "

+ parser.getPositionDescription());

} **else** {

mUnavailableFeatures.add(fname);

}

XmlUtils.skipCurrentTag(parser);

**continue**;

} **else** **if** ("allow-in-power-save-except-idle".equals(name) && !onlyFeatures) {

String pkgname = parser.getAttributeValue(**null**, "package");

**if** (pkgname == **null**) {

Slog.w(TAG, "<allow-in-power-save-except-idle> without package in "

+ permFile + " at " + parser.getPositionDescription());

} **else** {

mAllowInPowerSaveExceptIdle.add(pkgname);

}

XmlUtils.skipCurrentTag(parser);

**continue**;

} **else** **if** ("allow-in-power-save".equals(name) && !onlyFeatures) {

String pkgname = parser.getAttributeValue(**null**, "package");

**if** (pkgname == **null**) {

Slog.w(TAG, "<allow-in-power-save> without package in " + permFile + " at "

+ parser.getPositionDescription());

} **else** {

mAllowInPowerSave.add(pkgname);

}

XmlUtils.skipCurrentTag(parser);

**continue**;

} **else** **if** ("fixed-ime-app".equals(name) && !onlyFeatures) {

String pkgname = parser.getAttributeValue(**null**, "package");

**if** (pkgname == **null**) {

Slog.w(TAG, "<fixed-ime-app> without package in " + permFile + " at "

+ parser.getPositionDescription());

} **else** {

mFixedImeApps.add(pkgname);

}

XmlUtils.skipCurrentTag(parser);

**continue**;

} **else** **if** ("app-link".equals(name)) {

String pkgname = parser.getAttributeValue(**null**, "package");

**if** (pkgname == **null**) {

Slog.w(TAG, "<app-link> without package in " + permFile + " at "

+ parser.getPositionDescription());

} **else** {

mLinkedApps.add(pkgname);

}

XmlUtils.skipCurrentTag(parser);

} **else** {

XmlUtils.skipCurrentTag(parser);

**continue**;

}

}

} **catch** (XmlPullParserException e) {

Slog.w(TAG, "Got exception parsing permissions.", e);

} **catch** (IOException e) {

Slog.w(TAG, "Got exception parsing permissions.", e);

} **finally** {

IoUtils.closeQuietly(permReader);

}

**for** (String fname : mUnavailableFeatures) {

**if** (mAvailableFeatures.remove(fname) != **null**) {

Slog.d(TAG, "Removed unavailable feature " + fname);

}

}

}

上面代码很容易理解,解析xml节点的数据到具体的List或Set中,在PMS的构造函数中就取出了mGlobalGids/mAvailableFeatures/mSystemPermissions出来,分别对应的TAG节点为

**<group** gid="" **></group>**

**<feature></feature>**

**<assign-permission></assign-permission>**

### ServiceThread

下面接着分析PackageManagerService的构造函数，这个是接收“手动安装”事件的，PackageHandler 会发送外ServiceThread是要接收外部安装请求的的

// 建立PackageHandler消息循环，用于处理外部的安装请求等消息

// 比如如adb install、packageinstaller安装APK时

mHandlerThread = **new** ServiceThread(TAG,

Process.THREAD\_PRIORITY\_BACKGROUND, **true** */\*allowIo\*/*); *//HandlerThread*

mHandlerThread.start();

mHandler = **new** PackageHandler(mHandlerThread.getLooper());

Watchdog.getInstance().addThread(mHandler, WATCHDOG\_TIMEOUT);

### SystemConfig保存至PMS的Settings

File dataDir = Environment.getDataDirectory(); *//dataDir = "/data/"*

mAppDataDir = **new** File(dataDir, "data"); *// = "/data/data"*

mAppInstallDir = **new** File(dataDir, "app"); *// = "/data/app"*

mAppLib32InstallDir = **new** File(dataDir, "app-lib"); *// = "/data/app-lib"*

mAsecInternalPath = **new** File(dataDir, "app-asec").getPath(); *// = "data/app-asec"*

mUserAppDataDir = **new** File(dataDir, "user"); *//= "/data/user"*

mDrmAppPrivateInstallDir = **new** File(dataDir, "app-private"); *// = "/data/app-private"*

sUserManager = **new** UserManagerService(context, **this**,

mInstallLock, mPackages);

*// Propagate permission configuration in to package manager.*

*//合并SystemConfig读取的permission到Settings下*

ArrayMap<String, SystemConfig.PermissionEntry> permConfig

= systemConfig.getPermissions(); *//permission标签下的*

**for** (**int** i=0; i<permConfig.size(); i++) {

SystemConfig.PermissionEntry perm = permConfig.valueAt(i);

BasePermission bp = mSettings.mPermissions.get(perm.name);

**if** (bp == **null**) {

bp = **new** BasePermission(perm.name, "android", BasePermission.TYPE\_BUILTIN);

mSettings.mPermissions.put(perm.name, bp);

}

**if** (perm.gids != **null**) {

bp.setGids(perm.gids, perm.perUser);

}

}

*//从SystemConfig中读取的libs存入共享库*

ArrayMap<String, String> libConfig = systemConfig.getSharedLibraries();

**for** (**int** i=0; i<libConfig.size(); i++) {

mSharedLibraries.put(libConfig.keyAt(i),

**new** SharedLibraryEntry(libConfig.valueAt(i), **null**));

}

mFoundPolicyFile = SELinuxMMAC.readInstallPolicy();

上面的代码主要就是通过上面的SystemConfig获取到的信息存在在Settings和PMS内部。

### mSettings.readLPw解析packages.xml

在读取完权限文件之后，PMS会在其构造函数中调用Settings的readLPw方法，读取应用包的设置文件。

*//这里会读取前面说的/data/system/packages.xml文件以及他的备份文件*

*//这里特别说明下，会把解析的application存放到mSetting的mPackages中，后面会用到*

mRestoredSettings = mSettings.readLPw(**this**, sUserManager.getUsers(**false**),

mSdkVersion, mOnlyCore);

String customResolverActivity = Resources.getSystem().getString(

R.string.config\_customResolverActivity);

**if** (TextUtils.isEmpty(customResolverActivity)) {

customResolverActivity = **null**;

} **else** {

mCustomResolverComponentName = ComponentName.unflattenFromString(

customResolverActivity);

}

**long** startTime = SystemClock.uptimeMillis();

EventLog.writeEvent(EventLogTags.BOOT\_PROGRESS\_PMS\_SYSTEM\_SCAN\_START,

startTime);

*// Set flag to monitor and not change apk file paths when*

*// scanning install directories.*

**final** **int** scanFlags = SCAN\_NO\_PATHS | SCAN\_DEFER\_DEX | SCAN\_BOOTING | SCAN\_INITIAL;

*//已经dexopt的apk存放位置*

**final** ArraySet<String> alreadyDexOpted = **new** ArraySet<String>();

*/\*\**

*\* Add everything in the in the boot class path to the*

*\* list of process files because dexopt will have been run*

*\* if necessary during zygote startup.*

*\*/*

**final** String bootClassPath = System.getenv("BOOTCLASSPATH");

**final** String systemServerClassPath = System.getenv("SYSTEMSERVERCLASSPATH");

*//系统库类不要优化 可以通过echo $BOOTCLASSPATH查看*

**if** (bootClassPath != **null**) {

String[] bootClassPathElements = splitString(bootClassPath, ':');

**for** (String element : bootClassPathElements) {

alreadyDexOpted.add(element);

}

} **else** {

Slog.w(TAG, "No BOOTCLASSPATH found!");

}

**if** (systemServerClassPath != **null**) {

String[] systemServerClassPathElements = splitString(systemServerClassPath, ':');

**for** (String element : systemServerClassPathElements) {

alreadyDexOpted.add(element);

}

} **else** {

Slog.w(TAG, "No SYSTEMSERVERCLASSPATH found!");

}

* /data/system/packages.xml
* /data/system/packages-backup.xml
* /data/system/packages.list
* /data/system/users/userid/package-restrictions.xml

对于packages.xml和packages.list在之前已经简单的介绍过了，packages-backup.xml是packages.xml的备份文件。在每次写packages.xml文件的时候，都会将旧的packages.xml文件先备份，这样做是为了防止写文件过程中文件以外损坏，还能从旧的文件中恢复。

readLPw方法负责读取packages.xml文件。它的逻辑是如果存在packages-backup.xml,就认为packages.xml已经损坏，将之删除。然后从packages-backup.xml中读取信息，用读取的信息构造一个PackageSetting对象，然后以包名为key，PackageSetting为value，保存在HashMap中。

final HashMap<String, PackageSetting> mPackages =

new HashMap<String, PackageSetting>();

现在可以总结下readLPw的执行过程：

* 读取packages.xml文件
* 调用readPackageRestrictionsLPr方法，读取package-restrictions.xml文件

上面代码开始时解析packages.xml,该文件保存了系统内安装了的APK的信息,然后就是添加一些库类到alreadyDexOpted这个List里面,目的是以后做dexopt的时候跳过这些不必要的优化。这里说明一下packages.xml中的字段的保存位置(以下是在Settings.java中)。

**package** -> mPackages(readPackageLPw()-->addPackageLPw()) *//重点*

permissions -> mPermissions(readPermissionsLPw())

permission-trees -> mPermissionTrees(readPermissionsLPw())

shared-user -> mSharedUsers(readSharedUserLPw()-->addSharedUserLPw())

updated-**package** -> mDisabledSysPackages(readDisabledSysPackageLPw()) *//这个标签是在OTA中添加的？删除也会有这个标记？*

renamed-**package** -> mRenamedPackages

disabled的程序安装升级是什么效果？系统升级是什么效果？

disabled系统app通过pm升级之后，会变成enable。。。空间会变大

卸载更新也不行。。

系统升级update.zip

系统升级，禁用是所有，init

Diable之后就

### 小结

* 构造DisplayMetrics类：描述界面显示，尺寸，分辨率，密度。构造完后并获取默认的信息保存到变量mMetrics中。
* 构造Settings类：这个是Android的全局管理者，用于协助PMS保存所有的安装包信息
* 保存Installer对象
* 获取系统配置信息：SystemConfig构造函数中会通过readPermissions()解析指定目录下的所有xml文件,然后把这些信息保存到systemConfig中，涉及的目录有如下：
  + /system/etc/sysconfig
  + /system/etc/permissions
  + /oem/etc/sysconfig
  + /oem/etc/permissions
* 创建名为PackageManager的handler线程，建立PackageHandler消息循环，用于处理外部的安装请求等消息
* 创建data下的各种目录，比如data/app, data/app-private等。
* 创建用户管理服务UserManagerService
* 把systemConfig关于xml中的标签所指的动态库保存到mSharedLibraries
* Settings.readLPw扫描解析packages.xml和packages-backup.xml

补充说明下**debug.separate\_processes**这个属性：  
这个属性你可以使用强制应用程序组件在自己的进程中运行，有两种方法可以使用这个：

|  |  |
| --- | --- |
| 1  2  3  4 | // 所有的进程都会受到影响  setprop debug.separate\_processes  // 指定进程受影响  setprop debug.separate\_processes“com.google.process.content, com.google.android.samples” |

这个属性一般不会用到。

## SCAN\_START

当mOnlyCore = false时，则scanDirLI()还会收集如下目录中的apk

• /data/app

• /data/app-private

### 扫描顺序

这个地方可以考虑并行扫描，加快启动速度，当然要考虑时序问题，比如同一个apk同一个版本在不同目录，目录的先后顺序就很重要了，因为系统的私有目录的app权限是很大的。

### mInstaller.dexopt

这里强调一下，会把package的信息存在Settings.mPackages中，并根据package标签下的installStatus字段判断app安装的状态，这在后面有用到。下面继续看PMS的构造函数。

// alreadyDexOpted该集合中存放的是已经优化或者不需要优先的文件

/将环境变量BOOTCLASSPATH所执行的文件加入alreadyDexOpted

//将环境变量SYSTEMSERVERCLASSPATH所执行的文件加入alreadyDexOpted

//添加以下两个文件添加到已优化集合

alreadyDexOpted.add(frameworkDir.getPath() + "/framework-res.apk");

alreadyDexOpted.add(frameworkDir.getPath() + "/core-libart.jar");

*//通过命令getprop ro.product.cpu.abilist查看设备支持的指令集*

**final** List<String> allInstructionSets = InstructionSets.getAllInstructionSets();

**final** String[] dexCodeInstructionSets =

getDexCodeInstructionSets(

allInstructionSets.toArray(**new** String[allInstructionSets.size()]));

*/\*\**

*\* Ensure all external libraries have had dexopt run on them.*

*\*/*

**if** (mSharedLibraries.size() > 0) {

*// NOTE: For now, we're compiling these system "shared libraries"*

*// (and framework jars) into all available architectures. It's possible*

*// to compile them only when we come across an app that uses them (there's*

*// already logic for that in scanPackageLI) but that adds some complexity.*

**for** (String dexCodeInstructionSet : dexCodeInstructionSets) {

**for** (SharedLibraryEntry libEntry : mSharedLibraries.values()) {

**final** String lib = libEntry.path;

**if** (lib == **null**) {

**continue**;

}

**try** {

**int** dexoptNeeded = DexFile.getDexOptNeeded(lib, **null**, dexCodeInstructionSet, **false**);

**if** (dexoptNeeded != DexFile.NO\_DEXOPT\_NEEDED) {

alreadyDexOpted.add(lib);

mInstaller.dexopt(lib, Process.SYSTEM\_UID, **true**, dexCodeInstructionSet, dexoptNeeded);

}

} **catch** (FileNotFoundException e) {

Slog.w(TAG, "Library not found: " + lib);

} **catch** (IOException e) {

Slog.w(TAG, "Cannot dexopt " + lib + "; is it an APK or JAR? "

+ e.getMessage());

}

}

}

}

这段代码主要就是执行dexopt的过程,并将优化过的apk/jar放入alreadyDexOpted,这里mInstaller内部调用installd守护进程完成dexopt

### alreadyDexOpted.add frameworkDir

File frameworkDir = **new** File(Environment.getRootDirectory(), "framework");

*// Gross hack for now: we know this file doesn't contain any*

*// code, so don't dexopt it to avoid the resulting log spew.*

alreadyDexOpted.add(frameworkDir.getPath() + "/framework-res.apk");

*// Gross hack for now: we know this file is only part of*

*// the boot class path for art, so don't dexopt it to*

*// avoid the resulting log spew.*

alreadyDexOpted.add(frameworkDir.getPath() + "/core-libart.jar");

*/\*\**

*\* There are a number of commands implemented in Java, which*

*\* we currently need to do the dexopt on so that they can be*

*\* run from a non-root shell.*

*\*/*

String[] frameworkFiles = frameworkDir.list();

**if** (frameworkFiles != **null**) {

*// TODO: We could compile these only for the most preferred ABI. We should*

*// first double check that the dex files for these commands are not referenced*

*// by other system apps.*

**for** (String dexCodeInstructionSet : dexCodeInstructionSets) {

**for** (**int** i=0; i<frameworkFiles.length; i++) {

File libPath = **new** File(frameworkDir, frameworkFiles[i]);

String path = libPath.getPath();

*// Skip the file if we already did it.*

**if** (alreadyDexOpted.contains(path)) {

**continue**;

}

*// Skip the file if it is not a type we want to dexopt.*

**if** (!path.endsWith(".apk") && !path.endsWith(".jar")) {

**continue**;

}

**try** {

**int** dexoptNeeded = DexFile.getDexOptNeeded(path, **null**, dexCodeInstructionSet, **false**);

**if** (dexoptNeeded != DexFile.NO\_DEXOPT\_NEEDED) {

mInstaller.dexopt(path, Process.SYSTEM\_UID, **true**, dexCodeInstructionSet, dexoptNeeded);

}

} **catch** (FileNotFoundException e) {

Slog.w(TAG, "Jar not found: " + path);

} **catch** (IOException e) {

Slog.w(TAG, "Exception reading jar: " + path, e);

}

}

}

}

这段代码和上面的实现几乎一样,读取/system/framework/下的几个目录的jar包和apk,判断是否需要进行优化,值得注意的是并没有加入列表alreadyDexOpted 。

### scanDirLI

到了这一步，才算真正的开始安装apk了

*// Collect vendor overlay packages.*

*// (Do this before scanning any apps.)*

*// For security and version matching reason, only consider*

*// overlay packages if they reside in VENDOR\_OVERLAY\_DIR.*

File vendorOverlayDir = **new** File(VENDOR\_OVERLAY\_DIR);*// = "/vendor/overlay"*

scanDirLI(vendorOverlayDir, PackageParser.PARSE\_IS\_SYSTEM

| PackageParser.PARSE\_IS\_SYSTEM\_DIR, scanFlags | SCAN\_TRUSTED\_OVERLAY, 0);

*// Find base frameworks (resource packages without code).*

scanDirLI(frameworkDir, PackageParser.PARSE\_IS\_SYSTEM *//system/framework*

| PackageParser.PARSE\_IS\_SYSTEM\_DIR

| PackageParser.PARSE\_IS\_PRIVILEGED,

scanFlags | SCAN\_NO\_DEX, 0);

*// Collected privileged system packages.*

**final** File privilegedAppDir = **new** File(Environment.getRootDirectory(), "priv-app");

scanDirLI(privilegedAppDir, PackageParser.PARSE\_IS\_SYSTEM

| PackageParser.PARSE\_IS\_SYSTEM\_DIR

| PackageParser.PARSE\_IS\_PRIVILEGED, scanFlags, 0);

*// Collect ordinary system packages.*

**final** File systemAppDir = **new** File(Environment.getRootDirectory(), "app");

scanDirLI(systemAppDir, PackageParser.PARSE\_IS\_SYSTEM

| PackageParser.PARSE\_IS\_SYSTEM\_DIR, scanFlags, 0);

*// Collect all vendor packages.*

File vendorAppDir = **new** File("/vendor/app");

**try** {

vendorAppDir = vendorAppDir.getCanonicalFile();

} **catch** (IOException e) {

*// failed to look up canonical path, continue with original one*

}

scanDirLI(vendorAppDir, PackageParser.PARSE\_IS\_SYSTEM

| PackageParser.PARSE\_IS\_SYSTEM\_DIR, scanFlags, 0);

*// Collect all OEM packages.*

**final** File oemAppDir = **new** File(Environment.getOemDirectory(), "app");

scanDirLI(oemAppDir, PackageParser.PARSE\_IS\_SYSTEM

| PackageParser.PARSE\_IS\_SYSTEM\_DIR, scanFlags, 0);

**if** (DEBUG\_UPGRADE) Log.v(TAG, "Running installd update commands");

mInstaller.moveFiles(); *//执行LocalStock发送movefiles命令*

上面的代码的逻辑就是扫描指定的目录，这里的目录包括下面这些

/vendor/overlay

/system/framework

/system/priv-app

/system/app

/vendor/app

/oem/app

也就是我们上文提到的系统APK的存放目录，扫描结束之后会把apk信息存放在mPackages(这里是PMS,区别于Settings的mPackages)。

### scanDirLI

下面我们来分析scanDirLI()的过程。

**private** **void** scanDirLI(File dir, **int** parseFlags, **int** scanFlags, **long** currentTime) {

**final** File[] files = dir.listFiles();

**if** (ArrayUtils.isEmpty(files)) {

Log.d(TAG, "No files in app dir " + dir);

**return**;

}

**if** (DEBUG\_PACKAGE\_SCANNING) {

Log.d(TAG, "Scanning app dir " + dir + " scanFlags=" + scanFlags

+ " flags=0x" + Integer.toHexString(parseFlags));

}

*//遍历该目录下的APK*

**for** (File file : files) {

**final** **boolean** isPackage = (isApkFile(file) || file.isDirectory())

&& !PackageInstallerService.isStageName(file.getName());

**if** (!isPackage) {

*// Ignore entries which are not packages*

**continue**;

}

**try** {

scanPackageLI(file, parseFlags | PackageParser.PARSE\_MUST\_BE\_APK,

scanFlags, currentTime, **null**);

} **catch** (PackageManagerException e) {

Slog.w(TAG, "Failed to parse " + file + ": " + e.getMessage());

*// Delete invalid userdata apps*

*// 删除无效的用户APK*

**if** ((parseFlags & PackageParser.PARSE\_IS\_SYSTEM) == 0 &&

e.error == PackageManager.INSTALL\_FAILED\_INVALID\_APK) {

logCriticalInfo(Log.WARN, "Deleting invalid package at " + file);

**if** (file.isDirectory()) {

mInstaller.rmPackageDir(file.getAbsolutePath());

} **else** {

file.delete();

}

}

}

}

}

上面代码核心就是遍历指定的文件夹，对文件夹内部的文件执行函数

### scanPackageLI

scanPackageLI(File,...) ，通过其注释我们了解到他是扫描包的。它的代码也比较长，下面选择其中一部分说明。

**private** PackageParser.Package scanPackageLI(File scanFile,

**int** parseFlags, **int** scanMode, **long** currentTime) {

......

String scanPath = scanFile.getPath();

parseFlags |= mDefParseFlags;

PackageParser pp = **new** PackageParser();

......

**final** PackageParser.Package pkg = pp.parsePackage(scanFile,

scanPath, mMetrics, parseFlags);

......

**return** scanPackageLI(pkg, parseFlags, scanMode | SCAN\_UPDATE\_SIGNATURE, currentTime);

}

为指定的文件创建PackageParser,将解析结果存入Package ,最后在调用函数scanPackageLI(Package,...)。

### PackageParser.Package.parsePackage-> parseMonolithicPackage

而在函数PackageParser.Package.parsePackage(...)中会判断scanFile是文件还是目录(Android分包)，会对他们做不同的处理，我们这里简单点，就看是文件的分支，当时文件时，会调用函数parseMonolithicPackage(packageFile, flags) ，下面分析这个函数。

**public** Package parseMonolithicPackage(File apkFile, **int** flags){

**final** AssetManager assets = **new** AssetManager();

**final** Package pkg = parseBaseApk(apkFile, assets, flags);

pkg.codePath = apkFile.getAbsolutePath();

**return** pkg;

}

这里的核心就是函数parseBaseApk(File,...)，根据名称感觉有些明朗了,不就解析APK嘛，看看到底是怎么实现的吧。

### PackageParser. parseBaseApk

**private** Package parseBaseApk(File apkFile, AssetManager assets, **int** flags){

....

Resources res = **null**;

XmlResourceParser parser = **null**;

res = **new** Resources(assets, mMetrics, **null**);

assets.setConfiguration(0, 0, **null**, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

Build.VERSION.RESOURCES\_SDK\_INT);

parser = assets.openXmlResourceParser(cookie, "AndroidManifest.xml");

**final** String[] outError = **new** String[1];

**final** Package pkg = parseBaseApk(res, parser, flags, outError);

...

**return** pkg;

}

卧槽，又是圈套，有调函数parseBaseApk(Resources,...)来解析，不过上面我们已经看见关键的AndroidManifest.xml已经出现了。通过阅读parseBaseApk(Resources,...)，我们发现他会解析AndroidManifest.xml中的一部分文件，这里大体包括以下标签

- application

- overlay

- key-sets

- permission-group

- permission-tree

- uses-permission

- uses-permission-sdk-m | uses-permission-sdk-23

- uses-configuration

- uses-feature

- feature-group

- uses-sdk

- supports-screens

- protected-broadcast

- instrumentation

- original-package

- adopt-permissions

- uses-gl-texture

- compatible-screens

- supports-input

- eat-comment

惭愧，好多标签没见过，查看官网发现官网并没有列举以上全部[AndroidManifest](https://developer.android.com/guide/topics/manifest/manifest-intro.html" \t "_blank) 。我们这里继续跟进application标签，发现他调用函数parseBaseApplication()。这个函数就是对Application内部四大组件进行解析。

### PackageParser .parseBaseApplication()

#### parsePackageItemInfo

hook应用的报名

outInfo.labelRes就好了！！！

我们这里选取activity的部分来看看。

*// 函数参数 Package owner*

**if** (tagName.equals("activity")) {

*//class Activity extends Component<ActivityIntentInfo>*

Activity a = parseActivity(owner, res, parser, attrs, flags, outError, **false**,

owner.baseHardwareAccelerated);

**if** (a == **null**) {

mParseError = PackageManager.INSTALL\_PARSE\_FAILED\_MANIFEST\_MALFORMED;

**return** **false**;

}

owner.activities.add(a);

}

### parseActivity()-RK

parseActivity()就是解析activity标签下的内容，比如Activity的Theme什么的，解析的过程主要是利用TypedArray,具体的属性可以看看[Activity](https://developer.android.com/guide/topics/manifest/activity-element.html" \t "_blank),很多属性都是存储在一个flags标记了,这样减少了类中成员字段过多，这在Android中使用的比较多,比如View中很多属性也是存在一个flags中。RK在此处留了一个bug

### scanPackageDirtyLI(),-app\_cnt，privader？

到此一个解析好了的Package就好了，不知不觉，已经偏了十万八千里，不要急，拉回来，上面我们讲到scanPackageLI(File,...)的最后调用了scanPackageLI(Package,...),那么这个函数有是做什么的呢？这个函数调用了scanPackageDirtyLI()。

这个函数的代码量也是相当吓人，这里不打算具体分析，主要工作就是调用mInstaller为app创建目录，也就是/data/data/pkname/这个目录，还有就是APK对应的libs的存放位置，App签名验证，收集APK要的权限，最重要的就是把解析信息存放到了PMS的mPackages变量中，意味着App安装成功了，后面会用到这个变量。

### possiblyDeletedUpdatedSystemApps

回到PackageManagerService的构造函数中来。

*// Prune any system packages that no longer exist.*

**final** List<String> possiblyDeletedUpdatedSystemApps = **new** ArrayList<String>();

**if** (!mOnlyCore) {

*//mSettings.mPackages来自与package.xml的package标签*

Iterator<PackageSetting> psit = mSettings.mPackages.values().iterator();

**while** (psit.hasNext()) {

PackageSetting ps = psit.next();

*/\**

*\* If this is not a system app, it can't be a*

*\* disable system app.*

*\*/*

**if** ((ps.pkgFlags & ApplicationInfo.FLAG\_SYSTEM) == 0) {

**continue**;

}

*/\**

*\* If the package is scanned, it's not erased.*

*\*/*

*//PMS的mPackages存放扫描过的APK*

**final** PackageParser.Package scannedPkg = mPackages.get(ps.name);

*//扫描到了 && packages.xml中存在*

**if** (scannedPkg != **null**) {

*/\**

*\* If the system app is both scanned and in the*

*\* disabled packages list, then it must have been*

*\* added via OTA. Remove it from the currently*

*\* scanned package so the previously user-installed*

*\* application can be scanned.*

*\*/*

*//package.xml的package标签和updated-package标签都包含这个pkg*

*//根据上面注释，意味着这个APK是通过OTA添加的，暂时移除*

**if** (mSettings.isDisabledSystemPackageLPr(ps.name)) {

logCriticalInfo(Log.WARN, "Expecting better updated system app for "

+ ps.name + "; removing system app. Last known codePath="

+ ps.codePathString + ", installStatus=" + ps.installStatus

+ ", versionCode=" + ps.versionCode + "; scanned versionCode="

+ scannedPkg.mVersionCode);

removePackageLI(ps, **true**);*//mPackages.remove(ps.name);*

mExpectingBetter.put(ps.name, ps.codePath);

}

**continue**;

}

*//没有扫描到，在package标签下，但不在updated-package标签下，说明该APP已经不存在了*

*//因此要删掉他的目录*

*//直接从mSettings.mPackages中移除*

**if** (!mSettings.isDisabledSystemPackageLPr(ps.name)) {

psit.remove();

logCriticalInfo(Log.WARN, "System package " + ps.name

+ " no longer exists; wiping its data");

removeDataDirsLI(**null**, ps.name);

} **else** {

*//没有扫描到，在package标签下，也在updated-package标签下*

*//可能由OTA引入（或删除？）*

**final** PackageSetting disabledPs = mSettings.getDisabledSystemPkgLPr(ps.name);

**if** (disabledPs.codePath == **null** || !disabledPs.codePath.exists()) {

possiblyDeletedUpdatedSystemApps.add(ps.name);

}

}

}

}

处理被用户隐藏的APP(前面讲的pm hide package),因为被隐藏的APP在package.xml还存在,这里就是把这些APP从保存他们的列表中移除。另外就是在package.xml中还有该APK,但是扫描系统目录发现这个APK已经不存在了的处理方式。执行完之后mSetting.mPackages剩下的就是无效的APK，我们需要将这些清除，于是就有了下面的几行代码

*//look for any incomplete package installations*

ArrayList<PackageSetting> deletePkgsList = mSettings.getListOfIncompleteInstallPackagesLPr();*//获取mSettings.mPackages中installStatus为未成功安装的App(前文有讲，package中installStatus=false的apk)*

*//clean up list*

**for**(**int** i = 0; i < deletePkgsList.size(); i++) {

*//clean up here*

cleanupInstallFailedPackage(deletePkgsList.get(i));

}

*//delete tmp files*

deleteTempPackageFiles();

*// Remove any shared userIDs that have no associated packages*

mSettings.pruneSharedUsersLPw();*//移除没有被关联的mSharedUsers*

上面的作用就是清除无效APK引入的文件夹等。系统APK装载完了，下面就开始装载用户APK,

### 小结

PMS\_SYSTEM\_SCAN\_START阶段主要做了如下工作：

• 首先将BOOTCLASSPATH，SYSTEMSERVERCLASSPATH这两个环境变量下的路径加入到不需要dex优化集合alreadyDexOpted中

• SYSTEMSERVERCLASSPATH：主要包括/system/framework目录下services.jar，ethernet-service.jar，wifi-service.jar这3个文件。

• BOOTCLASSPATH：该环境变量内容较多，不同ROM可能有所不同，常见内容包含/system/framework目录下的framework.jar，ext.jar，core-libart.jar，telephony-common.jar，ims-common.jar，core-junit.jar等文件。

• 获取共享库mSharedLibraries，判断是否需要dex优化，如果需要则进行dex优化，并加入到alreadyDexOpted列表中

• 添加framework-res.apk、core-libart.jar两个文件添加到已优化集合alreadyDexOpted中

• 将framework目录下，其他的apk或者jar，进行dex优化并加入已优化集合alreadyDexOpted中

• scanDirLI(): 扫描指定目录下的apk文件，最终调用PackageParser.parseBaseApk来完成AndroidManifest.xml文件的解析，生成Application, activity,service,broadcast, provider等信息

• 删除系统不存在的包 removePackageLI

• 清理安装失败的包 cleanupInstallFailedPackage

• 删除临时文件 deleteTempPackageFiles

• 移除不相干包中的所有共享userID

## DATA\_SCAN\_START

scanDirLI(mAppInstallDir, 0, scanFlags | SCAN\_REQUIRE\_KNOWN, 0);

scanDirLI(mDrmAppPrivateInstallDir, PackageParser.PARSE\_FORWARD\_LOCK,

scanFlags | SCAN\_REQUIRE\_KNOWN, 0);

上面的代码和前面扫描系统APK是一样的，这是目录和flags变了，逻辑是一样的。这里的目录包括

/data/app

/data/app-**private：为空的**

继续看构造函数。

*/\*\**

*\* Remove disable package settings for any updated system*

*\* apps that were removed via an OTA. If they're not a*

*\* previously-updated app, remove them completely.*

*\* Otherwise, just revoke their system-level permissions.*

*\*/*

*//在引进了用户app之后mPackages内容增加了，再看看是否有这些app*

**for** (String deletedAppName : possiblyDeletedUpdatedSystemApps) {

PackageParser.Package deletedPkg = mPackages.get(deletedAppName);

mSettings.removeDisabledSystemPackageLPw(deletedAppName);

String msg;

**if** (deletedPkg == **null**) { *//OTA删除*

msg = "Updated system package " + deletedAppName

+ " no longer exists; wiping its data";

removeDataDirsLI(**null**, deletedAppName);

} **else** { *//在用户app中找到了，当然会移除系统包标识*

msg = "Updated system app + " + deletedAppName

+ " no longer present; removing system privileges for "

+ deletedAppName;

deletedPkg.applicationInfo.flags &= ~ApplicationInfo.FLAG\_SYSTEM;

PackageSetting deletedPs = mSettings.mPackages.get(deletedAppName);

deletedPs.pkgFlags &= ~ApplicationInfo.FLAG\_SYSTEM;

}

logCriticalInfo(Log.WARN, msg);

}

*/\*\**

*\* Make sure all system apps that we expected to appear on*

*\* the userdata partition actually showed up. If they never*

*\* appeared, crawl back and revive the system version.*

*\*/*

**for** (**int** i = 0; i < mExpectingBetter.size(); i++) { *//有新包，更新APK*

**final** String packageName = mExpectingBetter.keyAt(i);

**if** (!mPackages.containsKey(packageName)) {

**final** File scanFile = mExpectingBetter.valueAt(i);

logCriticalInfo(Log.WARN, "Expected better " + packageName

+ " but never showed up; reverting to system");

**final** **int** reparseFlags;

*//不同目录flags不一样*

**if** (FileUtils.contains(privilegedAppDir, scanFile)) {

reparseFlags = PackageParser.PARSE\_IS\_SYSTEM

| PackageParser.PARSE\_IS\_SYSTEM\_DIR

| PackageParser.PARSE\_IS\_PRIVILEGED;

} **else** **if** (FileUtils.contains(systemAppDir, scanFile)) {

reparseFlags = PackageParser.PARSE\_IS\_SYSTEM

| PackageParser.PARSE\_IS\_SYSTEM\_DIR;

} **else** **if** (FileUtils.contains(vendorAppDir, scanFile)) {

reparseFlags = PackageParser.PARSE\_IS\_SYSTEM

| PackageParser.PARSE\_IS\_SYSTEM\_DIR;

} **else** **if** (FileUtils.contains(oemAppDir, scanFile)) {

reparseFlags = PackageParser.PARSE\_IS\_SYSTEM

| PackageParser.PARSE\_IS\_SYSTEM\_DIR;

} **else** {

Slog.e(TAG, "Ignoring unexpected fallback path " + scanFile);

**continue**;

}

*//加入mStting.mPackages*

mSettings.enableSystemPackageLPw(packageName);

**try** {

scanPackageLI(scanFile, reparseFlags, scanFlags, 0, **null**);

} **catch** (PackageManagerException e) {

Slog.e(TAG, "Failed to parse original system package: "

+ e.getMessage());

}

}

}

mExpectingBetter.clear();

上面这段代码就是删除被OTA移除app的目录，更新新引入的App的目录。

*// Now that we know all of the shared libraries, update all clients to have*

*// the correct library paths.*

updateAllSharedLibrariesLPw(); *//为需要sharelibs的apk关联libs,放在pkg.usesLibraryFiles*

**for** (SharedUserSetting setting : mSettings.getAllSharedUsersLPw()) {

*// NOTE: We ignore potential failures here during a system scan (like*

*// the rest of the commands above) because there's precious little we*

*// can do about it. A settings error is reported, though.*

adjustCpuAbisForSharedUserLPw(setting.packages, **null** */\* scanned package \*/*,

**false** */\* force dexopt \*/*, **false** */\* defer dexopt \*/*);

}

*// Now that we know all the packages we are keeping,*

*// read and update their last usage times.*

mPackageUsage.readLP();*//读/data/system/package-usage.list*

## PMS\_SCAN\_END

* 当sdk版本不一致时，需要更新权限
* 当这是ota后的首次启动，正常启动则需要清除目录的缓存代码
* 当权限和其他默认项都完成更新，则清理相关信息
* 信息写回packages.xml文件

EventLog.writeEvent(EventLogTags.BOOT\_PROGRESS\_PMS\_SCAN\_END,

SystemClock.uptimeMillis());

Slog.i(TAG, "Time to scan packages: "

+ ((SystemClock.uptimeMillis()-startTime)/1000f)

+ " seconds");

*// If the platform SDK has changed since the last time we booted,*

*// we need to re-grant app permission to catch any new ones that*

*// appear. This is really a hack, and means that apps can in some*

*// cases get permissions that the user didn't initially explicitly*

*// allow... it would be nice to have some better way to handle*

*// this situation.*

**int** updateFlags = UPDATE\_PERMISSIONS\_ALL;

**if** (ver.sdkVersion != mSdkVersion) {

Slog.i(TAG, "Platform changed from " + ver.sdkVersion + " to "

+ mSdkVersion + "; regranting permissions for internal storage");

updateFlags |= UPDATE\_PERMISSIONS\_REPLACE\_PKG | UPDATE\_PERMISSIONS\_REPLACE\_ALL;

}

updatePermissionsLPw(**null**, **null**, updateFlags);*//Apk分配权限*

ver.sdkVersion = mSdkVersion;

*// clear only after permissions have been updated*

mExistingSystemPackages.clear();

mPromoteSystemApps = **false**;

*// If this is the first boot, and it is a normal boot, then*

*// we need to initialize the default preferred apps.*

*//第一次启动，初始化默认程序，如浏览器，email程序*

**if** (!mRestoredSettings && !onlyCore) {

mSettings.applyDefaultPreferredAppsLPw(**this**, UserHandle.USER\_OWNER);

applyFactoryDefaultBrowserLPw(UserHandle.USER\_OWNER);

primeDomainVerificationsLPw(UserHandle.USER\_OWNER);

}

*// If this is first boot after an OTA, and a normal boot, then*

*// we need to clear code cache directories.*

**if** (mIsUpgrade && !onlyCore) {

Slog.i(TAG, "Build fingerprint changed; clearing code caches");

**for** (**int** i = 0; i < mSettings.mPackages.size(); i++) {

**final** PackageSetting ps = mSettings.mPackages.valueAt(i);

**if** (Objects.equals(StorageManager.UUID\_PRIVATE\_INTERNAL, ps.volumeUuid)) {

deleteCodeCacheDirsLI(ps.volumeUuid, ps.name);

}

}

ver.fingerprint = Build.FINGERPRINT;

}

checkDefaultBrowser();

*// All the changes are done during package scanning.*

ver.databaseVersion = Settings.CURRENT\_DATABASE\_VERSION;

*// can downgrade to reader*

mSettings.writeLPr(); *//写package.xml*

EventLog.writeEvent(EventLogTags.BOOT\_PROGRESS\_PMS\_READY,

SystemClock.uptimeMillis());

mRequiredVerifierPackage = getRequiredVerifierLPr(); *//string*

mRequiredInstallerPackage = getRequiredInstallerLPr(); *//string*

mInstallerService = **new** PackageInstallerService(context, **this**); *//根据名字知道大概是app安装相关服务*

mIntentFilterVerifierComponent = getIntentFilterVerifierComponentNameLPr();

mIntentFilterVerifier = **new** IntentVerifierProxy(mContext,

mIntentFilterVerifierComponent);

} *// synchronized (mPackages)*

} *// synchronized (mInstallLock)*

## PMS\_READY

BOOT\_PROGRESS\_PMS\_READY阶段：

* 初始化PackageInstallerService
* GC回收下内存

*// Now after opening every single application zip, make sure they*

*// are all flushed. Not really needed, but keeps things nice and*

*// tidy.*

Runtime.getRuntime().gc();

*// Expose private service for system components to use.*

LocalServices.addService(PackageManagerInternal.class, **new** PackageManagerInternalImpl());

}

到此，PMS的构造函数就阅读完毕了。

## 1.1. SCAN\_START-开机安装方式

https://mr-cao.gitbooks.io/android/content/android-packagemanager.html

下面是一张流程图，描述了系统第一次开机过程中对/data/app目录扫描的过程。因为扫描是个很复杂的过程，所以在流程图中省略了一些异常的case，主要集中在扫描安装这条主线上，对于升级等流程的判断，以及签名信息不符合的判断都进行了省略。 

上图中标记红色的部分是扫描安装过程中几个关键的步骤。

* 在collectCertificate sLI方法中会对应用包的整数进行验证，检查应用包的完整性和合法性，防止应用程序包被篡改。
* mSetting.getPackageLPw方法会为Package创建对应的PackageSetting对象。这个对象中保存的信息最后会通过writeLPr写入到/data/system/packages.xml文件中去。
* createDataDirsLI方法会给installd发送消息，为应用程序创建对应的数据目录。
* copyNativeLibrariesForInternalApp会将应用程序包的动态库复制到对应的动态库目录。
* performDexOptLI方法也是调用installd发送消息，优化可执行代码。优化完毕的可执行代码存放在目录/data/dalvik-cache/目录下面。

上面提到的installd，将在下一个小节讲解。PackageManagerService扫描完APK之后，会在其内部建立复杂的数据结构。这里我们只以简单的一张类图感受下两个重要的类——PackageManagerService和Settings的关系。 

简单的介绍下上面的图：

* Package对象中的数据是从APK文件中扫描而得到，主要来源于AndroidManifest.xml文件
* PackageSetting的父类是PackageSettingBase,这个类中的数据除了第一次开机以外，都是从packages.xml或者packages-backup.xml
* Settings中以包名为key保存着所有的Package对应的设置信息；同时也保存着SharedUser相关的信息。应用程序的“记账簿”在开机过程中都会被读入到内存中，而这就是由Settings负责。

pms的启动到ready的大致流程如（图1）所示。



应用程序管理服务PMS安装应用程序的过程，其实就是解析析应用程序配置文件AndroidManifest.xml的过程，并从里面得到得到应用程序的相关信息，例如得到应用程序的组件Activity、Service、Broadcast Receiver和Content Provider等信息，有了这些信息后，通过ActivityManagerService这个服务，我们就可以在系统中正常地使用这些应用程序了。



**1.PMS.main()初始化注册**

将PMS服务初始化并注册到ServiceManager里面进行管理。

**2.建立java层的installer与c层的installd的socket联接**

建立Java层的installer与c层的installd的socket联接，使得在上层的install,remove,dexopt等功能最终由installd在底层实现；

**3.建立PackageHandler消息循环**

建立PackageHandler消息循环，用于处理外部的apk安装请求消息，如adb install,packageinstaller安装apk时会发送消息；

典型的比如INIT\_COPY和MCS\_BOUND等，在通过网络下载时候会调用。

**4. 成员变量readLp（）恢复上一次的安装信息**

由于Android每次启动的时候都需要安装一次信息，但是有些信息是保持不变的，例如Linux用户组Id，PMS 每次安装程序之后，都会把这些程序的信息保存下来，以便下次使用， 恢复上一次程序的安装信息是通过PMS 的成员变量mSetting的readLP()来实现的，恢复信息之后就开始扫描和安装app了。

检查/data/system/packages.xml是否存在，这个文件是在解析apk时由writeLP()创建的，里面记录了系统的permissions，以及每个apk的name,codePath,flags,ts,version,uesrid等信息，这些信息主要通过apk的AndroidManifest.xml解析获取，解析完apk后将更新信息写入这个文件并保存到flash，下次开机直接从里面读取相关信息添加到内存相关列表中。当有apk升级，安装或删除时会更新这个文件。

**5.jar的detopt优化**

检查**BootClassPath，mSharedLibraries**及/system/framework下的jar是否需要dexopt，需要的则通过dexopt进行优化；

**6.scanDirLI函数扫描特定目录的apk文件解析**

启动AppDirObserver线程监测**/system/framework,/system/app,/data/app,/data/app-privat**e目录的事件,主要监听add和remove事件。对于目录监听底层通过inotify机制实现，inotify 是一种文件系统的变化通知机制，如文件增加、删除等事件可以立刻让用户态得知,它为用户态监视文件系统的变化提供了强大的支持。当有add event时调用scanPackageLI(File , int , int)处理；当有remove event时调用removePackageLI()处理;

调用installer.install()进行安装工作,检查apk里的dex文件是否需要再优化,如果需要优化则通过辅助工具dexopt进行优化处理；将解析出的componet添加到pkg的对应列表里；对apk进行签名和证书校验,进行完整性验证。



**7.updatePermmisonLp函数分配权限**

这个函数为申请了特定资源访问权限的app，分配相应的用户组ID.

**8.writeLP()函数保存安装信息**

mSetting的writeLP（）将所获得应用程序的安装信息，保存在一个本地的配置文件中。以便下次安装的时候，将应用的信息回复过来。

## 总结

1.安装和卸载都是通过PackageManager，实质上是实现了PackageManager的远程服务PMS来完成具体的操作，所有细节和逻辑均可以在PMS中跟踪查看；

2.所有安装方式殊途同归，最终就回到PMS中，然后调用底层本地代码的installd来完成。

3.再看apk 的安装过程。回个我们再看apk的安装过程，主要分为如下几部

* 拷贝apk文件到指定目录
* 解压apk，拷贝文件，创建应用的数据目录
* 解析apk的AndroidManifinest.xml文件
* 向Launcher应用申请添加创建快捷方式

各个系统的版本如下:



# 源码分析-手动安装

PackageManager的installPackageWithVerificationAndEncryption最终是一个binder调用，是在PackageManagerService中负责实现。

在此方法中，获取一个Message对象，其id为INIT\_COPY，其obj为InstallParams。其中InstallParams对象根据installPackageWithVerificationAndEncryption的方法参数构造。INIT\_COPY消息是在PackageHandler中进行处理。对于此消息的处理流程为：

* 首先通过connectToService方法绑定IMediaContainerService服务；在绑定成功之后，发送MCS\_BOUND消息；IMeidaContainerService的实现是在frameworks/base/packages/DefaultContainerService/src/com/android/def/DefaultContainerService.java中。它负责原始apk文件的解密工作。
* 在PackageHandler消息中再次对MCS\_BOUND消息进行处理。这次就会调用InstallParams的startCopy方法，开启复制过程。startcopy方法又会调用handleStartCopy进行复制工作。主要做了如下工作：
  + 确定APK的安装位置
  + 检查磁盘空间是否足够
  + 以InstallParams构造一份InstallArgs对象。InstallArgs有两个子类：AsecInstallArgs和FileInstallArgs，对于安装于外部sd卡或者是有Forward Lock属性的应用，将会构造AsecInstallArgs，否则构造的是FileInstallArgs对象。
  + 检查APK是否需要验证（一般是不需要）
  + 调用InstallArgs对象的copyApk方法。

所以，handleStartCopy最重要的工作就是由InstallParams的copyApk来完成。假设在这个InstallParams引用是一个FileInstallArgs对象，那么其copy流程如下：

* 创建临时文件。对于最终安装在/data/app目录下面的应用，其临时文件的共享名为：/data/app/vmdl-XXX.tmp。这个XXX是一个随机数
* 调用IMediaContainerService的服务接口，从原始APK文件中提取出解密过的APK数据，写入到临时文件中
* 拷贝jni动态库到相应的目录

在InstallParams的startCopy处理完handleStartCopy之后，会调用handleRetureCode方法，此方法中将会进行触发真正的安装流程。

void handleReturnCode() {

if (mArgs != null) {

processPendingInstall(mArgs, mRet);

if (mTempPackage != null) {

if (!mTempPackage.delete()) {

Slog.w(TAG, "Couldn't delete temporary file: " +

mTempPackage.getAbsolutePath());

}

}

}

}

在handleReturnCode中又调用了processPendingInstall方法，这个方法主要做了以下几件事情：

* 调用FileInstallArgs对象的doPreInstall方法：如果之前的流程有错误，此方法将会执行清理动作
* 调用PackageManagerService的installPackageLI方法，开始安装过程
* 调用FileInstallArgs对象的doPostInstall方法：如果之前的流程有错误，此方法将会执行清理动作

installPackgeLI执行步骤如下：

* 从临时文件中提取出Package对象：final PackageParser.Package pkg = pp.parsePackage(tmpPackageFile, null, mMetrics, parseFlags)
* 提取证书信息，并验证
* 将临时文件更名为正式apk文件，修改动态库目录名
* 调用installNewPackageLI安装正式apk文件
  + scanPackageLI方法负责安装apk
  + updateSettingsLI负责更新“记账簿”

/\* Called when a downloaded package installation has been confirmed by the user \*/

public void installPackage(

final Uri packageURI, final IPackageInstallObserver observer, final int flags,final String installerPackageName)

这样就是从网络下载安装的入口了。



* 拷贝apk文件到指定目录

在Android系统中，apk安装文件是会被保存起来的，默认情况下，用户安装的apk首先会被拷贝到 /data/app 目录下。/data/app目录是用户有权限访问的目录，在安装apk的时候会自动选择该目录存放用户安装的文件，而系统出厂的apk文件则被放到了 /system 分区下（系统分区一般比较小3G左右）,包括 /system/app，/system/vendor/app，以及 /system/priv-app 等等，该分区只有Root权限的用户才能访问，这也就是为什么在**没有Root手机之前，我们无法删除系统出厂的app的原因了。**

|  |
| --- |
| **root@csb:/data # ls -al | grep app**  **drwxrwx--x system system 2017-09-16 09:38 app** |

* 解压apk，拷贝文件，创建应用的数据目录

为了加快app的启动速度，apk在安装的时候，会首先将app的可执行文件（dex）拷贝到 /data/dalvik-cache 目录，缓存起来。然后，在/data/data/目录下创建应用程序的数据目录（以应用的包名命名），存放应用的相关数据，如数据库、xml文件、cache、二进制的so动态库等等。

* 解析apk的AndroidManifinest.xml文件

Android系统中，也有一个类似注册表的东西，用来记录当前所有安装的应用的基本信息，每次系统安装或者卸载了任何apk文件，都会更新这个文件。这个文件位于如下目录：/data/system/packages.xml。系统在安装apk的过程中，会解析apk的AndroidManifinest.xml文件，提取出这个apk的重要信息写入到packages.xml文件中，这些信息包括：权限、应用包名、APK的安装位置、版本、userID等等。

由此，我们就知道了为啥一些**应用市场**和软件管理类的app能够很清楚地知道当前手机所安装的所有的app，以及这些**app的详细信息**了。

另外一件事就是Linux的用户Id和用户组Id，以便他可以获得合适的运行权限。以上这些都是由PackageServiceManager完成的，下面我们会重点介绍PackageServiceManager。

* 显示快捷方式

这些应用程序只是相当于在PMS服务注册好了，如果我们想要在Android桌面上看到这些应用程序，还需要有一个Home应用程序，负责从PMS服务中把这些安装好的应用程序取出来，并以友好的方式在桌面上展现出来，例如以快捷图标的形式。在Android系统中，负责把系统中已经安装的应用程序在桌面中展现出来的Home应用程序就是Launcher了

## 文件拷贝阶段

### installPackage

installPackage方法只是用当前用户安装应用，最后也会调用installPackageAsUser

@Override

public void installPackage(String originPath, IPackageInstallObserver2 observer,int installFlags, String installerPackageName, VerificationParams verificationParams,String packageAbiOverride) {

installPackageAsUser(originPath, observer, installFlags,installerPackageName, verificationParams,packageAbiOverride, UserHandle.getCallingUserId());

}

### installPackageAsUser

installPackageAsUser先检查调用进程是否有安装应用的权限，[再检查调用进程所属的用户是否有权限安装应用](file:///F:\key.guan\kgszgt\asys\%E4%B8%BA%E4%BD%95%E9%9C%80%E8%A6%81%E6%A3%80%E6%9F%A5%E7%94%A8%E6%88%B7?)，最后检查指定的用户是否被限制安装应用。如果参数installFlags带有INSTALL\_ALL\_USERS，则该应用将给系统中所有用户安装，否则只给指定用户安装。安装应用实践比较长，因此不可能在一个函数中完成。上面函数把数据保存在installParams然后发送了INIT\_COPY消息。通过PackageHandler的实例mHandler.sendMessage（msg）把信息发给继承Handler的类HandleMessage()方法会自动调用Packagemanager的安装方法installPackage（），发送消息时会传递一个InstallParams参数，InstallParams是继承自HandlerParams抽象类的，用来记录安装应用的参数。

@Override

public void installPackageAsUser(String originPath, IPackageInstallObserver2 observer,

int installFlags, String installerPackageName, VerificationParams verificationParams,

String packageAbiOverride, int userId) {

//检查调用进程的权限,比如PackageInstaller.apk这个系统应用就必须申请这个权限

mContext.enforceCallingOrSelfPermission(android.Manifest.permission.INSTALL\_PACKAGES, null);

//检查调用进程的用户是否有权限安装应用

final int callingUid = Binder.getCallingUid();

enforceCrossUserPermission(callingUid, userId, true, true, "installPackageAsUser");

//检查指定的用户是否被限制安装应用

// TODO DISALLOW\_INSTALL\_APPS 是安装黑名单

if (isUserRestricted(userId, UserManager.DISALLOW\_INSTALL\_APPS)) {

try {

if (observer != null) {

observer.onPackageInstalled("", INSTALL\_FAILED\_USER\_RESTRICTED, null, null);

}

} catch (RemoteException re) {

}

return;

}

//adb INSTALL\_FAILED\_USER\_RESTRICTED

if ((callingUid == Process.SHELL\_UID) || (callingUid == Process.ROOT\_UID)) {

installFlags |= PackageManager.INSTALL\_FROM\_ADB;

} else {

// Caller holds INSTALL\_PACKAGES permission, so we're less strict

// about installerPackageName.

installFlags &= ~PackageManager.INSTALL\_FROM\_ADB;

installFlags &= ~PackageManager.INSTALL\_ALL\_USERS;

}

//给所有用户安装

UserHandle user;

if ((installFlags & PackageManager.INSTALL\_ALL\_USERS) != 0) {

user = UserHandle.ALL;

} else {

user = new UserHandle(userId);

}

verificationParams.setInstallerUid(callingUid);

final File originFile = new File(originPath);

final OriginInfo origin = OriginInfo.fromUntrustedFile(originFile);

//保存参数到InstallParamsm,发送消息

**final Message msg = mHandler.obtainMessage(INIT\_COPY);**

msg.obj = new InstallParams(origin, observer, installFlags,

installerPackageName, verificationParams, user, packageAbiOverride);

mHandler.sendMessage(msg);

}

### doHandleMessage-INIT\_COPY

void doHandleMessage(Message msg) {

switch (msg.what) {

case INIT\_COPY: {

HandlerParams params = (HandlerParams) msg.obj;

int idx = mPendingInstalls.size();

if (DEBUG\_INSTALL) Slog.i(TAG, "init\_copy idx=" + idx + ": " + params);

// If a bind was already initiated we dont really

// need to do anything. The pending install

// will be processed later on.

if (!mBound) {

// If this is the only one pending we might

// have to bind to the service again.

if (!connectToService()) {//绑定DefaultContainerService

Slog.e(TAG, "Failed to bind to media container service");

params.serviceError();

return;

} else {//连接成功把安装信息保存到mPendingInstalls

// Once we bind to the service, the first

// pending request will be processed.

mPendingInstalls.add(idx, params);

}

} else {//如果已经绑定好了

mPendingInstalls.add(idx, params);

// Already bound to the service. Just make

// sure we trigger off processing the first request.

if (idx == 0) {

mHandler.sendEmptyMessage(MCS\_BOUND);

}

}

break;

}

INIT\_COPY消息的处理将绑定DefaultContainerService，因为这是一个异步的过程，要等待绑定的结果通过onServiceConnected返回，所以这里的安装参数放到了mPendingInstalls列表中。如果这个Service以前就绑定好了，现在就不需要再绑定，安装信息也会先放到mPendingInstalls。如果有多个安装请求同时到达，这里通过mPendingInstalls列表对他们进行排队。如果列表中只有一项，说明没有更多的安装请求，因此这种情况下回立即发出MCS\_BOUND消息。而onServiceConnected方法同样是发出MCS\_BOUND消息：

class DefaultContainerConnection implements ServiceConnection {

public void onServiceConnected(ComponentName name, IBinder service) {

if (DEBUG\_SD\_INSTALL) Log.i(TAG, "onServiceConnected");

IMediaContainerService imcs =

IMediaContainerService.Stub.asInterface(service);

mHandler.sendMessage(mHandler.obtainMessage(MCS\_BOUND, imcs));

}

public void onServiceDisconnected(ComponentName name) {

if (DEBUG\_SD\_INSTALL) Log.i(TAG, "onServiceDisconnected");

}

};

看下MCS\_BOUND的消息处理

case MCS\_BOUND: {

if (DEBUG\_INSTALL) Slog.i(TAG, "mcs\_bound");

if (msg.obj != null) {

mContainerService = (IMediaContainerService) msg.obj;

}

if (mContainerService == null) {//没有连接成功

// Something seriously wrong. Bail out

Slog.e(TAG, "Cannot bind to media container service");

for (HandlerParams params : mPendingInstalls) {

// Indicate service bind error

params.serviceError();//通知出错了

}

mPendingInstalls.clear();

} else if (mPendingInstalls.size() > 0) {

HandlerParams params = mPendingInstalls.get(0);

if (params != null) {

if (params.startCopy()) {//执行安装

// We are done... look for more work or to

// go idle.

if (DEBUG\_SD\_INSTALL) Log.i(TAG,

"Checking for more work or unbind...");

// Delete pending install

if (mPendingInstalls.size() > 0) {

mPendingInstalls.remove(0);//工作完成，删除第一项

}

if (mPendingInstalls.size() == 0) {//如果没有安装消息了，延时发送10秒MCS\_UNBIND消息

if (mBound) {

if (DEBUG\_SD\_INSTALL) Log.i(TAG,

"Posting delayed MCS\_UNBIND");

removeMessages(MCS\_UNBIND);

Message ubmsg = obtainMessage(MCS\_UNBIND);

// Unbind after a little delay, to avoid

// continual thrashing.

sendMessageDelayed(ubmsg, 10000);

}

} else {

// There are more pending requests in queue.

// Just post MCS\_BOUND message to trigger processing

// of next pending install.

if (DEBUG\_SD\_INSTALL) Log.i(TAG,

"Posting MCS\_BOUND for next work");

mHandler.sendEmptyMessage(MCS\_BOUND);//还有消息继续发送MCS\_BOUND消息

}

}

}

} else {

// Should never happen ideally.

Slog.w(TAG, "Empty queue");

}

break;

}

如果结束了我们看看MCS\_UNBIND消息的处理

case MCS\_UNBIND: {

// If there is no actual work left, then time to unbind.

if (DEBUG\_INSTALL) Slog.i(TAG, "mcs\_unbind");

if (mPendingInstalls.size() == 0 && mPendingVerification.size() == 0) {

if (mBound) {

if (DEBUG\_INSTALL) Slog.i(TAG, "calling disconnectService()");

disconnectService();//断开连接

}

} else if (mPendingInstalls.size() > 0) {

// There are more pending requests in queue.

// Just post MCS\_BOUND message to trigger processing

// of next pending install.

mHandler.sendEmptyMessage(MCS\_BOUND);

}

break;

}

MCS\_UNBIND消息的处理，如果处理的时候发现mPendingInstalls又有数据了，还是发送MCS\_BOUND消息继续安装，否则断开和DefaultContainerService的连接，安装结束。这个安装会尝试4次，超过4次就GG了  
下面我们看执行安装的函数startCopy：

final boolean startCopy() {

boolean res;

try {

if (DEBUG\_INSTALL) Slog.i(TAG, "startCopy " + mUser + ": " + this);

if (++mRetries > MAX\_RETRIES) {//重试超过4次退出

Slog.w(TAG, "Failed to invoke remote methods on default container service. Giving up");

mHandler.sendEmptyMessage(MCS\_GIVE\_UP);

handleServiceError();

return false;

} else {

handleStartCopy();

res = true;

}

} catch (RemoteException e) {

if (DEBUG\_INSTALL) Slog.i(TAG, "Posting install MCS\_RECONNECT");

mHandler.sendEmptyMessage(MCS\_RECONNECT);//安装出错，发送重新连接

res = false;

}

handleReturnCode();

return res;

}

### InstallParams.handleStartCopy

InstallParams 实现了抽象类HandlerParams

handleStartCopy()执行的工作如下：

* 判断安装标志位是否合法
* 判断安装空间是否足够
* 对安装位置的校验
* 判断是否需要对应用进行校验工作
* 如果校验成功，执行InstallArgs.copyApk()
* 如果无需校验，直接执行InstallArgs.copyApk()

handleStartCopy函数先通过DefaultContainerService调用了getMinimallPackageInfo来确定安装位置是否有足够的空间，并在PackageInfoLite对象的recommendedIntallLocation记录错误原因。发现空间不够，会调用installer的freecache方法来释放一部分空间。  
// 首先对安装的标志位进行判断，如果既有内部安装标志，又有外部安装标志，那么就设置  
//PackageManager.INSTALL\_FAILED\_INVALID\_INSTALL\_LOCATION返回值  
再接下来handleStartCopy有很长一段都在处理apk的校验，这个校验过程是通过发送Intent ACTION\_PACKAGE\_NEEDS\_VERIFICATION给系统中所有接受该Intent的应用来完成。如果无需校验，直接调用InstallArgs对象的copyApk方法。

这个方法比较长，分段来看。

ret = PackageManager.INSTALL\_SUCCEEDED

final StorageManager storage = StorageManager.from(mContext);

final long lowThreshold = storage.getStorageLowBytes(

Environment.getDataDirectory());

final long sizeBytes = mContainerService.calculateInstalledSize(

origin.resolvedPath, isForwardLocked(), packageAbiOverride);

if (mInstaller.freeCache(null, sizeBytes + lowThreshold) >= 0) {

pkgLite = mContainerService.getMinimalPackageInfo(origin.resolvedPath,

installFlags, packageAbiOverride);

}

首先，如果需要的空间不够大，就调用Install的freeCache去释放一部分缓存。这里的mContainerService对应的binder服务端实现，在DefaultContainerService中。中间经过复杂（安装位置，pkgLite.recommendedIntallLocation，安装位置的校验，installLocationPoliy策略等）的判断处理之后，创建一个InstallArgs对象，如果前面的判断结果是能安装成功的话ret=PackageManager.INSTALL\_SUCCEEDED，进入分支。  
// TODO installLocationPoliy() 是位置的策略PackageINfoLite

if (ret == PackageManager.INSTALL\_SUCCEEDED) {

/\*

\* ADB installs appear as UserHandle.USER\_ALL, and can only be performed by

\* UserHandle.USER\_OWNER, so use the package verifier for UserHandle.USER\_OWNER.

\*/

int userIdentifier = getUser().getIdentifier();

if (userIdentifier == UserHandle.USER\_ALL

&& ((installFlags & PackageManager.INSTALL\_FROM\_ADB) != 0)) {

userIdentifier = UserHandle.USER\_OWNER;

}

/\*

\* Determine if we have any installed package verifiers. If we

\* do, then we'll defer to them to verify the packages.

\*/

final int requiredUid = mRequiredVerifierPackage == null ? -1

: getPackageUid(mRequiredVerifierPackage, userIdentifier);

if (!origin.existing && requiredUid != -1

&& isVerificationEnabled(userIdentifier, installFlags)) {

final Intent verification = new Intent(

Intent.ACTION\_PACKAGE\_NEEDS\_VERIFICATION);

verification.addFlags(Intent.FLAG\_RECEIVER\_FOREGROUND);

verification.setDataAndType(Uri.fromFile(new File(origin.resolvedPath)),

PACKAGE\_MIME\_TYPE);

verification.addFlags(Intent.FLAG\_GRANT\_READ\_URI\_PERMISSION);

final List<ResolveInfo> receivers = queryIntentReceivers(verification,

PACKAGE\_MIME\_TYPE, PackageManager.GET\_DISABLED\_COMPONENTS,

0 /\* TODO: Which userId? \*/);

if (DEBUG\_VERIFY) {

Slog.d(TAG, "Found " + receivers.size() + " verifiers for intent "

+ verification.toString() + " with " + pkgLite.verifiers.length

+ " optional verifiers");

}

final int verificationId = mPendingVerificationToken++;

verification.putExtra(PackageManager.EXTRA\_VERIFICATION\_ID, verificationId);

verification.putExtra(PackageManager.EXTRA\_VERIFICATION\_INSTALLER\_PACKAGE,

installerPackageName);

verification.putExtra(PackageManager.EXTRA\_VERIFICATION\_INSTALL\_FLAGS,

installFlags);

verification.putExtra(PackageManager.EXTRA\_VERIFICATION\_PACKAGE\_NAME,

pkgLite.packageName);

verification.putExtra(PackageManager.EXTRA\_VERIFICATION\_VERSION\_CODE,

pkgLite.versionCode);

if (verificationParams != null) {

if (verificationParams.getVerificationURI() != null) {

verification.putExtra(PackageManager.EXTRA\_VERIFICATION\_URI,

verificationParams.getVerificationURI());

}

if (verificationParams.getOriginatingURI() != null) {

verification.putExtra(Intent.EXTRA\_ORIGINATING\_URI,

verificationParams.getOriginatingURI());

}

if (verificationParams.getReferrer() != null) {

verification.putExtra(Intent.EXTRA\_REFERRER,

verificationParams.getReferrer());

}

if (verificationParams.getOriginatingUid() >= 0) {

verification.putExtra(Intent.EXTRA\_ORIGINATING\_UID,

verificationParams.getOriginatingUid());

}

if (verificationParams.getInstallerUid() >= 0) {

verification.putExtra(PackageManager.EXTRA\_VERIFICATION\_INSTALLER\_UID,

verificationParams.getInstallerUid());

}

}

final PackageVerificationState verificationState = new PackageVerificationState(

requiredUid, args);

mPendingVerification.append(verificationId, verificationState);

final List<ComponentName> sufficientVerifiers = matchVerifiers(pkgLite,

receivers, verificationState);

// Apps installed for "all" users use the device owner to verify the app

UserHandle verifierUser = getUser();

if (verifierUser == UserHandle.ALL) {

verifierUser = UserHandle.OWNER;

}

/\*

\* If any sufficient verifiers were listed in the package

\* manifest, attempt to ask them.

\*/

if (sufficientVerifiers != null) {

final int N = sufficientVerifiers.size();

if (N == 0) {

Slog.i(TAG, "Additional verifiers required, but none installed.");

ret = PackageManager.INSTALL\_FAILED\_VERIFICATION\_FAILURE;

} else {

for (int i = 0; i < N; i++) {

final ComponentName verifierComponent = sufficientVerifiers.get(i);

final Intent sufficientIntent = new Intent(verification);

sufficientIntent.setComponent(verifierComponent);

mContext.sendBroadcastAsUser(sufficientIntent, verifierUser);

}

}

}

final ComponentName requiredVerifierComponent = matchComponentForVerifier(

mRequiredVerifierPackage, receivers);

if (ret == PackageManager.INSTALL\_SUCCEEDED

&& mRequiredVerifierPackage != null) {

/\*

\* Send the intent to the required verification agent,

\* but only start the verification timeout after the

\* target BroadcastReceivers have run.

\*/

verification.setComponent(requiredVerifierComponent);

mContext.sendOrderedBroadcastAsUser(verification, verifierUser,

android.Manifest.permission.PACKAGE\_VERIFICATION\_AGENT,

new BroadcastReceiver() {

@Override

public void onReceive(Context context, Intent intent) {

final Message msg = mHandler

.obtainMessage(CHECK\_PENDING\_VERIFICATION);

msg.arg1 = verificationId;

mHandler.sendMessageDelayed(msg, getVerificationTimeout());

}

}, null, 0, null, null);

/\*

\* We don't want the copy to proceed until verification

\* succeeds, so null out this field.

\*/

mArgs = null;

}

} else {

/\*

\* No package verification is enabled, so immediately start

\* the remote call to initiate copy using temporary file.

\*/

ret = args.copyApk(mContainerService, true);

}

}

InstallArgs是个抽象类，一共有三个实现类MoveInstallArgs（针对已有文件的Move）、AsecInstallArgs（针对SD卡）和FileInstallArgs（针对内部存储），会在createInstallArgs()方法中根据不同的参数返回不同的实现类。接下来分析FileInstallArgs.copyApk()方法：

### FileInstallArgs.copyApk()

int copyApk(IMediaContainerService imcs, boolean temp) throws RemoteException {

// 已经执行过copy了

if (origin.staged) {

codeFile = origin.file;

resourceFile = origin.file;

return PackageManager.INSTALL\_SUCCEEDED;

}

try {

// 在/data/app/下面生成一个类似vmdl1354353418.tmp的临时文件

final File tempDir = mInstallerService.allocateStageDirLegacy(volumeUuid);

codeFile = tempDir;

resourceFile = tempDir;

} catch (IOException e) {

return PackageManager.INSTALL\_FAILED\_INSUFFICIENT\_STORAGE;

}

// 在imcs.copyPackage()中会调用target.open()，返回一个文件描述符

final IParcelFileDescriptorFactory target = new IParcelFileDescriptorFactory.Stub() {

@Override

public ParcelFileDescriptor open(String name, int mode) throws RemoteException {

if (!FileUtils.isValidExtFilename(name)) {

throw new IllegalArgumentException("Invalid filename: " + name);

}

try {

final File file = new File(codeFile, name);

final FileDescriptor fd = Os.open(file.getAbsolutePath(),

O\_RDWR | O\_CREAT, 0644);

Os.chmod(file.getAbsolutePath(), 0644);

return new ParcelFileDescriptor(fd);

} catch (ErrnoException e) {

throw new RemoteException("Failed to open: " + e.getMessage());

}

}

};

int ret = PackageManager.INSTALL\_SUCCEEDED;

// 调用DefaultContainerService.mBinder.copyPackage()方法复制文件到target.open()方法指定的文件中，也即是上面产生的临时文件

**ret = imcs.copyPackage(origin.file.getAbsolutePath(), target);**

if (ret != PackageManager.INSTALL\_SUCCEEDED) {

return ret;

}

final File libraryRoot = new File(codeFile, LIB\_DIR\_NAME);

NativeLibraryHelper.Handle handle = null;

try {

handle = NativeLibraryHelper.Handle.create(codeFile);

ret = NativeLibraryHelper.copyNativeBinariesWithOverride(handle, libraryRoot,

abiOverride);

} catch (IOException e) {

ret = PackageManager.INSTALL\_FAILED\_INTERNAL\_ERROR;

} finally {

IoUtils.closeQuietly(handle);

}

return ret;

}

而copyApk方法同样是调用DefaultContainerService的copyPackage将应用的文件复制到/data/app下，如果还有native动态库，也会把包在apk文件中的动态库提取出来。

执行完copyApk后，应用安装到了data/app目录下了。

### InstallParams.handleReturnCode()

在handleStartCopy()执行完之后，文件复制工作阶段的工作已经完成了，接下来会在startCopy()中调用handleReturnCode()->processPendingInstall()来进行应用的解析和装载。

## 解析应用阶段

这个阶段的工作是对安装包进行扫描优化，把应用转换成oat格式，然后装载到内存中去。

### processPendingInstall()

private void processPendingInstall(final InstallArgs args, final int currentStatus) {

// 以异步的方式执行安装，因为安装工作可能持续时间比较长，避免占用CPU

mHandler.post(new Runnable() {

public void run() {

// 防止重复调用

mHandler.removeCallbacks(this);

PackageInstalledInfo res = new PackageInstalledInfo();

res.returnCode = currentStatus;

res.uid = -1;

res.pkg = null;

res.removedInfo = new PackageRemovedInfo();

if (res.returnCode == PackageManager.INSTALL\_SUCCEEDED) {

// 如果前面返回的是执行成功的返回值

args.doPreInstall(res.returnCode);

synchronized (mInstallLock) {

// 开始安装应用，带LI后缀的函数执行时要带mInstallLock锁

installPackageLI(args, res);

}

// 执行doPostInstall()，这里主要分析一下FileInstallArgs.doPostInstall()

// 如果没有安装成功，这里会清除前面生成的临时文件

args.doPostInstall(res.returnCode, res.uid);

}

// 执行备份，在下面的情况下会执行备份：1.安装成功，2.是一个新的安装而不是一个升级的操作，3.新的安装包还没有执行过备份操作

final boolean update = res.removedInfo.removedPackage != null;

final int flags = (res.pkg == null) ? 0 : res.pkg.applicationInfo.flags;

boolean doRestore = !update

&& ((flags & ApplicationInfo.FLAG\_ALLOW\_BACKUP) != 0);

// Set up the post-install work request bookkeeping. This will be used

// and cleaned up by the post-install event handling regardless of whether

// there's a restore pass performed. Token values are >= 1.

int token;

if (mNextInstallToken < 0) mNextInstallToken = 1;

token = mNextInstallToken++;

PostInstallData data = new PostInstallData(args, res);

mRunningInstalls.put(token, data);

if (res.returnCode == PackageManager.INSTALL\_SUCCEEDED && doRestore) {

IBackupManager bm = IBackupManager.Stub.asInterface(

ServiceManager.getService(Context.BACKUP\_SERVICE));

if (bm != null) {

try {

if (bm.isBackupServiceActive(UserHandle.USER\_OWNER)) {

bm.restoreAtInstall(res.pkg.applicationInfo.packageName, token);

} else {

doRestore = false;

}

} catch (RemoteException e) {

} catch (Exception e) {

doRestore = false;

}

} else {

doRestore = false;

}

}

if (!doRestore) {

// 发送POST\_INSTALL消息

Message msg = mHandler.obtainMessage(POST\_INSTALL, token, 0);

mHandler.sendMessage(msg);

}

}

});

}

processPendingInstall()方法内部是以异步的方式继续执行安装工作的，首先来调用installPackageLI()执行安装工作，然后调用doPostInstall()对前面的工作的返回结果进行处理，如果没有安装成功，执行清除的工作。然后再执行备份操作。  
下面来看一下installPackageLI()方法：

### installPackageLI()

installPackageLI()方法首先解析apk安装包，然后判断当前是否有安装该应用，然后根据不同的情况进行不同的处理，然后进行Dex优化操作。如果是升级安装，调用replacePackageLI()。如果是新安装，调用installNewPackageLI()。这两个方法会在下面详细介绍。

private void installPackageLI(InstallArgs args, PackageInstalledInfo res) {

final int installFlags = args.installFlags;

final String installerPackageName = args.installerPackageName;

final String volumeUuid = args.volumeUuid;

final File tmpPackageFile = new File(args.getCodePath());

final boolean forwardLocked = ((installFlags & PackageManager.INSTALL\_FORWARD\_LOCK) != 0);

final boolean onExternal = (((installFlags & PackageManager.INSTALL\_EXTERNAL) != 0)

|| (args.volumeUuid != null));

boolean replace = false;

int scanFlags = SCAN\_NEW\_INSTALL | SCAN\_UPDATE\_SIGNATURE;

if (args.move != null) {

scanFlags |= SCAN\_INITIAL;

}

res.returnCode = PackageManager.INSTALL\_SUCCEEDED;

// 创建apk解析器

final int parseFlags = mDefParseFlags | PackageParser.PARSE\_CHATTY

| (forwardLocked ? PackageParser.PARSE\_FORWARD\_LOCK : 0)

| (onExternal ? PackageParser.PARSE\_EXTERNAL\_STORAGE : 0);

PackageParser pp = new PackageParser();

pp.setSeparateProcesses(mSeparateProcesses);

pp.setDisplayMetrics(mMetrics);

final PackageParser.Package pkg;

try {

// 开始解析文件，解析apk的信息存储在PackageParser.Package中

pkg = pp.parsePackage(tmpPackageFile, parseFlags);

} catch (PackageParserException e) {

res.setError("Failed parse during installPackageLI", e);

return;

}

......

// 获取安装包的签名和AndroidManifest摘要

try {

pp.collectCertificates(pkg, parseFlags);

pp.collectManifestDigest(pkg);

} catch (PackageParserException e) {

res.setError("Failed collect during installPackageLI", e);

return;

}

if (args.manifestDigest != null) {

// 与installPackage()方法传递过来的VerificationParams获取的AndroidManifest摘要进行对比

if (!args.manifestDigest.equals(pkg.manifestDigest)) {

res.setError(INSTALL\_FAILED\_PACKAGE\_CHANGED, "Manifest digest changed");

return;

}

} else if (DEBUG\_INSTALL) {...}

// Get rid of all references to package scan path via parser.

pp = null;

String oldCodePath = null;

boolean systemApp = false;

synchronized (mPackages) {

// 判断是否是升级当前已有应用

if ((installFlags & PackageManager.INSTALL\_REPLACE\_EXISTING) != 0) {

String oldName = mSettings.mRenamedPackages.get(pkgName);

if (pkg.mOriginalPackages != null

&& pkg.mOriginalPackages.contains(oldName)

&& mPackages.containsKey(oldName)) {

// 如果当前应用已经被升级过

pkg.setPackageName(oldName);

pkgName = pkg.packageName;

replace = true;

} else if (mPackages.containsKey(pkgName)) {

// 当前应用没有被升级过

replace = true;

}

// 如果已有应用oldTargetSdk大于LOLLIPOP\_MR1(22)，新升级应用小于LOLLIPOP\_MR1，则不允许降级安装

// 因为AndroidM(23)引入了全新的权限管理方式：动态权限管理

if (replace) {

PackageParser.Package oldPackage = mPackages.get(pkgName);

final int oldTargetSdk = oldPackage.applicationInfo.targetSdkVersion;

final int newTargetSdk = pkg.applicationInfo.targetSdkVersion;

if (oldTargetSdk > Build.VERSION\_CODES.LOLLIPOP\_MR1

&& newTargetSdk <= Build.VERSION\_CODES.LOLLIPOP\_MR1) {

...

return;

}

}

}

PackageSetting ps = mSettings.mPackages.get(pkgName);

if (ps != null) {

if (shouldCheckUpgradeKeySetLP(ps, scanFlags)) {

// 判断签名是否一致

if (!checkUpgradeKeySetLP(ps, pkg)) {

...

return;

}

} else {

try {

verifySignaturesLP(ps, pkg);

} catch (PackageManagerException e) {

...

return;

}

}

oldCodePath = mSettings.mPackages.get(pkgName).codePathString;

if (ps.pkg != null && ps.pkg.applicationInfo != null) {

// 判断是否是系统应用

systemApp = (ps.pkg.applicationInfo.flags &

// 给origUsers赋值，此变量代表哪些用户以前已经安装过该应用

res.origUsers = ps.queryInstalledUsers(sUserManager.getUserIds(), true);

}

// Check whether the newly-scanned package wants to define an already-defined perm

int N = pkg.permissions.size();

for (int i = N-1; i >= 0; i--) {

PackageParser.Permission perm = pkg.permissions.get(i);

BasePermission bp = mSettings.mPermissions.get(perm.info.name);

if (bp != null) {

// If the defining package is signed with our cert, it's okay. This

// also includes the "updating the same package" case, of course.

// "updating same package" could also involve key-rotation.

final boolean sigsOk;

if (bp.sourcePackage.equals(pkg.packageName)

&& (bp.packageSetting instanceof PackageSetting)

&& (shouldCheckUpgradeKeySetLP((PackageSetting) bp.packageSetting,

scanFlags))) {

sigsOk = checkUpgradeKeySetLP((PackageSetting) bp.packageSetting, pkg);

} else {

sigsOk = compareSignatures(bp.packageSetting.signatures.mSignatures,

pkg.mSignatures) == PackageManager.SIGNATURE\_MATCH;

}

if (!sigsOk) {

// If the owning package is the system itself, we log but allow

// install to proceed; we fail the install on all other permission

// redefinitions.

if (!bp.sourcePackage.equals("android")) {

res.setError(INSTALL\_FAILED\_DUPLICATE\_PERMISSION, "Package "

+ pkg.packageName + " attempting to redeclare permission "

+ perm.info.name + " already owned by " + bp.sourcePackage);

res.origPermission = perm.info.name;

res.origPackage = bp.sourcePackage;

return;

} else {

pkg.permissions.remove(i);

}

}

}

}

}

// 系统应用不允许安装在SDCard上

if (systemApp && onExternal) {

res.setError(INSTALL\_FAILED\_INVALID\_INSTALL\_LOCATION,

"Cannot install updates to system apps on sdcard");

return;

}

// 下面将会进行Dex优化操作

if (args.move != null) {

// 如果是针对已有文件的Move，就不用在进行Dex优化了

scanFlags |= SCAN\_NO\_DEX;

scanFlags |= SCAN\_MOVE;

synchronized (mPackages) {

final PackageSetting ps = mSettings.mPackages.get(pkgName);

if (ps == null) {

res.setError(INSTALL\_FAILED\_INTERNAL\_ERROR,

"Missing settings for moved package " + pkgName);

}

pkg.applicationInfo.primaryCpuAbi = ps.primaryCpuAbiString;

pkg.applicationInfo.secondaryCpuAbi = ps.secondaryCpuAbiString;

}

} else if (!forwardLocked && !pkg.applicationInfo.isExternalAsec()) {

// 没有设置了PRIVATE\_FLAG\_FORWARD\_LOCK标志且不是安装在外部SD卡

// 使能 SCAN\_NO\_DEX 标志位，在后面的操作中会跳过 dexopt

scanFlags |= SCAN\_NO\_DEX;

try {

derivePackageAbi(pkg, new File(pkg.codePath), args.abiOverride,

true /\* extract libs \*/);

} catch (PackageManagerException pme) {

res.setError(INSTALL\_FAILED\_INTERNAL\_ERROR, "Error deriving application ABI");

return;

}

// 进行DexOpt操作，会调用install 的dexopt命令，优化后的文件放在 /data/dalvik-cache/ 下面

int result = mPackageDexOptimizer

.performDexOpt(pkg, null /\* instruction sets \*/, false /\* forceDex \*/,

false /\* defer \*/, false /\* inclDependencies \*/,

true /\* boot complete \*/);

if (result == PackageDexOptimizer.DEX\_OPT\_FAILED) {

res.setError(INSTALL\_FAILED\_DEXOPT, "Dexopt failed for " + pkg.codePath);

return;

}

}

// 重命名/data/app/下面应用的目录名字，调用getNextCodePath()来获取目录名称，类似com.android.browser-1

if (!args.doRename(res.returnCode, pkg, oldCodePath)) {

res.setError(INSTALL\_FAILED\_INSUFFICIENT\_STORAGE, "Failed rename");

return;

}

startIntentFilterVerifications(args.user.getIdentifier(), replace, pkg);

if (replace) {

// 如果是安装升级包，调用replacePackageLI

replacePackageLI(pkg, parseFlags, scanFlags | SCAN\_REPLACING, args.user,

installerPackageName, volumeUuid, res);

} else {

// 如果安装的新应用，调用installNewPackageLI

installNewPackageLI(pkg, parseFlags, scanFlags | SCAN\_DELETE\_DATA\_ON\_FAILURES,

args.user, installerPackageName, volumeUuid, res);

}

synchronized (mPackages) {

final PackageSetting ps = mSettings.mPackages.get(pkgName);

if (ps != null) {

// 安装完成后，给newUsers赋值，此变量代表哪些用户刚刚安装过该应用

res.newUsers = ps.queryInstalledUsers(sUserManager.getUserIds(), true);

}

}

}

pkg = pp.parsePackage(tmpPackageFile, parseFlags);前文已经分析过了

### doHandleMessage- POST\_INSTALL

processPendingInstall()方法中执行安装的最后是发送POST\_INSTALL消息，现在来看一下这个消息需要处理的事情：

case POST\_INSTALL: {

//从正在安装队列中将当前正在安装的任务删除

PostInstallData data = mRunningInstalls.get(msg.arg1);

mRunningInstalls.delete(msg.arg1);

boolean deleteOld = false;

if (data != null) {

InstallArgs args = data.args;

PackageInstalledInfo res = data.res;

if (res.returnCode == PackageManager.INSTALL\_SUCCEEDED) {

final String packageName = res.pkg.applicationInfo.packageName;

res.removedInfo.sendBroadcast(false, true, false);

Bundle extras = new Bundle(1);

extras.putInt(Intent.EXTRA\_UID, res.uid);

// 现在已经成功的安装了应用，在发送广播之前先授予一些必要的权限

// 这些权限在 installPackageAsUser 中创建 InstallParams 时传递的，为null

if ((args.installFlags

& PackageManager.INSTALL\_GRANT\_RUNTIME\_PERMISSIONS) != 0) {

grantRequestedRuntimePermissions(res.pkg, args.user.getIdentifier(),

args.installGrantPermissions);

}

// 看一下当前应用对于哪些用户是第一次安装，哪些用户是升级安装

int[] firstUsers;

int[] updateUsers = new int[0];

if (res.origUsers == null || res.origUsers.length == 0) {

// 所有用户都是第一次安装

firstUsers = res.newUsers;

} else {

firstUsers = new int[0];

// 这里再从刚刚已经安装该包的用户中选出哪些是以前已经安装过该包的用户

for (int i=0; i<res.newUsers.length; i++) {

int user = res.newUsers[i];

boolean isNew = true;

for (int j=0; j<res.origUsers.length; j++) {

if (res.origUsers[j] == user) {

// 找到以前安装过该包的用户

isNew = false;

break;

}

}

if (isNew) {

int[] newFirst = new int[firstUsers.length+1];

System.arraycopy(firstUsers, 0, newFirst, 0,

firstUsers.length);

newFirst[firstUsers.length] = user;

firstUsers = newFirst;

} else {

int[] newUpdate = new int[updateUsers.length+1];

System.arraycopy(updateUsers, 0, newUpdate, 0,

updateUsers.length);

newUpdate[updateUsers.length] = user;

updateUsers = newUpdate;

}

}

}

//为新安装用户发送广播ACTION\_PACKAGE\_ADDED

sendPackageBroadcast(Intent.ACTION\_PACKAGE\_ADDED,

packageName, extras, null, null, firstUsers);

final boolean update = res.removedInfo.removedPackage != null;

if (update) {

extras.putBoolean(Intent.EXTRA\_REPLACING, true);

}

//为升级安装用户发送广播ACTION\_PACKAGE\_ADDED

sendPackageBroadcast(Intent.ACTION\_PACKAGE\_ADDED,

packageName, extras, null, null, updateUsers);

if (update) {

// 如果是升级安装，还会发送ACTION\_PACKAGE\_REPLACED和ACTION\_MY\_PACKAGE\_REPLACED广播

sendPackageBroadcast(Intent.ACTION\_PACKAGE\_REPLACED,

packageName, extras, null, null, updateUsers);

sendPackageBroadcast(Intent.ACTION\_MY\_PACKAGE\_REPLACED,

null, null, packageName, null, updateUsers);

// 判断该包是否是设置了PRIVATE\_FLAG\_FORWARD\_LOCK标志或者是安装在外部SD卡

if (res.pkg.isForwardLocked() || isExternal(res.pkg)) {

int[] uidArray = new int[] { res.pkg.applicationInfo.uid };

ArrayList<String> pkgList = new ArrayList<String>(1);

pkgList.add(packageName);

sendResourcesChangedBroadcast(true, true,

pkgList,uidArray, null);

}

}

if (res.removedInfo.args != null) {

// 删除被替换应用的资源目录标记位

deleteOld = true;

}

// 针对Browser的一些处理

if (firstUsers.length > 0) {

if (packageIsBrowser(packageName, firstUsers[0])) {

synchronized (mPackages) {

for (int userId : firstUsers) {

mSettings.setDefaultBrowserPackageNameLPw(null, userId);

}

}

}

}

...

}

// 执行一次GC操作

Runtime.getRuntime().gc();

// 执行删除操作

if (deleteOld) {

synchronized (mInstallLock) {

res.removedInfo.args.doPostDeleteLI(true);

}

}

if (args.observer != null) {

try {

// 调用回调函数通知安装者此次安装的结果

Bundle extras = extrasForInstallResult(res);

args.observer.onPackageInstalled(res.name, res.returnCode,

res.returnMsg, extras);

} catch (RemoteException e) {...}

}

} else {...}

} break;

对POST\_INSTALL消息消息的处理主要就是一些权限处理、发送广播、通知相关应用处理安装结果，然后调用回调函数onPackageInstalled()，这个回调函数是调用installPackage()方法时作为参数传递进来的。

最后发送广播通知

*<li> {****@link*** *#ACTION\_PACKAGE\_ADDED}  
\* <li> {****@link*** *#ACTION\_PACKAGE\_REMOVED}*

### 小结

解析应用阶段的工作：

1. 解析apk信息
2. dexopt操作
3. 更新权限信息
4. 完成安装,发送Intent.ACTION\_PACKAGE\_ADDED广播

### 其他相关方法分析

#### getNextCodePath

|  |
| --- |
|  |

类似com.android.browser-1

#### replacePackageLI()

#### uid和gid分配方法

## 装载应用

## ref

<http://www.heqiangfly.com/2016/05/12/android-source-code-analysis-package-manager-installation/>

[https://guolei1130.github.io/2017/01/04/Android应用程序是如何安装的/](https://guolei1130.github.io/2017/01/04/Android%E5%BA%94%E7%94%A8%E7%A8%8B%E5%BA%8F%E6%98%AF%E5%A6%82%E4%BD%95%E5%AE%89%E8%A3%85%E7%9A%84/)  
[Android PackageManager相关源码分析之安装应用](http://www.heqiangfly.com/2016/05/12/android-source-code-analysis-package-manager-installation/)  
[PMS(Android5.1)深入分析（四）安装应用](http://www.aichengxu.com/android/2506357.htm)  
[Android应用程序安装过程解析(源码角度)](http://www.jianshu.com/p/21412a697eb0)  
<http://www.jianshu.com/p/21412a697eb0>  
<http://solart.cc/2016/10/30/install_apk/>  
一次测试的信息  
adb logcat -b system

10-03 17:39:21.892 I/PackageManager(20739): init\_copy idx=0: InstallParams{3c107196 file=/data/local/tmp/k.art.debug cid=null}

10-03 17:39:21.896 I/PackageManager(20739): mcs\_bound

10-03 17:39:21.896 I/PackageManager(20739): startCopy UserHandle{-1}: InstallParams{3c107196 file=/data/local/tmp/k.art.debug cid=null}

10-03 17:39:21.923 D/PackageManager(20739): installPackageLI: path=/data/app/vmdl828827845.tmp

10-03 17:39:22.027 D/PackageManager(20739): manifestDigest was not present, but parser got: ManifestDigest {mDigest=fe,da,41,e8,49,d6,cd,e5,10,16,26,df,83,1c,24

,cf,eb,1f,7a,fb,be,27,9f,2d,94,92,9c,ce,f2,6d,78,a1,}

10-03 17:39:22.027 W/PackageManager(20739): Package k.art.debug attempting to redeclare system permission android.permission.WRITE\_SETTINGS; ignoring new declar

ation

10-03 17:39:22.027 D/PackageManager(20739): Renaming /data/app/vmdl828827845.tmp to /data/app/k.art.debug-1

10-03 17:39:22.028 D/PackageManager(20739): installNewPackageLI: Package{5c4549c k.art.debug}

10-03 17:39:22.043 I/PackageManager(20739): Linking native library dir for /data/app/k.art.debug-1

10-03 17:39:22.043 D/PackageManager(20739): Resolved nativeLibraryRoot for k.art.debug to root=/data/app/k.art.debug-1/lib, isa=true

10-03 17:39:23.427 D/PackageManager(20739): New package installed in /data/app/k.art.debug-1

10-03 17:39:23.467 V/PackageManager(20739): BM finishing package install for 4

10-03 17:39:23.468 D/PackageManager(20739): Sending to user 0: act=android.intent.action.PACKAGE\_ADDED dat=package:k.art.debug flg=0x4000000 Bundle[{android.int

ent.extra.UID=10047, android.intent.extra.user\_handle=0}]

10-03 17:39:23.468 D/PackageManager(20739): java.lang.RuntimeException: here

10-03 17:39:23.468 D/PackageManager(20739): at com.android.server.pm.PMS.sendPackageBroadcast(PMS.java:8321)

10-03 17:39:23.468 D/PackageManager(20739): at com.android.server.pm.PMS$PackageHandler.doHandleMessage(PMS.java:1066)

10-03 17:39:23.468 D/PackageManager(20739): at com.android.server.pm.PMS$PackageHandler.handleMessage(PMS.java:824)

10-03 17:39:23.468 D/PackageManager(20739): at android.os.Handler.dispatchMessage(Handler.java:102)

10-03 17:39:23.468 D/PackageManager(20739): at android.os.Looper.loop(Looper.java:135)

10-03 17:39:23.468 D/PackageManager(20739): at android.os.HandlerThread.run(HandlerThread.java:61)

10-03 17:39:23.468 D/PackageManager(20739): at com.android.server.ServiceThread.run(ServiceThread.java:46)

10-03 17:39:31.964 I/PackageManager(20739): mcs\_unbind

10-03 17:39:31.965 I/PackageManager(20739): calling disconnectService()

# 方法名中的LI、LIF、LPw、LPr的含义

要想弄明白方法名中的LI、LIF、LPw、LPr的含义，需要先了解PackageManagerService内部使用的两个锁。因为LI、LIF、LPw、LPr中的L，指的是Lock，而后面跟的I和P指的是两个锁，I表示mInstallLock同步锁。P表示mPackages同步锁。LPw、LPr中的w表示writing，r表示reading。LIF中的F表示Freeze。

mPackages同步锁，是指操作mPackages时，用synchronized (mPackages) {}保护起来。mPackages同步锁用来保护内存中已经解析的包信息和其他相关状态。mPackages同步锁是细粒度的锁，只能短时间持有这个锁，因为争抢mPackages锁的请求很多，短时间持有mPackages锁，可以让其他请求等待的时间短些。

mInstallLock同步锁，是指安装App的时候，对安装的处理要用synchronized (mInstaller) {}保护起来。mInstallLock同步锁，用来保护所有对installd的访问。installd通常包含对应用数据的繁重操作。

由于installd是单线程的，并且installd的操作通常很慢，所以在已经持有mPackages同步锁的时候，千万不要再请求mInstallLock同步锁。反之，在已经持有mInstallLock同步锁的时候，可以去请求mPackages同步锁。

用代码表示出来，是这样的：

注意：不允许的情况：

synchronized (mPackages) {

synchronized (mInstaller) {

// 这种情况是不允许的。因为Install的处理时间会很长，导致对mPackages锁住的时间加长，会使得其他对mPackages操作的请求处于长时间等待。

}

}

允许的情况：

synchronized (mInstaller) {

synchronized (mPackages) {

// 这种情况是允许的。因为mPackages处理完之后，其他对mPackages操作的请求可以对mPackages处理，不需要等待太久。

// 由于处理Install的时间本身很长，synchronized (mPackages)又较快，所以不会对原本长时间持有mInstaller锁的情况有大的影响。

}

}

https://blog.csdn.net/u013553529/article/details/61962439

# 安装包权限

RequiredInstaller

private @NonNull String getRequiredInstallerLPr() {

final Intent intent = new Intent(Intent.ACTION\_INSTALL\_PACKAGE);

intent.addCategory(Intent.CATEGORY\_DEFAULT);

intent.setDataAndType(Uri.fromFile(new File("foo.apk")), PACKAGE\_MIME\_TYPE);

final List<ResolveInfo> matches = queryIntentActivitiesInternal(intent, PACKAGE\_MIME\_TYPE,

MATCH\_SYSTEM\_ONLY | MATCH\_DIRECT\_BOOT\_AWARE | MATCH\_DIRECT\_BOOT\_UNAWARE,

UserHandle.USER\_SYSTEM);

if (matches.size() == 1) {

ResolveInfo resolveInfo = matches.get(0);

if (!resolveInfo.activityInfo.applicationInfo.isPrivilegedApp()) {

throw new RuntimeException("The installer must be a privileged app");

}

return matches.get(0).getComponentInfo().packageName;

} else {

throw new RuntimeException("There must be exactly one installer; found " + matches);

}

}

private @NonNull String getRequiredUninstallerLPr() {

final Intent intent = new Intent(Intent.ACTION\_UNINSTALL\_PACKAGE);

intent.addCategory(Intent.CATEGORY\_DEFAULT);

intent.setData(Uri.fromParts(PACKAGE\_SCHEME, "foo.bar", null));

final ResolveInfo resolveInfo = resolveIntent(intent, null,

MATCH\_SYSTEM\_ONLY | MATCH\_DIRECT\_BOOT\_AWARE | MATCH\_DIRECT\_BOOT\_UNAWARE,

UserHandle.USER\_SYSTEM);

if (resolveInfo == null ||

mResolveActivity.name.equals(resolveInfo.getComponentInfo().name)) {

throw new RuntimeException("There must be exactly one uninstaller; found "

+ resolveInfo);

}

return resolveInfo.getComponentInfo().packageName;

}

如果删除了包安装器，系统无法启动

06-08 20:29:05.425 18574-18574/system\_process E/AndroidRuntime: \*\*\* FATAL EXCEPTION IN SYSTEM PROCESS: main

java.lang.RuntimeException: There must be exactly one installer; found []

at com.android.server.pm.PackageManagerService.getRequiredInstallerLPr(PackageManagerService.java:2945)

at com.android.server.pm.PackageManagerService.<init>(PackageManagerService.java:2780)

at com.android.server.pm.PackageManagerService.main(PackageManagerService.java:2007)

## 权限管概述

[**Android**](http://lib.csdn.net/base/android) 是一个权限分离的系统 。 这是利用 [**Linux**](http://lib.csdn.net/base/linux) 已有的权限管理机制，通过为每一个 Application 分配不同的 uid 和 gid ， 从而使得不同的 Application 之间的私有数据和访问（ native 以及 [**Java**](http://lib.csdn.net/base/javase) 层通过这种 sandbox 机制，都可以）达到隔离的目的 。 与此 同时， Android 还 在此基础上进行扩展，提供了 permission 机制，它主要是用来对 Application 可以执行的某些具体操作进行权限细分和访问控制，同时提供了 per-URI permission 机制，用来提供对某些特定的数据块进行 ad-hoc 方式的访问理

### Android Permission机制

一个权限主要包含三个方面的信息：权限的名称；属于的权限组；保护级别。一个权限组是指把权限按照功能分成的不同的集合。每一个权限组包含若干具体 权限，例如在 COST\_MONEY 组中包含 android.permission.SEND\_SMS ， android.permission.CALL\_PHONE 等和费用相关的权限。

        Android权限等级划分为normal,dangerous,signature,signatureOrSystem,system,development，不同的保护级别代表了程序要使用此权限时的认证方式。

        normal 的权限只要申请了就可以使用

        dangerous 的权限在安装时需要用户确认才可以使用  
        signature需要签名才能赋予权限，  
        signatureOrSystem需要签名或者系统级应用(放置在/system/app目录下)才能赋予权限，  
        system系统级应用(放置在/system/app目录下)才能赋予权限，系统权限的描述在frameworks/base/core/res/AndroidManifest.xml当中。  
        Package 的权限信息主要 通过在 AndroidManifest.xml 中通过一些标签来指定。如 <permission> 标签， <permission-group> 标签 <permission-tree> 等标签。如果 package 需要申请使用某个权限，那么需要使用 <use-permission> 标签来指定。

### Android 提供的Permission 接口

#### CheckPermission

        下面这一组接口主要用来检查某个调用（或者是其它 package 或者是自己）是否拥有访问某个 permission 的权限。参数中 pid 和 uid 可以指定，如果没有指定，那么 framework 会通过 Binder 来获取调用者的 uid 和 pid 信息，加以填充。返回值为 PackageManager.PERMISSION\_GRANTED 或者 PackageManager.PERMISSION\_DENIED   
public int checkPermission(String permission, int pid, int uid) // 检查某个 uid 和 pid 是否有 permission 权限  
public int checkCallingPermission(String permission) // 检查调用者是否有 permission 权限，如果调用者是自己那么返回 PackageManager.PERMISSION\_DENIED  
public int checkCallingOrSelfPermission(String permission) // 检查自己或者其它调用者是否有 permission 权限

        下面这一组和上面类似，如果遇到检查不通过时，会抛出异常，打印消息 。

public void enforcePermission(String permission, int pid, int uid, String message)

public void enforceCallingPermission(String permission, String message)

public void enforceCallingOrSelfPermission(String permission, String message)

checkCallingPermission(String permission)

@Override  
**public int** checkCallingPermission(String permission) {  
 **if** (permission == **null**) {  
 **throw new** IllegalArgumentException(**"permission is null"**);  
 }  
  
 **int** pid = Binder.*getCallingPid*();  
 **if** (pid != Process.*myPid*()) {  
 **return** checkPermission(permission, pid, Binder.*getCallingUid*());  
 }  
 **return** PackageManager.***PERMISSION\_DENIED***;  
}

@Override  
**public int** checkPermission(String permission, **int** pid, **int** uid) {  
 **if** (permission == **null**) {  
 **throw new** IllegalArgumentException(**"permission is null"**);  
 }  
  
 **try** {  
 **return** ActivityManagerNative.*getDefault*().checkPermission(  
 permission, pid, uid);  
 } **catch** (RemoteException e) {  
 **throw** e.rethrowFromSystemServer();  
 }  
}

#### CheckUriPermission

        为某个 package 添加访问 content Uri 的读或者写权限。

public void grantUriPermission(String toPackage, Uri uri, int modeFlags)  
public void revokeUriPermission(Uri uri, int modeFlags)  
        检查某个 pid 和 uid 的 package 是否拥有 uri 的读写权限，返回值表示是否被 granted 。  
public int checkUriPermission(Uri uri, int pid, int uid, int modeFlags)  
public int checkCallingUriPermission(Uri uri, int modeFlags)  
public int checkCallingOrSelfUriPermission(Uri uri, int modeFlags)  
public int checkUriPermission(Uri uri, String readPermission,String writePermission, int pid, int uid, int modeFlags)  
        检查某个 pid 和 uid 的 package 是否拥有 uri 的读写权限，如果失败则抛出异常，打印消息 。  
public void enforceUriPermission(Uri uri, int pid, int uid, int modeFlags, String message)  
public void enforceCallingUriPermission(Uri uri, int modeFlags, String message)  
public void enforceCallingOrSelfUriPermission(Uri uri, int modeFlags, String message)  
public void enforceUriPermission(Uri uri, String readPermission, String writePermission,int pid, int uid, int modeFlags, String message)

其中check开头的，只做检查。enforce开头的，不单检查，没有权限的还会抛出异常。

### PID 指进程ID.

PID是进程的身份标识，程序一旦运行，就会给应用分配一个独一无二的PID(ps:一个应用可能包含多个进程，每个进程有唯一的一个PID)

进程终止后PID会被系统收回，再次打开应用，会重新分配一个PID（新进程的PID一般比之前的号要大）

UID 指用户ID.

UID在Linux中就是用户的ID，表明时哪个用户运行了这个程序，主要用于权限的管理。而在Android 中又有所不同，因为android为单用户系统，这时UID 便被赋予了新的使命，数据共享，为了实现数据共享，Android init时是root权限，zygote出的系统systemserver时赋予system的uid，权限降为system权限，普通应用启动时被赋予

## checkUidPermission

1. 如果传入的 permission 名称为 null ，那么返回 PackageManager.PERMISSION\_DENIED 。  
2. 判断调用者 uid 是否符合要求 。  
1 ） 如果 uid 为 0 ，说明是 root 权限的进程，对权限不作控制。  
2 ） 如果 uid 为 system server 进程的 uid ，说明是 system server ，对权限不作控制。  
3 ） 如果是 ActivityManager 进程本身，对权限不作控制。  
4 ）如果调用者 uid 与参数传入的 req uid 不一致，那么返回 PackageManager.PERMISSION\_DENIED 。  
3. 如果通过 2 的检查后，再 调用 PackageManagerService.checkUidPermission ，判断 这个 uid 是否拥有相应的权限，分析如下 。  
1 ） 首先它通过调用 getUserIdLP ，去 PackageManagerService.Setting.mUserIds 数组中，根据 uid 查找 uid （也就是 package ）的权限列表。一旦找到，就表示有相应的权限。  
2 ） 如果没有找到，那么再去 PackageManagerService.mSystemPermissions 中找。这些信息是启动时，从 /system/etc/permissions/platform.xml 中读取的。这里记录了一些系统级的应用的 uid 对应的 permission 。  
3 ）返回结果 。

### 吊起原理

普通的应用调用需要系统权限的方法时，一般会报出Permission Denal的SecureException，这个检查的过程是如何的？我们从一个场景进行分析，monkey检测系统应用启动有一个接口

#### AMS

@Override  
**public void** setActivityController(IActivityController controller, **boolean** imAMonkey) {  
 enforceCallingPermission(android.Manifest.permission.SET\_ACTIVITY\_WATCHER,  
 **"setActivityController()"**);  
 **synchronized** (**this**) {  
 mController = controller;  
 mControllerIsAMonkey = imAMonkey;  
 Watchdog.getInstance().setActivityController(controller);  
 }  
}

**void** enforceCallingPermission(String permission, String func) {  
 **if** (checkCallingPermission(permission)  
 == PackageManager.PERMISSION\_GRANTED) {  
 **return**;  
 }  
  
 String msg = **"Permission Denial: "** + func + **" from pid="** + Binder.getCallingPid()  
 + **", uid="** + Binder.getCallingUid()  
 + **" requires "** + permission;  
 Slog.w(TAG, msg);  
 **throw new** SecurityException(msg);  
}

**int** checkCallingPermission(String permission) {  
 **return** checkPermission(permission,  
 Binder.getCallingPid(),  
 UserHandle.getAppId(Binder.getCallingUid()));  
}

**public int** checkPermission(String permission, **int** pid, **int** uid) {  
 **if** (permission == **null**) {  
 **return** PackageManager.PERMISSION\_DENIED;  
 }  
 **return** checkComponentPermission(permission, pid, uid, -1, **true**);  
}

**int** checkComponentPermission(String permission, **int** pid, **int** uid,  
 **int** owningUid, **boolean** exported) {  
 **if** (pid == MY\_PID) {//== myPid();  
 **return** PackageManager.PERMISSION\_GRANTED;  
 }  
 **return** ActivityManager.checkComponentPermission(permission, uid,  
 owningUid, exported);  
}

首先检查是否是MY\_PID，一般如果不是相同进程，这个pid是不同的。

如果是root或者system的uid，直接返回PERMISSION\_GRANTED。

system uid需要在manifest根节点中shareUserId中设置，并且使用目标系统的签名工具，手机厂商不同，这个签名工具一般是不同的。

#### AM

*/\*\** ***@hide*** *\*/***public static int** checkComponentPermission(String permission, **int** uid,  
 **int** owningUid, **boolean** exported) {  
 *// Root, system server get to do everything.* **final int** appId = UserHandle.getAppId(uid);  
 **if** (appId == Process.ROOT\_UID || appId == Process.SYSTEM\_UID) {  
 **return** PackageManager.PERMISSION\_GRANTED;  
 }  
 *// Isolated processes don't get any permissions.* **if** (UserHandle.isIsolated(uid)) {  
 **return** PackageManager.PERMISSION\_DENIED;  
 }  
 *// If there is a uid that owns whatever is being accessed, it has  
 // blanket access to it regardless of the permissions it requires.* **if** (owningUid >= 0 && UserHandle.isSameApp(uid, owningUid)) {  
 **return** PackageManager.PERMISSION\_GRANTED;  
 }  
 *// If the target is not exported, then nobody else can get to it.* **if** (!exported) {  
 */\*  
 RuntimeException here = new RuntimeException("here");  
 here.fillInStackTrace();  
 Slog.w(TAG, "Permission denied: checkComponentPermission() owningUid=" + owningUid,  
 here);  
 \*/* **return** PackageManager.PERMISSION\_DENIED;  
 }  
 **if** (permission == **null**) {  
 **return** PackageManager.PERMISSION\_GRANTED;  
 }  
 **try** {  
 **return** AppGlobals.getPackageManager()  
 .checkUidPermission(permission, uid);  
 } **catch** (RemoteException e) {  
 **throw** e.rethrowFromSystemServer();  
 }  
}

### checkUidPermission

**final int** callingUid = Binder.getCallingUid();  
**final int** callingUserId = UserHandle.getUserId(callingUid);  
**final boolean** isCallerInstantApp = getInstantAppPackageName(callingUid) != **null**;  
**final boolean** isUidInstantApp = getInstantAppPackageName(uid) != **null**;  
**final int** userId = UserHandle.getUserId(uid);  
**if** (!sUserManager.exists(userId)) {  
 **return** PackageManager.PERMISSION\_DENIED;  
}

HOOK权限的入口

**synchronized** (mPackages) {  
 Object obj = mSettings.getUserIdLPr(UserHandle.getAppId(uid));  
 **if** (obj != **null**) {  
 **if** (obj **instanceof** SharedUserSetting) {  
 **if** (isCallerInstantApp) {  
 **return** PackageManager.PERMISSION\_DENIED;  
 }  
 } **else if** (obj **instanceof** PackageSetting) {  
 **final** PackageSetting ps = (PackageSetting) obj;  
 **if** (filterAppAccessLPr(ps, callingUid, callingUserId)) {  
 **return** PackageManager.PERMISSION\_DENIED;  
 }  
 }  
 **final** SettingBase settingBase = (SettingBase) obj;  
 **final** PermissionsState permissionsState = settingBase.getPermissionsState();  
 **if** (permissionsState.hasPermission(permName, userId)) {  
 **if** (isUidInstantApp) {  
 BasePermission bp = mSettings.mPermissions.get(permName);  
 **if** (bp != **null** && bp.isInstant()) {  
 **return** PackageManager.PERMISSION\_GRANTED;  
 }  
 } **else** {  
 **return** PackageManager.PERMISSION\_GRANTED;  
 }  
 }  
 *// Special case: ACCESS\_FINE\_LOCATION permission includes ACCESS\_COARSE\_LOCATION* **if** (Manifest.permission.ACCESS\_COARSE\_LOCATION.equals(permName) && permissionsState  
 .hasPermission(Manifest.permission.ACCESS\_FINE\_LOCATION, userId)) {  
 **return** PackageManager.PERMISSION\_GRANTED;  
 }  
 } **else** {  
 ArraySet<String> perms = mSystemPermissions.get(uid);  
 **if** (perms != **null**) {  
 **if** (perms.contains(permName)) {  
 **return** PackageManager.PERMISSION\_GRANTED;  
 }  
 **if** (Manifest.permission.ACCESS\_COARSE\_LOCATION.equals(permName) && perms  
 .contains(Manifest.permission.ACCESS\_FINE\_LOCATION)) {  
 **return** PackageManager.PERMISSION\_GRANTED;  
 }  
 }  
 }  
}  
**return** PackageManager.PERMISSION\_DENIED;

一眼扫过去，关键中关键在于mSettings里面保存的这个SettingBase对象，它记录了PermissionsState也就是权限的授予情况。

android.permission.MANAGE\_DOCUMENTS

## grantRuntimePermissions

先不直接分析，我们从另外一边来看，看看是如何授权的。

授权有两个地方，一个是设置里面的入口，还有一个是申请权限弹框界面的入口，代码都在PackageInstaller里面，分别是ManagePermissionsActivity和GrantPermissionsActivity。就不仔细分析了最终授权和撤销都是在AppPermissionGroup这里实现的，grantRuntimePermissions和revokeRuntimePermissions两个方法。最终生效的代码还是在PackageManagerService里面

### 吊起

//授予权限是需要GRANT\_RUNTIME\_PERMISSIONS权限的

mContext.enforceCallingOrSelfPermission(

 android.Manifest.permission.GRANT\_RUNTIME\_PERMISSIONS,

 "grantRuntimePermission");

## revokeRuntimePermission

## 默认授予规则DefaultPermissionGrantPolicy

Google会在第一次开机时打开一些特定app的runtime permission，具体实现在DefaultPermissionGrantPolicy.java中

原则上,是不允许客户自行修改默认打开某些权限,会过不了GTS。若坚持要修改,GTS的部分需要自行向google waive。

默认授予是在PackageManagerService执行systemReady的时候执行的，主要是这个类DefaultPermissionGrantPolicy，名字也一目了然

public void grantDefaultPermissions(int userId) {

//针对系统组件和Privileged的应用做默认权限的处理

grantPermissionsToSysComponentsAndPrivApps(userId);

//对符合系统处理原则的模块进行默认权限的处理

grantDefaultSystemHandlerPermissions(userId);

grantDefaultPermissionExceptions(userId);

}

并且把相关信息写在 /data/system/users/0/runtime\_permissions.xml 中去  
下次开机就不需要再次申请了。

那么有理由相信，systemui 是有权限申请到 Location 的 runtime permission 的



### grantPermissionsToSysComponentsAndPrivApps

//针对系统组件和Privileged的应用做默认权限的处理

private void grantPermissionsToSysComponentsAndPrivApps(int userId) {

Log.i(TAG, "Granting permissionsto platform components for user " + userId);

synchronized (mService.mPackages) {

for (PackageParser.Package pkg :mService.mPackages.values()) {

//遍历所有的package，如果是系统组的，或者是platform签名的特权应用和persistent应用就可以默认授予

if(!isSysComponentOrPersistentPlatformSignedPrivAppLPr(pkg)

||!doesPackageSupportRuntimePermissions(pkg)

||pkg.requestedPermissions.isEmpty()) {

continue;

}

Set<String> permissions =new ArraySet<>();

final int permissionCount =pkg.requestedPermissions.size();

for (int i = 0; i <permissionCount; i++) {

String permission =pkg.requestedPermissions.get(i);

BasePermission bp =mService.mSettings.mPermissions.get(permission);

if (bp != null &&bp.isRuntime()) {

permissions.add(permission);

}

}

if (!permissions.isEmpty()) {

grantRuntimePermissionsLPw(pkg, permissions, true, userId);

}

}

}

}

#### isSysComponentOrPersistentPlatformSignedPrivAppLPr

g

isSysComponentOrPersistentPlatformSignedPrivAppLPr(PackageParser.Package pkg) {

//小于10000的系统进程

if(UserHandle.getAppId(pkg.applicationInfo.uid) < FIRST\_APPLICATION\_UID) {

return true;

}

if(!pkg.isPrivilegedApp()) { //这里我心存疑惑不确定这一类APP的范围

return false;

}

//下面是对当前禁用的APP如果不是persistent的就不用授予了

PackageSetting sysPkg =mService.mSettings.getDisabledSystemPkgLPr(pkg.packageName);

if(sysPkg != null) {

if((sysPkg.pkg.applicationInfo.flags & ApplicationInfo.FLAG\_PERSISTENT) == 0){

return false;

}

} else if((pkg.applicationInfo.flags & ApplicationInfo.FLAG\_PERSISTENT) == 0) {

return false;

}

//必须还要满足platform签名

Return PackageManagerService.compareSignatures(mService.mPlatformPackage.mSignatures,

pkg.mSignatures) == PackageManager.SIGNATURE\_MATCH;

}

#### grantRuntimePermissionsLPw

grantRuntimePermissionsLPw(pkg, permissions, true, false, userId);

授予权限的同时会控制一下权限的Flags，这个东西记录了当前授予的这个运行时权限的状态，主要有以下几种

### grantDefaultSystemHandlerPermissions

//对符合系统处理原则的模块进行默认权限的处理

这个方法好长好长，其实说白了就是针对满足一些系统固有的规则（必须应用安装，校验，联系人存储、下载、拨号等等）的应用授予最基本的权限（其实就是强关联的权限，比如downloadprovider授予读写存储的权限和网络权限），同时这个权限并不是fix的是可以关闭的。

// SetupWizard

PackageParser.Package setupPackage = getSystemPackageLPr(

mService.mSetupWizardPackage);

if (setupPackage != null

&& doesPackageSupportRuntimePermissions(setupPackage)) {

grantRuntimePermissionsLPw(setupPackage, PHONE\_PERMISSIONS, userId);

grantRuntimePermissionsLPw(setupPackage, CONTACTS\_PERMISSIONS, userId);

grantRuntimePermissionsLPw(setupPackage, LOCATION\_PERMISSIONS, userId);

grantRuntimePermissionsLPw(setupPackage, CAMERA\_PERMISSIONS, userId);

}

// Downloads UI

Intent downloadsUiIntent = new Intent(DownloadManager.ACTION\_VIEW\_DOWNLOADS);

PackageParser.Package downloadsUiPackage = getDefaultSystemHandlerActivityPackageLPr(

downloadsUiIntent, userId);

if (downloadsUiPackage != null

&& doesPackageSupportRuntimePermissions(downloadsUiPackage)) {

grantRuntimePermissionsLPw(downloadsUiPackage, STORAGE\_PERMISSIONS, true, userId);

}

// Gallery

Intent galleryIntent = new Intent(Intent.ACTION\_MAIN);

galleryIntent.addCategory(Intent.CATEGORY\_APP\_GALLERY);

PackageParser.Package galleryPackage = getDefaultSystemHandlerActivityPackageLPr(

galleryIntent, userId);

if (galleryPackage != null

&& doesPackageSupportRuntimePermissions(galleryPackage)) {

grantRuntimePermissionsLPw(galleryPackage, STORAGE\_PERMISSIONS, userId);

}

// Browser

PackageParser.Package browserPackage = null;

String defaultBrowserPackage = mService.getDefaultBrowserPackageName(userId);

if (defaultBrowserPackage != null) {

browserPackage = getPackageLPr(defaultBrowserPackage);

}

if (browserPackage == null) {

Intent browserIntent = new Intent(Intent.ACTION\_MAIN);

browserIntent.addCategory(Intent.CATEGORY\_APP\_BROWSER);

browserPackage = getDefaultSystemHandlerActivityPackageLPr(

browserIntent, userId);

}

if (browserPackage != null

&& doesPackageSupportRuntimePermissions(browserPackage)) {

grantRuntimePermissionsLPw(browserPackage, LOCATION\_PERMISSIONS, userId);

}

// Location

if (locationPackageNames != null) {

for (String packageName : locationPackageNames) {

PackageParser.Package locationPackage = getSystemPackageLPr(packageName);

if (locationPackage != null

&& doesPackageSupportRuntimePermissions(locationPackage)) {

grantRuntimePermissionsLPw(locationPackage, CONTACTS\_PERMISSIONS, userId);

grantRuntimePermissionsLPw(locationPackage, CALENDAR\_PERMISSIONS, userId);

grantRuntimePermissionsLPw(locationPackage, MICROPHONE\_PERMISSIONS, userId);

grantRuntimePermissionsLPw(locationPackage, PHONE\_PERMISSIONS, userId);

grantRuntimePermissionsLPw(locationPackage, SMS\_PERMISSIONS, userId);

grantRuntimePermissionsLPw(locationPackage, LOCATION\_PERMISSIONS,

true, userId);

grantRuntimePermissionsLPw(locationPackage, CAMERA\_PERMISSIONS, userId);

grantRuntimePermissionsLPw(locationPackage, SENSORS\_PERMISSIONS, userId);

grantRuntimePermissionsLPw(locationPackage, STORAGE\_PERMISSIONS, userId);

}

}

}

### grantDefaultPermissionExceptions

ref

## 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)

[Android源码设置default application](http://www.dxjia.cn/2016/03/08/aosp-default-application-setting/)

[Android偏好设置及findPreferredActivity分析](http://blog.csdn.net/aaajj/article/details/52262433)

[淘宝应用是如何干预 Android 的默认应用设置的](https://www.zhihu.com/question/27313953)

Android 6.0 运行时权限检查分析

<https://blog.csdn.net/hehui1860/article/details/52335551>

Android Permission权限控制机制

<https://blog.csdn.net/lzpdz/article/details/54646390>

# AppOpsManager.java

**public static final int *OP\_WRITE\_SETTINGS*** = 23;

# 应用签名

Hook方案; <http://www.trinea.cn/android/google-play-fc-when-download/>

checkSignatures

@Override

public int checkSignatures(String pkg1, String pkg2) {

synchronized (mPackages) {

final PackageParser.Package p1 = mPackages.get(pkg1);

final PackageParser.Package p2 = mPackages.get(pkg2);

if (p1 == null || p1.mExtras == null

|| p2 == null || p2.mExtras == null) {

return PackageManager.SIGNATURE\_UNKNOWN\_PACKAGE;

}

return compareSignatures(p1.mSignatures, p2.mSignatures);

}

}

## checkUidSignatures(int uid1, int uid2)

f

@Override

public int checkUidSignatures(int uid1, int uid2) {

// Map to base uids.

uid1 = UserHandle.getAppId(uid1);

uid2 = UserHandle.getAppId(uid2);

// reader

synchronized (mPackages) {

Signature[] s1;

Signature[] s2;

Object obj = mSettings.getUserIdLPr(uid1);

if (obj != null) {

if (obj instanceof SharedUserSetting) {

s1 = ((SharedUserSetting)obj).signatures.mSignatures;

} else if (obj instanceof PackageSetting) {

s1 = ((PackageSetting)obj).signatures.mSignatures;

} else {

return PackageManager.SIGNATURE\_UNKNOWN\_PACKAGE;

}

} else {

return PackageManager.SIGNATURE\_UNKNOWN\_PACKAGE;

}

obj = mSettings.getUserIdLPr(uid2);

if (obj != null) {

if (obj instanceof SharedUserSetting) {

s2 = ((SharedUserSetting)obj).signatures.mSignatures;

} else if (obj instanceof PackageSetting) {

s2 = ((PackageSetting)obj).signatures.mSignatures;

} else {

return PackageManager.SIGNATURE\_UNKNOWN\_PACKAGE;

}

} else {

return PackageManager.SIGNATURE\_UNKNOWN\_PACKAGE;

}

return compareSignatures(s1, s2);

}

}

## compareSignatures

df

static int compareSignatures(Signature[] s1, Signature[] s2) {

if (s1 == null) {

return s2 == null

? PackageManager.SIGNATURE\_NEITHER\_SIGNED

: PackageManager.SIGNATURE\_FIRST\_NOT\_SIGNED;

}

if (s2 == null) {

return PackageManager.SIGNATURE\_SECOND\_NOT\_SIGNED;

}

if (s1.length != s2.length) {

return PackageManager.SIGNATURE\_NO\_MATCH;

}

// Since both signature sets are of size 1, we can compare without HashSets.

if (s1.length == 1) {

return s1[0].equals(s2[0]) ?

PackageManager.SIGNATURE\_MATCH :

PackageManager.SIGNATURE\_NO\_MATCH;

}

ArraySet<Signature> set1 = new ArraySet<Signature>();

for (Signature sig : s1) {

set1.add(sig);

}

ArraySet<Signature> set2 = new ArraySet<Signature>();

for (Signature sig : s2) {

set2.add(sig);

}

// Make sure s2 contains all signatures in s1.

if (set1.equals(set2)) {

return PackageManager.SIGNATURE\_MATCH;

}

return PackageManager.SIGNATURE\_NO\_MATCH;

}

# APP查询

## getDefaultHomeActivity

*// Auto install icon*PackageManager pm = context.getPackageManager();  
ResolveInfo matched = pm.resolveActivity(  
 **new** Intent(Intent.ACTION\_MAIN)  
 .setComponent(cn).addCategory(Intent.CATEGORY\_LAUNCHER),  
 PackageManager.MATCH\_DEFAULT\_ONLY);

ApplicationPackageManager **extends** PackageManager

@Override  
**public** ResolveInfo resolveActivity(Intent intent, **int** flags) {  
 **return** resolveActivityAsUser(intent, flags, mContext.getUserId());  
}

**public** ResolveInfo resolveActivityAsUser(Intent intent, **int** flags, **int** userId) {  
 **try** {  
 **return** mPM.resolveIntent(  
 intent,  
 intent.resolveTypeIfNeeded(mContext.getContentResolver()),  
 flags,  
 userId);  
 } **catch** (RemoteException e) {  
 **throw** e.rethrowFromSystemServer();  
 }  
}

## resolveIntent

resolveIntentInternal

**private** Intent getHomeIntent() {  
 Intent intent = **new** Intent(Intent.ACTION\_MAIN);  
 intent.addCategory(Intent.CATEGORY\_HOME);  
 intent.addCategory(Intent.CATEGORY\_DEFAULT);  
 **return** intent;  
}  
  
**private** IntentFilter getHomeFilter() {  
 IntentFilter filter = **new** IntentFilter(Intent.ACTION\_MAIN);  
 filter.addCategory(Intent.CATEGORY\_HOME);  
 filter.addCategory(Intent.CATEGORY\_DEFAULT);  
 **return** filter;  
}

# 情景分析

## 查询运用列表&&隐藏app图标

<**intent-filter**>  
 <**action android:name="android.intent.action.MAIN"** />  
 <**category android:name="android.intent.category.LAUNCHER"** />  
 *<!--在applist 设置的时候不调用不显示 -->*<**data  
 android:host="AuthActivity"  
 android:scheme="kji"**/>

也就是这个MainActivity可以响应Uri为**kji**://AuthActivity的特定 Intent

但是为什么加入这个之后app就不显示图标了呢？

因为我们把app的入口Activity申明为由接收隐士的Intent来启动，这样自然也就不会显示图标了。

好了，如果需要显示图标的话而且也必须响应上述Uri的话，那么我们可以在该Activity中再加入一个intent-filter标签，将data元素移到新建的标签中。这样应用既可以显示图标又可以响应特定的Uri了。

public void starAct()

{

Intent intent = new Intent( );

ComponentName cn = new ComponentName("com.key.poc", "com.key.poc.MainActivity");

intent.setComponent(cn);

Uri uri=Uri.parse("kji.**AuthActivity** ");//此处应与B程序中Data中标签一致

intent.setData(uri);

startActivity(intent);

*// Auto install icon*PackageManager pm = context.getPackageManager();  
ResolveInfo matched = pm.resolveActivity(  
 **new** Intent(Intent.ACTION\_MAIN)  
 .setComponent(cn).addCategory(Intent.CATEGORY\_LAUNCHER),  
 PackageManager.MATCH\_DEFAULT\_ONLY);

## activity-alias

https://blog.csdn.net/ahence/article/details/51648768

<activity

android:name=".shadow.Shadow"

android:launchMode="singleInstance"

/>

<activity-alias

android:name="Shadow.FlyForbid"

android:targetActivity=".shadow.Shadow"

android:enabled="true"

android:icon="@drawable/ic\_launcher"

android:label="@string/shadow\_flyforbid\_name">

<intent-filter>

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

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

</intent-filter>

</activity-alias>

# 受限管理

package-restrictions.xml保存着受限制的APP的状态，比如某个APP处于disable状态，或者某个APP具有更高的优先级等。目录：

/data/system/users/userid/package-restrictions.xml，

data/system/users/0/package-restrictions.xml

初始化时机，调用readPackageRestrictionsLPr方法，读取package-restrictions.xml文件

## 使能管理

关于enable的含义可以参考：PackageManager.java中定义的常量：

public static final int COMPONENT\_ENABLED\_STATE\_DEFAULT = 0;

public static final int COMPONENT\_ENABLED\_STATE\_ENABLED = 1;

public static final int COMPONENT\_ENABLED\_STATE\_DISABLED = 2;

public static final int COMPONENT\_ENABLED\_STATE\_DISABLED\_USER = 3;

### pm disable

执行脚本：**用pm脚本可以模拟测试，比如**

$adb shell pm disable com.android.providers.drm

<pkg name="com.android.providers.drm" enabled="2" />

Adb shell pm disable com.android.provision/com.android.provision.DefaultActivity

运行上述命令之后，package-restrictions.xml文件就会存在一条受限制的记录：

<?xml version='1.0' encoding='utf-8' standalone='yes' ?>

<package-restrictions>

<pkg name="com.android.provision">

<disabled-components>

<item name="com.android.provision.DefaultActivity" />

</disabled-components>

</pkg>

### Api使能

mPm.getApplicationEnabledSetting(pkName)

mPm.setApplicationEnabledSetting(pkName, PackageManager.COMPONENT\_ENABLED\_STATE\_ENABLED, 0);

### appUI使能-禁用白名单

**设置--应用管理：可停用，但停用后无法进入桌面！！！**  
intent filter verification service 意图过滤器验证服务

com.android.statementservice

在系统设置：Utils.*isSystemPackage*(getContext().getResources(), **mPm**, **mPackageInfo**)来判断是否禁用，因此，我们需要修改两种，一个是系统设置的，另一种是adb shell的，

归结到一点：在PMS添加保护白名单

### 源码分析

调用PackageManager的setApplicationEnabledSetting（）最终会进入到PackageManagerService的setEnabledSetting（）函数：

在这个函数中会检查APP是否有android.Manifest.permission.CHANGE\_COMPONENT\_ENABLED\_STATE权限。

而这个权限是只有system app才能使用的，这也是为什么APP需要系统签名

#### PMS. setApplicationEnabledSetting

@Override  
**public void** setApplicationEnabledSetting(String appPackageName,  
 **int** newState, **int** flags, **int** userId, String callingPackage) {  
 **if** (!sUserManager.exists(userId)) **return**;  
 **if** (callingPackage == **null**) {  
 callingPackage = Integer.toString(Binder.getCallingUid());  
 }  
 setEnabledSetting(appPackageName, **null**, newState, flags, userId, callingPackage);  
}

#### PMS. setEnabledSetting

Fa

setEnabledSetting(**final** String packageName, String className, **int** newState,  
 **final int** flags, **int** userId, String callingPackage) {

##### newState合法性检查

**if** (!(newState == COMPONENT\_ENABLED\_STATE\_DEFAULT  
 || newState == COMPONENT\_ENABLED\_STATE\_ENABLED  
 || newState == COMPONENT\_ENABLED\_STATE\_DISABLED  
 || newState == COMPONENT\_ENABLED\_STATE\_DISABLED\_USER  
 || newState == COMPONENT\_ENABLED\_STATE\_DISABLED\_UNTIL\_USED)) {  
 **throw new** IllegalArgumentException(**"Invalid new component state: "** + newState);  
}

##### app权限检查

Process.SYSTEM\_UID或者自己禁用自己是可以的

PackageSetting pkgSetting;  
**final int** uid = Binder.getCallingUid();  
**final int** permission;  
**if** (uid == Process.SYSTEM\_UID) {  
 permission = PackageManager.PERMISSION\_GRANTED;  
} **else** {  
 permission = mContext.checkCallingOrSelfPermission(  
 android.Manifest.permission.CHANGE\_COMPONENT\_ENABLED\_STATE);  
}  
enforceCrossUserPermission(uid, userId,  
 **false** */\* requireFullPermission \*/*, **true** */\* checkShell \*/*, **"set enabled"**);  
**final boolean** allowedByPermission = (permission == PackageManager.PERMISSION\_GRANTED);

INTERACT\_ACROSS\_USERS\_FULL是啥？？

##### App 检查，无ui的app不允许

*// writer***synchronized** (mPackages) {  
 pkgSetting = mSettings.mPackages.get(packageName);  
 **if** (pkgSetting == **null**) {  
 **if** (className == **null**) {  
 **throw new** IllegalArgumentException(**"Unknown package: "** + packageName);  
 }  
 **throw new** IllegalArgumentException(  
 **"Unknown component: "** + packageName + **"/"** + className);  
 }  
}

##### 用户限制

*// Limit who can change which apps***if** (!UserHandle.isSameApp(uid, pkgSetting.appId)) {  
 *// Don't allow apps that don't have permission to modify other apps* **if** (!allowedByPermission) {  
 **throw new** SecurityException(  
 **"Permission Denial: attempt to change component state from pid="** + Binder.getCallingPid()  
 + **", uid="** + uid + **", package uid="** + pkgSetting.appId);  
 }  
 *// Don't allow changing protected packages.* **if** (mProtectedPackages.isPackageStateProtected(userId, packageName)) {  
 **throw new** SecurityException(**"Cannot disable a protected package: "** + packageName);  
 }  
}

mProtectedPackages.isPackageStateProtected(userId, packageName)

是啥呢？？？

测试一下》》

##### SHELL\_UID

**if** (uid == Process.SHELL\_UID) {  
 *// Shell can only change whole packages between ENABLED and DISABLED\_USER states* **int** oldState = pkgSetting.getEnabled(userId);  
 **if** (className == **null** &&  
 (oldState == COMPONENT\_ENABLED\_STATE\_DISABLED\_USER  
 || oldState == COMPONENT\_ENABLED\_STATE\_DEFAULT  
 || oldState == COMPONENT\_ENABLED\_STATE\_ENABLED)  
 &&  
 (newState == COMPONENT\_ENABLED\_STATE\_DISABLED\_USER  
 || newState == COMPONENT\_ENABLED\_STATE\_DEFAULT  
 || newState == COMPONENT\_ENABLED\_STATE\_ENABLED)) {  
 *// ok* } **else** {  
 **throw new** SecurityException(  
 **"Shell cannot change component state for "** + packageName + **"/"** + className + **" to "** + newState);  
 }  
}

##### className == null 使能

pkgSetting.setEnabled(newState, userId, callingPackage);

##### component

*// We're dealing with a component level state change  
 // First, verify that this is a valid class name.* PackageParser.Package pkg = pkgSetting.pkg;  
 **if** (pkg == **null** || !pkg.hasComponentClassName(className)) {  
 **if** (pkg != **null** &&  
 pkg.applicationInfo.targetSdkVersion >=  
 Build.VERSION\_CODES.JELLY\_BEAN) {  
 **throw new** IllegalArgumentException(**"Component class "** + className  
 + **" does not exist in "** + packageName);  
 } **else** {  
 Slog.w(TAG, **"Failed setComponentEnabledSetting: component class "** + className + **" does not exist in "** + packageName);  
 }  
 }  
 **switch** (newState) {  
 **case** COMPONENT\_ENABLED\_STATE\_ENABLED:  
 **if** (!pkgSetting.enableComponentLPw(className, userId)) {  
 **return**;  
 }  
 **break**;  
 **case** COMPONENT\_ENABLED\_STATE\_DISABLED:  
 **if** (!pkgSetting.disableComponentLPw(className, userId)) {  
 **return**;  
 }  
 **break**;  
 **case** COMPONENT\_ENABLED\_STATE\_DEFAULT:  
 **if** (!pkgSetting.restoreComponentLPw(className, userId)) {  
 **return**;  
 }  
 **break**;  
 **default**:  
 Slog.e(TAG, **"Invalid new component state: "** + newState);  
 **return**;  
 }  
}

##### 写入本地文件

scheduleWritePackageRestrictionsLocked

##### 发送状态广播

components = mPendingBroadcasts.get(userId, packageName);  
 **final boolean** newPackage = components == **null**;  
 **if** (newPackage) {  
 components = **new** ArrayList<String>();  
 }  
 **if** (!components.contains(componentName)) {  
 components.add(componentName);  
 }  
 **if** ((flags&PackageManager.DONT\_KILL\_APP) == 0) {  
 sendNow = **true**;  
 *// Purge entry from pending broadcast list if another one exists already  
 // since we are sending one right away.* mPendingBroadcasts.remove(userId, packageName);  
 } **else** {  
 **if** (newPackage) {  
 mPendingBroadcasts.put(userId, packageName, components);  
 }  
 **if** (!mHandler.hasMessages(SEND\_PENDING\_BROADCAST)) {  
 *// Schedule a message* mHandler.sendEmptyMessageDelayed(SEND\_PENDING\_BROADCAST, BROADCAST\_DELAY);  
 }  
 }  
}  
  
**long** callingId = Binder.clearCallingIdentity();  
**try** {  
 **if** (sendNow) {  
 packageUid = UserHandle.getUid(userId, pkgSetting.appId);  
 sendPackageChangedBroadcast(packageName,  
 (flags&PackageManager.DONT\_KILL\_APP) != 0, components, packageUid);  
 }  
} **finally** {  
 Binder.restoreCallingIdentity(callingId);  
}

### 失能之后

Start APP和查询都看不到的原因？？？

## 优先级

默认选择偏好，比如默认的Launcher或者默认的USB接入设备

|  |
| --- |
| <preferred-activities>  <item name="dji.system.launcher/.Launcher" match="100000" always="false" set="0">  <filter>  <action name="android.intent.action.MAIN" />  <cat name="android.intent.category.HOME" />  <cat name="android.intent.category.DEFAULT" />  </filter>  </item>  <item name="dji.system.launcher/.Launcher" match="100000" **always="true" set="2"**>  <set name="dji.system.launcher/.Launcher" />  <set name="dji.prof.args.tiny/.ui.main.MainActivity" />  <filter>  <action name="android.intent.action.MAIN" />  <cat name="android.intent.category.HOME" />  <cat name="android.intent.category.DEFAULT" />  </filter>  </item>  </preferred-activities>  <persistent-preferred-activities /> |

当系统有多个Launcher的时候，我们

业务入口，

1.Provision.apk

2.所以在AMS的startHomeActivityLocked启动Launcher的时候会先启动Provision应用，然后才是启动HomeLauncher

3. 在PackageManagerService构造函数中，会读取packages.xml文件:mRestoredSettings = mSettings.readLPw();所以可以在其前面添加，这个才是最本质的解决办法，但是这个地方需要考虑生命周期问题，此时AMS还没有启动，但是pMS默认的函数需要检查权限。解决办法就是不调用相关的 权限检查。

persist.default\_launcher = pkname

报错信息如下:

: \*\*\* FATAL EXCEPTION IN SYSTEM PROCESS: main

E/AndroidRuntime( 461): java.lang.NullPointerException: Attempt to invoke interface method 'int android.app.IActivityManager.checkPermission(java.lang.String, int, int)' on a null object reference

E/AndroidRuntime( 461): at android.app.ContextImpl.checkPermission(ContextImpl.java:1882)

E/AndroidRuntime( 461): at android.app.ContextImpl.checkCallingOrSelfPermission(ContextImpl.java:1923)

E/AndroidRuntime( 461): at com.android.server.pm.PackageManagerService.addPreferredActivityInternal(PackageManagerService.java:12145)

//ams == null=>checkCallingOrSelfPermission->checkPermission

//replacePreferredActivity(mHomeFilter, IntentFilter.MATCH\_CATEGORY\_EMPTY, mHomeComponentSet, mHomeComponentSet[prefIndex], userId);

//addPreferredActivity(mHomeFilter, IntentFilter.MATCH\_CATEGORY\_EMPTY, mHomeComponentSet, mHomeComponentSet[prefIndex], userId);

## chooseBestActivity

# Installer（PMS）

当守护进程installd启动完成后（frameworks/native/cmds/installd/），上层framework便可以通过socket跟该守护进程进行通信。 在SystemServer启动服务的过程中创建Installer对象，便会有一次跟installd通信的过程

[-> SystemServer.java]

private void startBootstrapServices() {

//启动installer服务【见小节3.0】

Installer installer = mSystemServiceManager.startService(Installer.class);

...

}

[-> Installer.java]

public Installer(Context context) {

super(context);

//创建InstallerConnection对象

mInstaller = new InstallerConnection();

}

public void onStart() {

Slog.i(TAG, "Waiting for installd to be ready.");

//【见小节3.1】

mInstaller.waitForConnection();

}

先创建Installer对象，再调用onStart()方法，该方法中主要工作是等待socket通道建立完成。

### 3.1 waitForConnection

[-> InstallerConnection.java]

public void waitForConnection() {

for (;;) {

//【见小节3.2】

if (execute("ping") >= 0) {

return;

}

Slog.w(TAG, "installd not ready");

SystemClock.sleep(1000);

}

}

通过循环地方式，每次休眠1s

### 3.2 execute

[-> InstallerConnection.java]

public int execute(String cmd) {

//【见小节3.3】

String res = transact(cmd);

try {

return Integer.parseInt(res);

} catch (NumberFormatException ex) {

return -1;

}

}

### 3.3 transact

[-> InstallerConnection.java]

public synchronized String transact(String cmd) {

//【见小节3.4】

if (!connect()) {

return "-1";

}

//【见小节3.5】

if (!writeCommand(cmd)) {

if (!connect() || !writeCommand(cmd)) {

return "-1";

}

}

//读取应答消息【3.6】

final int replyLength = readReply();

if (replyLength > 0) {

String s = new String(buf, 0, replyLength);

return s;

} else {

return "-1";

}

}

### 3.4 connect

private boolean connect() {

if (mSocket != null) {

return true;

}

Slog.i(TAG, "connecting...");

try {

mSocket = new LocalSocket();

LocalSocketAddress address = new LocalSocketAddress("installd",

LocalSocketAddress.Namespace.RESERVED);

mSocket.connect(address);

mIn = mSocket.getInputStream();

mOut = mSocket.getOutputStream();

} catch (IOException ex) {

disconnect();

return false;

}

return true;

}

### 3.5 writeCommand

private boolean writeCommand(String cmdString) {

final byte[] cmd = cmdString.getBytes();

final int len = cmd.length;

if ((len < 1) || (len > buf.length)) {

return false;

}

buf[0] = (byte) (len & 0xff);

buf[1] = (byte) ((len >> 8) & 0xff);

try {

mOut.write(buf, 0, 2); //写入长度

mOut.write(cmd, 0, len); //写入具体命令

} catch (IOException ex) {

disconnect();

return false;

}

return true;

}

命令写入socket套接字，installd进程收到该命令后，便开始执行ping操作并返回结果。

### 3.6 readReply

private int readReply() {

//【见小节3.6.1】

if (!readFully(buf, 2)) {

return -1;

}

final int len = (((int) buf[0]) & 0xff) | ((((int) buf[1]) & 0xff) << 8);

if ((len < 1) || (len > buf.length)) {

disconnect();

return -1;

}

if (!readFully(buf, len)) {

return -1;

}

return len;

}

#### 3.6.1 readFully

private boolean readFully(byte[] buffer, int len) {

try {

Streams.readFully(mIn, buffer, 0, len);

} catch (IOException ioe) {

disconnect();

return false;

}

return true;

}

可见，一次transact过程为先connect()来判断是否建立socket连接，如果已连接则通过writeCommand() 将命令写入socket的mOut管道，等待从管道的mIn中readFully()读取应答消息

上层PKMS收集完相应信息，通过socket交给守护进程installd，该进程才是真正干活的进程。未完。。。

<http://gityuan.com/2016/11/13/android-installd/>

# uid

## android.uid.system

### apps

frameeok

AndroidManifest.xml (core\res): package="android" coreApp="true" android:sharedUserId="android.uid.system"

AndroidManifest.xml (core\tests\hosttests\test-apps\downloadmanagertestapp): android:sharedUserId="android.uid.system">

AndroidManifest.xml (core\tests\systemproperties): android:sharedUserId="android.uid.system">

AndroidManifest.xml (keystore\tests): android:sharedUserId="android.uid.system">

AndroidManifest.xml (packages\fusedlocation): android:sharedUserId="android.uid.system">

AndroidManifest.xml (packages\inputdevices): android:sharedUserId="android.uid.system">

AndroidManifest.xml (packages\keyguard): android:sharedUserId="android.uid.systemui"

AndroidManifest.xml (packages\settingsprovider): android:sharedUserId="android.uid.system">

AndroidManifest.xml (packages\systemui): android:sharedUserId="android.uid.systemui"

AndroidManifest.xml (packages\wallpaperbackup): android:sharedUserId="android.uid.system" >

AndroidManifest.xml (tests\applaunch): android:sharedUserId="android.uid.system" >

AndroidManifest.xml (tests\dataidletest): android:sharedUserId="android.uid.system">

AndroidManifest.xml (tests\networksecurityconfigtest): android:sharedUserId="android.uid.system">

## android.uid.systemui

AndroidManifest.xml (packages\keyguard): android:sharedUserId="android.uid.systemui"

AndroidManifest.xml (packages\systemui): android:sharedUserId="android.uid.systemui"

# 案例

## 错误码分析

### INSTALL\_FAILED\_TEST\_ONLY

#### 源码分析

**final int installFlags = args.installFlags;**

String pkgName = res.name = pkg.packageName;

if ((pkg.applicationInfo.flags&ApplicationInfo.FLAG\_TEST\_ONLY) != 0) {

if ((installFlags & PackageManager.INSTALL\_ALLOW\_TEST) == 0) {

res.setError(INSTALL\_FAILED\_TEST\_ONLY, "installPackageLI");

return;

}

}

#### 解决办法

只要在AndroidStudio3.0内，点击绿三角运行Run，跑出来的apk，一定是TEST\_ONLY的

先用Build > Build APK(s)生成APK，再在命令行下使用ADB安装APK。

检查\app\src\main\AndroidMainfest.xml中是否有testOnly属性为true，如果有去掉或者改为false

2、检查Android Studio和gradle版本是否为alpha版本，换为稳定版本

3、检查是否关闭Instant Run功能，关掉此功能

4、检查是否使用alpha版本的依赖库，换成稳定版本

5、如果以上均无法解决，就在gradle.properties文件中添加android.injected.testOnly=false

#### 方法2：

adb push \*.apk /tmp

adb shell pm install -t /tmp/\*.apk

### INSTALL FAILED CONFLICTING PROVIDER

#### 问题原因

在Android中authority要求必须是唯一的：不同applicationId&&同authority报错，同一个applicationId&&不同authority不会报错。

比如你在定义一个provider时需要为它指定一个唯一的authority。如果你在安装一个带有provider的应用时，系统会检查当前已安装应用的authority是否和你要安装应用的authority相同，如果相同则会弹出上述警告，并且安装失败。

比如2个应用都是fileprovider，**com.facebook.app.FacebookContentProvider**

<provider

android:name="android.support.v4.content.FileProvider"

android:authorities="com.facebook.app.FacebookContentProvider "

android:grantUriPermissions="true"

android:exported="false">

<meta-data

android:name="android.support.FILE\_PROVIDER\_PATHS"

android:resource="@xml/file\_paths" />

</provider>

<provider  
 android:name="com.facebook.FacebookContentProvider"  
 android:authorities="com.facebook.app.FacebookContentProvider"  
 android:exported="true"

tools:replace="android:authorities" />

Log查看

|  |
| --- |
| 09-17 11:01:20.288 W/PackageManager( 488): Package couldn't be installed in /data/app/kapp4.v4-2  09-17 11:01:20.288 W/PackageManager( 488): com.android.server.pm.PackageManagerException: Can't install because provider name com.facebook.app.FacebookContentProvider111111111（in kapp4.v4) is already used by kapp3  09-17 11:01:20.288 W/PackageManager( 488): at com.android.server.pm.PMS.scanPackageDirtyLI(PMS.java:5872)  09-17 11:01:20.288 W/PackageManager( 488): at com.android.server.pm.PMS.scanPackageLI(PMS.java:5564)  09-17 11:01:20.288 W/PackageManager( 488): at com.android.server.pm.PMS.replaceSystemPackageLI(PMS.java:10755)  09-17 11:01:20.288 W/PackageManager( 488): at com.android.server.pm.PMS.replacePackageLI(PMS.java:10581)  09-17 11:01:20.288 W/PackageManager( 488): at com.android.server.pm.PMS.installPackageLI(PMS.java:11045)  09-17 11:01:20.288 W/PackageManager( 488): at com.android.server.pm.PMS.access$2700(PMS.java:249)  09-17 11:01:20.288 W/PackageManager( 488): at com.android.server.pm.PMS$7.run(PMS.java:8953) |

#### 解决方案

在定义provider是，使用软编码的形式，如下：

<provider

android:name="android.support.v4.content.FileProvider"

android:authorities="${applicationId}.fileprovider"

android:grantUriPermissions="true"

android:exported="false">

<meta-data

android:name="android.support.FILE\_PROVIDER\_PATHS"

android:resource="@xml/file\_paths" />

</provider>

上述代码中通过${applicationId}.fileprovider的形式来指定provider的authorities，所以该provider的authorities会根据applicationId的不同而不同，从而避免了authorities的冲突问题。

<**provider  
 android:name="com.facebook.FacebookContentProvider"  
 android:authorities="com.facebook.app.FacebookContentProvider${facebookId}"  
 android:exported="true"  
 tools:replace="android:authorities"** />

INSTALL\_FAILED\_CONFLICTING\_PROVIDER问题完美解决方案 <http://www.jianshu.com/p/ad8c066e9166>

#### 错误方法

删除了phone.debug.go也没用？

-d有无也没有用

cnt=1000 可用？

pm list packages | grep dji.go.v4 none

root@cs500c:/system/priv-app/DJI-GO4 # ls -al

-rw-r--r-- root root 135979984 2017-09-15 09:59 DJI-GO4.apk

<package name="dji.go.v4" codePath="/system/priv-app/DJI-GO4" version="25701"

clear data reboot 恢复称为cs版本

C:\Users\key.guan\DJI-Launcher\DJI-Launcher.apk: 1 file pushed. 3.5 MB/s (17748532 bytes in 4.824s)

#### 源码分析

1. **if** ((scanFlags & SCAN\_NEW\_INSTALL) != 0) {
2. **for** (**int** i=0; i<pkg.providers.size(); i++) {
3. PackageParser.Provider p = pkg.providers.get(i);
4. **if** (p.info.authority != **null**) {
5. String names[] = p.info.authority.split(";");
6. **for** (**int** j = 0; j < names.length; j++) {
7. if (mProvidersByAuthority.containsKey(names[j])) {
8. PackageParser.Provider other = mProvidersByAuthority.get(names[j]);
9. **final** String otherPackageName = ((other != **null** && other.getComponentName()
10. != **null**) ? other.getComponentName().getPackageName() : "?");
11. **throw** **new** PackageManagerException(**INSTALL\_FAILED\_CONFLICTING\_PROVIDER**,
12. "Can't install because provider name " + names[j] +
13. " (in package " + pkg.applicationInfo.packageName
14. + ") is already used by " + otherPackageName);
15. }
16. }
17. }
18. }
19. }

### INSTALL\_FAILED\_DUPLICATE\_PERMISSION

adb install /Users/bob.wang/softdep2/dji-phantom3-a/build/outputs/apk/DJI-Phantom3-A-dpadcs-debug.apk   
/Users/bob.wang/softdep2/dji-phantom3-a/bu...hed. 9.7 MB/s (180326736 bytes in 17.768s  
pkg: /data/local/tmp/DJI-Phantom3-A-dpadcs-debug.apk  
Failure [INSTALL\_FAILED\_DUPLICATE\_PERMISSION perm=dji.permission.broadcast pkg=dji.go.v4.debug]

### INSTALL\_FAILED\_UPDATE\_INCOMPATIBLE签名不合法

W/PackageManager: Package dji.go.v4 signatures do not match the previously installed version; ignoring!

F:\key.guan\dev\gitGo4\DJI-Phantom3-A\build\outputs\apk\DJI-Phantom3-A-dpadcs-debug.apk: 1 file pushed. 4.5 MB/s (180307582 bytes in 38.365s

pkg: /data/local/tmp/DJI-Phantom3-A-dpadcs-debug.apk

Failure [INSTALL\_FAILED\_UPDATE\_INCOMPATIBLE]

签名不同造成的吧

？？？？

C:\Users\key.guan\Desktop\A>adb install -r F:\key.guan\dev\jenkens\DJI-171212-161-28376\_officialInnerfactory\_sec.apk

Failed to install F:\key.guan\dev\jenkens\DJI-171212-161-28376\_officialInnerfactory\_sec.apk: Failure [INSTALL\_FAILED\_DUPLICATE\_PERMISSION: Package dji.go.v4.fac

tory attempting to redeclare permission dji.permission.broadcast already owned by dji.go.v4.debug]

C:\Users\key.guan\Desktop\A>

C:\Users\key.guan\Desktop\A>

C:\Users\key.guan\Desktop\A>

C:\Users\key.guan\Desktop\A>

C:\Users\key.guan\Desktop\A>

C:\Users\key.guan\Desktop\A>

C:\Users\key.guan\Desktop\A>

C:\Users\key.guan\Desktop\A>adb uninstall dji.go.v4.debug

Success

C:\Users\key.guan\Desktop\A>adb install -r F:\key.guan\dev\jenkens\DJI-171212-161-28376\_officialInnerfactory\_sec.apk

### 安装版本过低

安装app个数的限制

### INSTALL\_FAILED\_DEXOPT

#### 问题描述

>adb pull /system/priv-app/A/A.apk

>adb install -r A.apk

Failure [INSTALL\_FAILED\_DEXOPT]

|  |
| --- |
| 11-23 13:14:58.054 514-552/system\_process I/ActivityManagerService: Start proc 1362:com.android.defcontainer/u0a2 for service com.android.defcontainer/.DefaultContainerService  11-23 13:14:58.118 1362-1378/com.android.defcontainer D/DefContainer: Copying /data/local/tmp/kpad-Launcher.apk to base.apk  11-23 13:14:58.654 514-552/system\_process D/PackageManager: Renaming /data/app/vmdl442937071.tmp to /data/app/kpad.system.launcher-1  11-23 13:14:58.654 514-535/system\_process I/ActivityManagerService: Force stopping kpad.system.launcher appid=1000 user=-1: replace sys pkg  11-23 13:14:58.655 514-535/system\_process I/ActivityManagerService: Killing 942:kpad.system.launcher/1000 (adj 0): stop kpad.system.launcher  11-23 13:14:58.655 514-552/system\_process W/PackageManager: Trying to update system app code path from /system/app/kpad-Launcher to /data/app/kpad.system.launcher-1  11-23 13:14:58.660 514-569/system\_process D/WifiService: Client connection lost with reason: 4  11-23 13:14:58.661 514-535/system\_process I/ActivityManagerService: Force finishing activity 3 ActivityRecord{32ab6df9 u0 kpad.system.launcher/.Launcher t3}  11-23 13:14:58.662 514-552/system\_process I/PackageManager: Running dexopt on: /data/app/kpad.system.launcher-1/base.apk pkg=kpad.system.launcher isa=arm vmSafeMode=false  11-23 13:14:58.662 514-535/system\_process D/ActivityManagerService: moveHomeStackTaskToTop: moving TaskRecord{1d452549 #3 A=kpad.system.launcher U=0 sz=1}  11-23 13:14:58.663 1381-1381/? E/SchedPolicy: open of /dev/blkio/bg\_non\_interactive/tasks failed: Permission denied  11-23 13:14:58.663 514-535/system\_process D/ActivityManagerService: moveHomeStackTaskToTop: moving TaskRecord{1d452549 #3 A=kpad.system.launcher U=0 sz=1}  11-23 13:14:58.665 514-764/system\_process W/ActivityManagerService: Spurious death for ProcessRecord{27434e4e 942:kpad.system.launcher/1000}, curProc for 942: null  11-23 13:14:58.712 1381-1381/? I/dex2oat: Error opening the directory: /system/lib/plugins  11-23 13:14:58.712 1381-1381/? I/dex2oat: /system/bin/dex2oat --zip-fd=5 --zip-location=/data/app/kpad.system.launcher-1/base.apk --oat-fd=6 --oat-location=/data/dalvik-cache/arm/data@app@kpad.system.launcher-1@base.apk@classes.dex --instruction-set=arm --instruction-set-features=div --runtime-arg -Xms64m --runtime-arg -Xmx512m --swap-fd=7  11-23 13:14:58.718 1381-1381/? E/dex2oat: Failed to open dex from file descriptor for zip file '/data/app/kpad.system.launcher-1/base.apk': Entry not found  11-23 13:14:58.943 171-171/? E/installd: DexInv: --- END '/data/app/kpad.system.launcher-1/base.apk' --- status=0x0100, process failed  11-23 13:14:58.945 514-552/system\_process W/PackageManager: Package couldn't be installed in /data/app/kpad.system.launcher-1  com.android.server.pm.PackageManagerException: scanPackageLI  at com.android.server.pm.PMS.scanPackageDirtyLI(PMS.java:6268)  at com.android.server.pm.PMS.scanPackageLI(PMS.java:5564)  at com.android.server.pm.PMS.replaceSystemPackageLI(PMS.java:10755)  at com.android.server.pm.PMS.replacePackageLI(PMS.java:10581)  at com.android.server.pm.PMS.installPackageLI(PMS.java:11063)  at com.android.server.pm.PMS.access$2700(PMS.java:249)  at com.android.server.pm.PMS$7.run(PMS.java:8953)  at android.os.Handler.handleCallback(Handler.java:739)  at android.os.Handler.dispatchMessage(Handler.java:95)  at android.os.Looper.loop(Looper.java:135)  at android.os.HandlerThread.run(HandlerThread.java:61)  at com.android.server.ServiceThread.run(ServiceThread.java:46)  11-23 13:14:58.948 514-552/system\_process E/PackageManager: Unable to get canonical file java.io.IOException: Cannot create /system/app/kpad-Launcher/lib  11-23 13:14:59.076 514-552/system\_process I/art: Explicit concurrent mark sweep GC freed 16714(1031KB) AllocSpace objects, 4(179KB) LOS objects, 33% free, 6MB/9MB, paused 936us total 63.379ms  11-23 13:14:59.081 1349-1349/? I/art: System.exit called, status: 1 |

#### Packages.xml异常

<package name="dji.go.v4" codePath="/system/priv-app/DJI-AS" nativeLibraryPath="/data/app/dji.go.v4-1/lib" primaryCpuAbi="armeabi-v7a" flags="1075363397" ft="15fe43b89a0" it="15fe43b89a0" ut="15fe43b89a0" version="27398" userId="10001">

Apk搞错了。。。。

## 扫描加速

scanDirTracedLI采用并行扫描IO

Runtime.*getRuntime*().availableProcessors()\*2，全部资源都来扫描吧

## 不是bug

W/PackageParser: k-baseapk,tagName=uses-sdk

01-21 09:21:57.126 4320-4371/com.dpad.launcher W/PackageParser: k-baseapk,tagName=meta-data

01-21 09:21:57.126 4320-4371/com.dpad.launcher W/PackageParser: k-baseapk,Unknown element under <manifest>: meta-data at /data/data/com.dpad.launcher/cache/skins/skin-white.apk Binary XML file line #11

01-21 09:21:57.126 4320-4371/com.dpad.launcher W/PackageParser: k-baseapk,tagName=application

under <manifest>:节点下的meta-data是不被识别的。。

## lib如何链接到正确的so目录

问题描述： 安装最新的版本MG2，在执行任务的界面，natvie奔溃

问题分析：新的编解码框架@max 使用APP的一个so的api接口获得连接目录，mContext.getFilesDir().getParent() + "/lib"获取的是data分区的lib，由于是系统预装的都在system分区下，故无法找到

解决办法：app对so库的链接改为mContext.getApplicationInfo().nativeLibraryDir;

原理分析：

Android5.1（3288）：

系统预装： /data/data/dji.prof.mg2/lib-> /data/app-lib/dji.prof.mg2(不存在的目录)

Android7.1（3399）：

增强了预装app的data下lib链接有效性，废弃了/data/app-lib/dji.prof.mg2目录

系统预装：/data/**user/0/**dji.prof.mg2/lib->**/system/priv-app/mg2/lib/arm**

手动安装：/data/user/0/dji.prof.mg2/**lib -> /data/app/dji.prof.mg2-1/lib/arm**

**因此，在3288上通过**mContext.getFilesDir().getParent() + "/lib"必然得到的一个空的lib目录，但是在3399上则可以到系统的正常目录

技术学习：

/data/**user/0/**dji.prof.mg2/lib是什么时候创建的？什么时候链接到**priv-app**目录的

**/system/priv-app/mg2/lib/arm**

1|ag411:/data/user/0/dji.prof.mg2 # ls -al | grep lib

lrwxrwxrwx 1 root root 28 2013-01-18 16:50 **lib -> /system/priv-app/mg2/lib/arm**

drwx------ 2 u0\_a43 u0\_a43 3488 2020-01-14 19:27 lib-main

/data/app-lib/目录目前为空，是为了兼容5.1么

nativeLibraryDir是什么时候复制的，为什么系统预装和用户安装都能能够准确更新

lib-main是啥，是进入界面才有的工作，为啥要这么设计？

### Lib库更新原理：PkgMS.setNativeLibraryPaths

A：是什么时候复制的，为什么系统预装和用户安装都能能够准确更新

对于刚出厂的系统预装app

**final boolean** bundledApp = info.isSystemApp() && !info.isUpdatedSystemApp();

**final** String libDir = is64Bit ? LIB64\_DIR\_NAME : LIB\_DIR\_NAME;

**if** (bundledApp) {

info.nativeLibraryRootDir = Environment.buildPath(**new** File(apkRoot), libDir,  
 apkName).getAbsolutePath();//**/system/priv-app/mg2/lib**

**else** { **/data/app/dji.prof.mg2-1/lib**  
 **final** String apkName = deriveCodePathName(codePath);  
 info.nativeLibraryRootDir = **new** File(mAppLib32InstallDir, apkName)  
 .getAbsolutePath();  
}  
info.nativeLibraryDir = info.nativeLibraryRootDir;

### lib-main是啥

lib-main是啥，是进入界面才有的工作，为啥要这么设计？这是用户行为

SoLoader 是facebook推出的一个So文件加载库，它封装了System.loadLibrary并且能够处理So文件的依赖。

当React Native应用第一次启动时，SoLoader会在/data/data/package\_name下创建一个lib-main文件夹并根据规则提取apk中的So文件到其中。

我们都知道，Android应用所需要的So文件一般被放在/data/data/package\_name/lib目录中，这个目录的实际地址我们可以通过ApplicationInfo的nativeLibraryDir获取到。

我们都知道，同一个应用加载的所有So库必须是同一个指令集，如果混用将会报错崩溃。因此，SoLoader不仅遵循了Android系统的So文件提取与加载的策略，同时为了将自己的提取与加载逻辑与系统行为隔离开，特意创建了一个lib-main目录，可谓用心良苦。

<https://zhuanlan.zhihu.com/p/78631609>

这个是第三方应用，mg用了react native吧，然而这些没有什么用的，建议删除之，避免浪费空间

## 本地库（Native Library）的选择流程--Android6.0

### 什么是Native Library？

Native Library，一般我们译为本地库或原生库，是由C/C++编写的动态库(.so)，并通过JNI(Java Native Interface)机制为java层提供接口。应用一般会出于性能、安全等角度考虑将相关逻辑用C/C++实现并编译为库的形式提供接口，供上层或其他模块调用。

### 为什么需要本地库（Native Library）的选择？

我们知道本地库（Native Library）是由C/C++源码编译后的动态库文件，其编译是C/C++源码根据其要运行的目标平台的指令集翻译成对应的机器码的过程，不同的处理器架构，需要编译出相应平台的动态库，才能被正确的执行。因此为了让app能在不同的处理器平台上都能正确的运行，一般app都会将其C/C++源文件为常见平台（armeabi，armeabi-v7a等）都编译出相应的本地库文件，在打包成apk文件时，将这些不同平台的库都打包进apk的lib子目录中，每一种abi存放一个目录，目录名为abi的类型名。因此，一个apk文件的lib子目录中就可能同时包含多种abi库文件情况，当一个用户在自己的设备上**安装**该app时，系统就需要根据自己的所支持的abi类型，为app选择适当abi类型的库文件进行安装，以确保app在系统上正常运行。

### Android6.0，本地库（Native Library）的选择流程

上文提到本地库（Native Library）的选择流程是在app被**安装**的时候进行的，下面就以此为切入点，来看看在android6.0安装app时，是怎么为其选择合适的本地库（Native Library）的。

首先，我们来看一下PMS(PackageManagerService)的scanPackageDirtyLI函数，它是负责app安装时签名验证、本地库（Native Library）的选择安装等系列工作的，这里我们重点看一下本地库（Native Library）的选择流程：

frameworks/base/services/core/java/com/android/server/pm/PackageManagerService.java

private PackageParser.Package scanPackageDirtyLI(PackageParser.Package pkg, int parseFlags, int scanFlags, long currentTime, UserHandle user) throw PackageManagerException{

....

/\*计算cpuOverride的值，pkg.cpuAbiOverride来自安装参数指定，

pkg.cpuAbiOverrided的优先级高于pkgSetting.cpuAbiOverrideString，

只要pkg.cpuAbiOverrided不等于NativeLibraryHelper.CLEAR\_ABI\_OVERRIDE

\*／

final String cpuAbiOverride = deriveAbiOverride(pkg.cpuAbiOverride, pkgSetting);

if ((scanFlags & SCAN\_NEW\_INSTALL) == 0) {

/\* 这里执行abi的选择和本地库的安装操作，最后一个参数表示是否要将本地库抽取出来，这里为true，表示需要\*/

derivePackageAbi(pkg, scanFile, cpuAbiOverride, true /\* extract libs \*/);

}else{

....

}

....

}

可以看到在scanPackageDirtyLI中主要是调用了derivePackageAbi执行abi的选择和安装操作，接下来我们看一下 derivePackageAbi的实现：  
frameworks/base/services/core/java/com/android/server/pm/PackageManagerService.java

public void derivePackageAbi(PackageParser.Package pkg, File scanFile,

String cpuAbiOverride, boolean extractLibs)

throws PackageManagerException {

setNativeLibraryPaths(pkg);//主要是初始化和设置与本地相关的几个变量，如本地库的目录等

....

NativeLibraryHelper.Handle handle = null;

try {

handle = NativeLibraryHelper.Handle.create(scanFile); //打开apk文件

final File nativeLibraryRoot = new File(nativeLibraryRootStr);

// 将primaryCpuAbi和secondaryCpui都先置为null，以便于后面重新计算abi

pkg.applicationInfo.primaryCpuAbi = null;

pkg.applicationInfo.secondaryCpuAbi = null;

if (isMultiArch(pkg.applicationInfo)) {

// pkg.applicationInfo.pkgFlags设置了FLAG\_MULTIARCH标志位，

//即表示该app的代码会被其他app进程加载，因此就可能需要同时安装32位和64位的本地库

// Warn if we've set an abiOverride for multi-lib packages..

// By definition, we need to copy both 32 and 64 bit libraries for

// such packages.

...

int abi32 = PackageManager.NO\_NATIVE\_LIBRARIES;

int abi64 = PackageManager.NO\_NATIVE\_LIBRARIES;

if (Build.SUPPORTED\_32\_BIT\_ABIS.length > 0) {

//系统支持32位的abi

if (extractLibs) {

//表示需要为该app选择、\*\*安装\*\*合适的abi

abi32 = NativeLibraryHelper.copyNativeBinariesForSupportedAbi(handle,

nativeLibraryRoot, Build.SUPPORTED\_32\_BIT\_ABIS,

useIsaSpecificSubdirs);//调用NativeLibraryHelp的copyNativeBinariesForSupportedAbi进行abi的选择和安装

} else {

//只需选择（计算）出合适的abi即可，\*\*不用执行安装\*\*操作

abi32 = NativeLibraryHelper.findSupportedAbi(handle, Build.SUPPORTED\_32\_BIT\_ABIS);//调用NativeLibraryHelp的findSupportedAbi进行abi的选择 --- 这是本文关注的重点

}

}

maybeThrowExceptionForMultiArchCopy(

"Error unpackaging 32 bit native libs for multiarch app.", abi32);

//系统支持64位的abi。流程和32位abi选择流程相同，只是将调用参数中的候选abi列表改为了64位的，

//即Build.SUPPORTED\_32\_BIT\_ABIS改为Build.SUPPORTED\_64\_BIT\_ABIS

if (Build.SUPPORTED\_64\_BIT\_ABIS.length > 0) {

if (extractLibs) {

abi64 = NativeLibraryHelper.copyNativeBinariesForSupportedAbi(handle,

nativeLibraryRoot, Build.SUPPORTED\_64\_BIT\_ABIS,

useIsaSpecificSubdirs);

} else {

abi64 = NativeLibraryHelper.findSupportedAbi(handle, Build.SUPPORTED\_64\_BIT\_ABIS);

}

}

maybeThrowExceptionForMultiArchCopy(

"Error unpackaging 64 bit native libs for multiarch app.", abi64);

//根据计算结果，对primaryCpuAbi和secondaryCpuAbi进行赋值，注意这里查找结果abi32，abi64分别是在Build.SUPPORTED\_32\_BIT\_ABIS和Build.SUPPORTED\_64\_BIT\_ABIS列表中的下标

/\*赋值的规则如下：

1. 如果同时找到64位和32位的abi，则primaryCpuAbi设为64位的，secondaryCpuAbi设为32位的

2. 只找到64位但是没有找到32位的abi，则primaryCpuAbi设为64位的，secondaryCpuAbi保持为null

3. 如果没有找到64位但是找到了32位合适的abi，primaryCpuAbi设为32为的abi，secondaryCpuabi保持为null

4. 64位和32位的abi都没找到，则primaryCpuAbi和secondaryCpuAbi都保持为null

\*/

if (abi64 >= 0) {

pkg.applicationInfo.primaryCpuAbi = Build.SUPPORTED\_64\_BIT\_ABIS[abi64];

}

if (abi32 >= 0) {

final String abi = Build.SUPPORTED\_32\_BIT\_ABIS[abi32];

if (abi64 >= 0) {

pkg.applicationInfo.secondaryCpuAbi = abi;

} else {

pkg.applicationInfo.primaryCpuAbi = abi;

}

}

} else {

//app的代码不会被其他app进程载入

/\*

初始化候选abi列表：abiList。cpuAbiOverride优先级高于Build.SUPPORTED\_ABIS，

前者是根据安装参数和PackageSetting计算出来的，后者是系统编译时生成的

\*／

String[] abiList = (cpuAbiOverride != null) ?

new String[] { cpuAbiOverride } : Build.SUPPORTED\_ABIS;

...

//以下逻辑与if语句中的逻辑相同，copyRet存储合适的abi在候选abi列表（abiList）中的下标

final int copyRet;

if (extractLibs) {

copyRet = NativeLibraryHelper.copyNativeBinariesForSupportedAbi(handle,

nativeLibraryRoot, abiList, useIsaSpecificSubdirs);

} else {

copyRet = NativeLibraryHelper.findSupportedAbi(handle, abiList);

}

if (copyRet < 0 && copyRet != PackageManager.NO\_NATIVE\_LIBRARIES) {

throw new PackageManagerException(INSTALL\_FAILED\_INTERNAL\_ERROR,

"Error unpackaging native libs for app, errorCode=" + copyRet);

}

//对primaryCpuAbi进行赋值

if (copyRet >= 0) {

pkg.applicationInfo.primaryCpuAbi = abiList[copyRet];

} else if (copyRet == PackageManager.NO\_NATIVE\_LIBRARIES && cpuAbiOverride != null) {

//NativeLibraryHelper没找到合适的abi，但是cpuAbiOverride不为null，则primaryCpuAbi设为cpuAbiOverride

pkg.applicationInfo.primaryCpuAbi = cpuAbiOverride;

} else if (needsRenderScriptOverride) {

pkg.applicationInfo.primaryCpuAbi = abiList[0];

}

}

} catch (IOException ioe) {

Slog.e(TAG, "Unable to get canonical file " + ioe.toString());

} finally {

//关闭apk文件

IoUtils.closeQuietly(handle);

}

// 计算出本地库，再一次更新本地库目录及其相关信息

setNativeLibraryPaths(pkg);

}

接下来我们就到NativeLibraryHelper的findSupportedAbi中看看，具体是怎么进行abi选择的：  
frameworks/base/core/java/com/android/internal/content/NativeLibraryHelper.java

public static int findSupportedAbi(Handle handle, String[] supportedAbis) {

int finalRes = NO\_NATIVE\_LIBRARIES;

/\*handle为上文中PMS中derivePackageAbi调用NativeLibraryHelper.Handle.create

\* 打开的。因为存在Split APK机制，一个app有可能被拆分成几个apk情况，

\* create函数会打开一个app所有的apk文件，并存放在apkHandles数组中

\*／

//依次处理一个app的每一个apk文件

for (long apkHandle : handle.apkHandles) {

//直接调用nativeFindSupportedAbi对每一个apk进行本地库的选择

final int res = nativeFindSupportedAbi(apkHandle, supportedAbis);

if (res == NO\_NATIVE\_LIBRARIES) {

// 这个apk不存在本地库，什么也不做

} else if (res == INSTALL\_FAILED\_NO\_MATCHING\_ABIS) {

//apk存在本地库，但是不在系统支持的列表中，

//将返回值暂时先置为INSTALL\_FAILED\_NO\_MATCHING\_ABIS

if (finalRes < 0) {

finalRes = INSTALL\_FAILED\_NO\_MATCHING\_ABIS;

}

} else if (res >= 0) {

// apk存在本地库，且在系统支持的列表中，

// 如果其在系统支持列表的索引比之前apk找到的低，则更新返回值

if (finalRes < 0 || res < finalRes) {

finalRes = res;

}

} else {

// 未知错误，直接结束，返回相应的错误值

return res;

}

}

return finalRes;

}

总的来说findSupportedAbi还是比较简单，依次对一个app的所有apk调用nativeFindSupportedAbi来选择合适的本地库。 如果多个apk文件都找到了本地库，则根据其在supportedAbis索引值，选择索引较小的。因此可以看出系统支持的本地库优先级，在supportedAbis中是按高到低依次排列的。接下来我们就来看一下，对每一个apk进行本地库选择的nativeFindSupportedAbi。

在NativeLibraryHelper中，nativeFindSupportedAbi的定义如下

private native static int nativeFindSupportedAbi(long handle, String[] supportedAbis);

是一个native函数，其定义在：  
frameworks/base/core/java/jni/com\_android\_internal\_content\_NativeLibraryHelper.cpp

其中JNINativeMethod定义如下：

static JNINativeMethod gMethods[] = {

{

...

{"nativeFindSupportedAbi",

"(J[Ljava/lang/String;)I",

(void \*)com\_android\_internal\_content\_NativeLibraryHelper\_findSupportedAbi},

...

};

可见NativeLibraryHelper.java中调用的nativeFindSupportedAbi在com\_android\_internal\_content\_NativeLibraryHelper.cpp中定义的函数名为com\_android\_internal\_content\_NativeLibraryHelper\_findSupportedAbi，其定义如下：

static jint

com\_android\_internal\_content\_NativeLibraryHelper\_findSupportedAbi(JNIEnv \*env, jclass clazz, jlong apkHandle, jobjectArray javaCpuAbisToSearch)

{

return (jint) findSupportedAbi(env, apkHandle, javaCpuAbisToSearch);

}

该函数只是简单的调用findSupportedAbi，本身没有做工作。我们接着看findSupportedAbi的实现：

static int findSupportedAbi(JNIEnv \*env, jlong apkHandle, jobjectArray supportedAbisArray) {

/\*

\* 先将java层的传入的系统支持abi类型列表(String[])转化为C++层的表示(VectorM<SCopedUtfChars)，这样做的目的应该是为了便于后续字符串比较之类的操作更加方便。

\*/

const int numAbis = env->GetArrayLength(supportedAbisArray);

Vector<ScopedUtfChars\*> supportedAbis;

for (int i = 0; i < numAbis; ++i) {

supportedAbis.add(new ScopedUtfChars(env,

(jstring) env->GetObjectArrayElement(supportedAbisArray, i)));

}

//将java层NativeLibraryHelp.java中findSupportedAbi传入的apkHandle(long)转化为ZipFileRO\*

ZipFileRO\* zipFile = reinterpret\_cast<ZipFileRO\*>(apkHandle);

if (zipFile == NULL) {

return INSTALL\_FAILED\_INVALID\_APK;

}

/\* 创建NativeLibrariesIterator迭代器，该迭代器主要做了以下封装：

\* 1. 在打开的时候会自动打开apk文件的lib目录

\* 2. next操作会自动迭代apk文件lib目录中的本地库文件，自动跳过非本地库文件，并更新相关变量。

\*/

UniquePtr<NativeLibrariesIterator> it(NativeLibrariesIterator::create(zipFile));

if (it.get() == NULL) {

return INSTALL\_FAILED\_INVALID\_APK;

}

ZipEntryRO entry = NULL;

int status = NO\_NATIVE\_LIBRARIES;

//通过迭代器的next函数，依次迭代lib目录下的本地库文件

while ((entry = it->next()) != NULL) {

//找到了本地库，且初始化后未被修改过，则将返回值暂且设为INSTALL\_FAILED\_NO\_MATCHING\_ABIS

if (status == NO\_NATIVE\_LIBRARIES) {

status = INSTALL\_FAILED\_NO\_MATCHING\_ABIS;

}

const char\* fileName = it->currentEntry();//文件名，包括相对路径的目录名，如lib/armeabi/libxx.so

const char\* lastSlash = it->lastSlash();//路径中，最后一个`/`的位置

// 分别获取子目录名的起始地址和目录名长度-->这个目录名就是这个本地库文件的abi类型

const char\* abiOffset = fileName + APK\_LIB\_LEN;

const size\_t abiSize = lastSlash - abiOffset;

//检查候选abi列表是否包含该abi类型

for (int i = 0; i < numAbis; i++) {

const ScopedUtfChars\* abi = supportedAbis[i];

if (abi->size() == abiSize && !strncmp(abiOffset, abi->c\_str(), abiSize)) {

//该本地库文件的abi类型在候选abi列表中

if (((i < status) && (status >= 0)) || (status < 0) ) {

//该abi类型在候选abi列表的索引小于已经找到的值或者之前还未找到合适的abi类型，则更新返回值

status = i;

}

}

}

}

//回收内存

for (int i = 0; i < numAbis; ++i) {

delete supportedAbis[i];

}

return status;

}

可以看到，真正的本地库的abi类型选择是在native层完成的，核心逻辑其实也挺简单：即查看apk文件中lib子目录，看其包含了哪些子目录，每个目录名即一种abi类型；再查看这些abi类型是否在系统支持的abi列表改中，如果在则该abi就是app在系统中将要使用的abi类型，如果同时存在多种类型满足的情况，则选取索引在候选abi列表中较小的那个。

### 总结一下要点

* 如果app设置了FLAG\_MULTIARCH标志位（即，app要被载入其他app进程）, 需要同时选择64位和32位的abi，并根据结果，分别对primaryCpuAbi和secondaryCpuAbi赋值；否则，判断一下安装时是否指定了abi类型，如果指定了，就以该值作为abi候选列表，没有指定就用Build.SUPPORTED\_ABIS（它是一个String[]类型，每个元素是一种abi类型的名称，如 armeabi，armeabi-V7A，arm64-V8A，X86等），在系统编译时确定的。
* apk文件是zip文件类型，本地库文件就放在它的lib/子目录下，每一种abi类型的本地库文件存放一个目录，目录名就是该abi类型名。
* 一个app可能拆分成几个apk文件组成。在为app进行abi类型选择时，依次查看它每一个apk文件，选出最合适abi类型，如果一个app不同apk文件选出的abi类型不同，则为app选取在候选abi列表中索引最小的那个。apk文件的最合适abi类型的标准是：该abi类型在abi候选列表中，且索引最小。

作者：巫屋  
链接：https://www.jianshu.com/p/4aaa66c62fef  
来源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

## Ref

[深度探究apk安装过程](http://www.sohu.com/a/132599459_465908)

[Android应用程序安装过程解析(源码角度)](http://www.jianshu.com/p/21412a697eb0)

[pms包管理服务分析-初步理解](http://blog.csdn.net/lgglkk/article/details/54579396)

[pms包管理服务分析-PMS构造函数和包扫描过程](http://blog.csdn.net/lgglkk/article/details/54839662)

// TODO word

# queryIntentActivities

@Override  
**public** List<ResolveInfo> queryIntentActivities(Intent intent,  
 **int** flags) {  
 **return** queryIntentActivitiesAsUser(intent, flags, **mContext**.getUserId());  
}

**public** List<ResolveInfo> queryIntentActivitiesAsUser(Intent intent,  
 **int** flags, **int** userId) {  
 **try** {  
 ParceledListSlice<ResolveInfo> parceledList =  
 **mPM**.queryIntentActivities(intent,  
 intent.resolveTypeIfNeeded(**mContext**.getContentResolver()),  
 flags, userId);  
 **if** (parceledList == **null**) {  
 **return** Collections.*emptyList*();  
 }  
 **return** parceledList.getList();  
 } **catch** (RemoteException e) {  
 **throw** e.rethrowFromSystemServer();  
 }  
}

IPackageManager pm = ActivityThread.*getPackageManager*();  
**if** (pm != **null**) {  
 *// Doesn't matter if we make more than one instance.* **return** (**mPackageManager** = **new** ApplicationPackageManager(**this**, pm));  
}

**public static** IPackageManager getPackageManager() {  
 **if** (sPackageManager != **null**) {  
 **return** sPackageManager;  
 } **else** {  
 IBinder b = ServiceManager.getService(**"package"**);  
 sPackageManager = Stub.asInterface(b);  
 **return** sPackageManager;  
 }  
}

## PMS .queryIntentActivities

@Override  
**public** @NonNull ParceledListSlice<ResolveInfo> queryIntentActivities(Intent intent,  
 String resolvedType, **int** flags, **int** userId) {  
 **try** {  
 Trace.traceBegin(TRACE\_TAG\_PACKAGE\_MANAGER, **"queryIntentActivities"**);  
  
 **return new** ParceledListSlice<>(  
 queryIntentActivitiesInternal(intent, resolvedType, flags, userId));  
 } **finally** {  
 Trace.traceEnd(TRACE\_TAG\_PACKAGE\_MANAGER);  
 }  
}

## PMS.queryIntentActivitiesInternal

# 其他系统打包

加固为什么有问题，主要也是路径找不到了

Cnt是怎么限制

普通版是怎么限制的

奇巧淫技比较多一些

So是怎么拷贝的，之前的地图问题，帮帮加密的问题

# 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)

# 系统签名原理

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

在其中添加LOCAL\_CERTIFICATE := platform 或 shared 或 media

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？

### 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'

}

}

# Dump

# Pm

frameworks/base/cmds/pm/src/com/android/commands/pm/Pm.java

pm clear packagename 在root设备才生效！

pm list packages –f

查看所有已安装的应用的包名

pm disable <package name>

禁用启用app

* pm list packages [-f] [-d] [-e] [-s] [-3] [-i] [-u] [--user USER\_ID] [FILTER]  
  查看应用列表,**pm即PackageManager**

| **参数** | **解释** |
| --- | --- |
| -f | 显示应用关联的 apk 文件 |
| -d | 只显示 disabled 的应用 |
| -e | 只显示 enabled 的应用 |
| -s | 只显示系统应用 |
| -3 | 只显示第三方应用 |
| -i | 显示应用的 installer |
| -u | 包含已卸载应用 |
| <FILTER> | 包名包含 <FILTER> 字符串 |

# REF

主要参考：】

[PackageManagerService启动流程](https://maoao530.github.io/2017/01/10/packagemanager/)

pm 基本使用

# 搜索

|  |  |
| --- | --- |
| google | android 应用程序的安装和显示过程. android5.1 packagemanagerservice |
| 360 | android 应用程序的安装和显示过程 |

filetype:ppt packagemanagerservice

filetype:ppt android应用程序的安装过程

filetype:ppt PMS

filetype:ppt APK

filetype:ppt packagemanagerservice源码

filetype:ppt Android framework源码

# 未完成的链接

[APP权限管理](http://blog.csdn.net/zhejiang9/article/details/52412387)

[Android App安装过程学习笔记](http://palanceli.com/2016/10/23/2016/1015Installer1/)

[Android PackageManager Service详解（5.1源码）（四）](http://blog.csdn.net/zhejiang9/article/details/52412410)

# 连接1213

<http://blog.csdn.net/yxhuang2008/article/details/51783394>

<http://blog.csdn.net/xueshanhaizi/article/details/51077303>