# apk安装过程

1.系统应用安装

2.网络下载应用安装

3. ADB工具安装

4.第三方应用安装



## 概述

众所周知，Android应用最终是打包成.apk格式（其实就是一个压缩包），然后安装至手机并运行的。APK即Android Package的缩写。

Android系统在启动的过程中，会启动一个应用程序管理服务PackageManagerService，这个服务负责扫描系统中特定的目录，找到里面的应用程序文件，即以Apk为后缀的文件，然后对这些文件进解析，得到应用程序的相关信息，完成应用程序的安装过程。

**应用程序管理服务PackageManagerService安装应用程序的过程，其实就是解析析应用程序配置文件AndroidManifest.xml的过程，并从里面得到得到应用程序的相关信息，例如得到应用程序的组件Activity、Service、Broadcast Receiver和Content Provider等信息，有了这些信息后，通过ActivityManagerService这个服务，我们就可以在系统中正常地使用这些应用程序了**。

### 安装有如下四种方式

一般而言，Android应用安装有如下四种方式：

* 系统应用安装：开机时加载系统的APK和应用，没有安装界面；
* 网络下载应用安装：通过各种market应用完成，没有安装界面；
* ADB工具安装：即通过Android的SDK开发tools里面的adb.exe程序安装，没有安装界面；
* 第三方应用安装：通过SD卡里的APK文件安装(比如双击APK文件触发)，有安装界面，系统默认已经安装了一个安装卸载应用的程序，即由packageinstaller.apk应用处理安装及卸载过程的界面。

安装过程：复制APK安装包到data/app目录下，解压并扫描安装包，把dex文件(Dalvik字节码)保存到dalvik-cache目录，并data/data目录下创建对应的应用数据目录。

卸载过程：删除安装过程中在上述三个目录下创建的文件及目录。

### 应用安装涉及到的目录

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

/data/app ：用户程序安装的目录。安装时把apk文件复制到此目录

/data/data ：存放应用程序的数据

/data/dalvik-cache：将apk中的dex文件安装到dalvik-cache目录下(dex文件是dalvik虚拟机的可执行文件,当然，ART–Android Runtime的可执行文件格式为oat，启用ART时，系统会执行dex文件转换至oat文件)

/data/system ：该目录下的packages.xml文件，类似于Windows**的注册表**，这个文件是在解析apk时由writeLP()创建的，里面记录了系统的permissions，以及每个apk的name,codePath,flags,ts,version,uesrid等信息，这些信息主要通apk的AndroidManifest.xml解析获取，解析完apk后将更新信息写入这个文件并保存到flash，下次开机直接从里面读取相关信息添加到内存相关列表中。当有apk升级，安装或删除时会更新这个文件。/data/system/packages.xml中内容详解(这里列举的标签内容不一定完整，只是列举核心内容，packages.xml的完整定义详见官方文档)：

### 安装概述

#### 拷贝apk文件到指定目录

在Android系统中，apk安装文件是会被保存起来的，默认情况下，用户安装的apk首先会被拷贝到 /data/app 目录下。/data/app目录是用户有权限访问的目录，在安装apk的时候会自动选择该目录存放用户安装的文件，而系统出厂的apk文件则被放到了 /system 分区下,包括 /system/app，/system/vendor/app，以及 /system/priv-app 等等，该分区只有Root权限的用户才能访问，这也就是为什么在**没有Root手机之前，我们无法删除系统出厂的app的原因了。**

#### 解压apk，拷贝文件，创建应用的数据目录

为了加快app的启动速度，apk在安装的时候，会首先将app的可执行文件（dex）拷贝到 /data/dalvik-cache 目录，缓存起来。

然后，在/data/data/目录下创建应用程序的数据目录（以应用的包名命名），存放应用的相关数据，如数据库、xml文件、cache、二进制的so动态库等等。

#### 解析apk的AndroidManifinest.xml文件

Android系统中，也有一个类似注册表的东西，用来记录当前所有安装的应用的基本信息，每次系统安装或者卸载了任何apk文件，都会更新这个文件。这个文件位于如下目录：

/data/system/packages.xml。系统在安装apk的过程中，会解析apk的AndroidManifinest.xml文件，提取出这个apk的重要信息写入到packages.xml文件中，这些信息包括：权限、应用包名、APK的安装位置、版本、userID等等。

由此，我们就知道了为啥一些**应用市场**和软件管理类的app能够很清楚地知道当前手机所安装的所有的app，以及这些**app的详细信息**了。

另外一件事就是Linux的用户Id和用户组Id，以便他可以获得合适的运行权限。以上这些都是由PackageServiceManager完成的，下面我们会重点介绍PackageServiceManager。

#### 显示快捷方式

这些应用程序只是相当于在PackageManagerService服务注册好了，如果我们想要在Android桌面上看到这些应用程序，还需要有一个Home应用程序，负责从PackageManagerService服务中把这些安装好的应用程序取出来，并以友好的方式在桌面上展现出来，例如以快捷图标的形式。在Android系统中，负责把系统中已经安装的应用程序在桌面中展现出来的Home应用程序就是Launcher了

## PackageManagerService启动过程

SystemServer组件是由Zygote进程负责启动的，启动的时候就会调用它的main函数，Android系统在启动的过程中，会启动一个应用程序管理服务PackageManagerService，这个服务负责扫描系统中特定的目录，找到里面的应用程序文件，即以Apk为后缀的文件，然后对这些文件进解析，得到应用程序的相关信息。应用程序管理服务PackageManagerService安装应用程序的过程，其实就是解析析应用程序配置文件AndroidManifest.xml的过程，并从里面得到得到应用程序的相关信息，例如得到应用程序的组件Activity、Service、Broadcast Receiver和Content Provider等信息，有了这些信息后，通过ActivityManagerService这个服务，我们就可以在系统中正常地使用这些应用程序了。应用程序管理服务PackageManagerService是系统启动的时候由SystemServer组件启动的，启后它就会执行应用程序安装的过程，因此，本文将从SystemServer启动PackageManagerService服务的过程开始分析系统中的应用程序安装的过程。



## PackageManagerService入口



## 安装详解

### 系统应用安装方式



**1.PackageManagerService.main()初始化注册**

将PackageManagerService服务初始化并注册到ServiceManager里面进行管理。

**2.建立java层的installer与c层的installd的socket联接**

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

**3.建立PackageHandler消息循环**

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

典型的比如INIT\_COPY和MCS\_BOUND等，在通过网络下载时候会调用。

**4. 成员变量readLp（）恢复上一次的安装信息**

由于Android每次启动的时候都需要安装一次信息，但是有些信息是保持不变的，例如Linux用户组Id，PackageManagerService 每次安装程序之后，都会把这些程序的信息保存下来，以便下次使用， 恢复上一次程序的安装信息是通过PackageManagerService 的成员变量mSetting的readLP()来实现的，恢复信息之后就开始扫描和安装app了。

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

**5.jar的detopt优化**

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

**6.scanDirLI函数扫描特定目录的apk文件解析**

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

调用installer.install()进行安装工作,检查apk里的dex文件是否需要再优化,如果需要优化则通过辅助工具dexopt进行优化处理；将解析出的componet添加到pkg的对应列表里；对apk进行签名和证书校验,进行完整性验证。



**7.updatePermmisonLp函数分配权限**

这个函数为申请了特定资源访问权限的app，分配相应的用户组ID.

**8.writeLP()函数保存安装信息**

mSetting的writeLP（）将所获得应用程序的安装信息，保存在一个本地的配置文件中。以便下次安装的时候，将应用的信息回复过来。

### 网络下载应用安装



### ADB工具安装

其入口函数源文件为pm.java

(源文件路径：android\frameworks\base\cmds\pm\src\com\android\commands\pm\pm.java)

其中\system\framework\pm.jar 包管理库

包管理脚本 \system\bin\pm 解析

showUsage就是使用方法

|  |
| --- |
| private static void showUsage() {  System.err.println("usage: pm [list|path|install|uninstall]");  System.err.println(" pm list packages [-f]");  System.err.println(" pm list permission-groups");  System.err.println(" pm list permissions [-g] [-f] [-d] [-u] [GROUP]");  System.err.println(" pm list instrumentation [-f] [TARGET-PACKAGE]");  System.err.println(" pm list features");  System.err.println(" pm path PACKAGE");  System.err.println(" pm install [-l] [-r] [-t] [-i INSTALLER\_PACKAGE\_NAME] [-s] [-f] PATH");  System.err.println(" pm uninstall [-k] PACKAGE");  System.err.println(" pm enable PACKAGE\_OR\_COMPONENT");  System.err.println(" pm disable PACKAGE\_OR\_COMPONENT");  System.err.println(" pm setInstallLocation [0/auto] [1/internal] [2/external]");  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*省略\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  } |

安装时候会调用 runInstall（）方法

private void runInstall() {

int installFlags = 0;

String installerPackageName = null;

String opt;

while ((opt=nextOption()) != null) {

if (opt.equals("-l")) {

installFlags |= PackageManager.INSTALL\_FORWARD\_LOCK;

} else **if (opt.equals("-r")) {**

**installFlags |= PackageManager.INSTALL\_REPLACE\_EXISTING;**

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

installerPackageName = nextOptionData();

if (installerPackageName == null) {

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

showUsage();

return;

}

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

installFlags |= PackageManager.INSTALL\_ALLOW\_TEST;

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

// Override if -s option is specified.

installFlags |= PackageManager.INSTALL\_EXTERNAL;

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

// Override if -s option is specified.

installFlags |= PackageManager.INSTALL\_INTERNAL;

} else {

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

showUsage();

return;

}

}

String apkFilePath = nextArg();

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

if (apkFilePath == null) {

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

showUsage();

return;

}

PackageInstallObserver obs = new PackageInstallObserver();

try {

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

installerPackageName);

synchronized (obs) {

while (!obs.finished) {

try {

obs.wait();

} catch (InterruptedException e) {

}

}

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

System.out.println("Success");

} else {

System.err.println("Failure ["

+ installFailureToString(obs.result)

+ "]");

}

}

} catch (RemoteException e) {

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

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

}

}

其中的 因为有

mPm = IpackageManager.Stub.asInterface(ServiceManager.getService("package"));

Stub是接口IPackageManage的静态抽象类，asInterface是返回IPackageManager代理的静态方法。因为class PackageManagerService extends IPackageManager.Stub所以mPm.installPackage 调用

/\* Called when a downloaded package installation has been confirmed by the user \*/

public void installPackage(

final Uri packageURI, final IPackageInstallObserver observer, final int flags,final String installerPackageName)

这样就是从网络下载安装的入口了。



### 第三方应用安装



## 总结

1.安装和卸载都是通过PackageManager，实质上是实现了PackageManager的远程服务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，最终就回到PackageManagerService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

## Ref

[深度探究apk安装过程](http://www.sohu.com/a/132599459_465908)

[Android应用程序安装过程解析(源码角度)](http://www.jianshu.com/p/21412a697eb0)

# build.prop系统参数属性详解

build.prop中参数包括

1. Dalvik虚拟机相关参数
2. 系统版本、定义相关参数、
3. 基本性能相关参数、
4. 基本耗电相关参数、
5. 扩展性能较调及附加功能开启

## Dalvik虚拟机相关参数

**dalvik.vm.** **heapstartsize**

本参数控制Dalvik虚拟机在启动一个应用程序之后为其分配的初始堆栈大小，可填写的值为**2m~48m**。例如：dalvik.vm.heapstartsize=8m，就表示应用程序启动后为其分配的初始堆栈大小为8兆字节。这里分配的内存容量会影响到整个系统对RAM的使用程度，和第一次使用应用程序时的流畅程序。这个值越大，系统消耗RAM则越快，但是应用程序打开后的反应也越快。值越小，系统的RAM剩余则越多，但是程序在启动后会很卡。建议值是8m，既可以保持140M左右的RAM，程序的反应速度也会大幅度提高。

**dalvik.vm.heapsize**

本参数控制Dalvik虚拟机给一个应用程序分配的最大堆栈量，可填写的值为12m~48m。例如：dalvik.vm.heapsize=48m，就表示应用程序在任意时刻内可以使用的最大堆栈大小为48兆字节。这里分配的内存容量会影响到整个系统对RAM的使用程序，和程序在运行一段时间后的反应速度。**这个值越大，系统消耗RAM则越快，但是程序会运行的非常稳定**，尤其是游戏和视频程序的内容加载速度可以大幅度提升。值越小，系统的RAM剩余则越多，但是程序会很卡，尤其是游戏在切换场景Loading的时候会花费很多的时间。若应用程序需要使用超过这个值的内存时，将会触发系统的垃圾收集器，系统和程序就会卡顿。建议值是40~40m。

**dalvik.vm.lockprof.threshold**

本参数控制Dalvik虚拟机调试记录程序内部锁资源争夺的阈值，默认值是500。多用于程序的数据统计，对性能较调意义不大。

**dalvik.vm.stack-trace-file**

本参数控制Dalvik虚拟机的堆栈记录调试文件。用于系统调试，一般用户对其调整无意义,默认为/data/anr/traces.txt，这是开发者分析anr的重要文件。

**dalvik.vm.execution-mode**

本参数控制Dalvik虚拟机的程序执行机制。可填写的值有"int:portable"、"int:fast"和"int:jit"。int:portable表示以兼容模式运行(脚本翻译模式)，此模式下程序的兼容性最高，但其执行效率最低(程序优化度依赖于dalvik虚拟机版本)。官方默认此模式。int:fast表示以快速自优化模式运行(脚本翻译和预优化混合)，此模式下程序的兼容性很高，执行效率也比较高。因为此时dalvik虚拟机允许程序使用自己的预定义优化模式和代码(包括C/C  /汇编代码)。推荐使用。**int:jit表示以Just-In-Time**模式运行(JIT模式)，此模式下程序的兼容性最差，**但程序一旦加载后其运行效率最高(与C/C  直接编写的程序效率无异)**，因为在此模式下dalvik虚拟机会预先将Java程序翻译成针对机器平台的本地语言(Native)，同时完全允许代码中的所有预优化和代码，**允许所有不安全的非托管代码，同时不严谨的程序如果运行在JIT模式可能会造成内存泄露**。但要注意，**很多Dalvik虚拟机并不支持此模式**(如官方2.2)。

**dalvik.vm.dexopt-flags**

本参数控制Dalvik虚拟机的程序代码校验和优化。可填写的值有m、v和o。m为标准选项，可以是m=y或m=n，若m=y则启用不安全代码的校验和托管代码的优化，兼容性和安全性最高，推荐使用；

v为校验选项，可与o并存，可以是v=a或v=n。若v=a则表示校验所有代码，v=n则关闭代码的校验

o为优化选项，可与v并存。可以是o=v或o=a。若o=v则表示优化以校验过的代码，o=a则表示优化所有代码。例如：dalvik.vm.dexopt-flags=m=ydalvik.vm.dexopt-flags=v=n,o=v注意，这个参数只会影响到安装APK之后或初次使用APK时生成dex文件时有效。若整个系统(包括应用程序)为odex化，则无意义。

**dalvik.vm.verify-bytecode**

本参数控制Dalvik虚拟机是否验证应用程序的可执行代码。可以与上一个参数配合使用。可填写的值为true和false。其具体意义与dalvik.vm.dexopt-flags的v=n一模一样。但可以与dalvik.vm.dexopt-flags配合使用以取得更好的效果。例如：dalvik.vm.dexopt-flags=v=n,o=vdalvik.vm.verify-bytecode=false这样可以令后来安装的apk文件可以被优化而不被检验。dalvik.vm.checkjni，本参数控制Dalvik虚拟机在调用外部jni链接库的时候是否对其做安全性检验。可填写的值为true和false。此参数会覆盖ro.kernel.android.checkjni。若值为true，会增加程序的兼容性和稳定性，但也会增加其加载和执行的时间。推荐为false。

**dalvik.vm.deadlock-predict**

本参数控制Dalvik虚拟机对程序死锁预测处理。可填写的值有off、warn和err。off表示关闭死锁预测功能(默认设置)。warn表示在继续程序运行的同时只记录该死锁预测(如果为真死锁就会出现程序假死现象，然后等N久出现关闭)。err表示预测到死锁时马上弹出FC。注意：有些Dalvik虚拟机版本并不支持此参数。

超级急速流畅型

dalvik.vm.startheapsize=16m

dalvik.vm.heapsize=48m

dalvik.vm.execution-mode=int:jit

dalvik.vm.dexopt-flags=v=n,o=v

dalvik.vm.verify-bytecode=false

dalvik.vm.checkjni=false

常用稳定加流畅型：

dalvik.vm.startheapsize=8m

dalvik.vm.heapsize=40m

dalvik.vm.execution-mode=int:fast

dalvik.vm.dexopt-flags=m=y

dalvik.vm.verify-bytecode=false

dalvik.vm.checkjni=false

超级稳定大内存型：

dalvik.vm.startheapsize=4m

dalvik.vm.heapsize=30m

dalvik.vm.execution-mode=int:portable

dalvik.vm.dexopt-flags=v=a,o=v

dalvik.vm.verify-bytecode=true

dalvik.vm.checkjni=true

**(以荣耀一代为例，RAM 512M，以下是像我一样追求速度的童鞋的推荐参数)**  
**dalvik.vm.startheapsize=16m  
dalvik.vm.heapsize=64m（RAM大的童鞋这个参数可以再大一些）  
dalvik.vm.execution-mode=int:jit  
dalvik.vm.dexopt-flags=v=n,o=v  
dalvik.vm.verify-bytecode=false  
dalvik.vm.checkjni=false**

# Path

编译生成的apk路径：out/target/product/ maguro/system/app

签名文件目录：/build/target/product/security

Android系统应用的源码路径：/packages/apps目录或其子目录中

源码目录/package/apps/PackageInstaller

# Task

Settings.Global

Settings源码解读

Wifi热点的ip的查询

# QA

1. adb install –r A.apk 执行原理？tmp目录作用，INSTALL\_FAILED\_ILLEGITIMATE\_APK各种安装报错原因，cnt的作用，为啥可以。
2. 系统目录结构是啥
3. App和系统签名的原理是什么？
4. 在sws下正常运行as，执行一次系统签名呢

# 参考

[Android系统应用的开发和测试](http://www.epubit.com.cn/book/onlinechapter/30644)