### 讲讲 PackageManagerService运行流程详解

本专栏专注分享大型Bat面试知识，后续会持续更新，喜欢的话麻烦点击一个star

**面试官: 讲讲 PackageManagerService运行流程详解**

**心理分析**：PMS与AMS是面试的高发区，面试官通过该问题想了解求职者的android工作年限。一般刚进入android行业 很难弄清楚ams与pms的。而能弄清楚ams与pms工作时间肯定不会短。如何弄清楚pms也成了拿到高薪的一块敲门砖

**求职者:**应该apk安装，apk管理，apk删除讲起，

本文涉及源码基于Android 6.0源码 如需看完本文，需要下载以下源码文件

frameworks/base/services/core/java/com/android/server/pm/PackageManagerService.java
frameworks/base/services/core/java/com/android/server/pm/PackageInstallerService.java
frameworks/base/services/core/java/com/android/server/pm/Settings.java
frameworks/base/services/core/java/com/android/server/SystemConfig
frameworks/base/core/java/android/content/pm/PackageManager.java

frameworks/base/core/android/java/content/pm/IPackageManager.aidl
frameworks/base/core/java/android/content/pm/PackageParser.java
frameworks/base/cmds/pm/src/com/android/commands/pm/Pm.java

frameworks/base/services/core/java/com/android/server/pm/Installer.java
frameworks/base/core/java/com/android/internal/os/InstallerConnection.java

## 一.概述

PackageManagerService(简称PKMS)，是Android系统中核心服务之一，管理着所有跟package相关的工作，常见的比如安装、卸载应用。 PKMS服务也是通过binder进行通信，IPackageManager.aidl由工具转换后自动生成binder的服务端IPackageManager.Stub和客户端IPackageManager.Stub.Proxy，具体关系如图：

package\_manager\_service

package\_manager\_service

* Binder服务端：PackageManagerService继承于IPackageManager.Stub；
* Binder客户端：ApplicationPackageManager(简称APM)的成员变量mPM继承于IPackageManager.Stub.Proxy; 本身APM是继承于PackageManager对象。

Android系统启动过程中，一路启动到SystemServer后，便可以启动framework的各大服务，本文将介绍PKMS的启动过程。

#### PKMS启动

SystemServer启动过程中涉及到的PKMS如下：

private void startBootstrapServices() {
 //启动installer服务
 Installer installer = mSystemServiceManager.startService(Installer.class);
 ...

 //处于加密状态则仅仅解析核心应用
 String cryptState = SystemProperties.get("vold.decrypt");
 if (ENCRYPTING\_STATE.equals(cryptState)) {
 mOnlyCore = true; // ENCRYPTING\_STATE = "trigger\_restart\_min\_framework"
 } else if (ENCRYPTED\_STATE.equals(cryptState)) {
 mOnlyCore = true; // ENCRYPTED\_STATE = "1"
 }

 //创建PKMS对象【见小节2.1】
 mPackageManagerService = PackageManagerService.main(mSystemContext, installer,
 mFactoryTestMode != FactoryTest.FACTORY\_TEST\_OFF, mOnlyCore);
 //PKMS是否首次启动
 mFirstBoot = mPackageManagerService.isFirstBoot();

 //【见小节3.1】
 mPackageManager = mSystemContext.getPackageManager();
 ...
}

PKMS.main()过程主要是创建PKMS服务，并注册到ServiceManager大管家。

private void startOtherServices() {
 ...
 //启动MountService，后续PackageManager会需要使用
 mSystemServiceManager.startService(MOUNT\_SERVICE\_CLASS);
 //【见小节3.2】
 mPackageManagerService.performBootDexOpt();
 ...

 // phase 500
 mSystemServiceManager.startBootPhase(SystemService.PHASE\_SYSTEM\_SERVICES\_READY);
 ...

 //【见小节3.3】
 mPackageManagerService.systemReady();
 ...
}

整个system\_server进程启动过程，涉及PKMS服务的主要几个动作如下，接下来分别讲解每个过程

* PKMS.main()
* PKMS.performBootDexOpt
* PKMS.systemReady

## 二、 PKMS.main

public static PackageManagerService main(Context context, Installer installer, boolean factoryTest, boolean onlyCore) {
 //初始化PKMS对象
 PackageManagerService m = new PackageManagerService(context, installer,
 factoryTest, onlyCore);
 //将package服务注册到ServiceManager，这是binder服务的常规注册流程
 ServiceManager.addService("package", m);
 return m;
}

该方法的主要功能创建PKMS对象，并将其注册到ServiceManager。 关于PKMS对象的构造方法很长，分为以下几个阶段，每个阶段会输出相应的EventLog： 除了阶段1的开头部分代码，后续代码都是同时持有同步锁mPackages和mInstallLock的过程中执行的。

public PackageManagerService(Context context, Installer installer, boolean factoryTest, boolean onlyCore) {

 阶段1：BOOT\_PROGRESS\_PMS\_START
 ...
 synchronized (mInstallLock) {
 synchronized (mPackages) {
 ...
 阶段2：BOOT\_PROGRESS\_PMS\_SYSTEM\_SCAN\_START
 阶段3：BOOT\_PROGRESS\_PMS\_DATA\_SCAN\_START
 阶段4：BOOT\_PROGRESS\_PMS\_SCAN\_END
 阶段5：BOOT\_PROGRESS\_PMS\_READY
 ...
 }
 }

 Runtime.getRuntime().gc();
 //暴露私有服务，用于系统组件的使用
 LocalServices.addService(PackageManagerInternal.class, new PackageManagerInternalImpl());
}

接下里分别说说这5个阶段：

### 2.1 PMS\_START

阶段1 PMS\_START有两部分组成，由无需加锁的前部分和同时持有两个锁的后半部分，先来说说前半部分：

EventLog.writeEvent(EventLogTags.BOOT\_PROGRESS\_PMS\_START,
 SystemClock.uptimeMillis());
mContext = context;
mFactoryTest = factoryTest;
mOnlyCore = onlyCore; //标记是否只加载核心服务
//对于eng版本则延迟执行dexopt操作
mLazyDexOpt = "eng".equals(SystemProperties.get("ro.build.type"));
mMetrics = new DisplayMetrics();
mSettings = new Settings(mPackages); //创建Settings对象【见小节2.1.1】

// 添加system, phone, log, nfc, bluetooth, shell这六种shareUserId到mSettings；
mSettings.addSharedUserLPw("android.uid.system", Process.SYSTEM\_UID,
 ApplicationInfo.FLAG\_SYSTEM, ApplicationInfo.PRIVATE\_FLAG\_PRIVILEGED);
mSettings.addSharedUserLPw("android.uid.phone", RADIO\_UID,
 ApplicationInfo.FLAG\_SYSTEM, ApplicationInfo.PRIVATE\_FLAG\_PRIVILEGED);
mSettings.addSharedUserLPw("android.uid.log", LOG\_UID,
 ApplicationInfo.FLAG\_SYSTEM, ApplicationInfo.PRIVATE\_FLAG\_PRIVILEGED);
mSettings.addSharedUserLPw("android.uid.nfc", NFC\_UID,
 ApplicationInfo.FLAG\_SYSTEM, ApplicationInfo.PRIVATE\_FLAG\_PRIVILEGED);
mSettings.addSharedUserLPw("android.uid.bluetooth", BLUETOOTH\_UID,
 ApplicationInfo.FLAG\_SYSTEM, ApplicationInfo.PRIVATE\_FLAG\_PRIVILEGED);
mSettings.addSharedUserLPw("android.uid.shell", SHELL\_UID,
 ApplicationInfo.FLAG\_SYSTEM, ApplicationInfo.PRIVATE\_FLAG\_PRIVILEGED);

long dexOptLRUThresholdInMinutes;
if (mLazyDexOpt) {
 dexOptLRUThresholdInMinutes = 30; //对于eng版本，则只会对30分钟之内使用过的app执行dex优化
} else {
 dexOptLRUThresholdInMinutes = 7 \* 24 \* 60; //否则，用户一周内使用过的app执行dex优化
}
mDexOptLRUThresholdInMills = dexOptLRUThresholdInMinutes \* 60 \* 1000;
...

mInstaller = installer; //保存installer对象
mPackageDexOptimizer = new PackageDexOptimizer(this); //用于dex优化
//运行在”android.fg"线程的handler对象
mMoveCallbacks = new MoveCallbacks(FgThread.get().getLooper());

mOnPermissionChangeListeners = new OnPermissionChangeListeners(
 FgThread.get().getLooper());

getDefaultDisplayMetrics(context, mMetrics);

//获取系统配置信息【见小节2.1.2】
SystemConfig systemConfig = SystemConfig.getInstance();
mGlobalGids = systemConfig.getGlobalGids();
mSystemPermissions = systemConfig.getSystemPermissions();
mAvailableFeatures = systemConfig.getAvailableFeatures();

这里有一个参数mDexOptLRUThresholdInMills用于决定执行dex优化操作的时间阈，这个参数用于后续的PKMS.performBootDexOpt()过程。

* 对于Eng版本，则只会对30分钟之内使用过的app执行dex优化；
* 对于非Eng版本，则会将用户最近一周内使用过的app执行dex优化；

接下来，再来看看后半部分：

synchronized (mInstallLock) {
synchronized (mPackages) {
 //创建名为“PackageManager”的handler线程
 mHandlerThread = new ServiceThread(TAG,
 Process.THREAD\_PRIORITY\_BACKGROUND, true /\*allowIo\*/);
 mHandlerThread.start();
 mHandler = new PackageHandler(mHandlerThread.getLooper());
 Watchdog.getInstance().addThread(mHandler, WATCHDOG\_TIMEOUT);

 //创建各种目录
 File dataDir = Environment.getDataDirectory();
 mAppDataDir = new File(dataDir, "data");
 mAppInstallDir = new File(dataDir, "app");
 mAppLib32InstallDir = new File(dataDir, "app-lib");
 mAsecInternalPath = new File(dataDir, "app-asec").getPath();
 mUserAppDataDir = new File(dataDir, "user");
 mDrmAppPrivateInstallDir = new File(dataDir, "app-private");
 //创建用户管理服务
 sUserManager = new UserManagerService(context, this,
 mInstallLock, mPackages);
 ...

 //获取共享库
 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));
 }
 ...
 mRestoredSettings = mSettings.readLPw(this, sUserManager.getUsers(false),
 mSdkVersion, mOnlyCore);
 ...
 }
}

这个过程涉及的几个重要变量：

|  |  |
| --- | --- |
| 变量 | 所对应目录 |
| mAppDataDir | /data/data |
| mAppLib32InstallDir | /data/app-lib |
| mAsecInternalPath | /data/app-asec |
| mUserAppDataDir | /data/user |
| mAppInstallDir | /data/app |
| mDrmAppPrivateInstallDir | /data/app-private |

#### 2.1.1 创建Settings

Settings(Object lock) {
 this(Environment.getDataDirectory(), lock);
}

Settings(File dataDir, Object lock) {
 mLock = lock;

 mRuntimePermissionsPersistence = new RuntimePermissionPersistence(mLock);

 mSystemDir = new File(dataDir, "system");
 mSystemDir.mkdirs(); //创建/data/system
 FileUtils.setPermissions(mSystemDir.toString(),
 FileUtils.S\_IRWXU|FileUtils.S\_IRWXG
 |FileUtils.S\_IROTH|FileUtils.S\_IXOTH,
 -1, -1);
 mSettingsFilename = new File(mSystemDir, "packages.xml");
 mBackupSettingsFilename = new File(mSystemDir, "packages-backup.xml");
 mPackageListFilename = new File(mSystemDir, "packages.list");
 FileUtils.setPermissions(mPackageListFilename, 0640, SYSTEM\_UID, PACKAGE\_INFO\_GID);

 mStoppedPackagesFilename = new File(mSystemDir, "packages-stopped.xml");
 mBackupStoppedPackagesFilename = new File(mSystemDir, "packages-stopped-backup.xml");
}

此处mSystemDir是指目录/data/system，在该目录有以下5个文件：

|  |  |
| --- | --- |
| 文件 | 功能 |
| packages.xml | 记录所有安装app的信息 |
| packages-backup.xml | 备份文件 |
| packages-stopped.xml | 记录系统被强制停止的文件 |
| packages-stopped-backup.xml | 备份文件 |
| packages.list | 记录应用的数据信息 |

#### 2.1.2 SC.getInstance

[-> SystemConfig.java]

public static SystemConfig getInstance() {
 synchronized (SystemConfig.class) {
 if (sInstance == null) {
 sInstance = new SystemConfig();
 }
 return sInstance;
 }
 }

SystemConfig() {
 //读取权限目录【见小节2.1.3】
 readPermissions(Environment.buildPath(
 Environment.getRootDirectory(), "etc", "sysconfig"), false);
 readPermissions(Environment.buildPath(
 Environment.getRootDirectory(), "etc", "permissions"), false);
 readPermissions(Environment.buildPath(
 Environment.getOemDirectory(), "etc", "sysconfig"), true);
 readPermissions(Environment.buildPath(
 Environment.getOemDirectory(), "etc", "permissions"), true);
}

readPermissions()解析指定目录下的所有xml文件，比如将标签<library>所指的动态库保存到 PKMS的成员变量mSharedLibraries。可见，SystemConfig创建过程是对以下这四个目录中的所有xml进行解析:

* /system/etc/sysconfig
* /system/etc/permissions
* /oem/etc/sysconfig
* /oem/etc/permissions

#### 2.1.3 SC.readPermissions

[-> SystemConfig.java]

void readPermissions(File libraryDir, boolean onlyFeatures) {
 ...

 File platformFile = null;
 for (File f : libraryDir.listFiles()) {
 if (f.getPath().endsWith("etc/permissions/platform.xml")) {
 platformFile = f; //platform.xml文件
 continue;
 }

 if (!f.getPath().endsWith(".xml")) {
 continue;
 }

 if (!f.canRead()) {
 continue;
 }
 readPermissionsFromXml(f, onlyFeatures);
 }

 if (platformFile != null) {
 readPermissionsFromXml(platformFile, onlyFeatures);
 }
 }

该方法是解析指定目录下所有的具有可读权限的，且以xml后缀文件。

### 2.2 PMS\_SYSTEM\_SCAN\_START

long startTime = SystemClock.uptimeMillis();
EventLog.writeEvent(EventLogTags.BOOT\_PROGRESS\_PMS\_SYSTEM\_SCAN\_START,
 startTime);

final int scanFlags = SCAN\_NO\_PATHS | SCAN\_DEFER\_DEX | SCAN\_BOOTING | SCAN\_INITIAL;
//该集合中存放的是已经优化或者不需要优先的文件
final ArraySet<String> alreadyDexOpted = new ArraySet<String>();

final String bootClassPath = System.getenv("BOOTCLASSPATH");
final String systemServerClassPath = System.getenv("SYSTEMSERVERCLASSPATH");

//将环境变量BOOTCLASSPATH所执行的文件加入alreadyDexOpted
if (bootClassPath != null) {
 String[] bootClassPathElements = splitString(bootClassPath, ':');
 for (String element : bootClassPathElements) {
 alreadyDexOpted.add(element);
 }
}

//将环境变量SYSTEMSERVERCLASSPATH所执行的文件加入alreadyDexOpted
if (systemServerClassPath != null) {
 String[] systemServerClassPathElements = splitString(systemServerClassPath, ':');
 for (String element : systemServerClassPathElements) {
 alreadyDexOpted.add(element);
 }
}
...

//此处共享库是由SystemConfig实例化过程赋值的
if (mSharedLibraries.size() > 0) {
 for (String dexCodeInstructionSet : dexCodeInstructionSets) {
 for (SharedLibraryEntry libEntry : mSharedLibraries.values()) {
 final String lib = libEntry.path;
 ...
 int dexoptNeeded = DexFile.getDexOptNeeded(lib, dexCodeInstructionSet,
 "speed", false);
 if (dexoptNeeded != DexFile.NO\_DEXOPT\_NEEDED) {
 alreadyDexOpted.add(lib);
 //执行dexopt操作【见小节2.2.1】
 mInstaller.dexopt(lib, Process.SYSTEM\_UID, dexCodeInstructionSet,
 dexoptNeeded, DEXOPT\_PUBLIC /\*dexFlags\*/);
 }
 }
 }
}

//此处frameworkDir目录为/system/framework
File frameworkDir = new File(Environment.getRootDirectory(), "framework");

//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");

String[] frameworkFiles = frameworkDir.list();
if (frameworkFiles != null) {
 for (String dexCodeInstructionSet : dexCodeInstructionSets) {
 for (int i=0; i<frameworkFiles.length; i++) {
 File libPath = new File(frameworkDir, frameworkFiles[i]);
 String path = libPath.getPath();
 //跳过已优化集合中的文件
 if (alreadyDexOpted.contains(path)) {
 continue;
 }
 //跳过后缀不为apk和jar的文件
 if (!path.endsWith(".apk") && !path.endsWith(".jar")) {
 continue;
 }

 int dexoptNeeded = DexFile.getDexOptNeeded(path, dexCodeInstructionSet,
 "speed", false);
 if (dexoptNeeded != DexFile.NO\_DEXOPT\_NEEDED) {
 //执行dexopt操作【见小节2.2.1】
 mInstaller.dexopt(path, Process.SYSTEM\_UID, dexCodeInstructionSet,
 dexoptNeeded, DEXOPT\_PUBLIC /\*dexFlags\*/);
 }

 }
 }
}

final VersionInfo ver = mSettings.getInternalVersion();
mIsUpgrade = !Build.FINGERPRINT.equals(ver.fingerprint);
mPromoteSystemApps = mIsUpgrade && ver.sdkVersion <= Build.VERSION\_CODES.LOLLIPOP\_MR1;

if (mPromoteSystemApps) {
 Iterator<PackageSetting> pkgSettingIter = mSettings.mPackages.values().iterator();
 while (pkgSettingIter.hasNext()) {
 PackageSetting ps = pkgSettingIter.next();
 if (isSystemApp(ps)) {
 mExistingSystemPackages.add(ps.name);
 }
 }
}

//收集供应商包名：/vendor/overlay
File vendorOverlayDir = new File(VENDOR\_OVERLAY\_DIR);
scanDirLI(vendorOverlayDir, PackageParser.PARSE\_IS\_SYSTEM
 | PackageParser.PARSE\_IS\_SYSTEM\_DIR, scanFlags | SCAN\_TRUSTED\_OVERLAY, 0);

//收集包名：/system/framework
scanDirLI(frameworkDir, PackageParser.PARSE\_IS\_SYSTEM
 | PackageParser.PARSE\_IS\_SYSTEM\_DIR
 | PackageParser.PARSE\_IS\_PRIVILEGED,
 scanFlags | SCAN\_NO\_DEX, 0);

//收集私有的系统包名：/system/priv-app
final File privilegedAppDir = new File(Environment.getRootDirectory(), "priv-app");
scanDirLI(privilegedAppDir, PackageParser.PARSE\_IS\_SYSTEM
 | PackageParser.PARSE\_IS\_SYSTEM\_DIR
 | PackageParser.PARSE\_IS\_PRIVILEGED, scanFlags, 0);

//收集一般地系统包名：/system/app
final File systemAppDir = new File(Environment.getRootDirectory(), "app");
scanDirLI(systemAppDir, PackageParser.PARSE\_IS\_SYSTEM
 | PackageParser.PARSE\_IS\_SYSTEM\_DIR, scanFlags, 0);

//收集私有供应商包名：/vendor/priv-app
final File privilegedVendorAppDir = new File(Environment.getVendorDirectory(), "priv-app");
scanDirLI(privilegedVendorAppDir, PackageParser.PARSE\_IS\_SYSTEM
 | PackageParser.PARSE\_IS\_SYSTEM\_DIR
 | PackageParser.PARSE\_IS\_PRIVILEGED, scanFlags, 0);

//收集所有的供应商包名：/vendor/app
File vendorAppDir = new File(Environment.getVendorDirectory(), "app");
vendorAppDir = vendorAppDir.getCanonicalFile();
scanDirLI(vendorAppDir, PackageParser.PARSE\_IS\_SYSTEM
 | PackageParser.PARSE\_IS\_SYSTEM\_DIR, scanFlags, 0);

//收集所有OEM包名：/oem/app
final File oemAppDir = new File(Environment.getOemDirectory(), "app");
scanDirLI(oemAppDir, PackageParser.PARSE\_IS\_SYSTEM
 | PackageParser.PARSE\_IS\_SYSTEM\_DIR, scanFlags, 0);

//移除文件
mInstaller.moveFiles();

//删除不在存在的系统包
final List<String> possiblyDeletedUpdatedSystemApps = new ArrayList<String>();
if (!mOnlyCore) {
 Iterator<PackageSetting> psit = mSettings.mPackages.values().iterator();
 while (psit.hasNext()) {
 PackageSetting ps = psit.next();

 if ((ps.pkgFlags & ApplicationInfo.FLAG\_SYSTEM) == 0) {
 continue;
 }

 final PackageParser.Package scannedPkg = mPackages.get(ps.name);
 if (scannedPkg != null) {
 if (mSettings.isDisabledSystemPackageLPr(ps.name)) {
 removePackageLI(ps, true);
 mExpectingBetter.put(ps.name, ps.codePath);
 }
 continue;
 }

 if (!mSettings.isDisabledSystemPackageLPr(ps.name)) {
 psit.remove();
 removeDataDirsLI(null, ps.name);
 } else {
 final PackageSetting disabledPs = mSettings.getDisabledSystemPkgLPr(ps.name);
 if (disabledPs.codePath == null || !disabledPs.codePath.exists()) {
 possiblyDeletedUpdatedSystemApps.add(ps.name);
 }
 }
 }
}

//清理所有安装不完整的包
ArrayList<PackageSetting> deletePkgsList = mSettings.getListOfIncompleteInstallPackagesLPr();
for(int i = 0; i < deletePkgsList.size(); i++) {
 cleanupInstallFailedPackage(deletePkgsList.get(i));
}
//删除临时文件
deleteTempPackageFiles();

//移除不相干包中的所有共享userID
mSettings.pruneSharedUsersLPw();

**环境变量:** 那可通过adb shell env来查看系统所有的环境变量及相应值。也可通过命令**adb shell echo $SYSTEMSERVERCLASSPATH**。

* 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。

**dexopt():** 执行dex优化操作的文件有以下几类

* mSharedLibraries：该共享库下的所有文件，是由SystemConfig构造函数中赋值的；
* /system/framework：该目录的所有apk和jar文件，去除位于alreadyDexOpted中的文件。 具体有哪些文件不包括呢？比如services.jar, framework.jar, framework-res.apk, core-libart.jar.

**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。

1. /vendor/overlay
2. /system/framework
3. /system/priv-app
4. /system/app
5. /vendor/priv-app
6. /vendor/app
7. /oem/app

#### 2.2.1 dexopt

[-> Installer.java]

public int dexopt(String apkPath, int uid, String instructionSet, int dexoptNeeded, int dexFlags) {
 if (!isValidInstructionSet(instructionSet)) {
 return -1;
 }
 //[见小节2.2.2]
 return mInstaller.dexopt(apkPath, uid, instructionSet, dexoptNeeded, dexFlags);
}

#### 2.2.2 IC.dexopt

[-> InstallerConnection.java]

public int dexopt(String apkPath, int uid, String instructionSet, int dexoptNeeded, int dexFlags) {
 return dexopt(apkPath, uid, "\*", instructionSet, dexoptNeeded,
 null, dexFlags);
}

public int dexopt(String apkPath, int uid, String pkgName, String instructionSet, int dexoptNeeded, String outputPath, int dexFlags) {
 StringBuilder builder = new StringBuilder("dexopt");
 builder.append(' ');
 builder.append(apkPath);
 builder.append(' ');
 builder.append(uid);
 builder.append(' ');
 builder.append(pkgName);
 builder.append(' ');
 builder.append(instructionSet);
 builder.append(' ');
 builder.append(dexoptNeeded);
 builder.append(' ');
 builder.append(outputPath != null ? outputPath : "!");
 builder.append(' ');
 builder.append(dexFlags);
 return execute(builder.toString()); }

通过socket发送给installd守护进程来执行相应的dexopt操作。

### 2.3 PMS\_DATA\_SCAN\_START

if (!mOnlyCore) { //处理非系统app
 EventLog.writeEvent(EventLogTags.BOOT\_PROGRESS\_PMS\_DATA\_SCAN\_START, SystemClock.uptimeMillis());
 //收集/data/app包名
 scanDirLI(mAppInstallDir, 0, scanFlags | SCAN\_REQUIRE\_KNOWN, 0);
 //收集/data/app-private包名
 scanDirLI(mDrmAppPrivateInstallDir, PackageParser.PARSE\_FORWARD\_LOCK,
 scanFlags | SCAN\_REQUIRE\_KNOWN, 0);

 for (String deletedAppName : possiblyDeletedUpdatedSystemApps) {
 PackageParser.Package deletedPkg = mPackages.get(deletedAppName);
 mSettings.removeDisabledSystemPackageLPw(deletedAppName);

 String msg;
 if (deletedPkg == null) {
 removeDataDirsLI(null, deletedAppName);
 } else {
 deletedPkg.applicationInfo.flags &= ~ApplicationInfo.FLAG\_SYSTEM;

 PackageSetting deletedPs = mSettings.mPackages.get(deletedAppName);
 deletedPs.pkgFlags &= ~ApplicationInfo.FLAG\_SYSTEM;
 }
 }

 for (int i = 0; i < mExpectingBetter.size(); i++) {
 final String packageName = mExpectingBetter.keyAt(i);
 if (!mPackages.containsKey(packageName)) {
 final File scanFile = mExpectingBetter.valueAt(i);

 final int reparseFlags;
 if (FileUtils.contains(privilegedAppDir, scanFile)) {
 reparseFlags = PackageParser.PARSE\_IS\_SYSTEM
 | PackageParser.PARSE\_IS\_SYSTEM\_DIR
 | PackageParser.PARSE\_IS\_PRIVILEGED;
 } else if (FileUtils.contains(systemAppDir, scanFile)) {
 reparseFlags = PackageParser.PARSE\_IS\_SYSTEM
 | PackageParser.PARSE\_IS\_SYSTEM\_DIR;
 } else if (FileUtils.contains(vendorAppDir, scanFile)) {
 reparseFlags = PackageParser.PARSE\_IS\_SYSTEM
 | PackageParser.PARSE\_IS\_SYSTEM\_DIR;
 } else if (FileUtils.contains(oemAppDir, scanFile)) {
 reparseFlags = PackageParser.PARSE\_IS\_SYSTEM
 | PackageParser.PARSE\_IS\_SYSTEM\_DIR;
 } else {
 continue;
 }

 mSettings.enableSystemPackageLPw(packageName);

 try {
 // 扫描包名
 scanPackageLI(scanFile, reparseFlags, scanFlags, 0, null);
 } catch (PackageManagerException e) {
 ...
 }
 }
 }
}
mExpectingBetter.clear();

updateAllSharedLibrariesLPw();

for (SharedUserSetting setting : mSettings.getAllSharedUsersLPw()) {
 adjustCpuAbisForSharedUserLPw(setting.packages, null /\* scanned package \*/,
 false /\* force dexopt \*/, false /\* defer dexopt \*/,
 false /\* boot complete \*/);
}

mPackageUsage.readLP();

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

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

### 2.4 PMS\_SCAN\_END

EventLog.writeEvent(EventLogTags.BOOT\_PROGRESS\_PMS\_SCAN\_END,
 SystemClock.uptimeMillis());

int updateFlags = UPDATE\_PERMISSIONS\_ALL;
if (ver.sdkVersion != mSdkVersion) {
 updateFlags |= UPDATE\_PERMISSIONS\_REPLACE\_PKG | UPDATE\_PERMISSIONS\_REPLACE\_ALL;
}
//当sdk版本不一致时，需要更新权限
updatePermissionsLPw(null, null, StorageManager.UUID\_PRIVATE\_INTERNAL, updateFlags);
ver.sdkVersion = mSdkVersion;

if (!onlyCore && (mPromoteSystemApps || !mRestoredSettings)) {
 for (UserInfo user : sUserManager.getUsers(true)) {
 mSettings.applyDefaultPreferredAppsLPw(this, user.id);
 applyFactoryDefaultBrowserLPw(user.id);
 primeDomainVerificationsLPw(user.id);
 }
}

//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
if (mIsUpgrade && !onlyCore) {
 for (int i = 0; i < mSettings.mPackages.size(); i++) {
 final PackageSetting ps = mSettings.mPackages.valueAt(i);
 if (Objects.equals(StorageManager.UUID\_PRIVATE\_INTERNAL, ps.volumeUuid)) {
 deleteCodeCacheDirsLI(ps.volumeUuid, ps.name);
 }
 }
 ver.fingerprint = Build.FINGERPRINT;
}

checkDefaultBrowser();
//当权限和其他默认项都完成更新，则清理相关信息
mExistingSystemPackages.clear();
mPromoteSystemApps = false;

ver.databaseVersion = Settings.CURRENT\_DATABASE\_VERSION;
//信息写回packages.xml文件
mSettings.writeLPr();

### 2.5 PMS\_READY

EventLog.writeEvent(EventLogTags.BOOT\_PROGRESS\_PMS\_READY,
 SystemClock.uptimeMillis());

mRequiredVerifierPackage = getRequiredVerifierLPr();
mRequiredInstallerPackage = getRequiredInstallerLPr();
//【见小节2.5.1】
mInstallerService = new PackageInstallerService(context, this);

mIntentFilterVerifierComponent = getIntentFilterVerifierComponentNameLPr();
mIntentFilterVerifier = new IntentVerifierProxy(mContext,
 mIntentFilterVerifierComponent);

PKMS初始化完成阶段，还会创建一个PackageInstaller服务。

#### 2.5.1 创建PKIS服务

[-> PackageInstallerService]

public PackageInstallerService(Context context, PackageManagerService pm) {
 mContext = context;
 mPm = pm;
 //创建名为”PackageInstaller“的Handler线程
 mInstallThread = new HandlerThread(TAG);
 mInstallThread.start();

 mInstallHandler = new Handler(mInstallThread.getLooper());

 mCallbacks = new Callbacks(mInstallThread.getLooper());

 mSessionsFile = new AtomicFile(
 new File(Environment.getSystemSecureDirectory(), "install\_sessions.xml"));
 mSessionsDir = new File(Environment.getSystemSecureDirectory(), "install\_sessions");
 mSessionsDir.mkdirs();

 synchronized (mSessions) {
 readSessionsLocked();

 reconcileStagesLocked(StorageManager.UUID\_PRIVATE\_INTERNAL);

 final ArraySet<File> unclaimedIcons = newArraySet(
 mSessionsDir.listFiles());

 for (int i = 0; i < mSessions.size(); i++) {
 final PackageInstallerSession session = mSessions.valueAt(i);
 unclaimedIcons.remove(buildAppIconFile(session.sessionId));
 }

 for (File icon : unclaimedIcons) {
 icon.delete();
 }
 }
}

### 小节

PKMS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；
1. 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;
1. 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;
1. PMS\_SCAN\_END阶段：
* 将上述信息写回/data/system/packages.xml;
1. PMS\_READY阶段：
* 创建服务PackageInstallerService；

## 三、操作PKMS

### 3.1 getPackageManager

[-> ContextImpl.java]

public PackageManager getPackageManager() {
 if (mPackageManager != null) {
 return mPackageManager;
 }

 //【见小节3.1.1】
 IPackageManager pm = ActivityThread.getPackageManager();
 if (pm != null) {
 //创建ApplicationPackageManager对象
 return (mPackageManager = new ApplicationPackageManager(this, pm));
 }

 return null;
}

获取PKMS服务，并创建ApplicationPackageManager对象

#### 3.1.1 AT.getPackageManager

[-> ActivityThread.java]

public static IPackageManager getPackageManager() {
 if (sPackageManager != null) {
 return sPackageManager;
 }
 IBinder b = ServiceManager.getService("package");
 sPackageManager = IPackageManager.Stub.asInterface(b);
 return sPackageManager;
}

### 3.2 PKMS.performBootDexOpt

[-> PackageManagerService.java]

public void performBootDexOpt() {
 // 确保只有system或者root uid有权限执行该方法
 enforceSystemOrRoot("Only the system can request dexopt be performed");

 //运行在同一个进程,此处拿到的MountService的服务端
 IMountService ms = PackageHelper.getMountService();
 if (ms != null) {
 final boolean isUpgrade = isUpgrade(); //处于更新状态，则执行fstrim
 boolean doTrim = isUpgrade;
 if (doTrim) {
 Slog.w(TAG, "Running disk maintenance immediately due to system update");
 } else {
 //interval默认值为3天
 final long interval = android.provider.Settings.Global.getLong(
 mContext.getContentResolver(),
 android.provider.Settings.Global.FSTRIM\_MANDATORY\_INTERVAL,
 DEFAULT\_MANDATORY\_FSTRIM\_INTERVAL);
 if (interval > 0) {
 final long timeSinceLast = System.currentTimeMillis() - ms.lastMaintenance();
 if (timeSinceLast > interval) {
 doTrim = true; //距离上次fstrim时间超过3天，则执行fstrim
 }
 }
 }
 //此处ms是指MountService，该过程发送消息H\_FSTRIM给handler，然后再向vold发送fstrim命令
 if (doTrim) {
 ms.runMaintenance();
 }
 }

 final ArraySet<PackageParser.Package> pkgs;
 synchronized (mPackages) {
 //清空延迟执行dexopt操作的app,获取dexopt操作的app集合
 pkgs = mPackageDexOptimizer.clearDeferredDexOptPackages();
 }

 if (pkgs != null) {
 ArrayList<PackageParser.Package> sortedPkgs = new ArrayList<PackageParser.Package>();

 for (Iterator<PackageParser.Package> it = pkgs.iterator(); it.hasNext();) {
 PackageParser.Package pkg = it.next();
 //将pkgs中的核心app添加到sortedPkgs
 if (pkg.coreApp) {
 sortedPkgs.add(pkg);
 it.remove();
 }
 }

 //获取监听PRE\_BOOT\_COMPLETE的系统app集合
 Intent intent = new Intent(Intent.ACTION\_PRE\_BOOT\_COMPLETED);
 ArraySet<String> pkgNames = getPackageNamesForIntent(intent);

 for (Iterator<PackageParser.Package> it = pkgs.iterator(); it.hasNext();) {
 PackageParser.Package pkg = it.next();
 //将pkg中监听PRE\_BOOT\_COMPLETE的app添加到sortedPkgs
 if (pkgNames.contains(pkg.packageName)) {
 sortedPkgs.add(pkg);
 it.remove();
 }
 }

 //获取pkgs中最近一周使用过的app[见小节3.2.1]
 filterRecentlyUsedApps(pkgs);

 //将最近一周的app添加到sortedPkgs
 for (PackageParser.Package pkg : pkgs) {
 sortedPkgs.add(pkg);
 }

 if (mLazyDexOpt) {
 filterRecentlyUsedApps(sortedPkgs);
 }

 int i = 0;
 int total = sortedPkgs.size();
 File dataDir = Environment.getDataDirectory();
 long lowThreshold = StorageManager.from(mContext).getStorageLowBytes(dataDir);
 ...

 for (PackageParser.Package pkg : sortedPkgs) {
 long usableSpace = dataDir.getUsableSpace();
 if (usableSpace < lowThreshold) {
 break;
 }
 //[见小节3.2.2]
 performBootDexOpt(pkg, ++i, total);
 }
 } }

该方法主要功能：

* 当处于升级或者3天未执行fstrim，则本次会否执行fstrim操作；
* 对sortedPkgs中的app执行dexopt优化，其中包含：
* mDeferredDexOpt中的核心app；
* mDeferredDexOpt中监听PRE\_BOOT\_COMPLETE的app；
* mDeferredDexOpt中最近一周使用过的app;

#### 3.2.1 PKMS.filterRecentlyUsedApps

private void filterRecentlyUsedApps(Collection<PackageParser.Package> pkgs) {

 if (mLazyDexOpt || (!isFirstBoot() && mPackageUsage.isHistoricalPackageUsageAvailable())) {
 int total = pkgs.size();
 int skipped = 0;
 long now = System.currentTimeMillis();
 for (Iterator<PackageParser.Package> i = pkgs.iterator(); i.hasNext();) {
 PackageParser.Package pkg = i.next();
 // 过滤出最近使用过的app
 long then = pkg.mLastPackageUsageTimeInMills;
 if (then + mDexOptLRUThresholdInMills < now) {
 i.remove();
 skipped++;
 }
 }
 }
 }

获取最近使用的app,其中mDexOptLRUThresholdInMills：

* 对于Eng版本，则只会对30分钟之内使用过的app执行dex优化；
* 对于用户版本，则会将用户最近一周内使用过的app执行dex优化；

#### 3.2.2 PKMS.performBootDexOpt

 private void performBootDexOpt(PackageParser.Package pkg, int curr, int total) {
 if (!isFirstBoot()) {
 ActivityManagerNative.getDefault().showBootMessage(
 mContext.getResources().getString(R.string.android\_upgrading\_apk,
 curr, total), true);
 }
 PackageParser.Package p = pkg;
 synchronized (mInstallLock) {
 //[见小节3.2.3]
 mPackageDexOptimizer.performDexOpt(p, null /\* instruction sets \*/,
 false /\* force dex \*/, false /\* defer \*/, true /\* include dependencies \*/,
 false /\* boot complete \*/, false /\*useJit\*/);
 }
}

#### 3.2.3 performDexOpt

[-> PackageDexOptimizer.java]

int performDexOpt(PackageParser.Package pkg, String[] instructionSets, boolean forceDex, boolean defer, boolean inclDependencies, boolean bootComplete, boolean useJit) {
 ArraySet<String> done;
 if (inclDependencies && (pkg.usesLibraries != null || pkg.usesOptionalLibraries != null)) {
 done = new ArraySet<String>();
 done.add(pkg.packageName);
 } else {
 done = null;
 }
 synchronized (mPackageManagerService.mInstallLock) {
 final boolean useLock = mSystemReady;
 if (useLock) {
 mDexoptWakeLock.setWorkSource(new WorkSource(pkg.applicationInfo.uid));
 mDexoptWakeLock.acquire();
 }
 try {
 // 最终还是调用[小节2.2.1]的操作
 return performDexOptLI(pkg, instructionSets, forceDex, defer, bootComplete,
 useJit, done);
 } finally {
 if (useLock) {
 mDexoptWakeLock.release();
 }
 }
 }
 }

这个过程最终还是调用[小节2.2.1]的dexopt操作.

### 3.3 PKMS.systemReady

public void systemReady() {
 mSystemReady = true;
 ...

 synchronized (mPackages) {
 ArrayList<PreferredActivity> removed = new ArrayList<PreferredActivity>();
 for (int i=0; i<mSettings.mPreferredActivities.size(); i++) {
 PreferredIntentResolver pir = mSettings.mPreferredActivities.valueAt(i);
 removed.clear();
 for (PreferredActivity pa : pir.filterSet()) {
 if (mActivities.mActivities.get(pa.mPref.mComponent) == null) {
 removed.add(pa);
 }
 }
 if (removed.size() > 0) {
 for (int r=0; r<removed.size(); r++) {
 PreferredActivity pa = removed.get(r);
 pir.removeFilter(pa);
 }
 mSettings.writePackageRestrictionsLPr(
 mSettings.mPreferredActivities.keyAt(i));
 }
 }

 for (int userId : UserManagerService.getInstance().getUserIds()) {
 if (!mSettings.areDefaultRuntimePermissionsGrantedLPr(userId)) {
 grantPermissionsUserIds = ArrayUtils.appendInt(
 grantPermissionsUserIds, userId);
 }
 }
 }

 sUserManager.systemReady(); //多用户服务

 //升级所有已获取的默认权限
 for (int userId : grantPermissionsUserIds) {
 mDefaultPermissionPolicy.grantDefaultPermissions(userId);
 }

 //处理所有等待系统准备就绪的消息
 if (mPostSystemReadyMessages != null) {
 for (Message msg : mPostSystemReadyMessages) {
 msg.sendToTarget();
 }
 mPostSystemReadyMessages = null;
 }

 //观察外部存储设备
 final StorageManager storage = mContext.getSystemService(StorageManager.class);
 storage.registerListener(mStorageListener);

 mInstallerService.systemReady();
 mPackageDexOptimizer.systemReady();

 MountServiceInternal mountServiceInternal = LocalServices.getService(MountServiceInternal.class);
 mountServiceInternal.addExternalStoragePolicy(...);
}

## 四. 总结

这个过程会有“PackageManager”线程和“andorid.fg”线程

### 4.1 核心文件

|  |  |
| --- | --- |
| 文件 | 功能 |
| /data/data/ | App数据目录 |
| /data/user/ | App数据目录 |
| /data/app/ | App安装目录 |
| /data/system/packages.xml | 所有安装app信息 |
| /data/system/packages-stopped.xml | 所有强制停止app信息 |
| /data/system/packages.list | 所有安装app信息 |

Android系统有很多目录可以存放app，如下所示：

|  |  |
| --- | --- |
| 目录 | App类别 |
| /vendor/overlay | 系统App |
| /system/framework | 系统App |
| /system/priv-app | 系统App |
| /system/app | 系统App |
| /vendor/priv-app | 系统App |
| /vendor/app | 系统App |
| /oem/app | 系统App |
| /data/app | 普通App |
| /data/app-private | 普通App |

### 4.2 dexopt

startBootstrapServices()执行到创建PKMS的过程中会对以下目录进行dexopt操作：

* mSharedLibraries：该共享库下的所有文件是由SystemConfig构造函数中，以下4个目录下的所有xml文件中的标签

<library>

所指的动态库。

* /system/etc/sysconfig
* /system/etc/permissions
* /oem/etc/sysconfig
* /oem/etc/permissions
* /system/framework：该目录的所有apk和jar文件，去除位于alreadyDexOpted中的文件，其中alreadyDexOpted：
* SYSTEMSERVERCLASSPATH环境变量：/system/framework目录下services.jar，ethernet-service.jar，wifi-service.jar这3个文件
* BOOTCLASSPATH环境变量：/system/framework目录下的framework.jar，ext.jar，core-libart.jar等等
* /system/framework/framework-res.apk
* /system/framework/core-libart.jar

startOtherServices()执行到PKMS.performBootDexOpt过程，也是进行dexopt操作：

* mDeferredDexOpt中的核心app；
* mDeferredDexOpt中监听PRE\_BOOT\_COMPLETE的app；
* mDeferredDexOpt中最近一周使用过的app;

最后，dexopt操作通过socket发送给守护进程installd来完成，下一篇文章介绍installd的功能。