# PMS 自顶向下分析

# 1. 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。

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

## 1.1. 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

## 1.2. 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[k1]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：

```
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

## 1.3. PMS_START

```
mLazyDexOpt = "eng".equals(SystemProperties.get("ro.build.type"));
1.          mMetrics = new DisplayMetrics();//DisplayMetrics 是一个描述界面显示，尺寸，分辨率，密度的
2.          mSettings = new Settings(context);
3.          mSettings.addSharedUserLPw("android.uid.system", Process.SYSTEM_UID,
4.                  ApplicationInfo.FLAG_SYSTEM, ApplicationInfo.PRIVATE_FLAG_PRIVILEGED);
5.          mSettings.addSharedUserLPw("android.uid.phone", RADIO_UID,
6.                  ApplicationInfo.FLAG_SYSTEM, ApplicationInfo.PRIVATE_FLAG_PRIVILEGED);
7.          mSettings.addSharedUserLPw("android.uid.log", LOG_UID,
8.                  ApplicationInfo.FLAG_SYSTEM, ApplicationInfo.PRIVATE_FLAG_PRIVILEGED);
    mSettings.addSharedUserLPw("android.uid.nfc", NFC_UID,
9.                  ApplicationInfo.FLAG_SYSTEM, ApplicationInfo.PRIVATE_FLAG_PRIVILEGED);
10.         mSettings.addSharedUserLPw("android.uid.bluetooth", BLUETOOTH_UID,
11.                 ApplicationInfo.FLAG_SYSTEM, ApplicationInfo.PRIVATE_FLAG_PRIVILEGED);
12.         mSettings.addSharedUserLPw("android.uid.shell", SHELL_UID,
                ApplicationInfo.FLAG_SYSTEM, ApplicationInfo.PRIVATE_FLAG_PRIVILEGED);
13.  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
14.
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();
}
```

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

## 1.3.1. 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 |
| --- |
| -mSettingsFilename: File<br>-mBackupSettingsFilename: File<br>-mPackageListFilename: File<br>-mStoppedPackagesFilename: File<br>-mBackupStoppedPackagesFilename: File<br>-mPackages: ArrayMap<String, PackageSetting><br>-mDisabledSysPackages: ArrayMap<String, PackageSetting><br>-mVerifierDeviceIdentity: VerifierDeviceIdentity<br>-mPreferredActivities: SparseArray<PreferredIntentResolver><br>-mPersistentPreferredActivities: SparseArray<PersistentPreferredIntentResolver><br>-mCrossProfileIntentResolvers: SparseArray<CrossProfileIntentResolver><br>-mSharedUsers: ArrayMap<String, SharedUserSetting><br>-mUserIds: ArrayList<Object><br>-mOtherUserIds: SparseArray<Object><br>-mPastSignatures: ArrayList<Signature><br>-mPermissions: ArrayMap<String, BasePermission><br>-mPermissionTrees: ArrayMap<String, BasePermission><br>-mPackagesToBeCleaned: ArrayList<PackageCleanItem><br>-mRenamedPackages: ArrayMap<String, String><br>-mPendingPackages: ArrayList<PendingPackage><br>-mRestrictPackages: ArrayList<String><br>-mSystemDir: File<br>-mKeySetManagerService: KeySetManagerService |
| *+addPackageLPw():PackageSetting*<br>*+initializeRestrictionConf():void*<br>*+readRestrictionConfgLPw():void*<br>*+initRestrictions():void*<br>*+getPackageLPw():PackageSetting*<br>*+setInstallerPackageName():void*<br>*+getSharedUserLPw():SharedUserSetting*<br>*+getAllSharedUsersLPw():Collection<SharedUserSetting>*<br>*+disableSystemPackageLPw():boolean*<br>*+enableSystemPackageLPw():PackageSetting*<br>*+removeDisabledSystemPackageLPw():void*<br>*+addSharedUserLPw():SharedUserSetting*<br>*+isAdbInstallDisallowed():boolean*<br>*+insertPackageSettingLPw():void*<br>*+addPackageSettingLPw():void*<br>*+updateSharedUserPermsLPw():void*<br>*+removePackageLPw():int*<br>*+replacePackageLPw():void*<br>*+addUserIdLPw():boolean*<br>*+getUserIdLPr():Object*<br>*+removeUserIdLPw():void*<br>*+replaceUserIdLPw():void*<br>*+editPreferredActivitiesLPw():PreferredIntentResolver*<br>*+editCrossProfileIntentResolverLPw():CrossProfileIntentResolver*<br>*+getRestrictionsConfigFile():File*<br>*+getUserPackagesStateFile():File*<br>*+getUserPackagesStateBackupFile():File*<br>*+writeAllUsersPackageRestrictionsLPr():void* |

## 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.     }
4.
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_G
    ID);
17.
18.          // Deprecated: Needed for migration
19.          mStoppedPackagesFilename = new File(mSystemDir, "packages-stopped.xml");
20.          mBackupStoppedPackagesFilename = new File(mSystemDir, "packages-stopped-backup.
    xml");
21.     }
22.
```

//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[KG2]

## 配置文件 package.xml

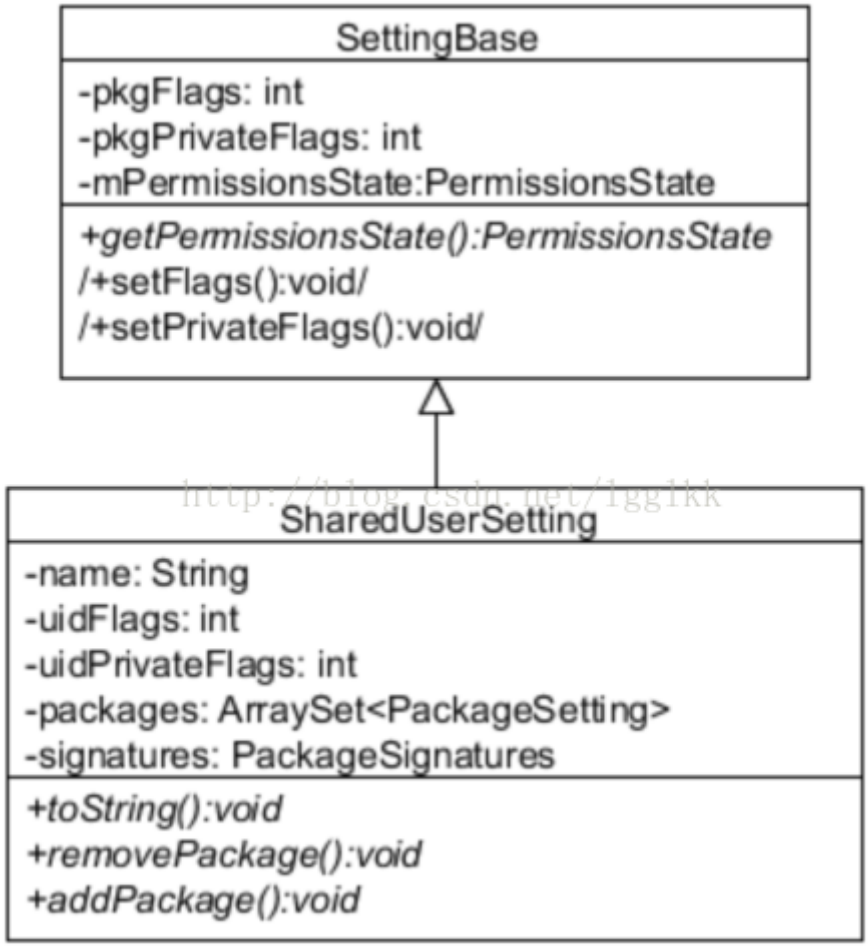
/data/system/packages.xml 通过它可以看到系统安装的所有软件包，以及软件包的信息

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**[k3]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 pkgPrivat
eFlags) {
       //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"
```
[KG4]

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` 。

## 1.3.2. 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[KG5]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[KG6]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~~[KG7]
```

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("con
fig")) {
            throw new XmlPullParserException("Unexpected start tag in " + permFile
                    + ": found " + parser.getName() + ", expected 'permissions' or 'conf
ig'");
        }
        //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) && !onlyFeature
s) {
                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>
```

### 1.3.3. 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);
```

### 1.3.4. 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-pri
vate"
        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_BUILTI
N);
```

```
                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。

### 1.3.5. 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.unflattenF
romString(
                    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<Stri
ng>();

        /**
         * 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("SYSTEM
SERVERCLASSPATH");

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

        if (bootClassPath != null) {
            String[] bootClassPathElements = splitString(bootClassP
ath, ':');
            for (String element : bootClassPathElements) {
                alreadyDexOpted.add(element);
            }
        } else {
            Slog.w(TAG, "No BOOTCLASSPATH found!");
        }
```

```
        if (systemServerClassPath != null) {
            String[] systemServerClassPathElements = splitString(sy
stemServerClassPath, ':');
            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[KG8]
```

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[KG9]，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[KG10]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" />
```

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()-->addSharedUserLP
w())

updated-package -> mDisabledSysPackages[KG11](readDisabledSysPackag
eLPw()) //这个标签是在 OTA 中添加的？删除也会有这个标记？

renamed-package -> mRenamedPackages
```

## 1.3.6. 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：

```
// 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。

## 1.4. SCAN_START

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

- /data/app
- /data/app-private

### 1.4.1. 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[KG12]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 add
s some complexity.
            for (String dexCodeInstructionSet : dexCodeInstructionS
ets) {
                for (SharedLibraryEntry libEntry : mSharedLibraries.
values()) {
                    final String lib = libEntry.path;
                    if (lib == null) {
                        continue;
                    }

                    try {
                        int dexoptNeeded = DexFile.getDexOptNeeded(li
b, 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

### 1.4.2. `alreadyDexOpted.add` frameworkDir

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

        // Gross hack for now: we know this file doesn't contain an
y
```

```
        // code, so don't dexopt it to avoid the resulting log spe
w.
        alreadyDexOpted.add(frameworkDir.getPath() + "/framework-r
es.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, whic
h
         * 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 prefer
red ABI. We should
            // first double check that the dex files for these comma
nds are not referenced
            // by other system apps.
            for (String dexCodeInstructionSet : dexCodeInstructionS
ets) {
                for (int i=0; i<frameworkFiles.length; i++) {
                    File libPath = new File(frameworkDir, frameworkF
iles[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 de
xopt.
                    if (!path.endsWith(".apk") && !path.endsWith(".j
ar")) {
                        continue;
                    }
                    try {
```

```
                    int dexoptNeeded = DexFile.getDexOptNeeded(pa
th, null, dexCodeInstructionSet, false);
                    if (dexoptNeeded != DexFile.NO_DEXOPT_NEEDED)
 {
                        mInstaller.dexopt(path, Process.SYSTEM_UI
D, 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[KG13] 。

### 1.4.3. 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 | SCA
N_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.getRoot
Directory(), "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.getRootDire
ctory(), "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 origi
nal one
        }
        scanDirLI(vendorAppDir, PackageParser.PARSE_IS_SYSTEM
                | PackageParser.PARSE_IS_SYSTEM_DIR, scanFlags, 0);

        // Collect all OEM packages.
        final File oemAppDir = new File(Environment.getOemDirector
y(), "app");
        scanDirLI(oemAppDir, PackageParser.PARSE_IS_SYSTEM
                | PackageParser.PARSE_IS_SYSTEM_DIR, scanFlags, 0);

        if (DEBUG_UPGRADE) Log.v(TAG, "Running installd update comm
ands");
```

```
        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)。

### 1.4.4. 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=" + s
canFlags
                   + " flags=0x" + Integer.toHexString(parseFlag
s));
       }

       //遍历该目录下的 APK
       for (File file : files) {
           final boolean isPackage = (isApkFile(file) || file.isDi
rectory())
                   && !PackageInstallerService.isStageName(file.get
Name());
           if (!isPackage) {
               // Ignore entries which are not packages
               continue;
           }
           try {
               scanPackageLI(file, parseFlags | PackageParser.PARS
E_MUST_BE_APK,
                       scanFlags, currentTime, null);
           } catch (PackageManagerException e) {
               Slog.w(TAG, "Failed to parse " + file + ": " + e.getM
essage());

               // Delete invalid userdata apps
```

```
                // 删除无效的用户 APK
                if ((parseFlags & PackageParser.PARSE_IS_SYSTEM) ==
0 &&
                        e.error == PackageManager.INSTALL_FAILED_INVA
LID_APK) {
                    logCriticalInfo(Log.WARN, "Deleting invalid pack
age 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

### 1.4.5. 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,...)。

### 1.4.6. 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)[KG14]，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。

### 1.4.7. 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,
                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 。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。

## 1.4.8. 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<ActivityIntentIn
fo>
                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);
            }
```

### 1.4.9. 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,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

### 1.4.10. 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。

### 1.4.11. 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.val
ues().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.g
et(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-install
ed
                     * application can be scanned.
                     */
                    //package.xml 的 package 标签和 updated-package[KG15]

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.nam
e)) {
                        logCriticalInfo(Log.WARN, "Expecting better u
pdated system app for "
```

```
                                + ps.name + "; removing system app.  La
st 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[KG16]

                //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.getD
isabledSystemPkgLPr(ps.name);
                    if (disabledPs.codePath == null || !disabledPs.c
odePath.exists()) {
                        possiblyDeletedUpdatedSystemApps.add(ps.nam
e);
                    }
```

```
                }
            }
        }
```

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[KG17]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.getLi
stOfIncompleteInstallPackagesLPr();//获取mSettings.mPackages中ins
tallStatus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 packag
es
        mSettings.pruneSharedUsersLPw();//移除没有被关联的mSharedUser
s
```

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,

## 1.4.12. 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

## 1.5. 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 permission
s.
            */
           //在引进了用户 app 之后 mPackages 内容增加了，再看看是否有这些 ap
p
           for (String deletedAppName : possiblyDeletedUpdatedSyst
emApps) {
               PackageParser.Package deletedPkg = mPackages.get(de
letedAppName);
               mSettings.removeDisabledSystemPackageLPw(deletedApp
Name);

               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 pri
vileges for "
                           + deletedAppName;

                   deletedPkg.applicationInfo.flags &= ~Application
Info.FLAG_SYSTEM;

                   PackageSetting deletedPs = mSettings.mPackages.g
et(deletedAppName);
                   deletedPs.pkgFlags &= ~ApplicationInfo.FLAG_SYST
EM;
               }
               logCriticalInfo(Log.WARN, msg);
           }

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

```
            * the userdata partition actually showed up. If they ne
ver
            * 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 " + p
ackageName
                           + " but never showed up; reverting to syst
em");

                   final int reparseFlags;

                   //不同目录flags不一样

                   if (FileUtils.contains(privilegedAppDir, scanFil
e)) {
                       reparseFlags = PackageParser.PARSE_IS_SYSTEM
                               | PackageParser.PARSE_IS_SYSTEM_DIR
                               | PackageParser.PARSE_IS_PRIVILEGED;
                   } else if (FileUtils.contains(systemAppDir, scan
File)) {
                       reparseFlags = PackageParser.PARSE_IS_SYSTEM
                               | PackageParser.PARSE_IS_SYSTEM_DIR;
                   } else if (FileUtils.contains(vendorAppDir, scan
File)) {
                       reparseFlags = PackageParser.PARSE_IS_SYSTEM
                               | PackageParser.PARSE_IS_SYSTEM_DIR;
                   } else if (FileUtils.contains(oemAppDir, scanFil
e)) {
                       reparseFlags = PackageParser.PARSE_IS_SYSTEM
                               | PackageParser.PARSE_IS_SYSTEM_DIR;
                   } else {
                       Slog.e(TAG, "Ignoring unexpected fallback pat
h " + scanFile);
                       continue;
```

```
                }

                //加入 mStting.mPackages
                mSettings.enableSystemPackageLPw(packageName);

                try {
                    scanPackageLI(scanFile, reparseFlags, scanFla
gs, 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 l

ibs,放在 pkg.usesLibraryFiles


        for (SharedUserSetting setting : mSettings.getAllSharedUse
rsLPw()) {
            // NOTE: We ignore potential failures here during a syst
em scan (like
            // the rest of the commands above) because there's preci
ous little we
            // can do about it. A settings error is reported, thoug
h.
            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
```

## 1.6. 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, UserHandl
e.USER_OWNER);
            applyFactoryDefaultBrowserLPw(UserHandle.USER_OWNER);
            primeDomainVerificationsLPw(UserHandle.USER_OWNER);
        }

        // If this is first boot after an OTA, and a normal boot, th
en
        // we need to clear code cache directories.
        if (mIsUpgrade && !onlyCore) {
            Slog.i(TAG, "Build fingerprint changed; clearing code c
aches");
            for (int i = 0; i < mSettings.mPackages.size(); i++) {
                final PackageSetting ps = mSettings.mPackages.valueA
t(i);
                if (Objects.equals(StorageManager.UUID_PRIVATE_INTE
RNAL, 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(); //str
ing
        mRequiredInstallerPackage = getRequiredInstallerLPr(); //s
tring

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

        mIntentFilterVerifierComponent = getIntentFilterVerifierCo
mponentNameLPr();
        mIntentFilterVerifier = new IntentVerifierProxy(mContext,
                mIntentFilterVerifierComponent);

    } // synchronized (mPackages)
    } // synchronized (mInstallLock)
```

## 1.7. 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 th
ey
    // are all flushed.  Not really needed, but keeps things nice a
nd
```

```
    // tidy.
    Runtime.getRuntime().gc();

    // Expose private service for system components to use.
    LocalServices.addService(PackageManagerInternal.class, new Pa
ckageManagerInternalImpl());
}
```

到此，PMS 的构造函数就阅读完毕了。

### 1.8. 小结

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

# 1. 源码分析-手动安装

## 1.1. 文件拷贝阶段

### 1.1.1. 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());

}
```

## 1.1.2. 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，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;[KG18]

        } 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);

    }
```

### 1.1.3. 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;

        }
```

## 1.1.4. InstallParams.handleStartCopy

# InstallParams 实现了抽象类 HandlerParams

handleStartCopy()执行的工作如下：[KG19]

- 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[KG20]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_PACKA
GE,

                            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_PENDIN
G_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：

## 1.1.5. 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。

### 1.1.6. 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。

## 1.2. 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。

### 1.2.1. 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：

## 1.2.2. installPackageLI()[KG21]

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)) {[KG22]
        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);[KG23]
            } 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[KG24].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) {[KG25]
    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) {[KG26]
        // 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[KG27](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[KG28]
        int result = mPackageDexOptimizer[KG29]
                .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)[KG30]) {
        res.setError(INSTALL_FAILED_INSUFFICIENT_STORAGE, "Failed rename");
        return;
    }
    startIntentFilterVerifications[KG31](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[KG32](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
```

## 1.2.3. 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();[KG33][KG34]
    // 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。

## 1.2.4. 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

## 1.2.5. 其他相关方法分析

### getNextCodePath

类似 com.android.browser-1

1.2.5.1. replacePackageLI()

uid 和 gid 分配方法

# 1.3. 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/
Android PackageManager 相关源码分析之安装应用
PMS(Android5.1)深入分析（四）安装应用
Android 应用程序安装过程解析(源码角度)
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()
```

# 2. 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。

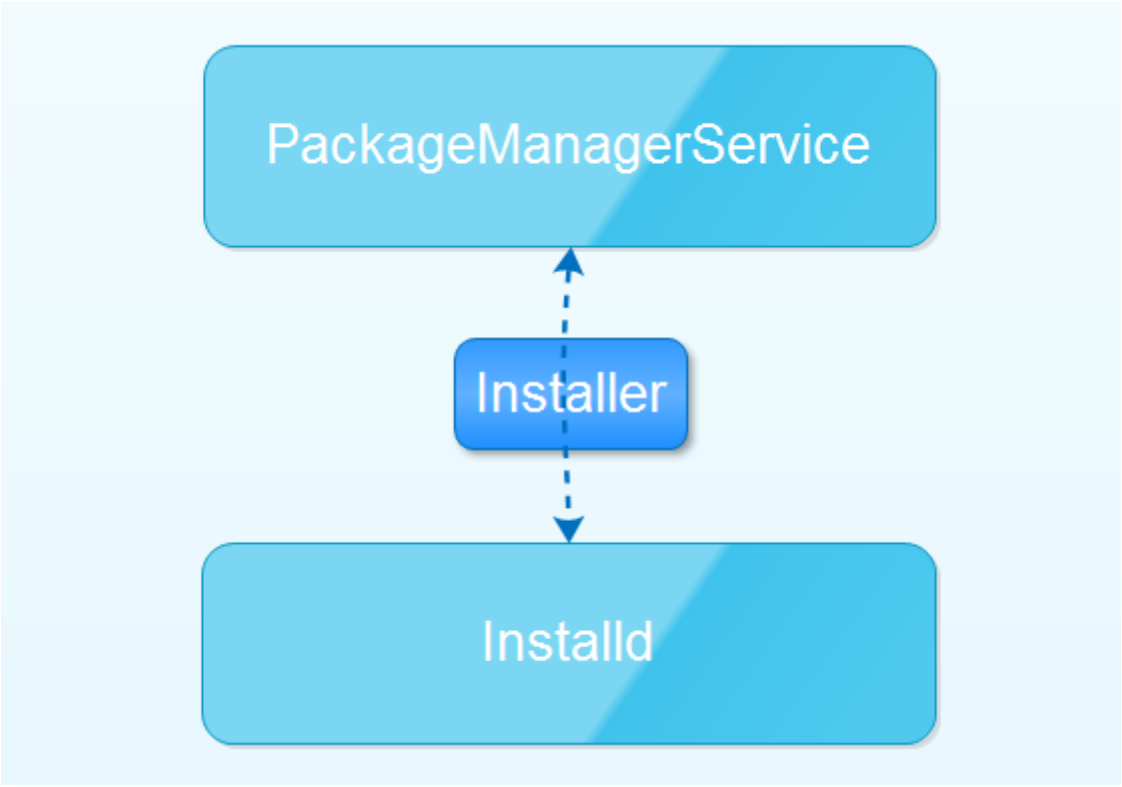
在 installd 的入口函数中，首先是做一些初始化的工作，然后放弃自己的 root 用户身份，改变了自己的用户类型和组用户类型，但同时它保留了设置目录权限，以及修改目录属主的权限：

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。

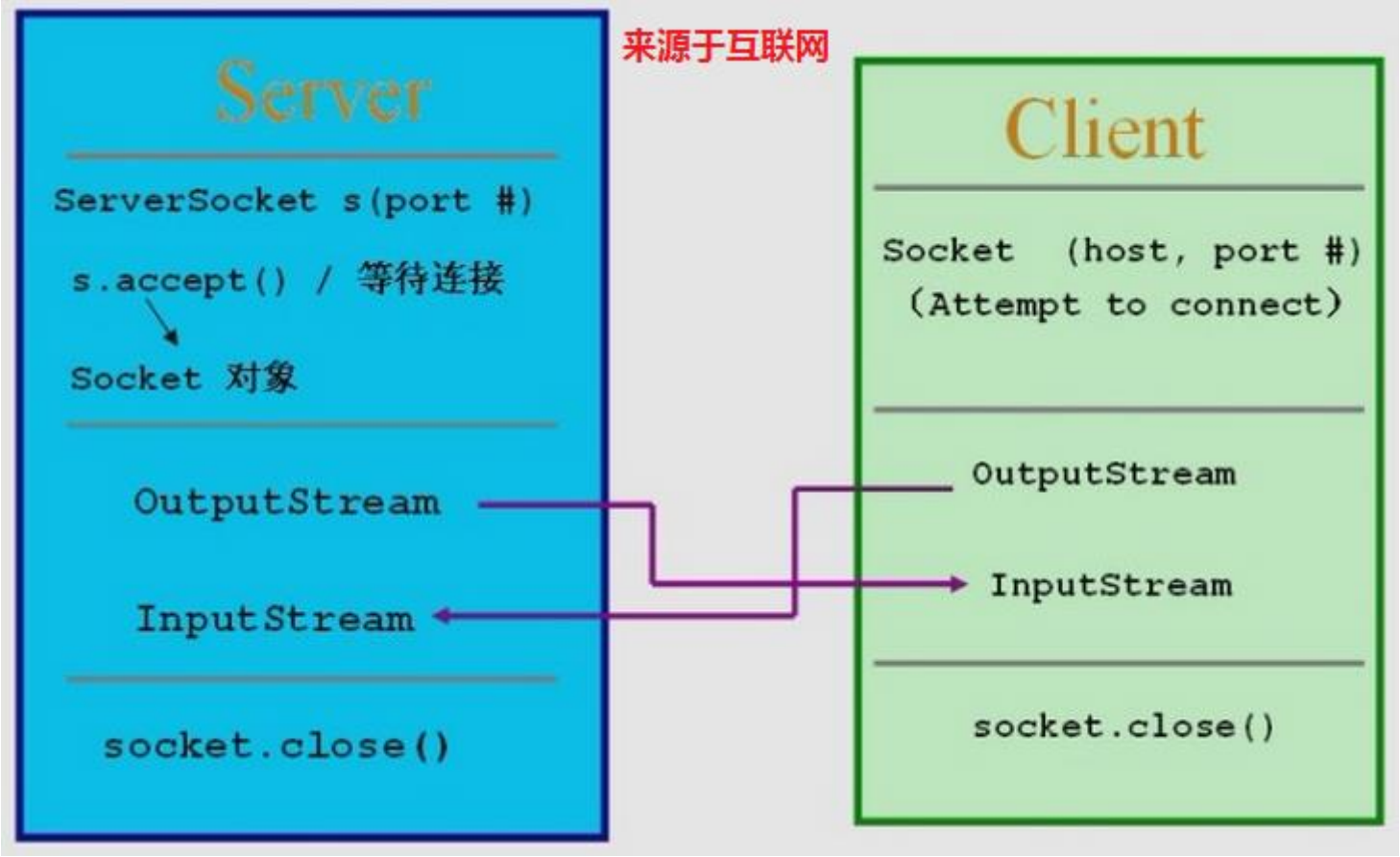
- do_install 函数为应用创建了以下目录，在 install 过程中，同时为创建的目录设置 uid 和 gid。
  - 数据目录:/data/data/packageName/
  - lib 目录：/data/app-lib/packageName
  - lib 符号链接:/data/data/packageName/lib → /data/app-lib/packageName
- do_dexopt 中调用 dexopt 函数处理 apk 文件。主要流程为：
  - 首先判断 apk 所在目录下面是否存在同名的 odex 文件，如果存在就直接返回；
  - 以 apk 文件的路径为名创建 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@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。




```
00135: struct cmdinfo cmds[] = {
00136:         { "ping",                0, do_ping },
00137:         { "install",             3, do_install },
00138:         { "dexopt",              3, do_dexopt },
00139:         { "movedex",             2, do_move_dex },
00140:         { "rmdex",               1, do_rm_dex },
00141:         { "remove",              2, do_remove },
00142:         { "rename",              2, do_rename },
00143:         { "fixuid",              3, do_fixuid },
00144:         { "freecache",           1, do_free_cache },
00145:         { "rmcache",             2, do_rm_cache },
00146:         { "getsize",             5, do_get_size },
00147:         { "rmuserdata",          2, do_rm_user_data },
00148:         { "movefiles",           0, do_movefiles },
00149:         { "linklib",             3, do_linklib },
00150:         { "mkuserdata",          3, do_mk_user_data },
00151:         { "rmuser",              1, do_rm_user },
00152:         { "cloneuserdata",       3, do_clone_user_data },
00153: };
```

installer 对应 installd

LocalSocket 通信（右）：




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[KG35]中定义

```
service installd /system/bin/installd
    class main
    socket installd stream 600 system system
```

frameworks/native/cmds/installd/

```
shell@zs600b:/ $ ps in
USER      PID   PPID  VSIZE  RSS     WCHAN    PC         NAME
root      1     0     752    396     ffffffff 00000000 S /init
install   169   1     1168   380     ffffffff 00000000 S /system/bin/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() [KG36]> 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()[KG37] < 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)[KG38] < 0) {
        ALOGE("Could not open selinux status; exiting.\n");
        exit(1);
    }

    drop_privileges();//变更 installd 进程的权限[KG39]

    lsocket = android_get_control_socket(SOCKET_PATH);//从环境变量 ANDROID_SOCKET_INSTA
LLD 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);[KG40]

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")[KG41] < 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_SUBDI
R) < 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_ase
c_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_SUBDI
R);// /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_S
UBDIR);// /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[KG42]

```
# 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[KG43]

```
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_PREFI
X, "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_R
W) == -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_d
ir);

        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_di
r, 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, gi
d) != 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_OVERRID
E);
    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[KG44]，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()，[KG45]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[KG46]
    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>[KG47]
    if (create_pkg_path_in_dir(applibdir, &android_app_lib_dir, pkgname, PKG_DIR_POSTF
IX)) {
        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)[KG48] {//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", libsymlin
k);
                return -1;
            }
        } else if (S_ISLNK(libStat.st_mode)) {
            if (unlink(libsymlink) < 0) {
                ALOGE("couldn't unlink lib directory during install for: %s", libsymlin
k);
                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()[KG49]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

```
root@zs600b:/data/data # ls -al | grep ten
drwxr-x--x u0_a44   u0_a44            2017-12-10 13:51 com.tencent.mobileqq
```

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

# 3. 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();[KG50]
}

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();[KG51]
    } 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[KG52]
    } 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/