**PMS自顶向下分析**

# 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 |



### 在PackageInstallerActivity.onCreate中

1. 从Intent对象获取Package URL、Scheme信息

2. 校验Scheme，根据Scheme不同的值进行逻辑处理。Scheme为file或者package

|  |
| --- |
| *//scheme只有两个值：file或package*  **if** (scheme != **null** && !**"file"**.equals(scheme) && !**"package"**.equals(scheme)) {  setPmResult(PackageManager.INSTALL\_FAILED\_INVALID\_URI);  finish();  **return**;  }  *// 未知源安装检测*  **if** ((requestFromUnknownSource) && (!isInstallingUnknownAppsAllowed())) {  *//ask user to enable setting first*  showDialogInner(***DLG\_UNKNOWN\_APPS***);  **return**;  } |

### InstallAppProgress

|  |
| --- |
| **if** (**"package"**.equals(**mPackageURI**.getScheme())) {  **try** {  pm.installExistingPackage(**mAppInfo**.**packageName**);  observer.packageInstalled(**mAppInfo**.**packageName**,  PackageManager.INSTALL\_SUCCEEDED);  } **catch** (PackageManager.NameNotFoundException e) {  observer.packageInstalled(**mAppInfo**.**packageName**,  PackageManager.INSTALL\_FAILED\_INVALID\_APK);  }  } **else** {  pm.installPackageWithVerificationAndEncryption(**mPackageURI**, observer, installFlags,  installerPackageName, verificationParams, **null**);  } |

pm.installExistingPackage()方法更新应用，反之如果scheme为file，则调用pm.installPackageWithVerificationAndEncryption()方法进行应用的安装。上面说到的两个方法都是PakcageManager中的方法，**两个方法都是静默安装**，在安装的时候不会出现任何的提示。但是由于上述两个方法在PackageManager中是被注释为@hide的，所以，普通的Android应用中无法调用和访问。静默安装是一个异步过程，所以，无论安装成功或者安装失败，都会向用户弹出结果，所以我们在调用方法中看到有个行参是observer，为PackageInstallObserver实例。PackageInstallObserver主要就是处理安装的结果，其中定义了Handler变量mHandler来进行UI操作，用于实现对用户提示。

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

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



## 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);**  } |



## 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;

}

1

11

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zygotefork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 。

### 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。

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。这里举一个例子：

$adb shell pm disable com.android.providers.drm

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：

<pkg name="com.android.providers.drm" enabled="2" />

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;

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

<?xml version='1.0' encoding='utf-8' standalone='yes' ?>

<package-restrictions>

<pkg name="com.dji.industry.pilot" enabled="2" enabledCaller="dji.system.launcher" />

<pkg name="com.android.provision">

<disabled-components>

<item name="com.android.provision.DefaultActivity" />

</disabled-components>

</pkg>

<pkg name="dji.go.v4" enabled="1" />

<pkg name="dji.pilot" enabled="1" />

**用pm脚本可以模拟测试，比如**

enable com.android.provision/com.android.provision.DefaultActivity

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

### 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

### 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) 。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()

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),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。

### 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()

# installd

在应用程序的管理工作中，有时候需要对存储设备做一些操作，比如创建目录修改目录权限等，这些操作有的是需要特权级的权限。PackageManagerService存活在system\_server进程中，这个进程的用户为system，它是没有特权级的权限的，所以我猜想，出于安全的角度考虑，Android单独将一部分需要特权的工作，转交给installd进程去完成。

在installd的入口函数中，首先是做一些初始化的工作，然后放弃自己的root用户身份，改变了自己的用户类型和组用户类型，但同时它保留了设置目录权限，以及修改目录属主的权限：

所以installd进程创建的socket名为：/dev/socket/installd， 600表示socket的用户权限为可读可写，system表示用户和用户组。

在PackageManagerService扫描安装应用的过程中，给installd先后发送了如下消息：

* install
* dexopt

这三个消息分别由do\_install，do\_dexopt函数来处理。

* do\_install函数为应用创建了以下目录，在install过程中，同时为创建的目录设置uid和gid。
  + 数据目录:/data/data/packageName/
  + lib目录：/data/app-lib/packageName
  + lib符号链接:/data/data/packageName/lib → /data/app-lib/packageName
* do\_dexopt中调用dexopt函数处理apk文件。主要流程为：
  + 首先判断apk所在目录下面是否存在同名的odex文件，如果存在就直接返回；
  + 以apk文件的路径为名创建dex路径。比如在debug版本中，/system/app/Settings.apk的dex文件将会保存在： /data/dalvik-cache/system@[app@Settings.apk](mailto:app@Settings.apk)@classes.dex文件中。dex文件命名规则是/data/dalvik-cache字符串拼接apk的路径，再拼“/classs.dex"。然后将”/data/dalvik-cache/“字符串后面的‘/’替换为@字符。
  + 最后调用系统工具/system/bin/dexopt来提取dex文件。





LocalSocket通信（右）：   


他是什么，有什么用，怎么用，

哪里来的，为什么会出来，存在的合理性，背景

是什么，现状

前景，哪里去，哪里会用到，怎么用

因为PMS所在进程SystemServer属于system用户组，没有root权限，不能完成在文件系统中更改目录、复制、删除文件、APK文件优化等操作，因此这里引入了installd进程

installd是一个native的守护进程，在init.rc中定义

service installd /system/bin/installd

class main

socket installd stream 600 system system

frameworks/native/cmds/installd/



ps | grep installd

install 404 1 2264 704 00b57be4 8b895b18 S /system/bin/installd

## 1 main()入口函数

installd.c

int main(const int argc, const char \*argv[]) {

char buf[BUFFER\_MAX];//BUFFER\_MAX:1024: input buffer for commands

struct sockaddr addr;

socklen\_t alen;

int lsocket, s, count;

int selinux\_enabled = (is\_selinux\_enabled() > 0);

ALOGI("installd firing up\n");

union selinux\_callback cb;

cb.func\_log = log\_callback;

selinux\_set\_callback(SELINUX\_CB\_LOG, cb);

if (initialize\_globals() < 0) {//初始化全局变量，创建目录

ALOGE("Could not initialize globals; exiting.\n");

exit(1);

}

if (initialize\_directories() < 0) {//初始化系统目录

ALOGE("Could not create directories; exiting.\n");

exit(1);

}

if (selinux\_enabled && selinux\_status\_open(true) < 0) {

ALOGE("Could not open selinux status; exiting.\n");

exit(1);

}

drop\_privileges();//变更installd进程的权限

lsocket = android\_get\_control\_socket(SOCKET\_PATH);//从环境变量ANDROID\_SOCKET\_INSTALLD中获取用于监听的本地socket，SOCKET\_PATH== installd

if (lsocket < 0) {

ALOGE("Failed to get socket from environment: %s\n", strerror(errno));

exit(1);

}

if (listen(lsocket, 5)) {//监听socket

ALOGE("Listen on socket failed: %s\n", strerror(errno));

exit(1);

}

fcntl(lsocket, F\_SETFD, FD\_CLOEXEC);

for (;;) {

alen = sizeof(addr);

s = accept(lsocket, &addr, &alen);//接收连接

if (s < 0) {

ALOGE("Accept failed: %s\n", strerror(errno));

continue;

}

fcntl(s, F\_SETFD, FD\_CLOEXEC);

ALOGI("new connection\n");

for (;;) {

unsigned short count;

if (readx(s, &count, sizeof(count))) {//读取命令的长度

ALOGE("failed to read size\n");

break;

}

if ((count < 1) || (count >= BUFFER\_MAX)) {//如果命令长度错误则停止处理

ALOGE("invalid size %d\n", count);

break;

}

if (readx(s, buf, count)) {

ALOGE("failed to read command\n");

break;

}

buf[count] = 0;

if (selinux\_enabled && selinux\_status\_updated() > 0) {

selinux\_android\_seapp\_context\_reload();

}

if (execute(s, buf)) break;//执行命令

}

ALOGI("closing connection\n");

close(s);//关闭连接

}

return 0;

}

installd就是监听一个本地的socket，这个socket通过在init.rc文件中指定服务属性的方式创建。如果有socket连接进来，则通过socket读取命令字符串，然后执行命令。

在main函数中，installd通过调用initialize\_globals()和initialize\_directories()来完成初始化的工作。initialize\_globals()函数将设置安装应用需要用到的目录名；initialize\_directories()则创建所有用户的安装目录。

在上面的方法中都使用到了dir\_rec\_t结构体，那么这个结构体是如何定义的呢？如下：

typedef struct {

char\* path;

size\_t len;

} dir\_rec\_t;

在这个结构体中有两个成员变量：path、len，分别表示文件路径和文件路径长度。

同样使用到了android\_system\_dirs，定义如下：

dir\_rec\_array\_t android\_system\_dirs;

typedef struct {

size\_t count;

dir\_rec\_t\* dirs;

} dir\_rec\_array\_t;

在这个结构体中有两个成员变量：count、dirs；其中dirs又是dir\_rec\_t的结构体类型。

## 2、初始化全局变量initialize\_globals

int initialize\_globals() {

// Get the android data directory.从环境变量中读取数据存储路径，android\_data\_dir=/data/

***// 数据目录/data/***

if (get\_path\_from\_env(&android\_data\_dir, "ANDROID\_DATA") < 0) {

return -1;

}

// Get the android app directory.得到应用程序安装目录：android\_app\_dir=/data/app/

***// app目录/data/app/***

if (copy\_and\_append(&android\_app\_dir, &android\_data\_dir, APP\_SUBDIR) < 0) {

return -1;

}

// Get the android protected app directory.得到应用程序私有目录：android\_app\_private\_dir=/data/app-private/

***// 受保护的app目录/data/priv-app/***

if (copy\_and\_append(&android\_app\_private\_dir, &android\_data\_dir, PRIVATE\_APP\_SUBDIR) < 0) {

return -1;

}

// Get the android app native library directory.android\_app\_lib\_dir=/data/app-lib/

***// app本地库目录/data/app-lib/***

if (copy\_and\_append(&android\_app\_lib\_dir, &android\_data\_dir, APP\_LIB\_SUBDIR) < 0) {

return -1;

}

// Get the sd-card ASEC mount point.从环境变量中取得sdcard ASEC的挂载点，android\_asec\_dir=/mnt/asec

***// sdcard挂载点/mnt/asec***

if (get\_path\_from\_env(&android\_asec\_dir, "ASEC\_MOUNTPOINT") < 0) {

return -1;

}

// Get the android media directory.android\_media\_dir=/data/media/

***// 多媒体目录/data/media***

if (copy\_and\_append(&android\_media\_dir, &android\_data\_dir, MEDIA\_SUBDIR) < 0) {

return -1;

}

***// 系统和厂商目录***

// Take note of the system and vendor directories.定义android\_system\_dirs变量，并分配存储空间

android\_system\_dirs.count = 4;

android\_system\_dirs.dirs = calloc(android\_system\_dirs.count, sizeof(dir\_rec\_t));

if (android\_system\_dirs.dirs == NULL) {

ALOGE("Couldn't allocate array for dirs; aborting\n");

return -1;

}

dir\_rec\_t android\_root\_dir;

if (get\_path\_from\_env(&android\_root\_dir, "ANDROID\_ROOT") < 0) {//android\_root\_dir=/system/

ALOGE("Missing ANDROID\_ROOT; aborting\n");

return -1;

}

android\_system\_dirs.dirs[0].path = build\_string2(android\_root\_dir.path, APP\_SUBDIR);// /system/app

android\_system\_dirs.dirs[0].len = strlen(android\_system\_dirs.dirs[0].path);

android\_system\_dirs.dirs[1].path = build\_string2(android\_root\_dir.path, PRIV\_APP\_SUBDIR);// /system/priv-app

android\_system\_dirs.dirs[1].len = strlen(android\_system\_dirs.dirs[1].path);

android\_system\_dirs.dirs[2].path = "/vendor/app/";

android\_system\_dirs.dirs[2].len = strlen(android\_system\_dirs.dirs[2].path);

android\_system\_dirs.dirs[3].path = "/oem/app/";

android\_system\_dirs.dirs[3].len = strlen(android\_system\_dirs.dirs[3].path);

return 0;

}

上面函数中使用到了ANDROID\_DATA、ASEC\_MOUNTPOINT、ANDROID\_ROOT，这些环境变量是在哪里定义的呢？在Android的启动脚本Init.environ.rc.in中配置了环境变量，如下：system/core/rootdir/init.environ.rc.in

# set up the global environment

on init

export PATH /sbin:/vendor/bin:/system/sbin:/system/bin:/system/xbin

export ANDROID\_BOOTLOGO 1

export ANDROID\_ROOT /system

export ANDROID\_ASSETS /system/app

export ANDROID\_DATA /data

export ANDROID\_STORAGE /storage

export ASEC\_MOUNTPOINT /mnt/asec

export LOOP\_MOUNTPOINT /mnt/obb

export BOOTCLASSPATH %BOOTCLASSPATH%

export SYSTEMSERVERCLASSPATH %SYSTEMSERVERCLASSPATH%

从代码中可以看出，initialize\_globals()函数主要是初始化一些全局变量，这些全局变量初始化为一些安装的路径。

## 3、初始化目录initialize\_directories

int initialize\_directories() {

int res = -1;

// Read current filesystem layout version to handle upgrade paths

char version\_path[PATH\_MAX];

snprintf(version\_path, PATH\_MAX, "%s.layout\_version", android\_data\_dir.path);

int oldVersion;// //读取当前文件系统版本

if (fs\_read\_atomic\_int(version\_path, &oldVersion) == -1) {

oldVersion = 0;

}

int version = oldVersion;

// **/data/user**

char \*user\_data\_dir = build\_string2(android\_data\_dir.path, SECONDARY\_USER\_PREFIX);

**// /data/data**

char \*legacy\_data\_dir = build\_string2(android\_data\_dir.path, PRIMARY\_USER\_PREFIX);

**// /data/user/0**

char \*primary\_data\_dir = build\_string3(android\_data\_dir.path, SECONDARY\_USER\_PREFIX, "0");

if (!user\_data\_dir || !legacy\_data\_dir || !primary\_data\_dir) {

goto fail;

}

// Make the /data/user directory if necessary

if (access(user\_data\_dir, R\_OK) < 0) {

if (mkdir(user\_data\_dir, 0711) < 0) {//如果目录不存在，创建/data/user/目录

goto fail;

}

if (chown(user\_data\_dir, AID\_SYSTEM, AID\_SYSTEM) < 0) {//设置属性组

goto fail;

}

if (chmod(user\_data\_dir, 0711) < 0) {//设置读写权限

goto fail;

}

}

// Make the /data/user/0 symlink to /data/data if necessary

//将/data/user/0链接到/data/data

if (access(primary\_data\_dir, R\_OK) < 0) {

if (symlink(legacy\_data\_dir, primary\_data\_dir)) {

goto fail;

}

}

//处理data/media 相关

if (version == 0) {

// Introducing multi-user, so migrate /data/media contents into /data/media/0

ALOGD("Upgrading /data/media for multi-user");

// Ensure /data/media。重新创建/data/media目录

if (fs\_prepare\_dir(android\_media\_dir.path, 0770, AID\_MEDIA\_RW, AID\_MEDIA\_RW) == -1) {

goto fail;

}

// /data/media.tmp

char media\_tmp\_dir[PATH\_MAX];

snprintf(media\_tmp\_dir, PATH\_MAX, "%smedia.tmp", android\_data\_dir.path);

// Only copy when upgrade not already in progress

if (access(media\_tmp\_dir, F\_OK) == -1) {

if (rename(android\_media\_dir.path, media\_tmp\_dir) == -1) {

ALOGE("Failed to move legacy media path: %s", strerror(errno));

goto fail;

}

}

// Create /data/media again

if (fs\_prepare\_dir(android\_media\_dir.path, 0770, AID\_MEDIA\_RW, AID\_MEDIA\_RW) == -1) {

goto fail;

}

if (selinux\_android\_restorecon(android\_media\_dir.path, 0)) {

goto fail;

}

// /data/media/0

char owner\_media\_dir[PATH\_MAX];

snprintf(owner\_media\_dir, PATH\_MAX, "%s0", android\_media\_dir.path);

// Move any owner data into place

if (access(media\_tmp\_dir, F\_OK) == 0) {

if (rename(media\_tmp\_dir, owner\_media\_dir) == -1) {

ALOGE("Failed to move owner media path: %s", strerror(errno));

goto fail;

}

}

// Ensure media directories for any existing users

DIR \*dir;

struct dirent \*dirent;

char user\_media\_dir[PATH\_MAX];

dir = opendir(user\_data\_dir);

if (dir != NULL) {

while ((dirent = readdir(dir))) {

if (dirent->d\_type == DT\_DIR) {

const char \*name = dirent->d\_name;

// skip "." and ".."

if (name[0] == '.') {

if (name[1] == 0) continue;

if ((name[1] == '.') && (name[2] == 0)) continue;

}

// /data/media/<user\_id>

snprintf(user\_media\_dir, PATH\_MAX, "%s%s", android\_media\_dir.path, name);

if (fs\_prepare\_dir(user\_media\_dir, 0770, AID\_MEDIA\_RW, AID\_MEDIA\_RW) == -1) {

goto fail;

}

}

}

closedir(dir);

}

version = 1;

}

// /data/media/obb

char media\_obb\_dir[PATH\_MAX];

snprintf(media\_obb\_dir, PATH\_MAX, "%sobb", android\_media\_dir.path);

if (version == 1) {

// Introducing /data/media/obb for sharing OBB across users; migrate

// any existing OBB files from owner.

ALOGD("Upgrading to shared /data/media/obb");

// /data/media/0/Android/obb

char owner\_obb\_path[PATH\_MAX];

snprintf(owner\_obb\_path, PATH\_MAX, "%s0/Android/obb", android\_media\_dir.path);

// Only move if target doesn't already exist

if (access(media\_obb\_dir, F\_OK) != 0 && access(owner\_obb\_path, F\_OK) == 0) {

if (rename(owner\_obb\_path, media\_obb\_dir) == -1) {

ALOGE("Failed to move OBB from owner: %s", strerror(errno));

goto fail;

}

}

version = 2;

}

if (ensure\_media\_user\_dirs(0) == -1) {

ALOGE("Failed to setup media for user 0");

goto fail;

}

if (fs\_prepare\_dir(media\_obb\_dir, 0770, AID\_MEDIA\_RW, AID\_MEDIA\_RW) == -1) {

goto fail;

}

if (ensure\_config\_user\_dirs(0) == -1) {

ALOGE("Failed to setup misc for user 0");

goto fail;

}

if (version == 2) {

ALOGD("Upgrading to /data/misc/user directories");

char misc\_dir[PATH\_MAX];

snprintf(misc\_dir, PATH\_MAX, "%smisc", android\_data\_dir.path);

char keychain\_added\_dir[PATH\_MAX];

snprintf(keychain\_added\_dir, PATH\_MAX, "%s/keychain/cacerts-added", misc\_dir);

char keychain\_removed\_dir[PATH\_MAX];

snprintf(keychain\_removed\_dir, PATH\_MAX, "%s/keychain/cacerts-removed", misc\_dir);

DIR \*dir;

struct dirent \*dirent;

dir = opendir(user\_data\_dir);

if (dir != NULL) {

while ((dirent = readdir(dir))) {

const char \*name = dirent->d\_name;

// skip "." and ".."

if (name[0] == '.') {

if (name[1] == 0) continue;

if ((name[1] == '.') && (name[2] == 0)) continue;

}

uint32\_t user\_id = atoi(name);

// /data/misc/user/<user\_id>

if (ensure\_config\_user\_dirs(user\_id) == -1) {

goto fail;

}

char misc\_added\_dir[PATH\_MAX];

snprintf(misc\_added\_dir, PATH\_MAX, "%s/user/%s/cacerts-added", misc\_dir, name);

char misc\_removed\_dir[PATH\_MAX];

snprintf(misc\_removed\_dir, PATH\_MAX, "%s/user/%s/cacerts-removed", misc\_dir, name);

uid\_t uid = multiuser\_get\_uid(user\_id, AID\_SYSTEM);

gid\_t gid = uid;

if (access(keychain\_added\_dir, F\_OK) == 0) {

if (copy\_dir\_files(keychain\_added\_dir, misc\_added\_dir, uid, gid) != 0) {

ALOGE("Some files failed to copy");

}

}

if (access(keychain\_removed\_dir, F\_OK) == 0) {

if (copy\_dir\_files(keychain\_removed\_dir, misc\_removed\_dir, uid, gid) != 0) {

ALOGE("Some files failed to copy");

}

}

}

closedir(dir);

if (access(keychain\_added\_dir, F\_OK) == 0) {

delete\_dir\_contents(keychain\_added\_dir, 1, 0);

}

if (access(keychain\_removed\_dir, F\_OK) == 0) {

delete\_dir\_contents(keychain\_removed\_dir, 1, 0);

}

}

version = 3;

}

// Persist layout version if changed

if (version != oldVersion) {

if (fs\_write\_atomic\_int(version\_path, version) == -1) {

ALOGE("Failed to save version to %s: %s", version\_path, strerror(errno));

goto fail;

}

}

// Success!

res = 0;

fail:

free(user\_data\_dir);

free(legacy\_data\_dir);

free(primary\_data\_dir);

return res;

}

## 4、变更installd进程的权限

我们看下drop\_privileges()，代码如下：

static void drop\_privileges() {

if (prctl(PR\_SET\_KEEPCAPS, 1) < 0) {//保留进程的权限

ALOGE("prctl(PR\_SET\_KEEPCAPS) failed: %s\n", strerror(errno));

exit(1);

}

if (setgid(AID\_INSTALL) < 0) {//设置进程的gid为“install”

ALOGE("setgid() can't drop privileges; exiting.\n");

exit(1);

}

if (setuid(AID\_INSTALL) < 0) {//设置进程的uid为"install"

ALOGE("setuid() can't drop privileges; exiting.\n");

exit(1);

}

struct \_\_user\_cap\_header\_struct capheader;

struct \_\_user\_cap\_data\_struct capdata[2];

memset(&capheader, 0, sizeof(capheader));

memset(&capdata, 0, sizeof(capdata));

capheader.version = \_LINUX\_CAPABILITY\_VERSION\_3;

capheader.pid = 0;

capdata[CAP\_TO\_INDEX(CAP\_DAC\_OVERRIDE)].permitted |= CAP\_TO\_MASK(CAP\_DAC\_OVERRIDE);

capdata[CAP\_TO\_INDEX(CAP\_CHOWN)].permitted |= CAP\_TO\_MASK(CAP\_CHOWN);

capdata[CAP\_TO\_INDEX(CAP\_SETUID)].permitted |= CAP\_TO\_MASK(CAP\_SETUID);

capdata[CAP\_TO\_INDEX(CAP\_SETGID)].permitted |= CAP\_TO\_MASK(CAP\_SETGID);

capdata[CAP\_TO\_INDEX(CAP\_FOWNER)].permitted |= CAP\_TO\_MASK(CAP\_FOWNER);

capdata[0].effective = capdata[0].permitted;

capdata[1].effective = capdata[1].permitted;

capdata[0].inheritable = 0;

capdata[1].inheritable = 0;

if (capset(&capheader, &capdata[0]) < 0) {//设置进程的权限

ALOGE("capset failed: %s\n", strerror(errno));

exit(1);

}

}

drop\_privileges()使用系统调用prctl来保留进程的能力。可参考Linux内核，虽然installd的uid和gid都变成了AID\_INSTALL，但是保留了5项能力，使得installd无需root用户身份也能够完成安装过程。

上面函数中用到了\_\_user\_cap\_data\_struct结构体，我们看下这个结构体是如何定义的，在android\_filesystem\_capability.h文件中定义如下：

typedef struct \_\_user\_cap\_data\_struct {

\_\_u32 effective;

\_\_u32 permitted;

\_\_u32 inheritable;

} \_\_user \*cap\_user\_data\_t;

typedef struct \_\_user\_cap\_header\_struct {

\_\_u32 version;

int pid;

} \_\_user \*cap\_user\_header\_t;

## 5、execute()

main()中通过调用execute()函数来执行命令。

static int execute(int s, char cmd[BUFFER\_MAX]) {

char reply[REPLY\_MAX];

char \*arg[TOKEN\_MAX+1];

unsigned i;

unsigned n = 0;

unsigned short count;

int ret = -1;

reply[0] = 0;

arg[0] = cmd;

while (\*cmd) {

if (isspace(\*cmd)) {

\*cmd++ = 0;

n++;

arg[n] = cmd;

if (n == TOKEN\_MAX) {

goto done;

}

}

if (\*cmd) {

cmd++; //计算参数个数

}

}

for (i = 0; i < sizeof(cmds) / sizeof(cmds[0]); i++) {

if (!strcmp(cmds[i].name,arg[0])) {

if (n != cmds[i].numargs) {

**//参数个数不匹配，直接返回**

ALOGE("%s requires %d arguments (%d given)\n",

cmds[i].name, cmds[i].numargs, n);

} else {

//执行相应的命令，func为函数指针

ret = cmds[i].func(arg + 1, reply);

}

goto done;

}

}

done:

if (reply[0]) {

n = snprintf(cmd, BUFFER\_MAX, "%d %s", ret, reply);

} else {

n = snprintf(cmd, BUFFER\_MAX, "%d", ret);

}

if (n > BUFFER\_MAX) n = BUFFER\_MAX;

count = n;

//将命令执行后的返回值写入socket套接字

if (writex(s, &count, sizeof(count))) return -1;

if (writex(s, cmd, count)) return -1;

return 0;

};

execute()函数执行过程就是查找下面并执行下面的cmds标准命令字符串对应的函数：

struct cmdinfo cmds[] = {

{ "ping", 0, do\_ping },

{ "install", 4, do\_install },

{ "dexopt", 6, do\_dexopt },

{ "markbootcomplete", 1, do\_mark\_boot\_complete },

{ "movedex", 3, do\_move\_dex },

{ "rmdex", 2, do\_rm\_dex },

{ "remove", 2, do\_remove },

{ "rename", 2, do\_rename },

{ "fixuid", 3, do\_fixuid },

{ "freecache", 1, do\_free\_cache },

{ "rmcache", 2, do\_rm\_cache },

{ "rmcodecache", 2, do\_rm\_code\_cache },

{ "getsize", 7, do\_get\_size },

{ "rmuserdata", 2, do\_rm\_user\_data },

{ "movefiles", 0, do\_movefiles },

{ "linklib", 3, do\_linklib },

{ "mkuserdata", 4, do\_mk\_user\_data },

{ "mkuserconfig", 1, do\_mk\_user\_config },

{ "rmuser", 1, do\_rm\_user },

{ "idmap", 3, do\_idmap },

{ "restorecondata", 3, do\_restorecon\_data },

{ "patchoat", 5, do\_patchoat },

};

dexopt：将应用优化为dex格式。

rmdex：删除apk文件。

remove：卸载应用。

rename：更改应用数据目录的名称。

fixuid：更改应用数据目录的uid。

freecache：清除/cache目录下的文件。

rmcache：删除/cache目录下的某个应用的目录。

getsize：计算一个应用占用的空间大小，包括apk大小、数据目录、cache目录等。

rmuserdata：删除一个user的所有安装的应用。

movefiles：执行/system/etc/updatecmds目录下的移动目录的脚本。

linklib：为动态库建立符合连接。

mkuserdata：为一个user创建目录。

mkuserconfig：为一个user创建配置文件。

rmuser：删除一个user的所有文件。

idmap：对两个apk进程执行idmap操作。

restorecondata：恢复目录的SELinux安全上下文。

patchoat：将应用优化为oat格式。

此命令表总共有25的命令，该表中第二列是指命令所需的参数个数，第三列是指命令所指向的函数。 不同的Android版本该表格都会有所不同

## 6、分析install命令

install命令执行的函数是install()，如下：

//commands.c

int install(const char \*pkgname, uid\_t uid, gid\_t gid, const char \*seinfo)

{

char pkgdir[PKG\_PATH\_MAX];// 256

char libsymlink[PKG\_PATH\_MAX];

char applibdir[PKG\_PATH\_MAX];

struct stat libStat;

Xandy:app\_install\_cnt

if ((uid < AID\_SYSTEM) || (gid < AID\_SYSTEM)) {//检查uid、gid

ALOGE("invalid uid/gid: %d %d\n", uid, gid);

return -1;

}

//得到应用的数据目录名/data/data/<包名>

if (create\_pkg\_path(pkgdir, pkgname, PKG\_DIR\_POSTFIX, 0)) {

ALOGE("cannot create package path\n");

return -1;

}

//得到应用的动态库目录名/data/data/<包名>/lib

if (create\_pkg\_path(libsymlink, pkgname, PKG\_LIB\_POSTFIX, 0)) {

ALOGE("cannot create package **lib symlink origin** path\n");

return -1;

}

//得到/app-lib目录下的符号链接的名称/data/app-lib/<包名>

if (create\_pkg\_path\_in\_dir(applibdir, &android\_app\_lib\_dir, pkgname, PKG\_DIR\_POSTFIX)) {

ALOGE("cannot create package **lib symlink dest** path\n");

return -1;

}

//创建应用的数据目录

if (mkdir(pkgdir, 0751) < 0) {

ALOGE("cannot create dir '%s': %s\n", pkgdir, strerror(errno));

return -1;

}

if (chmod(pkgdir, 0751) < 0) {//修改目录权限

ALOGE("cannot chmod dir '%s': %s\n", pkgdir, strerror(errno));

unlink(pkgdir);

return -1;

}

//检查符号链接是否已经存在

if (lstat(libsymlink, &libStat) < 0) {

if (errno != ENOENT) {

ALOGE("couldn't stat lib dir: %s\n", strerror(errno));

return -1;

}

} else {

if (S\_ISDIR(libStat.st\_mode)) {

if (delete\_dir\_contents(libsymlink, 1, NULL) < 0) {

ALOGE("couldn't delete lib directory during install for: %s", libsymlink);

return -1;

}

} else if (S\_ISLNK(libStat.st\_mode)) {

if (unlink(libsymlink) < 0) {

ALOGE("couldn't unlink lib directory during install for: %s", libsymlink);

return -1;

}

}

}

//为目录设置SELinux的安全上下文

if (selinux\_android\_setfilecon(pkgdir, pkgname, seinfo, uid) < 0) {

ALOGE("cannot setfilecon dir '%s': %s\n", pkgdir, strerror(errno));

unlink(libsymlink);

unlink(pkgdir);

return -errno;

}

//创建符号链接

if (symlink(applibdir, libsymlink) < 0) {

ALOGE("couldn't symlink directory '%s' -> '%s': %s\n", libsymlink, applibdir,

strerror(errno));

unlink(pkgdir);

return -1;

}

//修改目录的gid和uid

if (chown(pkgdir, uid, gid) < 0) {

ALOGE("cannot chown dir '%s': %s\n", pkgdir, strerror(errno));

unlink(libsymlink);

unlink(pkgdir);

return -1;

}

return 0;

}

上面多次用到create\_pkg\_path()函数，我们分析一下：

/\*\*

\* Create the package path name for a given package name with a postfix for

\* a certain userid. Returns 0 on success, and -1 on failure.

\*/

int create\_pkg\_path(char path[PKG\_PATH\_MAX],//PKG\_PATH\_MAX=256

const char \*pkgname,

const char \*postfix,//后缀

userid\_t userid)

{

size\_t userid\_len;

char\* userid\_prefix;

if (userid == 0) {

userid\_prefix = PRIMARY\_USER\_PREFIX;// 前缀 data/

userid\_len = 0;

} else {

userid\_prefix = SECONDARY\_USER\_PREFIX;// 前缀 user/

userid\_len = snprintf(NULL, 0, "%d", userid);// userid

}

const size\_t prefix\_len = android\_data\_dir.len + strlen(userid\_prefix)

+ userid\_len + 1 /\*slash\*/;// /data/data/ 计算前缀的长度

char prefix[prefix\_len + 1];

char \*dst = prefix;

size\_t dst\_size = sizeof(prefix);

if (append\_and\_increment(&dst, android\_data\_dir.path, &dst\_size) < 0 // /data/

|| append\_and\_increment(&dst, userid\_prefix, &dst\_size) < 0) {

ALOGE("Error building prefix for APK path");

return -1;

}

if (userid != 0) {

int ret = snprintf(dst, dst\_size, "%d/", userid);

if (ret < 0 || (size\_t) ret != userid\_len + 1) {

ALOGW("Error appending UID to APK path");

return -1;

}

}

dir\_rec\_t dir;

dir.path = prefix;

dir.len = prefix\_len;

return create\_pkg\_path\_in\_dir(path, &dir, pkgname, postfix);

}

上述函数中用到了append\_and\_increment函数，这个函数的作用就是调用strlcpy函数，将src源字符串复制到dst目标字符串。

int append\_and\_increment(char\*\* dst, const char\* src, size\_t\* dst\_size) {

ssize\_t ret = strlcpy(\*dst, src, \*dst\_size);//将src复制到dst中

if (ret < 0 || (size\_t) ret >= \*dst\_size) {

return -1;

}

\*dst += ret;

\*dst\_size -= ret;

return 0;

}

create\_pkg\_path函数最后调用了create\_pkg\_path\_in\_dir函数，如下：

int create\_pkg\_path\_in\_dir(char path[PKG\_PATH\_MAX],

const dir\_rec\_t\* dir,

const char\* pkgname,

const char\* postfix)

{

const size\_t postfix\_len = strlen(postfix);

const size\_t pkgname\_len = strlen(pkgname);

if (pkgname\_len > PKG\_NAME\_MAX) {

return -1;

}

if (is\_valid\_package\_name(pkgname) < 0) {

return -1;

}

if ((pkgname\_len + dir->len + postfix\_len) >= PKG\_PATH\_MAX) {

return -1;

}

char \*dst = path;

size\_t dst\_size = PKG\_PATH\_MAX;

if (append\_and\_increment(&dst, dir->path, &dst\_size) < 0

|| append\_and\_increment(&dst, pkgname, &dst\_size) < 0

|| append\_and\_increment(&dst, postfix, &dst\_size) < 0) {

ALOGE("Error building APK path");

return -1;

}

return 0;

}

install()命令创建了应用的数据目录，并把目录的uid和gid改成参数传递的值。同时在/data/app-lib目录下创建了一个符号链接，指向应用的本地动态库的安装路径。

751



<http://lib.csdn.net/article/android/63896>

# Installer

当守护进程installd启动完成后，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/

# 案例

### 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. }

## 错误码分析

## 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]

## 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。。。。

## 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

# 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

# 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

# 扩展：

如何写一个系统服务

Stub，aldl

PPTcopy

# 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>