# 基本概念

## 名词解释

**PMS**: **即**PackageManagerService,在fremwork真正执行apk安装的类。

AMS: **ActivityManagerService**

**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。

四大组件：**Activity、Service、Broadcast Receiver（Broadcast）和Content Provider(Provider)**

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

[**http://androidxref.com/**](http://androidxref.com/)

[**http://blog.csdn.net/yxhuang2008/article/details/51783394**](http://blog.csdn.net/yxhuang2008/article/details/51783394)

**http://blog.csdn.net/xueshanhaizi/article/details/51077303**

## 相关目录

/system/framework/framework-res.apk 保存的是应用是资源类型；

1. /system/priv-app 目录是保存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
2. /vendor/app 目录是保存设备厂商提供的应用程序。

/system/app ：系统自带的应用程序，获得adb root权限才能删除

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

/data/app ：用户程序安装的目录。安装时把apk文件复制到此目录

/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，以及软件包的信息.类似于Windows**的注册表**，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。

记录系统中所有已经安装的应用信息，包括基本信息，签名和权限。

## 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。

# 安装方式

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. 第三方应用安装：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. ADB工具安装：即通过Android的SDK开发tools里面的adb.exe程序安装，没有安装界面；
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); |

### 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

*//从此处可以看到，scheme只有两个值：file或package，为空呢* **if** (scheme != **null** && !**"file"**.equals(scheme) && !**"package"**.equals(scheme)) {  
 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())) {  
 *//scheme为package时，调用更新方法更新程序* **try** {  
 pm.installExistingPackage(**mAppInfo**.**packageName**);  
 observer.packageInstalled(**mAppInfo**.**packageName**,  
 PackageManager.INSTALL\_SUCCEEDED);  
 } **catch** (PackageManager.NameNotFoundException e) {  
 observer.packageInstalled(**mAppInfo**.**packageName**,  
 PackageManager.INSTALL\_FAILED\_INVALID\_APK);  
 }  
 } **else** {  
 *//scheme为file时,调用安装方法* pm.installPackageWithVerificationAndEncryption(**mPackageURI**, observer, installFlags,  
 installerPackageName, verificationParams, **null**);  
 }

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就好

## pm安装

pm(AS安装)和adb install（最终也是调用pm）

$ 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

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\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  } |

安装时候会调用 runInstall（）方法

private void runInstall() {

int installFlags = 0;

String installerPackageName = null;

String opt;

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("-i")) {

installerPackageName = nextOptionData();

if (installerPackageName == null) {

System.err.println("Error: no value specified for -i");

showUsage();

return;

}

} else if (opt.equals("-t")) {

installFlags |= PackageManager.INSTALL\_ALLOW\_TEST;

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

// Override if -s option is specified.

installFlags |= PackageManager.INSTALL\_EXTERNAL;

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

// Override if -s option is specified.

installFlags |= PackageManager.INSTALL\_INTERNAL;

} else {

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

showUsage();

return;

}

}

String apkFilePath = nextArg();

System.err.println("\tpkg: " + apkFilePath);

if (apkFilePath == null) {

System.err.println("Error: no package specified");

showUsage();

return;

}

PackageInstallObserver obs = new PackageInstallObserver();

try {

**mPm.installPackage(Uri.fromFile(new File(apkFilePath)), obs, installFlags,**

**installerPackageName);**

synchronized (obs) {

while (!obs.finished) {

try {

obs.wait();

} catch (InterruptedException e) {

}

}

if (obs.result == PackageManager.INSTALL\_SUCCEEDED) {

System.out.println("Success");

} else {

System.err.println("Failure ["

+ installFailureToString(obs.result)

+ "]");

}

}

} catch (RemoteException e) {

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

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

}

}



## 开机安装

应用程序管理服务PMS是系统启动的时候由SystemServer组件启动的，启后它就会执行应用程序安装的过程.

## 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

Android系统在启动的过程中，启动一个应用程序管理服务PMS负责扫描系统中特定的目录，找到里面的应用程序文件，即以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。

应用程序管理服务PMS是系统启动的时候由SystemServer组件启动的，启后它就会执行应用程序安装的过程.

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。

## 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。

## 开机安装方式

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。

### PMS构造方法 ①

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

1. **public** PMS(Context context, Installer installer, **boolean** factoryTest, **boolean** onlyCore) {
2. EventLog.writeEvent(EventLogTags.BOOT\_PROGRESS\_PMS\_START, SystemClock.uptimeMillis());
3. mContext = context;
4. mFactoryTest = factoryTest;
5. mOnlyCore = onlyCore;
6. mLazyDexOpt = "eng".equals(SystemProperties.get("ro.build.type"));
7. mMetrics = **new** DisplayMetrics();
8. mSettings = **new** Settings(mPackages);
9. mSettings.addSharedUserLPw("android.uid.system", Process.SYSTEM\_UID,
10. ApplicationInfo.FLAG\_SYSTEM, ApplicationInfo.PRIVATE\_FLAG\_PRIVILEGED);
11. mSettings.addSharedUserLPw("android.uid.phone", RADIO\_UID,
12. ApplicationInfo.FLAG\_SYSTEM, ApplicationInfo.PRIVATE\_FLAG\_PRIVILEGED);
13. mSettings.addSharedUserLPw("android.uid.log", LOG\_UID,
14. ApplicationInfo.FLAG\_SYSTEM, ApplicationInfo.PRIVATE\_FLAG\_PRIVILEGED);
15. mSettings.addSharedUserLPw("android.uid.nfc", NFC\_UID,
16. ApplicationInfo.FLAG\_SYSTEM, ApplicationInfo.PRIVATE\_FLAG\_PRIVILEGED);
17. mSettings.addSharedUserLPw("android.uid.bluetooth", BLUETOOTH\_UID,
18. ApplicationInfo.FLAG\_SYSTEM, ApplicationInfo.PRIVATE\_FLAG\_PRIVILEGED);
19. mSettings.addSharedUserLPw("android.uid.shell", SHELL\_UID,
20. ApplicationInfo.FLAG\_SYSTEM, ApplicationInfo.PRIVATE\_FLAG\_PRIVILEGED);

构造方法内部首先获得一个系统级上下文mContext，并在实例化mSettings后，添加system, radio, log, nfc, bluetooth, shell  6种SharedUserSettings到mSettings。

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。

Settings类结构如图（2）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");
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. mBackupSettingsFilename = **new** File(mSystemDir, "packages-backup.xml");
14. mPackageListFilename = **new** File(mSystemDir, "packages.list");
15. FileUtils.setPermissions(mPackageListFilename, 0640, SYSTEM\_UID, PACKAGE\_INFO\_GID);
17. // Deprecated: Needed for migration
18. mStoppedPackagesFilename = **new** File(mSystemDir, "packages-stopped.xml");
19. mBackupStoppedPackagesFilename = **new** File(mSystemDir, "packages-stopped-backup.xml");
20. }

//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

#### **创建并添加SharedUserSetting**

 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。



### 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>** |

### **实例化SystemConfig并获得gids, systempermissions, features**

## 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:



# 拷贝安装Pms

/\* 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

## 网络下载应用安装

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

## 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  
//TODO 用户的概念？一个进程就是一个用户？

@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

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

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。

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：

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);

}

}

}

processPendingInstall()来进行应用的解析和装载

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。

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：

### doHandleMessage- POST\_INSTALL

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。

### 总结一下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

## 其他方法

### PackageParser

parseActivity

## 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()

# 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="fileprovider"

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

## 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

# 搜索

关键字：android 应用程序的安装和显示过程

360: android 应用程序的安装和显示过程