[在此处输入文章标题]

* 1、设计思想
* 2、PackageManagerService的抽象理解
* 3、PackageManagerService里面的数据结构
* 4、PackageManagerService的三大流程
* 5、PackageManagerService的体系结构

**一、设计思想**

如果你是Android 系统中的架构师，让你设计一个Android的安装系统中的PackageManagerService，你会怎么设计？ 既然要设计，咱们要首先弄清几个问题，我希望大家看下面的问题的时候，多想两个问题：1、如果让你设计，你怎么设计。这个"类"存在意义是什么?

* 1、为什么关机的时候手机是砖头，而开机后，所有APP都可以运行了，这是为什么？
* 2、Android系统是通过什么手段来加载已经安装到手机上应用的？
* 3、既然是加载，按照科学的架构设计，是不是应该存在一个管理者，来全局管理，那个这个类是什么？
* 4、在安装一个APK的时候，APK是"死的"，Android系统是怎么把它变成一个"活的"APP，他是怎么加载到内存中去的

那我们就来依次来看下这几个问题

**1、为什么关机的时候手机是砖头，而开机后，所有APP都可以运行了，它是怎么加载的?**

* 首先明确一点，手机关机以后，就是一个冰冷的砖头，只能用来"砸核桃"，那开机后，你点击桌面上的任何一个图片，都能开启一个APP，这说明在开机过程中，系统把已经安装好的APP加载到内存中，这到底是怎么做的？所以我们反推断，在安卓系统中肯定存在这么一块区域，用于存放已经安装的APP的信息，在开机的时候，通过系统扫描，这块区域，把对应的内容加载到内存中去。
* 其次，通过上面的分析，我们知道了在Android系统中存在这样一块区域，在开机的的时候，加载这块区域的信息，从而实现加载在内存中去。那么我们继续反推断，那这块区域的信息，是怎么来的？应该在安装这个APK的时候，把这个APK的信息写入到该区域的。这样就可以实现了在安卓系统一次安装后，在删除APK文件后，还可以运行APP了

其实上面的解答是基本上所有操作的系统的安装思路，大家可以想一下在Windows下是不是也是如此。

上面说的Android区域其实就是：**“/data目录”下的system目录，这个目录用来保存很多系统文件。主要工作是创建了5个位于目录/data/system的File对象**，分别是：

* packages.xml：记录了系统中所有安装的应用信息，包括基本信息、签名和权限
* pakcages-back.xml：packages.xml文件
* pakcages-stoped.xml：记录系统中被强制停止的运行的应用信息。系统在强制停止某个应用的时候，会将应用的信息记录在该文件中。
* pakcages-stoped-backup.xml：pakcages-stoped.xml文件的备份
* 保存普通应用的数据目录和uid等信息

这个5个文件中pakcages-back.xml和pakcages-stoped-backup.xml是备份文件。当Android对文件packages.xml和pakcages-stoped.xml写之前，会先把它们备份，如果写文件成功了，再把备份文件删除。如果写的时候，系统出问题了，重启后在需要读取这两个文件时，如果发现备份文件存在，会使用备份文件的内容，因为源文件可能已经损坏了。其中**packages.xml**是**PackageManagerServcie**启动时，需要用到的文件。  
我把我的Nexus 6P手机Root后，在/data/system 截屏如下：

/data/system目录.png

我把packages.xml导出来，文件内容太大，我就直接截屏了，内容如下：

截屏1.png

截屏2.png

图片看不清，可以看下面的缩减版

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

<packages>

<version sdkVersion="23" databaseVersion="3" fingerprint="google/angler/angler:6.0.1/MTC20L/3230295:user/release-keys" />

<version volumeUuid="primary\_physical" sdkVersion="23" databaseVersion="23" fingerprint="google/angler/angler:6.0.1/MTC19T/2741993:user/release-keys" />

<permission-trees>

<item name="com.google.android.googleapps.permission.GOOGLE\_AUTH" package="com.google.android.gsf" />

</permission-trees>

<permissions>

<item name="android.permission.REAL\_GET\_TASKS" package="android" protection="18" />

<item name="android.permission.REMOTE\_AUDIO\_PLAYBACK" package="android" protection="2" />

.....

<item name="com.android.voicemail.permission.ADD\_VOICEMAIL" package="android" protection="1" />

</permissions>

<package name="com.google.android.youtube" codePath="/system/app/YouTube" nativeLibraryPath="/system/app/YouTube/lib" primaryCpuAbi="arm64-v8a" publicFlags="945307205" privateFlags="0" ft="11e9134c000" it="11e9134c000" ut="11e9134c000" version="107560144" userId="10075">

<sigs count="1">

<cert index="0" key="" />

</sigs>

<perms>

<item name="com.google.android.c2dm.permission.RECEIVE" granted="true" flags="0" />

<item name="android.permission.USE\_CREDENTIALS" granted="true" flags="0" />

<item name="com.google.android.providers.gsf.permission.READ\_GSERVICES" granted="true" flags="0" />

<item name="com.google.android.youtube.permission.C2D\_MESSAGE" granted="true" flags="0" />

<item name="android.permission.MANAGE\_ACCOUNTS" granted="true" flags="0" />

<item name="android.permission.NFC" granted="true" flags="0" />

<item name="android.permission.CHANGE\_NETWORK\_STATE" granted="true" flags="0" />

<item name="android.permission.RECEIVE\_BOOT\_COMPLETED" granted="true" flags="0" />

<item name="com.google.android.gms.permission.AD\_ID\_NOTIFICATION" granted="true" flags="0" />

<item name="android.permission.INTERNET" granted="true" flags="0" />

<item name="android.permission.ACCESS\_NETWORK\_STATE" granted="true" flags="0" />

<item name="android.permission.VIBRATE" granted="true" flags="0" />

<item name="android.permission.ACCESS\_WIFI\_STATE" granted="true" flags="0" />

<item name="android.permission.WAKE\_LOCK" granted="true" flags="0" />

</perms>

<proper-signing-keyset identifier="11" />

<domain-verification packageName="com.google.android.youtube" status="0">

<domain name="youtu.be" />

<domain name="m.youtube.com" />

<domain name="youtube.com" />

<domain name="www.youtube.com" />

</domain-verification>

</package>

上面是我手机packages.xml的一个片段。我们看下里面的**"youtube"**应用。通过标签<package>记录了一个应用的基本信息，签名和声明的权限。

**(1)<package>表示包信息，下面我们就来解释下标签<package>中的属性**

* name表示应用的包名
* codePath表示的是apk文件的路径
* nativeLibraryPath表示应用的native库的存储路径
* flags是指应用的属性，如FLAG\_SYSTEM、FLAG\_PERSISTENT等
* it表示应用安装的时间
* ut表示应用最后一次修改的时间
* version表示应用的版本号
* userId表示所属于的id

**(2)<sign>表示应用的签名，下面我们就来解释下 标签<sign>中的属性**

* count表示标签中包含有多少个证书
* cert表示具体的证书的值

**(3)<perms>表示应用声明使用的权限，每一个子标签代表一项权限**

通过上面的内容，我们知道Android系统通过packages.xml文件来存储应用信息的，所以我们举一反三，新安装的APK，肯定是把新安装的APK相关信息写入这个packages.xml文件中，那么怎么把这个xml文件，映射到内存中的？ 那我们就来看第二个问题

**2、Android系统是通过什么手段来加载已经安装到手机上应用的**

上面提到了，应用的信息都存储在packages.xml中的<package>标签里面，那我们是怎么加载到内存中去的？大家平时是存储数据库的时候都是怎么做的？对的，一般都是一个实体类对应数据库中的一个表；其中每一个对象对应的是数据库中的一条数据。同理，Android系统也是这样设计的，<package>标签里面记录的包信息其实是一一对应的PackageSetting类。

PackageSetting类的继承关系.png

PackageSetting继承了PackageSettingBase类，PackageSettingBase类继承自GrantedPremisson类。应用的基本信息保存在PackageSettingBase类的成员变量中，声明的权限保存在GrantedPremissions类，签名则保存在SharedUserSetting类的成员变量signatures中。标签<package>所标识的应用PackageSetting对象都保存在Setting的mPackages中，定义如下：

// com.android.server.pm.Settings.java

final HashMap<String, PackageSetting> mPackages =

new HashMap<String, PackageSetting>();

在packages.xml中除了标签<package>，还有<updated-package>、<cleaning-package>和<renamed-package> 这三种标签。

* <updated-package> 标签表示升级包覆盖的系统应用，对应的是PackageSetting，在Settings里面同样用mPackages 变量表示
* <cleaning-package> 标签用来记录那些已经删除，但是数据目录还暂时保留的应用的信息。对应的是PackageCleanItem。在Settings里面用mPackagesToBeCleaned变量表示
* <renamed-package> 标签用来记录系统中改名的应用。它的记录信息都插入到mSettings的mRenamedPackages对象中。

其中mPackagesToBeCleaned和mRenamedPackages在mSettings.java的定义如下：

// com.android.server.pm.Settings.java

// Packages that have been uninstalled and still need their external

// storage data deleted.

final ArrayList<PackageCleanItem> mPackagesToBeCleaned = new ArrayList<PackageCleanItem>();

// Packages that have been renamed since they were first installed.

// Keys are the new names of the packages, values are the original

// names. The packages appear everwhere else under their original

// names.

final HashMap<String, String> mRenamedPackages = new HashMap<String, String>();

上面用大量的文笔说Settings，那么它是什么东西？下面就让我继续来看下一个问题

**3、既然是加载，按照科学的架构设计，是不是应该存在一个管理者，来全局管理，那个这个类是什么？**

这个类就是Settings

Settings是Android的包的全局管理者，用于协助PackageManagerService保存所有的安装包信息，同时用于存储系统执行过程中的一些设置，PackageManagerService和Settings之间的类图关系如下：

PackageManagerService和Settings的关系.png

[大图地址1](https://link.jianshu.com?t=http%3A%2F%2Fimg.voidcn.com%2Fvcimg%2F000%2F003%2F552%2F709_ec8_79b.jpg)

Settings里面有3个重要的成员变量：mShareUsers，mPackages，mSharedUsers 。如下：

final ArrayMap<String, SharedUserSetting> mSharedUsers =

new ArrayMap<String, SharedUserSetting>();

final ArrayMap<String, PackageSetting> mPackages =

new ArrayMap<String, PackageSetting>();

final ArraySet<String, SharedUserSetting> mSharedUsers =

new ArraySet<String, SharedUserSetting>();

* mShareUsers是一个以String类型的name为"key"，ShareUserSetting对象为"value"的ArrayMap。
* mPackages是一个以String类型的name为"key"，PackageSetting对象为"value"的ArrayMap。
* mSharedUsers 是一个以String类型的name(比如"android.uid.system")为"key"，以SharedUserSetting 对象为"value"的HashMap

其中ShareUserSetting类继承自GrantedPermissions ，内部包含一个ArraySet类型的packages ，这个packages保存了声明相同的shareUserId的Package的权限设置信息（PackageSetting ）通过上面的问题，我们知道PackageSetting继承自PackageSettingBase，同时PackageSetting中保存着package的多种信息。

如下图：

PackageSetting.png

上面提到了一个概念是SharedUserSetting，那么ShareUserSetting的作用什么是什么？那我们就来看下：

SharedUserSetting用来描述具有相同的sharedUserId的应用信息，它的成员变量packages保存了所有具有相同sharedUserId的应用信息引用。这些应用的签名时相同的，所有只需要在成员变量signatures中保存一份。通过这个对象，Android运行时很容易检索到某个应用拥有相同的sharedUserId的其他应用。其中应用的签名保存在ShardUserSetting类的成员变量signatures中。

我们在看系统应用的AndroidManifest.xml中会发现

<manifest

xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:androidprv="http://schemas.android.com/apk/prv/res/android"

package="com.android.settings" coreApp="true"

android:sharedUserId="android.uid.system"

android:versionCode="1"

android:versionName="1.0" >

</manifest>

shareUserId与UID相关，作用是：

* 1、两个或多个APK或者进程声明了同一种sharedUserId的APK可以共享批次的数据，并且可以在运行在同一进程中(相当于进程是系统的用户，某些进程可以归为同一用户使用，相当于Linux系统中的GroupId)
* 2、通过声明特定的sharedUserId，该APK所在的进程将被赋予指定的  
  　　2.通过声明特定的sharedUserId,该APK所在的进程将被赋予指定的UID，将被赋予该UID特定的权限。

在Settings中和用户有关的还有两个重要变量，即mUserIds 和mOtherUserIds

private final ArrayList<Object> mUserIds = new ArrayList<Object>();

private final SparseArray<Object> mOtherUserIds =

new SparseArray<Object>();

他们都是以UID为索引，得到对应的ShardUserSetting对象。更多的关于Android系统中关于"用户"的信息，在后面"用户模块"再单独讲解

**4、在安装一个APK的时候，APK是"死的"，Android系统是怎么把它变成一个"活的"APP，他是怎么加载到内存中去的**

**这里就不得不提一下PackageParser这个类，这个类负责在APK文件安装的时候，解析AndroidManifest。**

在Android中，Settings提供可持续化的包信息管理，PackageSetting是一个存储单元，表示一个pkg信息。我们在解析APK安装包的时候，会用到PackageParser，在PackageParser里面有一个字段是PackageParse.Package。这个PackageParse.Package其实是对应的上面packages.xml里面的<package>标签。同时PackageParse.Package也可以理解为pkg信息在内存中的一个实时信息，关机后变消失，重启后重新生成，所以PackageParse.Package中的信息一致保证最新。PackageParse.Packag、Settings和PackageSetting三者的关系如下：

关系.png

Settings中保存了一个包名和PackagesSetting的映射表，PackageParse.Package中的mExtras引用指向了对应的PackageSetting实例，而PackageParse中保存了一个PackageParse.Package列表

PackageParse.png

从上到下，介绍如下：  
1：PackageParser.Package对应一个apk完整的原始数据  
2：PackageSetting包含一个PackageParser对象实例，说明它也对应一个apk包的数据，不同的是，它还包括apk相关配置数据，比如apk内部哪些component是被disable等。  
3：Settings包含了PackageSetting对象列表，也就是说它包含了系统所有apk数据，还有就是PackageParser，顾名思义，负责APK数据解析  
4：PackageManagerService是全局的包管理器

**5、补充一点：**

Settings里面的主要关联关系如下图：

主要关联关系.png

**二、PackageManagerService的抽象理解**

**上面说了很多，我们再上升一个高度，PackageManagerService到底应该怎么去理解它？**

每一个组织结构，都有一套自己的管理机制，比如任何一家公司，都会存在下面三个元素：管理者(经理)、被管理者(员工)、管理机制(公司的规章制度及KPI考核等)。同理在Android的系统的世界里面，也有一家公司叫"包管理"。如果要研究Android的包管理机制，同样可以从以下几个角度来思考？

* 管理者是谁，他的职责是什么？
* 被管理者是谁，他的职责是什么？
* 管理机制是什么，它是如何运转的？

所谓包，其实就是一种文件的格式，比如APK包，JAR包等，在Android中存活着很多包，所有的应用程序都是APK包，很多构成Android运行环境的都是JAR包，还有一些以so为后缀的库文件，包管理者很重要的一个职责就是识别不同的包，统一维护这些包的信息。当有一个包进入(安装)或者离开(卸载)Android世界，都需要向包管理者申报，其他管理部分要获取包的具体信息，也都需要向包管理者申请。

如同一家公司是由人与人协作工作的，不同包之间也需要进行协作。既然有协作，自然就有协作的规范，一个包可以干什么，不可以干什么，都需要有一个明确的范围界定，这就是包管理中的权限设计。涉及到的内容非常广泛，Android的权限管理、SELinux，都是包管理中权限设计的组成部分。同理Android的世界就像一个井然有序的一家公司，既有包管理部门，也有其他各种管理部门，比如电源管理部门，窗口管理部门等等。大家不仅各司其职，而且也有来往。比如在APK安装到Activity的显示，看着很简单的过程，其实却需要大量的管理部门参与进来，不断地进行数据解析、封装、传递、呈现，其内部机制十分复杂。

现在大家想一下上面三个问题的答案，我详细大部分人的前两个答案是一致的，管理者是PackageManagerService，被管理是各种"包"，最后一个答案是各有千秋，这里是没有标准答案的，希望大家能自己找到自己的答案。

**三、PackageManagerService里面的数据结构**

PackageManagerService涉及的数据结构非常多，在分析源码时，很容易陷入各种数据结构之间的关系，难以自拔，以至于看不到包管理的全貌。我在这里简单的总结了一下各个数据结构的职能如下：

* PackageManangerService ：包管理的核心服务
* com.android.server.pm.Settings ：所有包的管理信息
  + com.android.server.pm.PackageSetting ：单一包的信息
  + com.android.server.pm.BasePermission ：系统中已有的权限
  + com.android.server.pm.PermissionState ：授权状态

—————————————分隔符—————————————

* PackageParser：包解析器
  + PackageParser.Package ：解析得到的包信息
  + PackageParser.Component ：组件的基类，其子类对应到AndroidManifest.xml中定义的不同组件
    - PackageParser.Activity 对应AndroidManifest.xml中定义<Activity>和<Receiver>标签
    - PackageParser.Service ：对应AndroidManifest.xml中定义<Service/>标签
    - PackageParser.Provider ：对应AndroidManifest.xml中定义<Provider/> 标签
    - PackageParser.Instrumentation ：对应AndroidManifest.xml中定义<Instrumentation/> 标签
    - PackageParser.Permission ：对应AndroidManifest.xml中定义<permission/> 标签
    - PackageParser.PermissionGroup ：对应AndroidManifest.xml中定义<permission-group/> 标签
  + PackageLite ：轻量的包信息
  + ApkLite ：轻量级的APK信息
  + IntentInfo ：组件所定义的<intent-filter/>信息，保存了每个<intent-filter/>节点的信息，是基类，它的子类是ActivityIntentInfo、ServiceIntentInfo和ProviderIntentInfo
    - ActivityIntentInfo ：保存<activity/>和<Receiver/>节点下的<intent-filter/>节点
    - ServiceIntentInfo ：保存<service/>节点下的<intent-filter/>节点
    - ProviderIntentInfo：保存<provider/> 节点下的<intent-filter/>节点

PackageParser的数据结构.png

—————————————分隔符—————————————

* PackageInfo ：跨进程传递的包数据，包解析时生成
  + PackageItemInfo ：一个应用包内所有组件项和通用信息的基类。提供最基本的属性集，如：label、icon、meta-data等。
    - ApplicationInfo：代表一个特定应用的基本信息，对应AndroidManifest里面的<application>
    - InstrumentationInfo：用作进行instrumentation的测试的片段，对应AndroidManifest里面的<instrumentation>
    - PermissionInfo：代表一个特定的权限，对应AndroidManifest里面的<permission/>
    - PermissionGroupInfo ：一个特定的权限组，对应AndroidManifest里面的<permission-group/>
    - ComponentInfo：代表一个应用内组件(如activityInfo、serviceInfo、ProviderInfo)通用信息的基类。一般不会直接使用该类，它设计为了不同应用的组件共享统一的定义。
      * ActivityInfo ：对应AndroidManifest.xml里面的注册的<activity/>标签和<receiver/>标签。代表一个Activity或者receiver
      * ServiceInfo ：对应AndroidManifest.xml里面的注册的<service/>标签。代表一个service
      * ProviderInfo ：对应AndroidManifest.xml里面的注册的<service/>标签。代表一个Provider
* Intent：根据特定的条件找到匹配的组件
* IntentFilter ：Intent过滤器
  + ResolveInfo
  + IntentResolver ：Intent解析器，其子类用于不同组件的Intent解析  
    保存了所有<activity/>或者<receiver/>节点信息。(Activity或者BroadcastReceiver信息就是用该自定义类保存的)
    - ActivityIntentResolver ：保存所有<activity/>和<receiver/>节点信息。(Activity或者BroadcastReceiver信息就是用该自定义类保存的)  
      保存了所有<service/>节点信息。(Service信息就是用该自定义类保存的)。
    - ServiceIntentResolver ：保存了所有<service /> 节点信息。(Service信息就是用该自定义类保存的)
    - ProviderIntentReslover：保存了所有 <provider /> 节点信息
* PackageHandler ：包管理的消息处理器
  + HandlerParams ：消息的数据载体
    - InstallParams ： 用于APK的安装
    - MeasureParams：用于查询某个已安装的APK占据存储空间的大小(例如在设置程序中得到某个APK的缓存文件大小)
    - MoveParams ：用于已安装APK的位置移动
  + InstallArgs ：APK的安装参数
    - FileInstallArgs ：针对是安装在内部存储的APK
    - AsecInstallArgs ：针对安装在SD卡上的APK
    - MoveInfoArgs ： 移动APK  
      这么庞大的数据结构，其各个数据结构的类图如下：

数据结构.png

[大图地址3](https://link.jianshu.com?t=http%3A%2F%2Fimg.voidcn.com%2Fvcimg%2F000%2F003%2F552%2F711_f4c_7d3.jpg)

**四、PackageManagerService的三大过程组**

如果大家想对Android系统有一个大致的了解，就必须要要了解PackageManagerService的三大流程

* 1、包扫描的过程组：  
  即Android将一个APK文件的静态信息转化为可以管理的数据结构
* 2、包安装的过程组：  
  即包管理接纳一个新成员的体现。
* 3、包查询的过程组：  
  即Intent的定义和解析是包查询的核心，通过包查询服务可以获取到一个包的信息

下面我们来一一进行简单的介绍

**(一)、包扫描过程组——即开机扫描过程**

**1、为什么要进行包扫描?**

扫描目录的目的：

扫描Android系统的几个目标文件中的APK，从而建立合适的数据结构以及管理诸如Package信息、四大组件、授权信息等各种信息。抽象的地看，PackageManagerService像一个工厂，它解析实际的物理文件(APK文件)，以及生成符合自己要求的产品。比如PackageManagerService将解析APK包中的AndroidManifest.xml，并根据其中声明的Activity标签来创建与此对应的对象，并保存到PackageParser.Package类型的变量中，然后通过PackageManagerService的scanPackageDirtyLI()方法将解析后的组件数据统计到PackageManagerService的本地变量中，用于管理查询调用，当系统中任意某个APK的package发生改变时，如卸载，升级等操作都会更新package的统计数据到PackageManagerService，PackageManagerService正式基于拥有系统中所有的Package的信息才能胜任"包管理"这个管理者的角色。PackageManagerService的工作流程相对简单，复杂的是其中用于保存各种信息的数据结构和它们的关联关系，以及对应影响结果的策略控制(比如系统应用和普通应用)

**2、包扫描过程组的不同理解**

如果把包扫描过程组看成一件事，那么这件事就是：

调用PackageManagerService类的静态方法main()方法来获取PackageManagerService对象

如果把包扫描过程组看成两件事，那么这两件事就是

1、创建PackageManagerService对象  
2、将PackageManagerService向ServiceManager注册，即加入SMS，方便后续其他进程或者app通过ServiceManager获得PackageManagerService服务。

如果把包扫描过程组看成三件事，那么这三件事是：

1、先读取保存在packages.xml中记录的系统关机前记录所有安装的APP信息， 将其保存在PackageManagerServiced中mSettings中的mPackages中。  
2、扫描指定的若干目录中的app，并把信息记录在PackageManagerServiced的mPackages中。  
3、最后上面的两者进行对比，看是否有升级的APP，然后进行相关处理，最后写入package.xml中

当然换一个角度，以扫描角度来看，也可以把包扫描分解成另外三个阶段：

* 扫描目标文件夹之前的准备工作
* 扫描目标文件夹
* 扫描目标文件夹之后的工作

如果把包扫描过程组看成四件事，那么这四件事是：

1、读取响应的配置文件  
2、优化APK和Jar包  
3、扫描系统中所有安装的应用  
4、把扫描出的所有应用信息进行保存

如果把包扫描过程组划分的更细，则我将其分为6大步骤

* 1、变量初始化，包括mSettings，mInstaller，mPackageDexOptimizer等等
* 2、读取配置文件
* 3、扫描系统Package，包含Dex优化
* 4、保存扫描信息
* 5、扫描非系统应用
* 6、更新数据

如果把包扫描过程组划分的更细，则我将其分为9大步  
第一步：创建**Settings**对象，并调用其**addSharedUserLPw()**方法，保存**ShareUserSetting**信息  
第二步：创建**Installer**对象，用于Native进程installd交互  
第三步：创建**ThreadHandler**线程，并以其**Looper**为参数创建**PackageHandler**对象，用于程序的安装和卸载  
第四步：根据**Installer**对象和**/data/user**文件对象创建**UserManager**对象，用于多用户管理  
第五步：调用**readPermissions()**方法，从**/system/etc/permissions**目录下的XML文件读取权限信息  
第六步：调用**Settings**对象的**readLPw()**方法解析**/data/system**目录下的文件：  
第七步：扫描**/system/frameworks**目录以及**BOOTCLASSPATH**和**platform.xml**定义的系统目录下的jar和APK文件是否需要dex优化，如果需要则调用**Installer.dexopt()**方法来发送消息给**installd**让它优化；如果任意一个文件执行了dex优化操作，删除**/data/dalvik-cache**目录下的缓存文件  
第八步：创建**AppDirObserver**对象监听**/system/frameworks、/system/app、/vendor/app(厂商定制)、/data/app、/data/app-private**5个目录，并调用scanDirLI()方法扫描其中的APK文件：  
第九步：汇总上面扫描XML和APK得到的信息，并写入文件;

**3、如果把包扫描过程组划分为"方法级"的流程，如下图：**

开机扫描流程.png

[大图地址3](https://link.jianshu.com?t=http%3A%2F%2Fimg.blog.csdn.net%2F20170828235549810)

**4、温馨提醒**

* 在packages.xml中<package>标签记录的APP的安装信息。有独立uid的APP，后面再反序列化的时候，会映射为PackageSetting对象，保存在mSettings的mPackages中；有sharedUid的APP，后面反序列化的时候，会映射为PendingPackage对象，保存在mSettings的mPendingPackages中。
* 对于<share-user>标签记录的的share uid信息，封装为SharedUserSetting对象，保存到mSettings里面的mSharedUser中，在此过程中遇到的uid和 shared uid都保存在mUserIds中，并让每个uid指向与之关联的PackageSetting对象，或者SharedUserSetting对象

**5、小结**

* PackageManagerService是伴随着系统进程启动而启动的，最终会构造一个PackageManagerService对象，此后，PackageManagerService将成为Android世界的包管理者，对外提供包的增、删、改、查的操作
* 在PackageManagerService的启动过程中，最重要的是对所有静态APK文件进行扫描，生成一个在内存中的数据结构Package，PackageManagerService实际上就是维护这所有在内存中的数据结构。已有的包的历史信息会写入磁石，PackageManagerService的Settings专门来管理写入磁盘的包信息。
* 所有包的信息扫描完成后，需要对应用进行授权，这是Android权限管理的一部分。随着 Android版本的升级，授权机制略有区别，总体框架是：每个APK都可以声明权限，并为权限设定保护级别，其他APK需要使用这些权限的时候，需要先申请，再由系统判定是否进行授权。

**(二)、包安装的过程组——即安装一个新的APK**

安装一个APK的其大致流程如下：

大致流程.png

通常，安装一个APK 通常分为以下4种方式

* 安装系统应用
* 网络下载应用安装
* ADB工具安装
* 第三方应用安装

下面我们就依次介绍下

**1、 安装系统应用**

系统的应用的安装主要在PackageManagerService的main方法里面进行操作的

其顺序如下：  
第一步：PackageManagerService.main()初始化注册  
第二步：建立Java层的installer与C层的intalld的socket联接  
第三步：建立PackageHandler消息循环  
第四步：调用成员变量mSettings的readLPw()方法恢复上一次的安装信息  
第五步：.jar文件的detopt优化  
第六步：scanDirLI函数扫描特定目录的APK文件解析  
第七步：updatePermissionsLPw()函数分配权限  
第八步：调用mSettings.writeLPr()保存安装信息

**2、 网络下载应用安装**

其顺序如下：  
第一步：调用PackageManagerService的installPackage方法  
第二步：上面的方法调用installPackageWithVerfication()，进行权限校验，发送INIT\_COPY的msg  
第三步：进入PackageManagerService的doHandleMessage方法的INIT\_COPY分支  
第四步：成功绑定了com.android.defcontainer.DefaultContainerService服务，进入MCS\_BOUND分支  
第五步：里面调用PackageManagerService中内部抽象类HandlerParams的子类InstallParams的startCopy方法。  
第六步：抽象类的HandlerParams的startCopy方法调用了HandlerParams子类的handleStartCopy和handlerReturnCode两个方法  
第七步：handlesStartCopy方法调用了InstallArgs的子类copyApk，它负责将下载的APK文件copy到/data/app  
第八步：handleReturnCode调用handleReturnCode方法  
第九步：调用PackageManagerService服务的installPackageLI(PackageParser.Package, int, int, UserHandle, String, String,PackageInstalledInfo)方法进行APK扫描。  
第十步：上面的方法判断是否APP应安装，调用installNewPackageLI或replacePackageLI方法  
第十一步：调用updateSettingsLI方法进行更新PackageManagerService的Settings  
第十二步：发送what值为POST\_INSTALL的Message给PackageHandler进行处理  
第十三步：发送what值为MCS\_UNBIND的Message给PackageHandler，进而调用PackageHandler.disconnectService()中断连接

**3、 ADB工具安装**

Android Debug Bridge (adb)是SDK自带的管理设备的工具，通过ADB命令的方式也可以为手机或者模拟器安装应用，其入口函数为pm.java

Android Debug Bridge (adb) 是SDK自带的管理设备的工具，通过ADB命令行的方式也可以为手机或模拟器安装应用，其入口函数源文件为[pm.java](https://link.jianshu.com?t=http%3A%2F%2Fandroidxref.com%2F6.0.1_r10%2Fxref%2Fframeworks%2Fbase%2Fcmds%2Fpm%2Fsrc%2Fcom%2Fandroid%2Fcommands%2Fpm%2FPm.java)

其顺序如下：  
第一步：pm.java的runInstall()方法  
第二步：参数不对会调用showUsage方法，弹出使用说明  
第三步：正常情况runInstall会调用mPm变量的installPackageWithVerification方法  
第四步：由于pm.java中的变量mPm是PackageManagerService的实例，所以实际上是调用PackageManagerService的installPackageWithVerfication()方法  
第五步：进入PackageManagerService的doHandleMessage方法的INIT\_COPY分支  
第六步：成功绑定了com.android.defcontainer.DefaultContainerService服务，进入MCS\_BOUND分支  
第七步：里面调用PackageManagerService中内部抽象类HandlerParams的子类InstallParams的startCopy方法。  
第八步：抽象类的HandlerParams的startCopy方法调用了HandlerParams子类的handleStartCopy和handlerReturnCode两个方法  
第九步：handlesStartCopy方法调用了InstallArgs的子类copyApk，它负责将下载的APK文件copy到/data/app  
第十步：handleReturnCode调用handleReturnCode方法  
第十一步：调用PackageManagerService服务的installPackageLI(PackageParser.Package, int, int, UserHandle, String, String,PackageInstalledInfo)方法进行APK扫描。  
第十二步：上面的方法判断是否APP应安装，调用installNewPackageLI或replacePackageLI方法  
第十三步：调用updateSettingsLI方法进行更新PackageManagerService的Settings  
第十四步：发送what值为POST\_INSTALL的Message给PackageHandler进行处理  
第十五步：发送what值为MCS\_UNBIND的Message给PackageHandler，进而调用PackageHandler.disconnectService()中断连接

ADB安装.png

**4、 第三方应用安装**

第一步：调用PackageInstallerActivity的onCreate方法初始化安装界面  
第二步：初始化界面以后调用initiateInstall方法  
第三步：上面的方法调用startInstallConfirm方法，弹出确认和取消安装的按钮  
第四步：点击确认按钮，打开新的activity：InstallAppProgress  
第五步：InstallAppProgress类初始化带有进度条的界面之后，调用PackageManager的installPackage方法  
第六步：PackageManager是PackageManagerService实例，所以就是调用PackageManagerService的installPackage方法  
第七步：调用PackageManagerService的installPackage方法  
第八步：上面的方法调用installPackageWithVerfication()，进行权限校验，发送INIT\_COPY的msg  
第九步：进入PackageManagerService的doHandleMessage方法的INIT\_COPY分支  
第十步：成功绑定了com.android.defcontainer.DefaultContainerService服务，进入MCS\_BOUND分支  
第十一步：里面调用PackageManagerService中内部抽象类HandlerParams的子类InstallParams的startCopy方法。  
第十二步：抽象类的HandlerParams的startCopy方法调用了HandlerParams子类的handleStartCopy和handlerReturnCode两个方法  
第十三步：handlesStartCopy方法调用了InstallArgs的子类copyApk，它负责将下载的APK文件copy到/data/app  
第十四步：handleReturnCode调用handleReturnCode方法  
第十五步：调用PackageManagerService服务的installPackageLI(PackageParser.Package, int, int, UserHandle, String, String,PackageInstalledInfo)方法进行APK扫描。  
第十六步：上面的方法判断是否APP应安装，调用installNewPackageLI或replacePackageLI方法  
第十七步：调用updateSettingsLI方法进行更新PackageManagerService的Settings  
第十八步：发送what值为POST\_INSTALL的Message给PackageHandler进行处理  
第十九步：发送what值为MCS\_UNBIND的Message给PackageHandler，进而调用PackageHandler.disconnectService()中断连接

流程.png

[点击放大查看高清无码大图](https://link.jianshu.com?t=http%3A%2F%2Fsolart.cc%2Fimages%2Finstall_apk_seq.png)

小结：

* 1、安装和卸载都是通过PackageManager，实质上是实现了PackageManagerService来完成具体的操作，所有细节和逻辑均可以在PackageManagerService中跟踪查看。
* 2、所有安装方式殊途同归，最终就是回到PackageManagerService中，然后调用底层本地代码的installd来完成的。
* 3、APK的安装过程主要分为以下几步：  
  - 拷贝到apk文件到指定目录  
  - 解压缩apk，拷贝文件，创建应用的数据目录  
  - 解析apk的AndroidManifest.xml文件  
  - 向Launcher应用申请添加创建快捷方式

**(三)、包查询的过程组——即解析Intent并找到其配备的组件**

**1、包管理是以什么样的形式对外提供服务那？**

在写应用程序时，我们通常会利用应用自身的上下文环境Context来获取包管理服务，如下：

// 获取一个PackageManager的对象实例

PackageManager pm = context.getPackageManager();

// 通过PackageManager对象获取指定包名的包信息

PackageInfo pi = pm.getPackageInfo("com.android.contacts", 0);

这么一段简单的代码，其实蕴含很多的深意

* 1、上面已经讲解过了PackageManagerService和其管理的各种数据结构，都是运行在系统进程之中。在应用进程中获取的PackageManager对象，只是PackageManagerService在应用进程中的一个代理，不同的应用进程都有不同的代理，意味着不同应用进程中的PackageManager是不同的，但是管理者PackageManagerService有且只有一个
* 2、运行在应用进程中的PackageManager要与运行在系统进程中的PackageManagerService进行通信，通信手段是Android中最常见的Binder机制。因此会有一个IPackageManager.aidl文件，用于描两者通信的接口。另外，应用进程中的PackageInfo对象。PackageInfo其实就是由系统进程传递到应用进程的对象

IPackageManager.png

PackageManagerService作为包管理的最核心组成部分，伴随着系统的启动而创建，并一直运行系统进程中。当应用程序需要获取包管理服务时，会生成一个PackageManager对PackageManagerService进行通信。在包解析时就会生成包信息，即XXInfo这一类数据结构，PackageManagerService将这些数据传递给需要的应用进程。

管理者对内设计了复杂的管理机制，对外封装了简单的使用接口。这种设计在Android中大量出现，比如ActivityManagerService、WindowManagerService、PowerManagerService等，基本所有的系统服务都遵循这种设计规范。对于应用程序而言，不需关心管理者的实现原理，只需要理解接口的使用场景

Android在全局定义了IPackageManager，接口，描述了包管理者对外提供的功能，运行在系统进程中的PackageManagerService实现了IPackageManager接口，作为包管理的服务端，客户端通过IPackageManager接口请求包服务。为了方便客户端进行包服务，Android做了多层的封装。应用进程作为客户端，通过PackageManager便可使用包服务，客户端实际存在的对象是ApplicationPackageManager，它封装了IPackageManager的所有接口。在应用进程来看，客户端和服务端的概念是模糊的，明确的只有运行环境的概念，即Context。包服务就存在于应用进程的运行环境中，需要时直接拿出来使用即可。

“运行环境(Context)”是Android的设计哲学之一，Android有意弱化进程，强化运行环境，这是面向应用开发者的设计。运行环境是什么并不是一个很好回到的问题。可以将其类比为我们的工作环境，当我们需要办公设备时，只需要向管理部门申请，并不需要关心办公设备如何采购，办公设备对一般的工作人员而言，就像是工作环境中天然存在的东西。

**2、包管理的具体服务形式——Intent的解析：**

在Android中，使用Intent来表达意图，最终会有一个响应者。当系统产生一个Intent后，如何找到它的响应者？这需要对Intent进行解析。作为所有包信息管理者的中枢，PackageManagerService自然有义务承担解析Intent的责任。要解析Intent，就需要了解Intent的结构，标识了一个Intent身份的信息由两部分构成：

* 主要信息：主要信息Action和Data。Action用于表明Intent所要执行的操作，譬如ACTION\_VIEW，ACTION\_EDIT；Data用于表明执行操作的数据，譬如联系人数据，数据是以URI来表达的。再举两个Action和Data成对出现的例子：  
  - ACTION\_VIEW：[content://contacts/people/1](https://link.jianshu.com?t=content%3A%2F%2Fcontacts%2Fpeople%2F1) ：标识查看联系人数据库中，ID为1的联系人信息  
  - ACTION\_DIAL：tel:119 ：表达拨打电话给119  
  **上面两个例子的URI并不一样，完整的URI格式为**[**scheme://host:port/path**](https://link.jianshu.com?t=scheme%3A%2F%2Fhost%3Aport%2Fpath)**。**
* 次要信息：除了主要标记的信息，Intent还可以附加很多额外的信息，比如Category，Type，Componten和Extra：  
  - Category：标识Intent的类别，譬如CATEGROY\_LAUNCHER标识对属于左面的图标这一类的对象执行操作  
  - Type：标识Intent所有操作的数据类型，就是MIMEType，譬如要操作PNG图片，那Type就可以设置为png  
  - Component：标识Intent要操作的对象  
  - Extra：标识Intent所传递的数据  
  上述这些数据都实现了Parcelable接口。

之所以Intent信息的主次之分，是因为解析Intent的规则需要有一个依据，主要信息是最能表达意图的，而次要信息则是解析规则的一个补充。这就像大家在做自我介绍的时候，总是先说姓名、籍贯这些主要的信息，再额外补充爱好、特长这些次要信息，这样一来在和其他人交朋友的时候，其他人就可以先根据籍贯、姓名锁定我。如果我们只介绍爱好、特长，那么别人锁定的范围就比较广，因为有相同爱好或者特长的人比较多。

Intent身份信息，其实就是Android的一种设计语言，譬如"打电话给119"，只需要发出Action为ACTION\_DIAL，URI为“tel:119”的Intent即可，剩下的就交给Android系统去理解这个意图。任何组件只要按照规则发生，都会被Android系统正确的理解。

而根据Intent的方式不同，可以将Intent分为两类：

* 显示(Explicit)：明确指明需要谁来响应Intent。这一类Intent的解析过程比较简单
* 隐式(Implicit)：有系统找出合适的目标来响应Intent。这一类Intent的解析过程比较复杂，由于目标明确，所以需要经过层层筛选才能找到最合适的响应者。

之所以Intent有显式和隐式之分，是因为解析Intent的方式不同，如果我指定要和某某交朋友，那么发出的这一类请求，就是显式Intent；如果没有指定交朋友的对象，只是说找到跟我爱好相同的人，那发出的这一类请求，就是隐式的Intent。对待这两种Intent显然有不同的解析方式。

**如果和"运行环境Context"一样，Intent也是面向应用程序设计，同样是弱化了了进程的概念。应用程序只需表明"我想要什么"，不需要关心索要的东西在什么地方，如何找到索要的东西。Intent是Android通信的手段之一，可以承载要传递的信息，至于信息怎么从发起进程传递到目标进程，应用程序可以毫不关心。**

Intent最后一个响应者是一个Android组件，Android组件都可以定义IntentFilter，前面说了包解析器的时候，说到了每一个Component类中都有一个IntentInfo对象的数组，而IntentInfo则是IntentFilter的子类。既然一个Android组件可以定义多个IntentFilter，那么Intent想要匹配到最终的组件，则需要通过组件所定义的所有IntentFilter：

IntentFilter.png

多个IntentFilter之间是"或"的关系，哪怕其他所有IntentFilter都匹配失败，只要有一个IntentFilter通过，最终Intent还是找到了可以响应的组件。

每一个IntentFilter就像是一个定义了白名单规则的过滤器，只有满足白名单的要求才会放行。Intent的过滤规则，其实就是针对Intent的身份信息的匹配规则，当Intent的身份信息与IntentFilter所规定的要求匹配上，则允许通过；否则，Intent就被过滤掉了。IntentFilter的过滤规则包含以下三个方面：

* Action：每个IntentFilter可以定义零个或多个<action标签>，如果Intent想要通过这个IntentFilter，则Intent所辖的Action需要匹配其中至少一个。
* Category：每一个IntentFilter可以定义零个或者多个<category>标签，如果Intent想要通过这个IntentFilter，则Intent所辖的Categroy必须是IntentFilter所定义的Category的子集，才能通过IntentFileter。譬如Intent设置了两个Category：CATEGORY\_LAUNCHER和CATEGORY\_MAIN，只有那些至少定义了两项Category的IntentFilter，才会放行该Intent。启动Activity时，会为Intent设置默认的Category，即CATEGORY\_DEFAULT。目标Activity需要添加一个category伪CATEGORY\_DEFAULT的IntentFilter来匹配这一类隐式的Intent。
* Data：每一个IntentFilter可以定义零个或多个<data>，数据可以通过类型(MIMEType)和位置(URI)来描述，如果Intent想要通过这个IntentFilter，则Intent所辖的Data需要匹配其中至少一个。

在了解Intent的身份信息和IntentFilter的规则定义之后，就可以介绍Intent解析的过程了，PackageManagerService有四大组件的Intent解析器，分别是ActivityIntentResolver用于解析发往Activity或Broadcast的Intent，ServiceIntentResolver用于解析发往Service的Intent，ProviderIntentResolver用于解析发往Provider的Intent，系统每收到一个Intent的解析请求时，就会使用对应的解析器，他们都是IntentResolver的子类。

IntentResolver的职能就是解析Intent，它包含了所有IntentFilter，同时有一个重要的成员函数queryIntent()，接受Intent作为参数，返回查询结构：一个ResolveInfo对象的数据。因为可能有多个组件来响应一个Intent，所以返回结果是一个数组。可想而知，该函数就是针对输入的Intent，按照前面所述的过滤规则，逐个与IntentFilter进行匹配，直到找到最终的响应者，便加入返回结果的列表。

* ResolveInfo是最终的Intent解析结果的数据结构，并不复杂，就是各类组件信息的一个包装。需要注意的是，其中的组件信息ActivityInfo、ProviderInfo、ServiceInfo只有一个不为空，这样就可以区分不同组件的解析结果。
* 解析"显式"的Intent，如果Intent中有设置Component，则说明已经显式的指明由谁来响应Intent。根据Component可以获取到对应的 ActivityInfo，再封装成ResolveInfo便可以作为结果返回了。
* 解析"隐式"的Intent，该处逻辑比较复杂，后面讲解Activity的启动流程时再详细讲解。

**3、小结**

* 包管理对外提供服务的形式基于Bidner机制，服务端是运行在系统进程中的PackageManagerService，包查询服务是使用范围很广的一类服务，很多其他服务都需要用到包信息，都是通过PackageManagerService获取的。
* 包查询服务的核心是Intent解析，PackageManagerService中实现了不同组件的解析器。针对一个输入的Intent，解析得到可以响应的组件。Android为此设计了IntentFilter机制，定义了Intent匹配规则，最终解析实现在IntentResolver.queryIntent()函数中

**五、PackageManagerService的体系结构**

image.png

[大图地址](https://link.jianshu.com?t=http%3A%2F%2Fwww.qingpingshan.com%2Fuploads%2Fallimg%2F170309%2F13402UH8-2.jpg)

**六、总结**

本片文章主要讲解了包管理的三个大的过程：**包扫描过程、包查询过程、包安装过程**，其中重点的是**包安装的过程**。我们再来复习一下：

APK的安装流程如下：

复制APK安装包到/data/app目录下，解压缩并扫描安装包，向资源管理器注入APK资源，解析AndroidManifest文件，并在/data/data目录下创建对应的应用数据目录，然后针对Dalvik/ART环境优化dex文件，保存到dalvik-cache目录，将AndroidManifest文件解析出的组件、权限注册到PackageManagerService并发送广播

具体流程如下：

├── PMS.installPackage()

└── PMS.installPackageAsUser()

|传递 InstallParams 参数

PackageHandler.doHandleMessage().INIT\_COPY

|

PackageHandler.doHandleMessage().MCS\_BOUND

├── HandlerParams.startCopy()

│ ├── InstallParams.handleStartCopy()

│ │ └──InstallArgs.copyApk()

│ └── InstallParams.handleReturnCode()

│ └── PMS.processPendingInstall()

│ ├── InstallArgs.doPreInstall()

│ ├── PMS.installPackageLI()

│ │ ├── PackageParser.parsePackage()

│ │ ├── PackageParser.collectCertificates()

│ │ ├── PackageParser.collectManifestDigest()

│ │ ├── PackageDexOptimizer.performDexOpt()

│ │ ├── InstallArgs.doRename()

│ │ │ └── InstallArgs.getNextCodePath()

│ │ ├── replacePackageLI()

│ │ │ ├── shouldCheckUpgradeKeySetLP()

│ │ │ ├── compareSignatures()

│ │ │ ├── replaceSystemPackageLI()

│ │ │ │ ├── killApplication()

│ │ │ │ ├── removePackageLI()

│ │ │ │ ├── Settings.disableSystemPackageLPw()

│ │ │ │ ├── createInstallArgsForExisting()

│ │ │ │ ├── deleteCodeCacheDirsLI()

│ │ │ │ ├── scanPackageLI()

│ │ │ │ └── updateSettingsLI()

│ │ │ └── replaceNonSystemPackageLI()

│ │ │ ├── deletePackageLI()

│ │ │ ├── deleteCodeCacheDirsLI()

│ │ │ ├── scanPackageLI()

│ │ │ └── updateSettingsLI()

│ │ └── installNewPackageLI()

│ │ ├── scanPackageLI()

│ │ └── updateSettingsLI()

│ ├── InstallArgs.doPostInstall()

│ ├── BackupManager.restoreAtInstall()

│ └── sendMessage(POST\_INSTALL)

│ |

│ PackageHandler.doHandleMessage().POST\_INSTALL

│ ├── grantRequestedRuntimePermissions()

│ ├── sendPackageBroadcast()

│ └── IPackageInstallObserver.onPackageInstalled()

└── PackageHandler.doHandleMessage().MCS\_UNBIND

└── PackageHandler.disconnectService()

至此，整个**APK安装流程详解**全部说完，谢谢！

作者：隔壁老李头  
链接：https://www.jianshu.com/p/f47e45602ad2  
來源：简书  
简书著作权归作者所有，任何形式的转载都请联系作者获得授权并注明出处。