Android PackageManagerService 分析（1.0）

概要

本篇主要分析了系统启动阶段包管理服务的启动流程，其中的几个接口在 apk 安装时也会被调用。包管理服务启动时主要做的工作大致有如下几方面：

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

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

3.解析/system/etc/permission 下 xml 文件(framework/base/data/etc/)，包括 platform.xml 和系统支持的各种硬件模块的 feature.主要工作：
(1)建立底层 user ids 和 group ids 同上层 permissions 之间的映射；可以指定一个权限与几个组 ID 对应。当一个 APK 被授予这个权限时，它也同时属于这几个组。
(2)给一些底层用户分配权限，如给 shell 授予各种 permission 权限；把一个权限赋予一个 UID，当进程使用这个 UID 运行时，就具备了这个权限。
(3) library,系统增加的一些应用需要 link 的扩展 jar 库；
(4) feature, 系统每增加一个硬件，都要添加相应的 feature. 将解析结果放入 mSystemPermissions,mSharedLibraries,mSettings.mPermissions,mAvailableFeatures 等几个集合中供系统查询和权限配置使用。

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

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

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

7.对于以上几个目录下的 apk 逐个解析，主要是解析每个 apk 的 AndroidManifest.xml 文件，处理 asset/res 等资源文件，建立起每个 apk 的配置结构信息，并将每个 apk 的配置信息添加到全局列表进行管理。调用 installer.install()进行安装工作,检查 apk 里的 dex 文件是否需要再优化,如果需要优化则通过辅助工

具 dexopt 进行优化处理；将解析出的 componet 添加到 pkg 的对应列表里；
对 apk 进行签名和证书校验,进行完整性验证。

8.将解析的每个 apk 的信息保存到 packages.xml 和 packages.list 文件里，
packages.list 记录了如下数据：pkgName，userId，debugFlag，dataPath（包的数据路径）

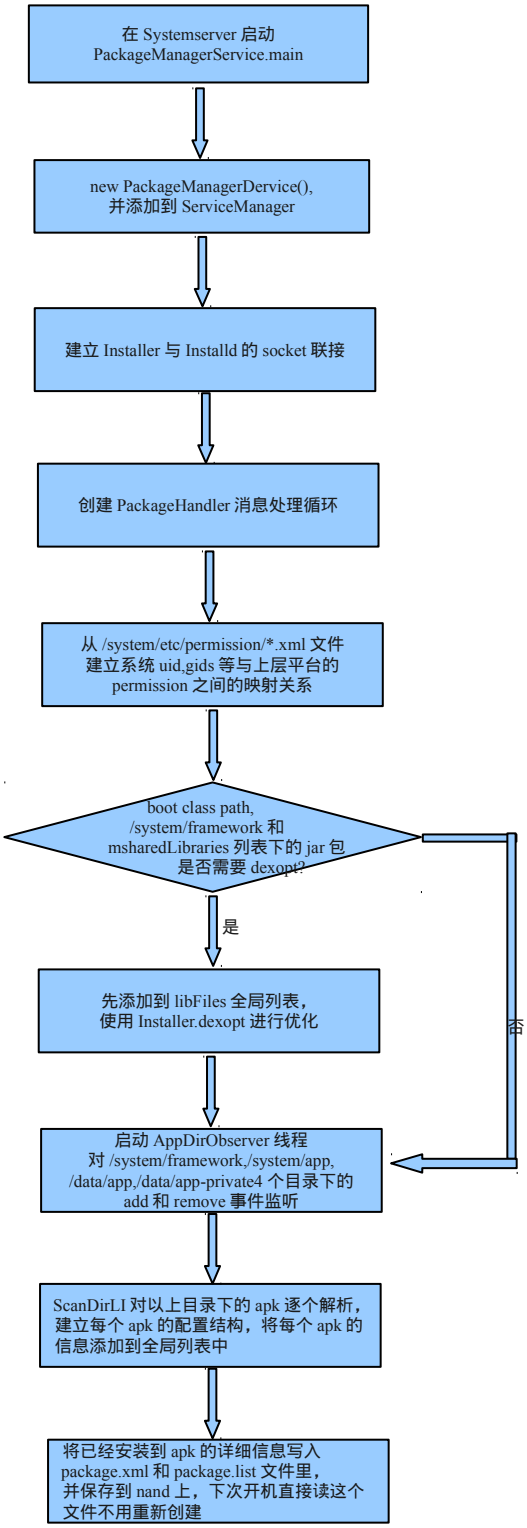


图 1 主流程图

详细分析

在 systemserver.java 中启动包管理服务
pm=PackageManagerService.main(context,factoryTest != SystemServer.FACTORY_TEST_OFF);
main 函数主要功能是构造 PackageManagerService 实例，然后添加到 ServiceManager 中。

```
    public static final IPackageManager main(Context context, boolean factoryTest) {
        PackageManagerService m = new PackageManagerService(context, factoryTest);
        ServiceManager.addService("package", m);
        return m;
    }
```

PackageManagerService(context, factoryTest) 是包管理服务的主进程。它完成了对/system/app,/data/app,/system/framework,/data/app-private 下的 apk 文件的解析。详细流程如下：

初始化过程：
判断 ro.build.type 是否等于 eng；
创建系统显示像素实例 mMetrics = new DisplayMetrics();

创建 mSettings 实例 mSettings = new Settings()，Settings 类是 PackageManagerService 的一个静态子类，它的作用主要是保持动态设置的信息，通过 Settings()构造函数在/data/system 下创建了三个文件名：packages.xml，packages-backup.xml(这个文件在 mSettings.writeLP()里被删除了)，packages.list。

mSettings 增加 android.uid.system，android.uid.phone，android.uid.log 三个共享用户 ID,同时授予其系统权限。

//建立 installer 与 installd 的 socket 联接
Installer installer = new Installer();
installer.ping() && Process.supportsProcesses()；
installd 完成以下一些命令

```
struct cmdinfo cmds[] = {
    { "ping",               0, do_ping },
    { "install",            3, do_install },
    { "dexopt",             3, do_dexopt },
    { "movedex",            2, do_move_dex },
    { "rmdex",              1, do_rm_dex },
    { "remove",             1, do_remove },
    { "rename",             2, do_rename },
    { "freecache",          1, do_free_cache },
    { "rmcache",            1, do_rm_cache },
    { "protect",            2, do_protect },
    { "getsize",            3, do_get_size },
    { "rmuserdata",         1, do_rm_user_data },
```

```
    { "movefiles",      0, do_movefiles },
};

//获取当前缺省的显示像素
WindowManager
wm=(WindowManager)context.getSystemService(Context.WINDOW_SERVICE);
Display d = wm.getDefaultDisplay();
d.getMetrics(mMetrics);

建立一个消息循环，用于处理 apk 安装时的请求消息处理（这些请求来自 adb install/push,
包安装器，android market 下载安装 apk 时发送的）
mHandlerThread.start();
mHandler = new PackageHandler(mHandlerThread.getLooper());
这个消息循环处理的消息事件如下：
    static final int SEND_PENDING_BROADCAST = 1;
    static final int MCS_BOUND = 3;
    static final int END_COPY = 4;
    static final int INIT_COPY = 5;
    static final int MCS_UNBIND = 6;
    static final int START_CLEANING_PACKAGE = 7;
    static final int FIND_INSTALL_LOC = 8;
    static final int POST_INSTALL = 9;
    static final int MCS_RECONNECT = 10;
    static final int MCS_GIVE_UP = 11;
    static final int UPDATED_MEDIA_STATUS = 12;
    static final int WRITE_SETTINGS = 13;

//创建/data/data 和/data/app-private 目录
File dataDir = Environment.getDataDirectory();//获得/data 目录
mAppDataDir = new File(dataDir, "data");
mDrmAppPrivateInstallDir = new File(dataDir, "app-private");

// Read permissions from /system/etc/permission directory.
//这些文件在 framework/base/data/etc
Void readPermissions()
{
      //解析/system/etc/permission/下的*.xml 文件，获取权限信息
      //最后解析该目录下的 platform.xml 文件，使该文件里的权限在栈顶出现，以便预先处
理
      //这个文件记录了系统级应用的 uid 及其拥有的权限
       File permFile = new File(Environment.getRootDirectory(),"etc/permissions/platform.xml");
       readPermissionsFromXml(permFile);
       //该函数的功能如下：
         通过 xml 解析器解释*.xml 文件，提取标签名“group”，"permission"，"assign-
```

permission"，"library"，"feature"并进行相应处理。在 platform.xml 中对底层的系统用户和组 ID(group ids)同上层的由平台管理的 permission 名字之间进行了关系映射，使它们关联起来。当一个应用被授予某个权限后，同时属于已知的组 ID，这个应用就可以进行允许这个组的文件系统操作，如（read,write,execute）。这里记录了一些系统级的应用的 uid 对应的 permission

//每个标签的含义：

group：安装到系统中的所有 APK 都具备的组 ID。

permission：可以指定一个权限与几个组 ID 对应。当一个 APK 被授予这个权限时，它也同时属于这几个组。

assign-permission：把一个权限赋予一个 UID，当进程使用这个 UID 运行时，就具备了这个权限。

library：为系统添加一些扩展库用的。对应的.jar 文件放在/system/framework/目录下。比如 Google Map 相关的库。

feature：每添加一个硬件，都要增加对应的 feature。将以上解析的结果对应放入 mGlobalGids，mSettings.mPermissions，mSystemPermissions,mSharedLibraries,,mAvailableFeatures 等几个 list 中供系统查询和权限配置使用。

}

//readLP()会判断/data/system/packages.xml 文件是否存在，如果不存在则返回 false,如果存在则进行解析，在系统第一次启动时 packages.xml 文件是不存在的，由 writeLP()创建该文件，并将该文件写到 nand 上，下次开机会直接读取并解析这个文件。解析的过程即是按照 xml 定义的标签，将对应的属性和值添加到全局列表中。packages.xml 文件中记录了系统安装的所有 apk 的属性权限的信息，当系统中的 apk 安装，删除或升级时，改文件就会被更新。

<permissions>标签定义了目前系统中定义的所有权限。主要分为两类：系统定义的（package 属性为 Android）和 APK 定义的（package 属性为 APK 的包名）

sharedUserId/userId:Android 系统启动一个普通的 APK 时，会为这个 APK 分配一个独立的 UID，这就是 userId。如果 APK 要和系统中其它 APK 使用相同的 UID 的话，那就是 sharedUserId。

perms:APK 的 AndroidManifest.xml 文件中，每使用一个<uses-permission>标签，<perms>标签中就会增加一项。

<shared-user>代表一个共享 UID，通常，共同实现一系列相似功能的 APK 共享一个 UID。<perms>标签中的 权限代表了这个共享 UID 的权限，所有使用的同一个共享 UID 的 APK 运行在同一进程中，这个进程的 UID 就是这个共享 UID，这些 APK 都具有这个共享 UID 的权限。

name:共享 UID 的名字，在 APK 的 Android:sharedUserId 属性中使用。

userId：使用这个共享 UID 的所有 APK 运行时所在的进程的 UID。

mRestoredSettings = mSettings.readLP();

//判断 boot class path 里的文件（jar 文件）是否需要 dexopt(判断标准是检查 DvmGlobals.bootClassPath 是否已经包含这个文件),如果需要先将文件添加到 libFiles 里，同时进行 dexopt：由 Installer 通过 socket 将命令传给 installd 的 run_dexopt,最终调用的

是/system/bin/dexopt 对 jar 包进行处理。如果已经进行了 dexopt 动作，则将/data/dalvik-cache 下的以 data 开头的文件删除，后续重新建立。如果外部库 mSharedLibraries 列表存在，也要检查列表中的元素是否需要 dexopt,如果需要则和 boot class path 进行相同处理。对于/system/framework 下 apk 和 jar 文件检查是否需要 dexopt.

```
String bootClassPath = System.getProperty("java.boot.class.path");
 if (bootClassPath != null) {
        String[] paths = splitString(bootClassPath, ':');
            for (int i=0; i<paths.length; i++) {
              try {
                      if (dalvik.system.DexFile.isDexOptNeeded(paths[i])) {//是否需要 dexopt
                          libFiles.add(lib);//添加到 libFiles 列表
                          mInstaller.dexopt(paths[i], Process.SYSTEM_UID, true);//进行 dexopt
                       }
                    }
             }
  }
//对 framework-res.apk 不进行 dexopt 直接添加到 libFiles
        libFiles.add(mFrameworkDir.getPath() + "/framework-res.apk");
```

// 启动 AppDirObserver 线程监测/system/framework， /system/app， /data/app， /data/app-private 几个目录的事件，主要监听的是 add 和 remove 事件。对于目录监听底层通过 inotify 机制实现，inotify 是在 2.6.13 中引入的新功能，它为用户态监视文件系统的变化提供了强大的支持；inotify 是一种文件系统的变化通知机制，如文件增加、删除等事件可以立刻让用户态得知，当监测到事件发生时该线程做何处理呢？

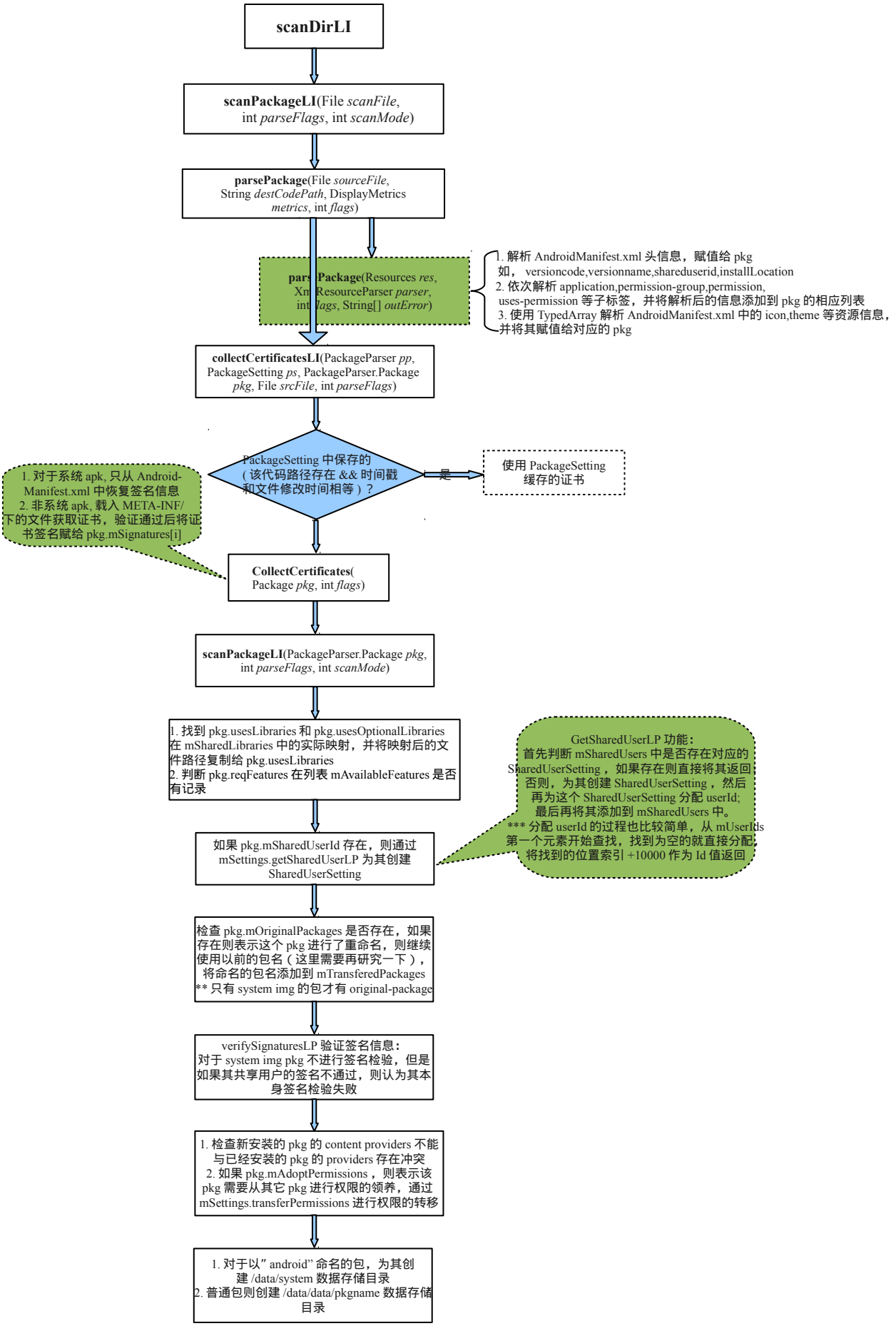
```
MframeworkInstallObserver =
                new AppDirObserver(mFrameworkDir.getPath(),OBSERVER_EVENTS, true);
mFrameworkInstallObserver.startWatching();
```

//调用 scanDirLI 解析以上目录下的 apk 文件，该函数是包管理服务的重要函数，在后面有详细的分析

```
    private void scanDirLI(File dir, int flags, int scanMode) {
        String[] files = dir.list();
        for (i=0; i<files.length; i++)
        {
            File file = new File(dir, files[i]);
            PackageParser.Package pkg =
                  scanPackageLI(file,flags|PackageParser.PARSE_MUST_BE_APK, scanMode);
            if (pkg == null && (flags & PackageParser.PARSE_IS_SYSTEM) == 0 &&
                    mLastScanError == PackageManager.INSTALL_FAILED_INVALID_APK) {
                // Delete the apk
                file.delete();
            }
        }
```

```
    }

//对于不存在的 system apk 调用以下函数删除掉
   Iterator<PackageSetting> psit = mSettings.mPackages.values().iterator();
   PackageSetting ps = psit.next();
   if ((ps.pkgFlags&ApplicationInfo.FLAG_SYSTEM) != 0
                    &&!mPackages.containsKey(ps.name)
                    && !mSettings.mDisabledSysPackages.containsKey(ps.name))
   {
         psit.remove();
         mInstaller.remove(ps.name);
   }

//在解析完以上目录下的 apk 后，更新应用的权限
       updatePermissionsLP(null, null, true, regrantPermissions, regrantPermissions);

//writeLP 会生成 packages.xml 和 packages.list 文件,packages.list 的数据格式是：
pkgName，userId，debugFlag，dataPath（包的数据路径）,packages.xml 保存了每个已经安
装 apk 的详尽的信息
       mSettings.writeLP();
以上是包管理服务在系统启动时做的全部工作。

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///
下面解析其中一个比较重要的函数 scanDirLI：

private void scanDirLI(File dir, int flags, int scanMode) {
        String[] files = dir.list();
        for (i=0; i<files.length; i++)
        {
            File file = new File(dir, files[i]);
            PackageParser.Package pkg =
                    scanPackageLI(file,flags|PackageParser.PARSE_MUST_BE_APK, scanMode);
            if (pkg == null && (flags & PackageParser.PARSE_IS_SYSTEM) == 0 &&
                    mLastScanError == PackageManager.INSTALL_FAILED_INVALID_APK) {
                // Delete the apk
                file.delete();
            }
        }
    }
这个函数结构比较简单，对监测的几个目录下的每一个 apk 文件继续通过
scanPackageLI(file,flags|PackageParser.PARSE_MUST_BE_APK, scanMode)进行解析，对不
```

存在且安装失败已经无效的非系统 apk 直接删除。



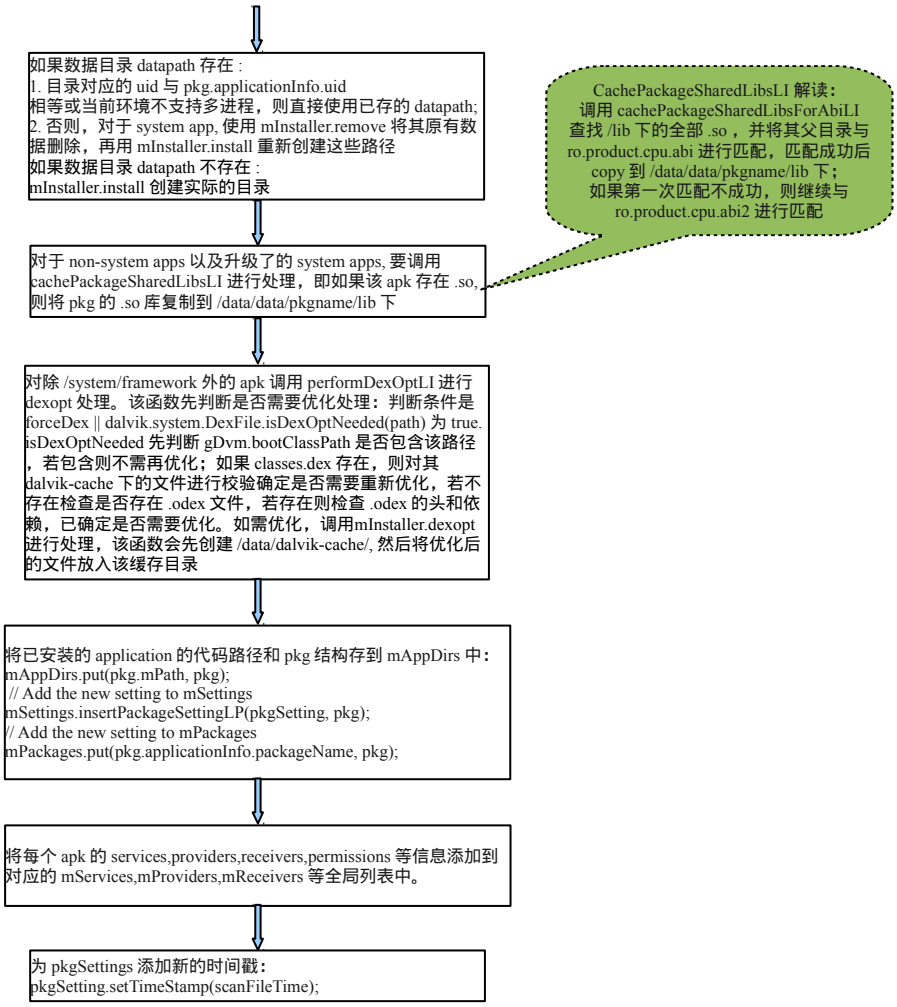


图 2 scanDirLI 流程分析

scanPackageLI 是一个重定义函数，它的作用是：用 PackageParser 的两个重定义函数 parsePackage 解析 package 的 asset,res，建立 asset 资源文件路径；解析 AndroidManifest.xml 文件，建立 PackageParser.Package 结构，这个结构保存了从 AndroidManifest.xml 解析出的 package 的信息。对 package 进行数字签名及完整性校验，

```
private PackageParser.Package scanPackageLI(File scanFile, int parseFlags, int scanMode)
{
    //实例化一个 PackageParser 对象
      PackageParser pp = new PackageParser(scanPath);

    // parsePackage 也是一个重定义函数，它主要做了三件事，一个是解析 apk 中的 asset 下的文件，一个是解析 res 下的文件，关于 asset 与 res 区别请参考：http://blog.csdn.net/hshm20517/archive/2011/06/02/6461890.aspx 。然后通过重定义函数 parsePackage(Resources res, XmlResourceParser parser, int flags, String[] outError) 对 apk 的 AndroidManifest.xml 进行解析,将每个标签对应的信息添加到每个包的相关列表中，如将标签 application 下的 activity 通过 pkg.activities.add(a) 添加到 package 的 activities 列表，将 service 添加到 owner.services.add(s)。
    PackageParser.Package pkg = pp.parsePackage(scanFile, scanPath, mMetrics, parseFlags);
```

//检查这个 package 是否已经存在，以及是否重命名过，以及该系统 package 是否可以被更新，如果可以被更新，则对比系统分区和 data 分区的 package 版本，如果系统分区的 package 高于 data 分区的版本，则保留系统分区的 package

//对 package 进行签名认证，如果是 system img 里的，只是通过 AndroidManifest.xml 获得签名，对签名校验，不会对全部文件进行有效性检查；否则，就要结合 META-INF/进行签名和有效性校验

collectCertificatesLI(pp, ps, pkg, scanFile, parseFlags);

//调用重定义函数继续进行解析,将每个 apk 解析出的标签信息添加到全局的列表里。如将每个 apk 的 recervers 列表里的元素 pkg.receivers.get(i), 通过 mReceivers.addActivity(a, "receiver")添加到全局列表 mReceivers 里

return scanPackageLI(pkg, parseFlags, scanMode | SCAN_UPDATE_SIGNATURE);

}

<补充知识>

res/raw 和 assets 区别

*res/raw 和 assets 的相同点：

1.两者目录下的文件在打包后会原封不动的保存在 apk 包中，不会被编译成二进制。

*res/raw 和 assets 的不同点：

1.res/raw 中的文件会被映射到 R.java 文件中，访问的时候直接使用资源 ID 即 R.id.filename；assets 文件夹下的文件不会被映射到 R.java 中，访问的时候需要 AssetManager 类。

2.res/raw 不可以有目录结构，而 assets 则可以有目录结构，也就是 assets 目录下可以再建立文件夹

*读取文件资源：

1.读取 res/raw 下的文件资源，通过以下方式获取输入流来进行写操作

InputStream is = getResources().openRawResource(R.id.filename);

2.读取 assets 下的文件资源，通过以下方式获取输入流来进行写操作

AssetManager am = null;

am = getAssets();

InputStream is = am.open("filename");

（用于内置文件但不知道文件名称，需要筛选出想要的文件然后拷贝到目标目录中，推荐内置在 assets 文件夹中）

1.res/raw 目录：

通过反射的方式得到 R.java 里面 raw 内部类里面所有的资源 ID 的名称，然后通过名称获取资源 ID 的值来读取我们想要的文件。

2.assets 目录：

getAssets().list("");来获取 assets 目录下所有文件夹和文件的名称，再通过这些名称读取我们想要的文件。

另，在处理 asset 时，android 限制最大的数据是 1M，超出后会报错误。

</>

public Package parsePackage(File sourceFile, String destCodePath, DisplayMetrics metrics, int flags)
{
	//解析/asset 下的文件
	assmgr = new AssetManager();
	int cookie = assmgr.addAssetPath(mArchiveSourcePath);
	parser = assmgr.openXmlResourceParser(cookie, "AndroidManifest.xml");

	//解析/Res 下的文件，通过 parsePackage 函数解析 AndroidManifest.xml 文件
	Resources res = new Resources(assmgr, metrics, null);
	pkg = parsePackage(res, parser, flags, errorText);

	// 设置代码路径和资源路径
	pkg.mPath = destCodePath;
	pkg.mScanPath = mArchiveSourcePath;
}

Package parsePackage(Resources res, XmlResourceParser parser, int flags, String[] outError)
{
解析 AndroidManifest.xml 里的各个标签,并对 pkg 的 mVersionCode,mSharedUserId, mSharedUserLabel,installLocation 等变量赋值。对于 application，permission-group，permission，permission-tree，uses-permission，uses-configuration，uses-feature，uses-sdk，supports-screens，protected-broadcast，instrumentation，original-package，adopt-permissions，eat-comment 等标签调用相关函数进行处理。解析出每个标签下的子标签的信息，然后将这些信息添加到每个 package 的对应列表中，如将 application 下的 activity 通过 pkg.activities.add(a)添加到 package 的 activities 列表。

	//将 pkg 返回
	return pkg;

}

private PackageParser.Package scanPackageLI(PackageParser.Package pkg, int parseFlags, int scanMode) {

	（1）Check all shared libraries and map to their actual file path.
	（2）check pkg.reqFeatures in mAvailableFeatures
	（3）Check and note if we are renaming from an original package name
	（4）Check if we are renaming from an original package name.
	对于 original package 不是太了解，还需要继续研究

	判断新装应用的 content providers 是否与已经安装应用的产生冲突。

```
        if (mPlatformPackage == pkg) {//如果包名以 android 开头的，则将应用的 dataDir 设为/data/system
            // The system package is special.
            dataPath = new File (Environment.getDataDirectory(), "system");
            pkg.applicationInfo.dataDir = dataPath.getPath();
        }else {
            // This is a normal package, need to make its data directory.
            dataPath = getDataPathForPackage(pkg);
        if (dataPath.exists()) {//如果路径存在
             使用 FileUtils.getPermissions 获取 dataPath 的权限，
            if (mOutPermissions[1] == pkg.applicationInfo.uid  || !Process.supportsProcesses())
                {
                    pkg.applicationInfo.dataDir = dataPath.getPath();
                } else {
                     mInstaller.remove(pkgName);//将该包删除
                     mInstaller.install(pkgName, pkg.applicationInfo.uid, pkg.applicationInfo.uid);
                      //重新安装该应用
                }
               pkg.applicationInfo.dataDir = dataPath.getPath();//dataDir 重新赋值
        }
        else
        {       //如果路径不存在则直接 install 安装
                mInstaller.install(pkgName, pkg.applicationInfo.uid, pkg.applicationInfo.uid);
         }

         // Perform shared library installation and dex validation and
        // optimization, if this is not a system app.
        performDexOptLI(pkg, forceDex);

          //如果新的应用已经安装，请求 ActivityManager 将旧的 kill 掉，以免使用时造成混乱
          if ((parseFlags & PackageManager.INSTALL_REPLACE_EXISTING) != 0) {
            killApplication(pkg.applicationInfo.packageName,pkg.applicationInfo.uid);
        }

            // Add the new setting to mSettings
            mSettings.insertPackageSettingLP(pkgSetting, pkg);
            // Add the new setting to mPackages
            mPackages.put(pkg.applicationInfo.packageName, pkg);
            // Make sure we don't accidentally delete its data.
            mSettings.mPackagesToBeCleaned.remove(pkgName);

            以下将每个包的 provider,service,activity 等信息添加到全局列表中
            mServices.addService(s);
```

```
            mReceivers.addActivity(a, "receiver");
            mActivities.addActivity(a, "activity");
            mPermissionGroups.put(pg.info.name, pg);
            permissionMap.put(p.info.name, bp);
            mInstrumentation.put(a.getComponentName(), a);
            mProtectedBroadcasts.add(pkg.protectedBroadcasts.get(i));
}
```