PKMS Android10.0 源码解读（目前安卓最新源码解读）

PackageManagerService 简称 PKMS

PKMS 是什么东西?

PackageManagerService（简称PMS）
安装
卸载
应用信息查询

答: PackageManagerService（简称 PKMS），是 Android 系统中核心服务之一，负责应用程序的**安装**，**卸载**，**信息查询**，等工作。

PKMS 概述信息:

Android系统启动时，会启动（应用程序管理服务器PKMS），此服务负责扫描系统中特定的目录，寻找里面的APK格式的文件，并对这些文件进行解析，然后得到应用程序相关信息，最后完成应用程序的安装

PKMS在安装应用过程中, 会全面解析应用程序的AndroidManifest.xml文件, 来得到Activity, Service, BroadcastReceiver, ContextProvider 等信息, 在结合PKMS服务就可以在OS中正常的使用应用程序了

在Android系统中, 系统启动时由SystemServer启动PKMS服务, 启动该服务后会执行应用程序的安装过程,

接下来后就会重点的介绍 (SystemServer启动PKMS服务的过程, 讲解在Android系统中安装应用程序的过程)

简单来需知：PKMS 与 AMS 一样，也是Android系统核心服务之一，非常非常的重要，主要完成以下核心功能：

1.解析AndroidNanifest.xml清单文件，解析清单文件中的所有节点信息

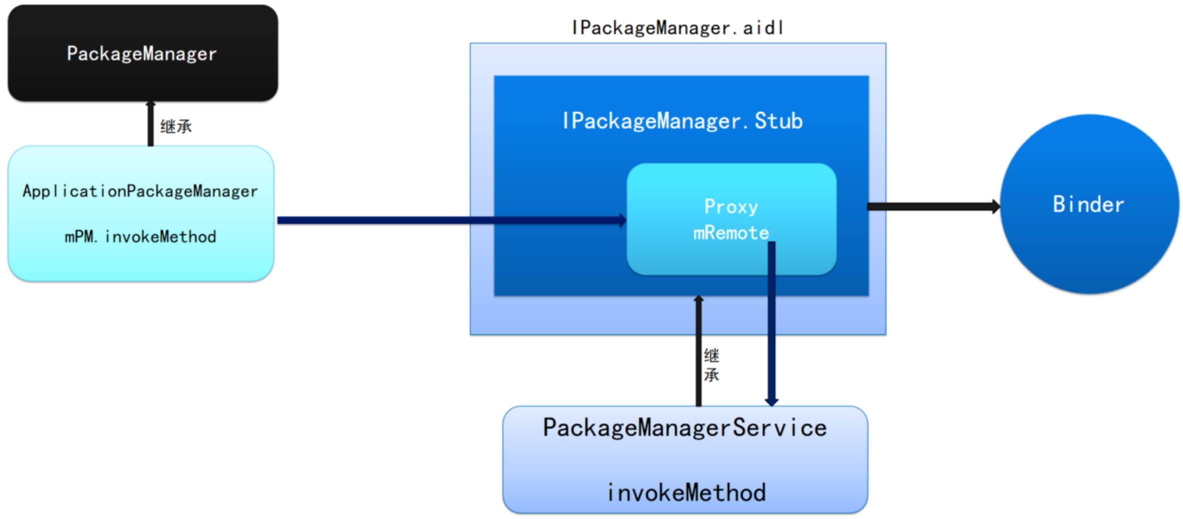
2.扫描.apk文件，安装系统应用，安装本地应用等

3.管理本地应用，主要有， 安装，卸载，应用信息查询 等

同学们 我们分析的核心源码路径地址如下：

```
/frameworks/base/core/java/android/app/ApplicationPackageManager.java
/frameworks/base/services/java/com/android/server/SystemServer.java
/frameworks/base/services/core/java/com/android/server/pm/PackageManagerService.java
/frameworks/base/services/core/java/com/android/server/pm/PackageDexOptimizer.java
/frameworks/base/services/core/java/com/android/server/pm/Installer.java
/frameworks/base/services/core/java/com/android/server/pm/Settings.java
/frameworks/base/services/core/java/com/android/server/pm/permission/BasePermission.java
/frameworks/base/services/core/java/com/android/server/pm/PackageManagerService.java
/frameworks/base/services/core/java/com/android/server/pm/permission/DefaultPermissionGrantPolicy.java
/frameworks/base/services/core/java/com/android/server/pm/permission/PermissionManagerService.java
/frameworks/base/core/java/android/content/pm/IPackageManager.aidl
/frameworks/base/core/java/android/content/pm/PackageManager.java
/frameworks/base/core/java/com/android/server/SystemConfig.java
```

一部曲 - PKMS角色位置：



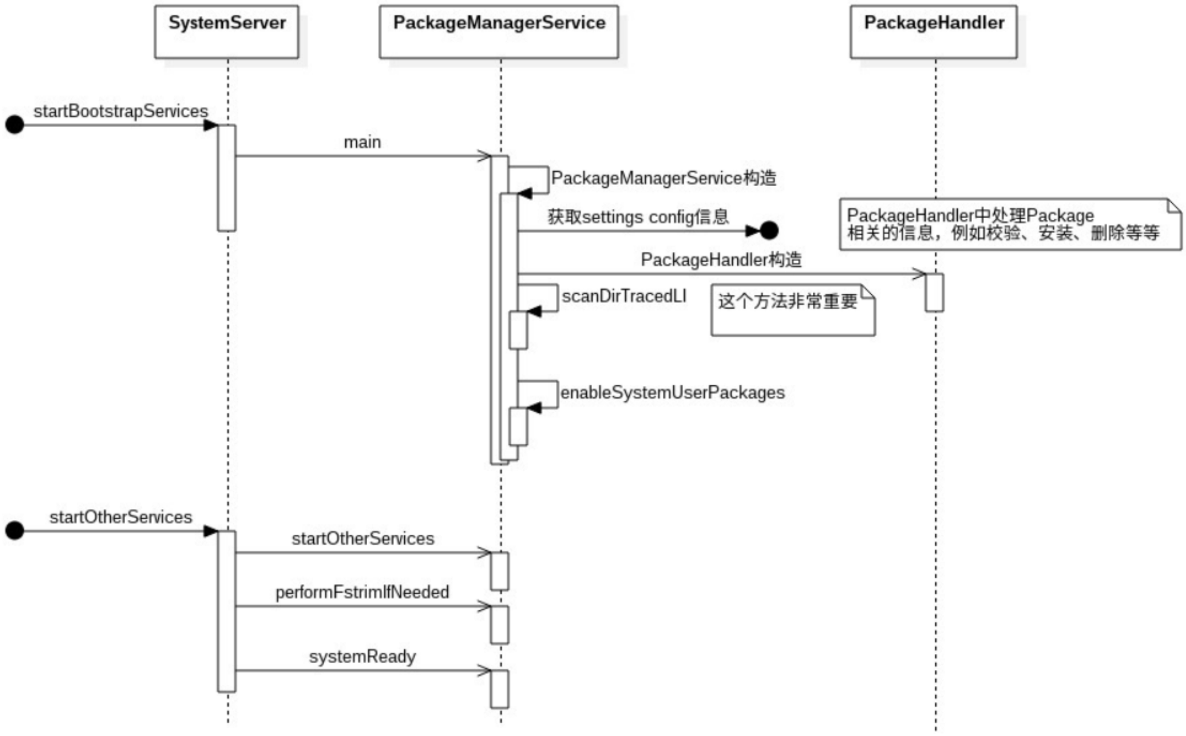
同学们注意：客户端可通过Context.getPackageManager()获得ApplicationPackageManager对象, 而mPM指向的是Proxy代理，当调用到mPM.方法后，将会调用到IPackageManager的Proxy代理方法，然后通过Binder机制中的mRemote与服务端PackageManagerService通信 并调用到PackageManagerService的方法；

自我总结：PKMS是属于Binder机制的服务端角色

接下来，我们就自己来手写一个**简单的PKMS**，同学们好不好

二步曲 - PKMS 启动过程分析：

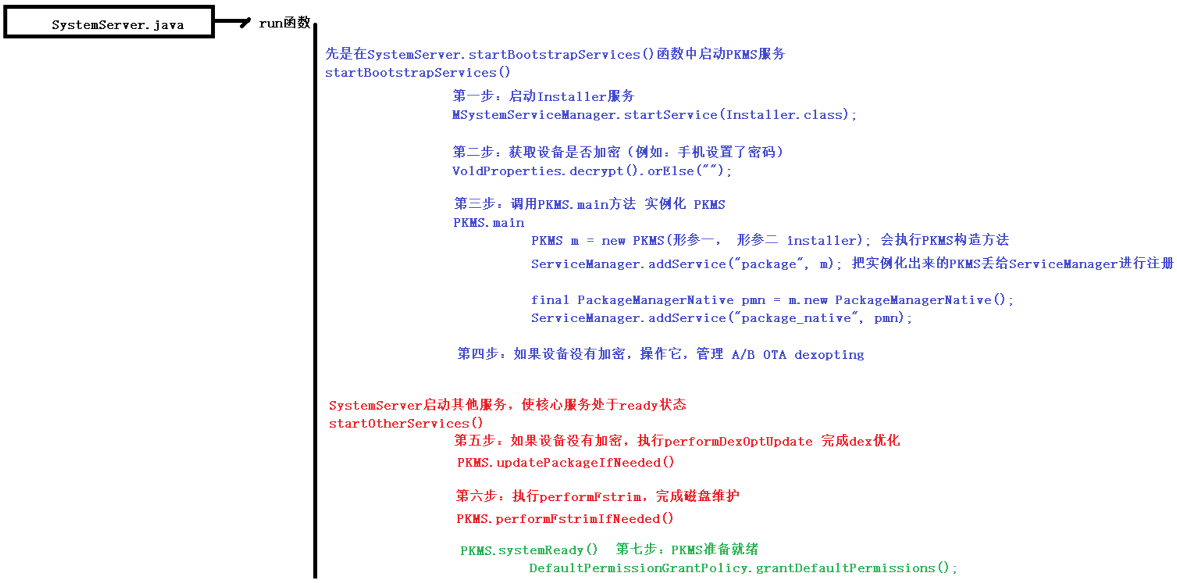
PKMS的过程图如下：



PKMS启动过程描述：

SystemServer启动PKMS： 先是在SystemServer.startBootstrapServices()函数中启动PKMS服务，再调用startOtherServices()函数中对dex优化，磁盘管理功能，让PKMS进入systemReady状态。

七步走，文字不如画图：



```
SystemServer.java  run函数

    先是在SystemServer.startBootstrapServices()函数中启动PKMS服务
    startBootstrapServices()
                第一步：启动Installer服务
                MSystemServiceManager.startService(Installer.class);

                第二步：获取设备是否加密（例如：手机设置了密码）
                VoldProperties.decrypt().orElse("");

                第三步：调用PKMS.main方法 实例化 PKMS
                PKMS.main
                        PKMS m = new PKMS(形参一， 形参二 installer); 会执行PKMS构造方法
                        ServiceManager.addService("package", m); 把实例化出来的PKMS丢给ServiceManager进行注册

                        final PackageManagerNative pmn = m.new PackageManagerNative();
                        ServiceManager.addService("package_native", pmn);

                 第四步：如果设备没有加密，操作它，管理 A/B OTA dexopting


    SystemServer启动其他服务，使核心服务处于ready状态
    startOtherServices()
                第五步：如果设备没有加密，执行performDexOptUpdate 完成dex优化
                 PKMS.updatePackageIfNeeded()

                第六步：执行performFstrim，完成磁盘维护
                PKMS.performFstrimIfNeeded()

                 PKMS.systemReady()  第七步：PKMS准备就绪
                        DefaultPermissionGrantPolicy.grantDefaultPermissions();
```

第一步 到 第四步：

startBootstrapServices()首先启动Installer服务，也就是安装器，随后判断当前的设备是否处于加密状态，如果是则只是解析核心应用，接着调用PackageManagerService的静态方法main来创建pms对象

第一步： 启动Installer服务

第二步：获取设备是否加密(手机设置密码)，如果设备加密了，则只解析"core"应用

第三步： 调用PKMS main方法初始化PackageManagerService，其中调用PackageManagerService()构造函数创建了PKMS对象

第四步： 如果设备没有加密，操作它。管理A/B OTA dexopting。

```
private void startBootstrapServices() {
    ...
    // 第一步：启动Installer
    // 阻塞等待installd完成启动，以便有机会创建具有适当权限的关键目录，如/data/user。
    // 我们需要在初始化其他服务之前完成此任务。
    Installer installer = mSystemServiceManager.startService(Installer.class);
    mActivityManagerService.setInstaller(installer);

    ...
    // 第二步：获取设别是否加密(手机设置密码)，如果设备加密了，则只解析"core"应用，mOnlyCore
= true，后面会频繁使用该变量进行条件判断
    String cryptState = VoldProperties.decrypt().orElse("");
    if (ENCRYPTING_STATE.equals(cryptState)) {
        Slog.w(TAG, "Detected encryption in progress - only parsing core apps");
        mOnlyCore = true;
    } else if (ENCRYPTED_STATE.equals(cryptState)) {
        Slog.w(TAG, "Device encrypted - only parsing core apps");
        mOnlyCore = true;
    }

    // 第三步：调用main方法初始化PackageManagerService
    mPackageManagerService = PackageManagerService.main(mSystemContext,
installer,
                mFactoryTestMode != FactoryTest.FACTORY_TEST_OFF, mOnlyCore);

    // PKMS是否是第一次启动
    mFirstBoot = mPackageManagerService.isFirstBoot();

    // 第四步：如果设备没有加密，操作它。管理A/B OTA dexopting。
    if (!mOnlyCore) {
        boolean disableOtaDexopt =
SystemProperties.getBoolean("config.disable_otadexopt",
                false);
        OtaDexoptService.main(mSystemContext, mPackageManagerService);
    }
    ...
}
```

第五步，第六步，第七步：

startOtherServices

第五步：执行 updatePackagesIfNeeded，完成dex优化；

第六步：执行 performFstrimIfNeeded，完成磁盘维护；

第七步：调用systemReady，准备就绪。

```
 private void startOtherServices() {
    ...
    if (!mOnlyCore) {
        ...
        // 第五步：如果设备没有加密，执行performDexOptUpgrade，完成dex优化；
        mPackageManagerService.updatePackagesIfNeeded();
    }
    ...
    // 第六步：最终执行performFstrim，完成磁盘维护
    mPackageManagerService.performFstrimIfNeeded();
    ...
    // 第七步：PKMS准备就绪
    mPackageManagerService.systemReady();
    ...
}
```

为什么是分析第三步，需要给同学们说清楚：

第三步细节：PKMS.main()

main函数主要工作：

(1) 检查Package编译相关系统属性

(2) 调用PackageManagerService构造方法

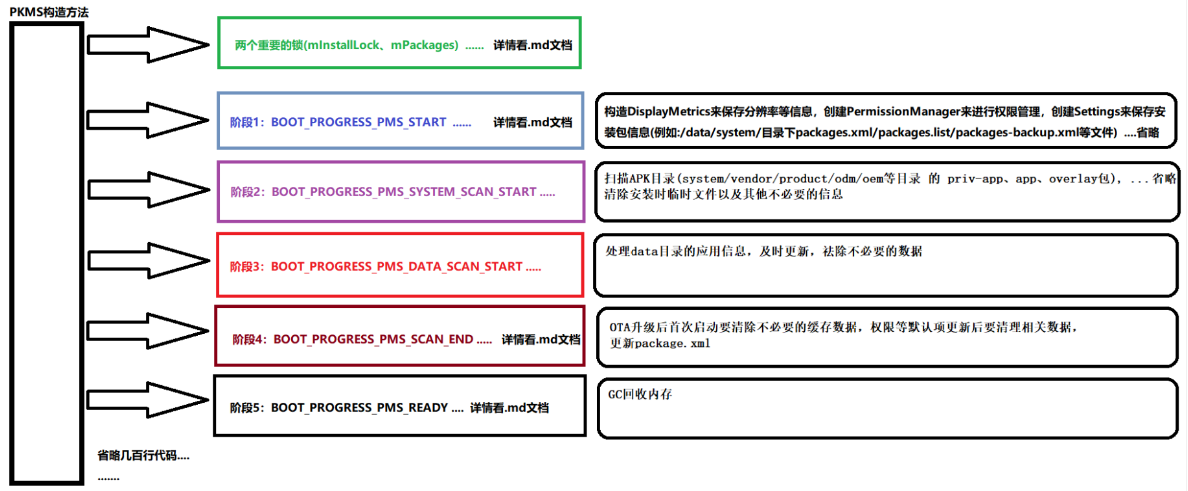
(3) 启用部分应用服务于多用户场景

(4) 往ServiceManager中注册”package”和”package_native”。

```
public static PackageManagerService main(Context context, Installer installer,
        boolean factoryTest, boolean onlyCore) {
    // (1)检查Package编译相关系统属性
    PackageManagerServiceCompilerMapping.checkProperties();

    //(2)调用PackageManagerService构造方法, 同学们可以参考【PKMS构造方法】
    PackageManagerService m = new PackageManagerService(context, installer,
            factoryTest, onlyCore);
    //(3)启用部分应用服务于多用户场景
    m.enableSystemUserPackages();

    //(4)往ServiceManager中注册”package”和”package_native”。
    ServiceManager.addService("package", m);
    final PackageManagerNative pmn = m.new PackageManagerNative();
    ServiceManager.addService("package_native", pmn);
    return m;
}
```

【PKMS构造方法】：

PKMS构造方法，整体描述图：



PKMS构造方法

两个重要的锁(mInstallLock、mPackages) …… 详情看.md文档

阶段1：BOOT_PROGRESS_PMS_START …… 详情看.md文档

构造DisplayMetrics来保存分辨率等信息，创建PermissionManager来进行权限管理，创建Settings来保存安装包信息(例如:/data/system/目录下packages.xml/packages.list/packages-backup.xml等文件) ....省略

阶段2：BOOT_PROGRESS_PMS_SYSTEM_SCAN_START .....

扫描APK目录(system/vendor/product/odm/oem等目录 的 priv-app、app、overlay包)，...省略
清除安装时临时文件以及其他不必要的信息

阶段3：BOOT_PROGRESS_PMS_DATA_SCAN_START .....

处理data目录的应用信息，及时更新，祛除不必要的数据

阶段4：BOOT_PROGRESS_PMS_SCAN_END ..... 详情看.md文档

OTA升级后首次启动要清除不必要的缓存数据，权限等默认项更新后要清理相关数据，
更新package.xml

阶段5：BOOT_PROGRESS_PMS_READY .... 详情看.md文档

GC回收内存

省略几百行代码....
....

KMS初始化时的核心部分为PKMS()构造函数的内容，我们下面就来分析该流程:

PKMS的构造函数中由

```
两个重要的锁(mInstallLock、mPackages):
    mInstallLock ：用来保护所有安装apk的访问权限，此操作通常涉及繁重的磁盘数据读写等操作，并且是单线程操作，故有时候会处理很慢，
    此锁不会在已经持有mPackages锁的情况下火的，反之，在已经持有mInstallLock锁的情况下，立即获取mPackages是安全的

    mPackages：用来解析内存中所有apk的package信息及相关状态。
```

和

```
5个阶段构成，下面会详细的来分析这些内容。
    阶段1：BOOT_PROGRESS_PMS_START

    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
```

阶段1，阶段2，阶段3，阶段4，阶段5 的 EventLog：

```
public PackageManagerService(Context context, Installer installer,
        boolean factoryTest, boolean onlyCore) {
        ...
        // 阶段1：BOOT_PROGRESS_PMS_START
        EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_PMS_START,
                SystemClock.uptimeMillis());

        // 阶段2：BOOT_PROGRESS_PMS_SYSTEM_SCAN_START
        EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_PMS_SYSTEM_SCAN_START,
                    startTime);
        ...
        // 阶段3：BOOT_PROGRESS_PMS_DATA_SCAN_STAR
        if (!mOnlyCore) {

 EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_PMS_DATA_SCAN_START,
                        SystemClock.uptimeMillis());
        }
        ...
        // 阶段4：BOOT_PROGRESS_PMS_SCAN_END
        EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_PMS_SCAN_END,
                    SystemClock.uptimeMillis());
        ...
        // 阶段5：BOOT_PROGRESS_PMS_READY
        EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_PMS_READY,
                    SystemClock.uptimeMillis());
}
```

阶段1细节：

(1) 构造 DisplayMetrics ，保存分辨率等相关信息；

(2) 创建Installer对象，与installd交互；

(3) 创建mPermissionManager对象，进行权限管理；

(4) 构造Settings类，保存安装包信息，清除路径不存在的孤立应用，主要涉及/data/system/目录的packages.xml，packages-backup.xml，packages.list，packages-stopped.xml，packages-stopped-backup.xml等文件。

(5) 构造PackageDexOptimizer及DexManager类，处理dex优化；

(6) 创建SystemConfig实例，获取系统配置信息，配置共享lib库；

(7) 创建PackageManager的handler线程，循环处理外部安装相关消息。

```
public PackageManagerService(...) {
    LockGuard.installLock(mPackages, LockGuard.INDEX_PACKAGES);
    EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_PMS_START,
                SystemClock.uptimeMillis());
    mContext = context;

    mFactoryTest = factoryTest; // 一般为false，即非工厂生产模式
    mOnlyCore = onlyCore; //标记是否只加载核心服务

    // 【同学们注意】(1) 构造 DisplayMetrics ，保存分辨率等相关信息；
    mMetrics = new DisplayMetrics(); // 分辨率配置

    // 【同学们注意】(2)创建Installer对象，与installd交互；
    mInstaller = installer; //保存installer对象

    //创建提供服务/数据的子组件。这里的顺序很重要,使用到了两个重要的同步锁：mInstallLock、
mPackages
    synchronized (mInstallLock) {
        synchronized (mPackages) {
            // 公开系统组件使用的私有服务
            // 本地服务
            LocalServices.addService(
                    PackageManagerInternal.class, new
PackageManagerInternalImpl());
            // 多用户管理服务
            sUserManager = new UserManagerService(context, this,
                    new UserDataPreparer(mInstaller, mInstallLock, mContext,
mOnlyCore), mPackages);
            mComponentResolver = new ComponentResolver(sUserManager,
                    LocalServices.getService(PackageManagerInternal.class),
                    mPackages);

            // 【同学们注意】 (3)创建mPermissionManager对象，进行权限管理；
            // 权限管理服务
            mPermissionManager = PermissionManagerService.create(context,
                    mPackages /*externalLock*/);
            mDefaultPermissionPolicy =
mPermissionManager.getDefaultPermissionGrantPolicy();

            //创建Settings对象
            mSettings = new Settings(Environment.getDataDirectory(),
                    mPermissionManager.getPermissionSettings(), mPackages);
        }
    }

    // 【同学们注意】(4)构造Settings类，保存安装包信息，清除路径不存在的孤立应用，主要涉
及/data/system/目录的packages.xml，packages-backup.xml，          packages.list，
packages-stopped.xml，packages-stopped-backup.xml等文件。
    // 添加system, phone, log, nfc, bluetooth, shell，se，networkstack 这8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);
    mSettings.addSharedUserLPw("android.uid.se", SE_UID,
            ApplicationInfo.FLAG_SYSTEM,
ApplicationInfo.PRIVATE_FLAG_PRIVILEGED);
    mSettings.addSharedUserLPw("android.uid.networkstack", NETWORKSTACK_UID,
            ApplicationInfo.FLAG_SYSTEM,
ApplicationInfo.PRIVATE_FLAG_PRIVILEGED);
    ...

    // 【同学们注意】(5)构造PackageDexOptimizer及DexManager类，处理dex优化；
    // DexOpt优化
    mPackageDexOptimizer = new PackageDexOptimizer(installer, mInstallLock,
context,
                "*dexopt*");
    mDexManager = new DexManager(mContext, this, mPackageDexOptimizer,
installer, mInstallLock);
    // ART虚拟机管理服务
    mArtManagerService = new ArtManagerService(mContext, this, installer,
mInstallLock);
    mMoveCallbacks = new MoveCallbacks(FgThread.get().getLooper());

    mViewCompiler = new ViewCompiler(mInstallLock, mInstaller);
    // 权限变化监听器
    mOnPermissionChangeListeners = new OnPermissionChangeListeners(
            FgThread.get().getLooper());
    mProtectedPackages = new ProtectedPackages(mContext);
    mApexManager = new ApexManager(context);

    // 获取默认分辨率
    getDefaultDisplayMetrics(context, mMetrics);
    // 【同学们注意】(6)创建SystemConfig实例，获取系统配置信息，配置共享lib库；
    //拿到SystemConfig()的对象，其中会调用SystemConfig的readPermissions()完成权限的读取
    SystemConfig systemConfig = SystemConfig.getInstance();
    synchronized (mInstallLock) {
            // writer
            synchronized (mPackages) {
                // 【同学们注意】(7)创建PackageManager的handler线程，循环处理外部安装相
关消息。
                // 启动"PackageManager"线程，负责apk的安装、卸载
                mHandlerThread = new ServiceThread(TAG,
                        Process.THREAD_PRIORITY_BACKGROUND, true /*allowIo*/);
                mHandlerThread.start();
                // 应用handler
                mHandler = new PackageHandler(mHandlerThread.getLooper());
                // 进程记录handler
                mProcessLoggingHandler = new ProcessLoggingHandler();
                // Watchdog监听ServiceThread是否超时：10分钟
                Watchdog.getInstance().addThread(mHandler, WATCHDOG_TIMEOUT);
                // Instant应用注册
                mInstantAppRegistry = new InstantAppRegistry(this);
                 // 共享lib库配置
                ArrayMap<String, SystemConfig.SharedLibraryEntry> libConfig
                        = systemConfig.getSharedLibraries();
                final int builtInLibCount = libConfig.size();
                for (int i = 0; i < builtInLibCount; i++) {
                    String name = libConfig.keyAt(i);
                    SystemConfig.SharedLibraryEntry entry =
libConfig.valueAt(i);
                    addBuiltInSharedLibraryLocked(entry.filename, name);
                }
                ...
                // 读取安装相关SELinux策略
                SELinuxMMAC.readInstallPolicy();

                // 返回栈加载
                FallbackCategoryProvider.loadFallbacks();

                // 【【同学们注意：这段代码 等下下面会分析】】
                //读取并解析/data/system下的XML文件
                mFirstBoot = !mSettings.readLPw(sUserManager.getUsers(false));
                // 清理代码路径不存在的孤立软件包
                final int packageSettingCount = mSettings.mPackages.size();
                for (int i = packageSettingCount - 1; i >= 0; i--) {
                    PackageSetting ps = mSettings.mPackages.valueAt(i);
                    if (!isExternal(ps) && (ps.codePath == null ||
!ps.codePath.exists())
                            && mSettings.getDisabledSystemPkgLPr(ps.name) !=
null) {
                        mSettings.mPackages.removeAt(i);
                        mSettings.enableSystemPackageLPw(ps.name);
                    }
                }

                // 如果不是首次启动，也不是CORE应用，则拷贝预编译的DEX文件
                if (!mOnlyCore && mFirstBoot) {
                    requestCopyPreoptedFiles();
                }
                ...
            } // synchronized (mPackages)
        }
}
```

同学们注意，此readLPw 是上面调下来的哦：

mSettings.readLPw

```
readLPw()会扫描下面5个文件
1) "/data/system/packages.xml"                  所有安装app信息
2) "/data/system/packages-backup.xml"           所有安装app信息之备份的信息记录
3) "/data/system/packages.list"                 所有安装app信息
4) "/data/system/packages-stopped.xml"          所有强制停止app信息
5) "/data/system/packages-stopped-backup.xml"   所有强制停止app信息之备份的信息记录

个文件共分为三组，简单的作用描述如下：
packages.xml：PKMS 扫描完目标文件夹后会创建该文件。当系统进行程序安装、卸载和更新等操作时，均
会更新该文件。该文件保存了系统中与 package 相关的一些信息。
packages.list：描述系统中存在的所有非系统自带的 APK 的信息。当这些程序有变动时，PKMS 就会更
新该文件。
packages-stopped.xml：从系统自带的设置程序中进入应用程序页面，然后在选择强制停止
（ForceStop）某个应用时，系统会将该应用的相关信息记录到此文件中。也就是该文件保存系统中被用户强
制停止的 Package 的信息。

这些目录的指向，都在Settings中的构造函数完成， 如下所示，得到目录后调用readLPw()进行扫描

// 同学们：先看Settings构造函数
Settings(File dataDir, PermissionSettings permission,
        Object lock) {
    mLock = lock;
    mPermissions = permission;
    mRuntimePermissionsPersistence = new RuntimePermissionPersistence(mLock);

    mSystemDir = new File(dataDir, "system");  //mSystemDir指向目录"/data/system"
    mSystemDir.mkdirs();  //创建 "/data/system"
    //设置权限
    FileUtils.setPermissions(mSystemDir.toString(),
            FileUtils.S_IRWXU|FileUtils.S_IRWXG
            |FileUtils.S_IROTH|FileUtils.S_IXOTH,
            -1, -1);

    //(1)指向目录"/data/system/packages.xml"
    mSettingsFilename = new File(mSystemDir, "packages.xml");
    //(2)指向目录"/data/system/packages-backup.xml"
    mBackupSettingsFilename = new File(mSystemDir, "packages-backup.xml");
    //(3)指向目录"/data/system/packages.list"
    mPackageListFilename = new File(mSystemDir, "packages.list");
    FileUtils.setPermissions(mPackageListFilename, 0640, SYSTEM_UID,
PACKAGE_INFO_GID);
    //(4)指向目录"/data/system/packages-stopped.xml"
    mStoppedPackagesFilename = new File(mSystemDir, "packages-stopped.xml");
    //(5)指向目录"/data/system/packages-stopped-backup.xml"
    mBackupStoppedPackagesFilename = new File(mSystemDir, "packages-stopped-
backup.xml");
}

// 同学们：在看readLPw函数
[Settings.java]
boolean readLPw(@NonNull List<UserInfo> users) {
    FileInputStream str = null;
    ...
    if (str == null) {
        str = new FileInputStream(mSettingsFilename);
    }
    //解析"/data/system/packages.xml"
    XmlPullParser parser = Xml.newPullParser();
    parser.setInput(str, StandardCharsets.UTF_8.name());

    int type;
    while ((type = parser.next()) != XmlPullParser.START_TAG
            && type != XmlPullParser.END_DOCUMENT) {
        ;
    }
    int outerDepth = parser.getDepth();
    while ((type = parser.next()) != XmlPullParser.END_DOCUMENT
            && (type != XmlPullParser.END_TAG || parser.getDepth() >
outerDepth)) {
        if (type == XmlPullParser.END_TAG || type == XmlPullParser.TEXT) {
            continue;
        }
        //根据XML的各个节点进行各种操作，例如读取权限、shared-user等
        String tagName = parser.getName();
        if (tagName.equals("package")) {
            readPackageLPw(parser);
        } else if (tagName.equals("permissions")) {
            mPermissions.readPermissions(parser);
        } else if (tagName.equals("permission-trees")) {
            mPermissions.readPermissionTrees(parser);
        } else if (tagName.equals("shared-user")) {
            readSharedUserLPw(parser);
        }...
    }
    str.close();
    ...
    return true;
}
```

阶段2细节：

(1) 从init.rc中获取环境变量BOOTCLASSPATH和SYSTEMSERVERCLASSPATH；

(2) 对于旧版本升级的情况，将安装时获取权限变更为运行时申请权限；

(3) 扫描system/vendor/product/odm/oem等目录的priv-app、app、overlay包；

(4) 清除安装时临时文件以及其他不必要的信息。

```
public PackageManagerService(Context context, Installer installer,
            boolean factoryTest, boolean onlyCore) {
    synchronized (mInstallLock) {
        synchronized (mPackages) {
            // 记录扫描开始时间
            long startTime = SystemClock.uptimeMillis();

 EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_PMS_SYSTEM_SCAN_START,
                    startTime);

            // 【同学们注意】 (1)从init.rc中获取环境变量BOOTCLASSPATH和
SYSTEMSERVERCLASSPATH；
            //获取环境变量，init.rc
            final String bootClassPath = System.getenv("BOOTCLASSPATH");
            final String systemServerClassPath =
System.getenv("SYSTEMSERVERCLASSPATH");
            ...
            // 获取system/framework目录
            File frameworkDir = new File(Environment.getRootDirectory(),
"framework");
            // 获取内部版本
            final VersionInfo ver = mSettings.getInternalVersion();
            // 判断fingerprint是否有更新
            mIsUpgrade = !Build.FINGERPRINT.equals(ver.fingerprint);
            ...
            // 【同学们注意】 (2)对于旧版本升级的情况，将安装时获取权限变更为运行时申请权限；

            // 对于Android M之前版本升级上来的情况，需将系统应用程序权限从安装升级到运行时
            mPromoteSystemApps =
                    mIsUpgrade && ver.sdkVersion <=
Build.VERSION_CODES.LOLLIPOP_MR1;
            // 对于Android N之前版本升级上来的情况，需像首次启动一样处理package
            mIsPreNUpgrade = mIsUpgrade && ver.sdkVersion <
Build.VERSION_CODES.N;
            mIsPreNMR1Upgrade = mIsUpgrade && ver.sdkVersion <
Build.VERSION_CODES.N_MR1;
            mIsPreQUpgrade = mIsUpgrade && ver.sdkVersion <
Build.VERSION_CODES.Q;
            // 在扫描之前保存预先存在的系统package的名称，不希望自动为新系统应用授予运行时权
限
            if (mPromoteSystemApps) {
                Iterator<PackageSetting> pkgSettingIter =
mSettings.mPackages.values().iterator();
                while (pkgSettingIter.hasNext()) {
                    PackageSetting ps = pkgSettingIter.next();
                    if (isSystemApp(ps)) {
                        mExistingSystemPackages.add(ps.name);
                    }
                }
            }
            // 准备解析package的缓存
            mCacheDir = preparePackageParserCache();
            // 设置flag，而不在扫描安装时更改文件路径
            int scanFlags = SCAN_BOOTING | SCAN_INITIAL;
            ...

            // 【同学们注意：】(3)扫描system/vendor/product/odm/oem等目录的priv-app、app、overlay包；
            //扫描以下路径：

 /vendor/overlay、/product/overlay、/product_services/overlay、/odm/overlay、/oem/overlay、/system/framework
            /system/priv-app、/system/app、/vendor/priv-app、/vendor/app、/odm/priv-app、/odm/app、/oem/app、/oem/priv-app、
            /product/priv-app、/product/app、/product_services/priv-app、/product_services/app、/product_services/priv-app
            // [ PMSapk的安装]
            scanDirTracedLI(new File(VENDOR_OVERLAY_DIR),...);
            scanDirTracedLI(new File(PRODUCT_OVERLAY_DIR),...);
            scanDirTracedLI(new File(PRODUCT_SERVICES_OVERLAY_DIR),...);
            scanDirTracedLI(new File(ODM_OVERLAY_DIR),...);
            scanDirTracedLI(new File(OEM_OVERLAY_DIR),...);
            ...
            final List<String> possiblyDeletedUpdatedSystemApps = new ArrayList<>();
            final List<String> stubSystemApps = new ArrayList<>();
            // 删掉不存在的package
            if (!mOnlyCore) {
                final Iterator<PackageParser.Package> pkgIterator = mPackages.values().iterator();
                while (pkgIterator.hasNext()) {
                    final PackageParser.Package pkg = pkgIterator.next();
                    if (pkg.isStub) {
                        stubSystemApps.add(pkg.packageName);
                    }
                }
                final Iterator<PackageSetting> psit = mSettings.mPackages.values().iterator();
                while (psit.hasNext()) {
                    PackageSetting ps = psit.next();
                    // 如果不是系统应用，则不被允许disable
                    if ((ps.pkgFlags & ApplicationInfo.FLAG_SYSTEM) == 0) {
                        continue;
                    }

                    // 如果应用被扫描，则不允许被擦除
                    final PackageParser.Package scannedPkg = mPackages.get(ps.name);
                    if (scannedPkg != null) {
                        // 如果系统应用被扫描且存在disable应用列表中，则只能通过OTA升级添加

                        if (mSettings.isDisabledSystemPackageLPr(ps.name)) {
                            ...
                            removePackageLI(scannedPkg, true);
                            mExpectingBetter.put(ps.name, ps.codePath);
                        }
                        continue;
                    }
                    ...
                }
            }
            // 【同学们注意】(4)清除安装时临时文件以及其他不必要的信息。
            // 删除临时文件
            deleteTempPackageFiles();
            // 删除没有关联应用的共享UID标识
            mSettings.pruneSharedUsersLPw();
            ...
        }
        ...
    }
    ...
}
```

阶段3细节：

对于不仅仅解析核心应用的情况下，还处理data目录的应用信息，及时更新，祛除不必要的数据。

```
public PackageManagerService(Context context, Installer installer,
            boolean factoryTest, boolean onlyCore) {
    synchronized (mInstallLock) {
        synchronized (mPackages) {
            ...
            if (!mOnlyCore) {

 EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_PMS_DATA_SCAN_START,
                        SystemClock.uptimeMillis());
                scanDirTracedLI(sAppInstallDir, 0, scanFlags |
SCAN_REQUIRE_KNOWN, 0);
                ...
                // 移除通过OTA删除的更新系统应用程序的禁用package设置
                // 如果更新不再存在，则完全删除该应用。否则，撤消其系统权限
                for (int i = possiblyDeletedUpdatedSystemApps.size() - 1; i >=
0; --i) {
                    final String packageName =
possiblyDeletedUpdatedSystemApps.get(i);
                    final PackageParser.Package pkg =
mPackages.get(packageName);
                    final String msg;

                    mSettings.removeDisabledSystemPackageLPw(packageName);
                    ...
                }
                // 确保期望在userdata分区上显示的所有系统应用程序实际显示
                // 如果从未出现过，需要回滚以恢复系统版本
                for (int i = 0; i < mExpectingBetter.size(); i++) {
                    final String packageName = mExpectingBetter.keyAt(i);
                    if (!mPackages.containsKey(packageName)) {
                        final File scanFile = mExpectingBetter.valueAt(i);
                        ...
                        mSettings.enableSystemPackageLPw(packageName);
                        try {
              //扫描APK
                            scanPackageTracedLI(scanFile, reparseFlags,
rescanFlags, 0, null);
                        } catch (PackageManagerException e) {
                            Slog.e(TAG, "Failed to parse original system
package: "
                                    + e.getMessage());
                        }
                    }
                }
                // 解压缩并安装任何存根系统应用程序。必须最后执行此操作以确保替换或禁用所有存
根
                installSystemStubPackages(stubSystemApps, scanFlags);
                ...
                // 获取storage manager包名
                mStorageManagerPackage = getStorageManagerPackageName();
                // 解决受保护的action过滤器。只允许setup wizard（开机向导）为这些action
设置高优先级过滤器
                mSetupWizardPackage = getSetupWizardPackageName();
                ...
                // 更新客户端以确保持有正确的共享库路径
                updateAllSharedLibrariesLocked(null,
Collections.unmodifiableMap(mPackages));
                ...
                // 读取并更新要保留的package的上次使用时间
                mPackageUsage.read(mPackages);
                mCompilerStats.read();
            }
        }
    }
}
```

阶段4细节：

(1) sdk版本变更，更新权限；

(2) OTA升级后首次启动，清除不必要的缓存数据；

(3) 权限等默认项更新完后，清理相关数据；

(4) 更新package.xml

```
public PackageManagerService(Context context, Installer installer,
            boolean factoryTest, boolean onlyCore) {
    synchronized (mInstallLock) {
        synchronized (mPackages) {
            ...
            EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_PMS_SCAN_END,
                    SystemClock.uptimeMillis());

            // 【同学们注意】 (1) sdk版本变更，更新权限；
            // 如果自上次启动以来，平台SDK已改变，则需要重新授予应用程序权限以捕获出现的任何
新权限
            final boolean sdkUpdated = (ver.sdkVersion != mSdkVersion);
            mPermissionManager.updateAllPermissions(
                    StorageManager.UUID_PRIVATE_INTERNAL, sdkUpdated,
mPackages.values(),
                    mPermissionCallback);
            ...
            // 如果这是第一次启动或来自Android M之前的版本的升级，并且它是正常启动，那需要在
所有已定义的用户中初始化默认的首选应用程序
            if (!onlyCore && (mPromoteSystemApps || mFirstBoot)) {
                for (UserInfo user : sUserManager.getUsers(true)) {
                    mSettings.applyDefaultPreferredAppsLPw(user.id);
                    primeDomainVerificationsLPw(user.id);
                }
            }
            // 在启动期间确实为系统用户准备存储，因为像SettingsProvider和SystemUI这样的核
心系统应用程序无法等待用户启动
            final int storageFlags;
            if (StorageManager.isFileEncryptedNativeOrEmulated()) {
                storageFlags = StorageManager.FLAG_STORAGE_DE;
            } else {
                storageFlags = StorageManager.FLAG_STORAGE_DE |
StorageManager.FLAG_STORAGE_CE;
            }
            ...
            // 【同学们注意】(2) OTA升级后首次启动，清除不必要的缓存数据；
            // 如果是在OTA之后首次启动，并且正常启动，那需要清除代码缓存目录，但不清除应用程
序配置文件
            if (mIsUpgrade && !onlyCore) {
                Slog.i(TAG, "Build fingerprint changed; clearing code caches");
                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)) {
                        // No apps are running this early, so no need to freeze
                        clearAppDataLIF(ps.pkg, UserHandle.USER_ALL,
                                FLAG_STORAGE_DE | FLAG_STORAGE_CE |
FLAG_STORAGE_EXTERNAL
                                        | Installer.FLAG_CLEAR_CODE_CACHE_ONLY);
                    }
                }
                ver.fingerprint = Build.FINGERPRINT;
            }

            //安装Android-Q前的非系统应用程序在Launcher中隐藏他们的图标
            if (!onlyCore && mIsPreQUpgrade) {
                Slog.i(TAG, "Whitelisting all existing apps to hide their
icons");
                int size = mSettings.mPackages.size();
                for (int i = 0; i < size; i++) {
                    final PackageSetting ps = mSettings.mPackages.valueAt(i);
                    if ((ps.pkgFlags & ApplicationInfo.FLAG_SYSTEM) != 0) {
                        continue;
                    }

 ps.disableComponentLPw(PackageManager.APP_DETAILS_ACTIVITY_CLASS_NAME,
                            UserHandle.USER_SYSTEM);
                }
            }

            // 【同学们注意】 (3) 权限等默认项更新完后，清理相关数据；
            // 仅在权限或其它默认配置更新后清除
            mExistingSystemPackages.clear();
            mPromoteSystemApps = false;
            ...
             // 所有变更均在扫描过程中完成
            ver.databaseVersion = Settings.CURRENT_DATABASE_VERSION;

            // 【同学们注意】(4) 更新package.xml
            //降级去读取
            mSettings.writeLPr();
        }
    }
}
```

阶段5细节：

GC回收内存 和一些细节而已

```
 public PackageManagerService(Context context, Installer installer,
            boolean factoryTest, boolean onlyCore) {
    synchronized (mInstallLock) {
        synchronized (mPackages) {
            ...
            EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_PMS_READY,
                    SystemClock.uptimeMillis());
            ...
            //PermissionController 主持 缺陷许可证的授予和角色管理，所以这是核心系统的一个
关键部分。
            mRequiredPermissionControllerPackage =
getRequiredPermissionControllerLPr();
            ...
            updateInstantAppInstallerLocked(null);
            // 阅读并更新dex文件的用法
            // 在PM init结束时执行此操作，以便所有程序包都已协调其数据目录
            // 此时知道了包的代码路径，因此可以验证磁盘文件并构建内部缓存
            // 使用文件预计很小，因此与其他活动（例如包扫描）相比，加载和验证它应该花费相当小
的时间
            final Map<Integer, List<PackageInfo>> userPackages = new HashMap<>
();
            for (int userId : userIds) {
                userPackages.put(userId, getInstalledPackages(/*flags*/ 0,
userId).getList());
            }
            mDexManager.load(userPackages);
            if (mIsUpgrade) {
                MetricsLogger.histogram(null, "ota_package_manager_init_time",
                        (int) (SystemClock.uptimeMillis() - startTime));
            }
        }
    }
    ...
    // 【同学们注意】GC回收内存
    // 打开应用之后，及时回收处理
    Runtime.getRuntime().gc();
    // 上面的初始扫描在持有mPackage锁的同时对installd进行了多次调用
    mInstaller.setWarnIfHeld(mPackages);
    ...
}
```

Derry 2020年9月10日00:59:54 保存

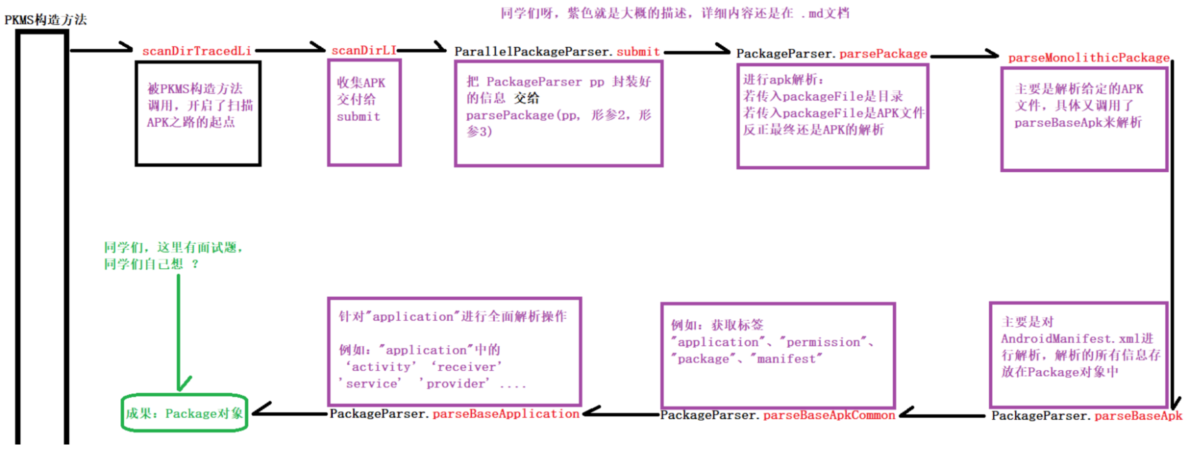
三部曲 - APK的扫描：

同学们注意：PKMS的构造函数中调用了 **scanDirTracedLI**方法 来扫描某个目录的apk文件。

同学们注意：Android10.0 和 其他低版本扫描的路径是不一样的：Android 10.0中，PKMS主要扫描以下路径的APK信息：

```
/vendor/overlay                    系统的APP类别
/product/overlay                   系统的APP类别
/product_services/overlay          系统的APP类别
/odm/overlay                       系统的APP类别
/oem/overlay                       系统的APP类别
/system/framework                  系统的APP类别
/system/priv-app                   系统的APP类别
/system/app                        系统的APP类别
/vendor/priv-app                   系统的APP类别
/vendor/app                        系统的APP类别
/odm/priv-app                      系统的APP类别
/odm/app                           系统的APP类别
/oem/app                           系统的APP类别
/oem/priv-app                      系统的APP类别
/product/priv-app                  系统的APP类别
/product/app                       系统的APP类别
/product_services/priv-app         系统的APP类别
/product_services/app              系统的APP类别
/product_services/priv-app         系统的APP类别
```

APK的扫描，整体描述图：


同学们呀，紫色就是大概的描述，详细内容还是在 .md文档
PKMS构造方法
scanDirTracedLi
被PKMS构造方法调用，开启了扫描APK之路的起点
scanDirLI
收集APK交付给submit
ParallelPackageParser.submit
把 PackageParser pp 封装好的信息 交给 parsePackage(pp, 形参2, 形参3)
PackageParser.parsePackage
进行apk解析：
若传入packageFile是目录
若传入packageFile是APK文件
反正最终还是APK的解析
parseMonolithicPackage
主要是解析给定的APK文件，具体又调用了parseBaseApk来解析
PackageParser.parseBaseApk
主要是对AndroidManifest.xml进行解析，解析的所有信息存放在Package对象中
PackageParser.parseBaseApkCommon
例如：获取标签"application"、"permission"、"package"、"manifest"
PackageParser.parseBaseApplication
针对"application"进行全面解析操作
例如："application"中的 'activity' 'receiver' 'service' 'provider' ....
成果：Package对象
同学们，这里有面试题，同学们自己想 ？

PKMS.**scanDirTracedLi**：首先加入了一些systtrace的日志追踪，然后调用scanDirLI()进行分析

```
private void scanDirTracedLI(File scanDir, final int parseFlags, int scanFlags,
long currentTime) {
    Trace.traceBegin(TRACE_TAG_PACKAGE_MANAGER, "scanDir [" +
scanDir.getAbsolutePath() + "]");
    try {
        // 【同学们注意】会调用此 scanDirLI函数
        scanDirLI(scanDir, parseFlags, scanFlags, currentTime);
    } finally {
        Trace.traceEnd(TRACE_TAG_PACKAGE_MANAGER);
    }
}
```

PKMS.**scanDirLI**： 使用了ParallelPackageParser的对象，ParallelPackageParser是一个队列，我们这里手机所有系统的apk，然后从这些队列里面取出apk，再调用**PackageParser** 解析进行解析：

```
private void scanDirLI(File scanDir, int parseFlags, int scanFlags, long
currentTime) {
    final File[] files = scanDir.listFiles();
    if (ArrayUtils.isEmpty(files)) {
        Log.d(TAG, "No files in app dir " + scanDir);
        return;
    }

    if (DEBUG_PACKAGE_SCANNING) {
        Log.d(TAG, "Scanning app dir " + scanDir + " scanFlags=" + scanFlags
                + " flags=0x" + Integer.toHexString(parseFlags));
    }
    // parallelPackageParser是一个队列，收集系统 apk 文件，
    // 然后从这个队列里面一个个取出 apk ，调用 PackageParser 解析
     try (ParallelPackageParser parallelPackageParser = new
ParallelPackageParser(
            mSeparateProcesses, mOnlyCore, mMetrics, mCacheDir,
            mParallelPackageParserCallback)) {
        // Submit files for parsing in parallel
        int fileCount = 0;
        for (File file : files) {
            // 是Apk文件，或者是目录
            final boolean isPackage = (isApkFile(file) || file.isDirectory())
                    && !PackageInstallerService.isStageName(file.getName());
            过滤掉非 apk 文件，如果不是则跳过继续扫描
            if (!isPackage) {
                // Ignore entries which are not packages
                continue;
            }
            // 把APK信息存入parallelPackageParser中的对象mQueue，PackageParser()函数
赋给了队列中的pkg成员
            // 【同学们注意】 这里的 submit 函数 很重要，下面就会分析此函数
            parallelPackageParser.submit(file, parseFlags);
            fileCount++;
        }

        // Process results one by one
        for (; fileCount > 0; fileCount--) {
        // 从parallelPackageParser中取出队列apk的信息
```

```
            ParallelPackageParser.ParseResult parseResult =
parallelPackageParser.take();
            Throwable throwable = parseResult.throwable;
            int errorCode = PackageManager.INSTALL_SUCCEEDED;

            if (throwable == null) {
                // TODO(toddke): move lower in the scan chain
                // Static shared libraries have synthetic package names
                if (parseResult.pkg.applicationInfo.isStaticSharedLibrary()) {
                    renameStaticSharedLibraryPackage(parseResult.pkg);
                }
                try {
                    //调用 scanPackageChildLI 方法扫描一个特定的 apk 文件
                    // 该类的实例代表一个 APK 文件，所以它就是和 apk 文件对应的数据结构。
                    scanPackageChildLI(parseResult.pkg, parseFlags, scanFlags,
                            currentTime, null);
                } catch (PackageManagerException e) {
                    errorCode = e.error;
                    Slog.w(TAG, "Failed to scan " + parseResult.scanFile + ": "
+ e.getMessage());
                }
            } else if (throwable instanceof
PackageParser.PackageParserException) {
                PackageParser.PackageParserException e =
(PackageParser.PackageParserException)
                        throwable;
                errorCode = e.error;
                Slog.w(TAG, "Failed to parse " + parseResult.scanFile + ": " +
e.getMessage());
            } else {
                throw new IllegalStateException("Unexpected exception occurred
while parsing "
                        + parseResult.scanFile, throwable);
            }

            // Delete invalid userdata apps
            //如果是非系统 apk 并且解析失败
            if ((scanFlags & SCAN_AS_SYSTEM) == 0 &&
                    errorCode != PackageManager.INSTALL_SUCCEEDED) {
                logCriticalInfo(Log.WARN,
                        "Deleting invalid package at " + parseResult.scanFile);
                 // 非系统 Package 扫描失败，删除文件
                removeCodePathLI(parseResult.scanFile);
            }
        }
    }
}
```

ParallelPackageParser.**submit**：

把扫描路径中的APK等内容，放入队列mQueue，

并把parsePackage() pp 赋给ParseResult，用于后面的调用

```
public void submit(File scanFile, int parseFlags) {
    mService.submit(() -> {
        ParseResult pr = new ParseResult();
        Trace.traceBegin(TRACE_TAG_PACKAGE_MANAGER, "parallel parsePackage [" +
scanFile + "]"); // 日志打印
        try {
            PackageParser pp = new PackageParser();
            pp.setSeparateProcesses(mSeparateProcesses);
            pp.setOnlyCoreApps(mOnlyCore);
            pp.setDisplayMetrics(mMetrics);
            pp.setCacheDir(mCacheDir);
            pp.setCallback(mPackageParserCallback);
            pr.scanFile = scanFile;
            // 并把parsePackage()与pp 赋值ParseResult，用于后面的调用
            pr.pkg = parsePackage(pp, scanFile, parseFlags); // 【同学们注意】
parsePackage下面会分析
        } catch (Throwable e) {
            pr.throwable = e;
        } finally {
            Trace.traceEnd(TRACE_TAG_PACKAGE_MANAGER);
        }
        try {
            // 把扫描路径中的APK等内容，放入队列mQueue
            mQueue.put(pr);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            // Propagate result to callers of take().
            // This is helpful to prevent main thread from getting stuck waiting
on
            // ParallelPackageParser to finish in case of interruption
            mInterruptedInThread = Thread.currentThread().getName();
        }
    });
}
```

通过 PackageParser.**parsePackage** 进行apk解析：

如果传入的packageFile是目录， 调用parseClusterPackage()解析

如果传入的packageFile是APK文件， 调用parseMonolithicPackage()解析

```
public Package parsePackage(File packageFile, int flags, boolean useCaches)
        throws PackageParserException {
  ...
    if (packageFile.isDirectory()) {
    //如果传入的packageFile是目录，调用parseClusterPackage()解析
        parsed = parseClusterPackage(packageFile, flags);
    } else {
    //如果是APK文件，就调用parseMonolithicPackage()解析
        parsed = parseMonolithicPackage(packageFile, flags);  // 【同学们注意】下面
我们分析此函数
    }
  ...
    return parsed;
}
```

PackageParser.**parseMonolithicPackage**()，它的作用是解析给定的APK文件，将其作为单个单块包处理，最终调用parseBaseApk()进行解析

```
public Package parseMonolithicPackage(File apkFile, int flags) throws
PackageParserException {
    final PackageLite lite = parseMonolithicPackageLite(apkFile, flags);
    if (mOnlyCoreApps) {
        if (!lite.coreApp) {
            throw new
PackageParserException(INSTALL_PARSE_FAILED_MANIFEST_MALFORMED,
                    "Not a coreApp: " + apkFile);
        }
    }

    final SplitAssetLoader assetLoader = new DefaultSplitAssetLoader(lite,
flags);
    try {
        // 对核心应用解析 【同学们注意】 最终调用parseBaseApk()进行解析，我们下面来分析
        final Package pkg = parseBaseApk(apkFile,
assetLoader.getBaseAssetManager(), flags);
        pkg.setCodePath(apkFile.getCanonicalPath());
        pkg.setUse32bitAbi(lite.use32bitAbi);
        return pkg;
    } catch (IOException e) {
        throw new
PackageParserException(INSTALL_PARSE_FAILED_UNEXPECTED_EXCEPTION,
                "Failed to get path: " + apkFile, e);
    } finally {
        IoUtils.closeQuietly(assetLoader);
    }
}
```

PackageParser.**parseBaseApk**()主要是对AndroidManifest.xml进行解析，解析后所有的信息放在Package对象中

```
 private Package parseBaseApk(File apkFile, AssetManager assets, int flags)
        throws PackageParserException {
    final String apkPath = apkFile.getAbsolutePath();
     ...
    XmlResourceParser parser = null;
    ...
        final int cookie = assets.findCookieForPath(apkPath);
        if (cookie == 0) {
            throw new PackageParserException(INSTALL_PARSE_FAILED_BAD_MANIFEST,
                    "Failed adding asset path: " + apkPath);
        }
        // 获得一个 XML 资源解析对象，该对象解析的是 APK 中的 AndroidManifest.xml 文件。
        parser = assets.openXmlResourceParser(cookie,
ANDROID_MANIFEST_FILENAME);
        final Resources res = new Resources(assets, mMetrics, null);

        final String[] outError = new String[1];

        // 再调用重载函数parseBaseApk()最终到parseBaseApkCommon()，解析
AndroidManifest.xml 后得到一个Package对象
```

```
        // 【同学们注意】解析后所有的信息放在Package对象中
        final Package pkg = parseBaseApk(apkPath, res, parser, flags, outError);
        ...
        pkg.setVolumeUuid(volumeUuid);
        pkg.setApplicationVolumeUuid(volumeUuid);
        pkg.setBaseCodePath(apkPath);
        pkg.setSigningDetails(SigningDetails.UNKNOWN);

        return pkg;
    ...
}
```

parseBaseApk -----> **parseBaseApkCommon** , parseBaseApk省略了

PackageParser.**parseBaseApkCommon** 从AndroidManifest.xml中获取标签名，解析标签中的各个item的内容，存入Package对象中

例如：获取标签 "application"、"permission"、"package"、"manifest" 同学们，太多了，省略了哈

```
 private Package parseBaseApkCommon(Package pkg, Set<String> acceptedTags,
Resources res,
        XmlResourceParser parser, int flags, String[] outError) throws
XmlPullParserException,
        IOException {
  TypedArray sa = res.obtainAttributes(parser,
            com.android.internal.R.styleable.AndroidManifest);
  //拿到AndroidManifest.xml 中的sharedUserId, 一般情况下有“android.uid.system”等信息
  String str = sa.getNonConfigurationString(
            com.android.internal.R.styleable.AndroidManifest_sharedUserId, 0);

  while ((type = parser.next()) != XmlPullParser.END_DOCUMENT
            && (type != XmlPullParser.END_TAG || parser.getDepth() >
outerDepth)) {
    //从AndroidManifest.xml中获取标签名
    String tagName = parser.getName();
    //如果读到AndroidManifest.xml中的tag是"application",执行parseBaseApplication()进
行解析
    if (tagName.equals(TAG_APPLICATION)) {
            if (foundApp) {
        ...
            }
            foundApp = true;

            // 解析"application"的信息，赋值给pkg
            // 【同学们注意】这里解析到的是"application" <application 包含了 四大组件，
下面分析此操作
            if (!parseBaseApplication(pkg, res, parser, flags, outError)) {
                return null;
            }
      ...
      //如果标签是"permission"
      else if (tagName.equals(TAG_PERMISSION)) {
            //进行"permission"的解析
            if (!parsePermission(pkg, res, parser, outError)) {
                return null;
            }
    ....
  }
 }
}
```

上面解析AndroidManifest.xml，会得到 "**application**"、"overlay"、"permission"、"uses-permission" 等信息

我们下面就针对"**application**"进行展开分析一下，进入 PackageParser.**parseBaseApplication**()函数

```
private boolean parseBaseApplication(Package owner, Resources res,
            XmlResourceParser parser, int flags, String[] outError)
 while ((type = parser.next()) != XmlPullParser.END_DOCUMENT
                && (type != XmlPullParser.END_TAG || parser.getDepth() >
innerDepth)) {
  // 获取"application"子标签的标签内容
  String tagName = parser.getName();
  // 如果标签是"activity"
    if (tagName.equals("activity")) { // 解析Activity的信息，把activity加入
Package对象
      Activity a = parseActivity(owner, res, parser, flags, outError,
cachedArgs, false,
          owner.baseHardwareAccelerated);
      if (a == null) {
        mParseError =
PackageManager.INSTALL_PARSE_FAILED_MANIFEST_MALFORMED;
        return false;
      }

      hasActivityOrder |= (a.order != 0);
      owner.activities.add(a);

    } else if (tagName.equals("receiver")) { // 如果标签是"receiver"，获取
receiver信息，加入Package对象
      Activity a = parseActivity(owner, res, parser, flags, outError,
cachedArgs, true, false);
      if (a == null) {
        mParseError =
PackageManager.INSTALL_PARSE_FAILED_MANIFEST_MALFORMED;
        return false;
      }

      hasReceiverOrder |= (a.order != 0);
      owner.receivers.add(a);

    }else if (tagName.equals("service")) { // 如果标签是"service"，获取service信
息，加入Package对象
      Service s = parseService(owner, res, parser, flags, outError,
cachedArgs);
      if (s == null) {
        mParseError =
PackageManager.INSTALL_PARSE_FAILED_MANIFEST_MALFORMED;
        return false;
      }
```

```
            hasServiceOrder |= (s.order != 0);
            owner.services.add(s);

        }else if (tagName.equals("provider")) { // 如果标签是"provider"，获取provider信息，加入Package对象
            Provider p = parseProvider(owner, res, parser, flags, outError, cachedArgs);
            if (p == null) {
                mParseError = PackageManager.INSTALL_PARSE_FAILED_MANIFEST_MALFORMED;
                return false;
            }

            owner.providers.add(p);
        }
    ...
  }
}
```

在 PackageParser 扫描完一个 APK 后，此时系统已经根据该 APK 中 AndroidManifest.xml，创建了一个完整的 Package 对象

APK的扫描，自我总结：

第一步：扫描APK，解析AndroidManifest.xml文件，得到清单文件各个标签内容

第二步：解析清单文件到的信息由 Package 保存。从该类的成员变量可看出，和 Android 四大组件相关的信息分别由 activites、receivers、providers、services 保存，由于一个 APK 可声明多个组件，因此 activites 和 receivers等均声明为 ArrayList

四部曲 - APK的安装：

安装步骤一: 把Apk的信息通过IO流的形式写入到PackageInstaller.Session中

安装步骤二: 调用PackageInstaller.Session的commit方法, 把Apk的信息交给PKMS处理

安装步骤三: 进行Apk的Copy操作, 进行安装

安装的三步走, 整体描述图:

第一步

安装步骤一：把Apk的信息通过IO流的形式写入到PackageInstaller.Session中

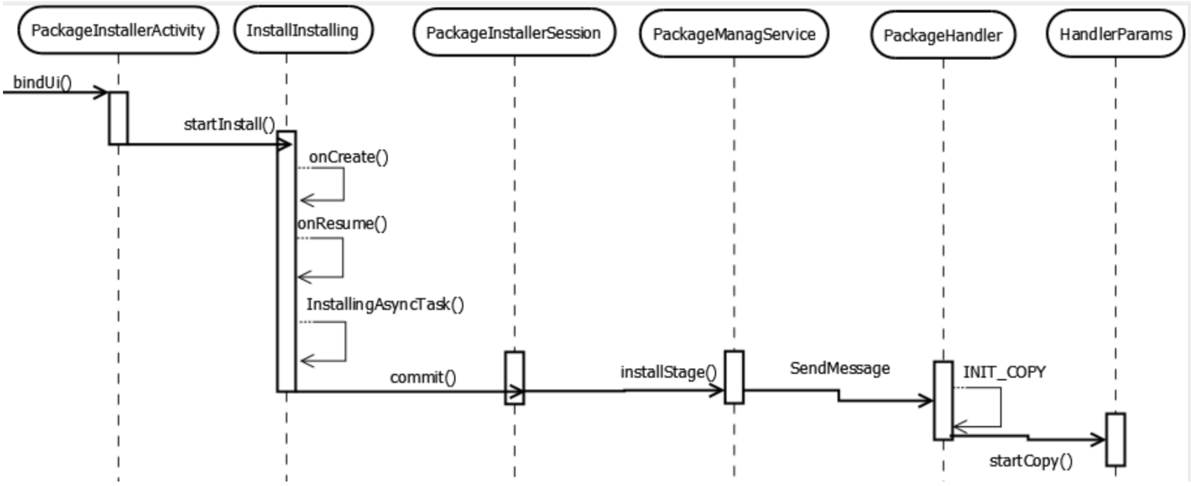
第二步

安装步骤二：调用PackageInstaller.Session的commit方法，
把Apk的信息交给PKMS处理

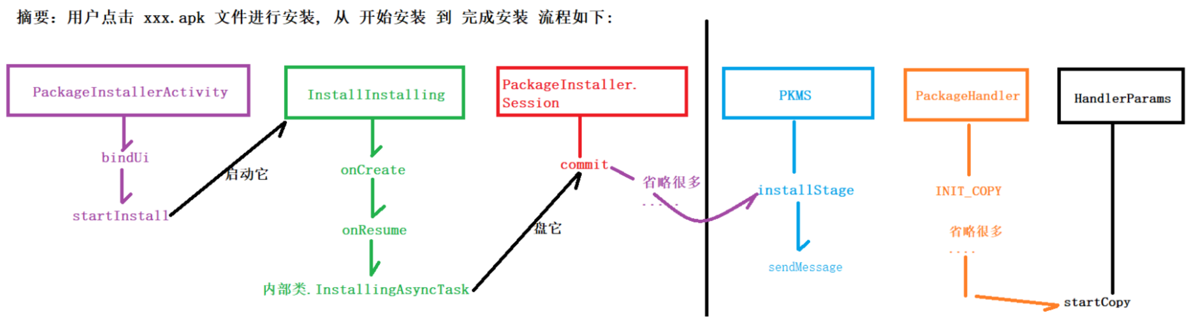
第三步

安装步骤三：进行Apk的Copy操作

用户点击 xxx.apk 文件进行安装, 从 开始安装 到 完成安装 流程如下:



APK的安装, 整体描述图:


摘要：用户点击 xxx.apk 文件进行安装，从 开始安装 到 完成安装 流程如下：

点击一个apk后，会弹出安装界面，点击确定按钮后，会进入**PackageInstallerActivity** 的 **bindUi**() 中的mAlert点击事件, 弹出的安装界面底部显示的是一个diaglog，主要由bindUi构成，上面有 "**取消**" 和 " **安装**" 两个按钮，点击安装后 调用startInstall()进行安装:

```
private void bindUi() {
    mAlert.setIcon(mAppSnippet.icon);
    mAlert.setTitle(mAppSnippet.label);
    mAlert.setView(R.layout.install_content_view);
    mAlert.setButton(DialogInterface.BUTTON_POSITIVE,
getString(R.string.install),
            (ignored, ignored2) -> {
                if (mOk.isEnabled()) {
                    if (mSessionId != -1) {
                        mInstaller.setPermissionsResult(mSessionId, true);
                        finish();
                    } else {
                        startInstall();  // 进行APK安装  [同学们注意] 下面开始分析
startInstall 做的事情
                    }
                }
            }, null);
    mAlert.setButton(DialogInterface.BUTTON_NEGATIVE,
getString(R.string.cancel),
            (ignored, ignored2) -> {
                // Cancel and finish
                setResult(RESULT_CANCELED);
                if (mSessionId != -1) {
                    //如果mSessionId存在，执行setPermissionsResult()完成取消安装
                    mInstaller.setPermissionsResult(mSessionId, false);
                }
                finish();
            }, null);
    setupAlert();

    mOk = mAlert.getButton(DialogInterface.BUTTON_POSITIVE);
    mOk.setEnabled(false);
}
```

startInstall方法组装了一个Intent，并跳转到 InstallInstalling 这个Activity，并关闭掉当前的PackageInstallerActivity。InstallInstalling主要用于向包管理器发送包的信息并处理包管理的回调:

```
private void startInstall() {
    // Start subactivity to actually install the application
    Intent newIntent = new Intent();
    newIntent.putExtra(PackageUtil.INTENT_ATTR_APPLICATION_INFO,
            mPkgInfo.applicationInfo);
    newIntent.setData(mPackageURI);
    // 设置Intent中的class为 InstallInstalling，用来进行Activity跳转
    // class InstallInstalling extends AlertActivity  [同学们注意] 下面会分析
InstallInstalling Activity
    newIntent.setClass(this, InstallInstalling.class);
    String installerPackageName = getIntent().getStringExtra(
            Intent.EXTRA_INSTALLER_PACKAGE_NAME);
    if (mOriginatingURI != null) {
        newIntent.putExtra(Intent.EXTRA_ORIGINATING_URI, mOriginatingURI);
    }
    if (mReferrerURI != null) {
        newIntent.putExtra(Intent.EXTRA_REFERRER, mReferrerURI);
    }
    if (mOriginatingUid != PackageInstaller.SessionParams.UID_UNKNOWN) {
        newIntent.putExtra(Intent.EXTRA_ORIGINATING_UID, mOriginatingUid);
    }
    if (installerPackageName != null) {
        newIntent.putExtra(Intent.EXTRA_INSTALLER_PACKAGE_NAME,
                installerPackageName);
    }
    if (getIntent().getBooleanExtra(Intent.EXTRA_RETURN_RESULT, false)) {
        newIntent.putExtra(Intent.EXTRA_RETURN_RESULT, true);
    }
    newIntent.addFlags(Intent.FLAG_ACTIVITY_FORWARD_RESULT);
    if(localLOGV) Log.i(TAG, "downloaded app uri="+mPackageURI);
    startActivity(newIntent);
    finish();
}
```

启动 **InstallInstalling**，进入onCreate, 重点是看onCreate函数中的**六步**:

```
protected void onCreate(@Nullable Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    ApplicationInfo appInfo = getIntent()
            .getParcelableExtra(PackageUtil.INTENT_ATTR_APPLICATION_INFO);
    mPackageURI = getIntent().getData();

    if ("package".equals(mPackageURI.getScheme())) {
        try {
            getPackageManager().installExistingPackage(appInfo.packageName);
            launchSuccess();
        } catch (PackageManager.NameNotFoundException e) {
            launchFailure(PackageManager.INSTALL_FAILED_INTERNAL_ERROR, null);
        }
    } else {
        //根据mPackageURI创建一个对应的File
        final File sourceFile = new File(mPackageURI.getPath());
        PackageUtil.AppSnippet as = PackageUtil.getAppSnippet(this, appInfo,
sourceFile);

        mAlert.setIcon(as.icon);
        mAlert.setTitle(as.label);
        mAlert.setView(R.layout.install_content_view);
        mAlert.setButton(DialogInterface.BUTTON_NEGATIVE,
getString(R.string.cancel),
                (ignored, ignored2) -> {
                    if (mInstallingTask != null) {
                        mInstallingTask.cancel(true);
                    }

                    if (mSessionId > 0) {
```

```
 getPackageManager().getPackageInstaller().abandonSession(mSessionId);
                        mSessionId = 0;
                    }

                    setResult(RESULT_CANCELED);
                    finish();
                }, null);
        setupAlert();
        requireViewById(R.id.installing).setVisibility(View.VISIBLE);

        // 第一步.如果savedInstanceState不为null，获取此前保存的mSessionId和
mInstallId，其中mSessionId是安装包的会话id，mInstallId是等待的安装事件id
        if (savedInstanceState != null) {
            mSessionId = savedInstanceState.getInt(SESSION_ID);
            mInstallId = savedInstanceState.getInt(INSTALL_ID);

            // Reregister for result; might instantly call back if result was
delivered while
            // activity was destroyed
            try {
                // 第二步.根据mInstallId向InstallEventReceiver注册一个观察者，
launchFinishBasedOnResult会接收到安装事件的回调，
                //无论安装成功或者失败都会关闭当前的Activity(InstallInstalling)。如果
savedInstanceState为null，代码的逻辑也是类似的
                InstallEventReceiver.addObserver(this, mInstallId,
                        this::launchFinishBasedOnResult);
            } catch (EventResultPersister.OutOfIdsException e) {
                // Does not happen
            }
        } else {
            // 第三步.创建SessionParams，它用来代表安装会话的参数,组装params
            PackageInstaller.SessionParams params = new
PackageInstaller.SessionParams(
                    PackageInstaller.SessionParams.MODE_FULL_INSTALL);
            params.setInstallAsInstantApp(false);

 params.setReferrerUri(getIntent().getParcelableExtra(Intent.EXTRA_REFERRER));
            params.setOriginatingUri(getIntent()
                    .getParcelableExtra(Intent.EXTRA_ORIGINATING_URI));

 params.setOriginatingUid(getIntent().getIntExtra(Intent.EXTRA_ORIGINATING_UID,
                    UID_UNKNOWN));
            params.setInstallerPackageName(getIntent().getStringExtra(
                    Intent.EXTRA_INSTALLER_PACKAGE_NAME));
            params.setInstallReason(PackageManager.INSTALL_REASON_USER);

            // 第四步.根据mPackageUri对包（APK）进行轻量级的解析，并将解析的参数赋值给
SessionParams
            File file = new File(mPackageURI.getPath());
            try {
                PackageParser.PackageLite pkg =
PackageParser.parsePackageLite(file, 0);
                params.setAppPackageName(pkg.packageName);
                params.setInstallLocation(pkg.installLocation);
                params.setSize(
                        PackageHelper.calculateInstalledSize(pkg, false,
params.abiOverride));
```

```
            } catch (PackageParser.PackageParserException e) {
                Log.e(LOG_TAG, "Cannot parse package " + file + ": Assuming
defaults.");
                Log.e(LOG_TAG,
                        "Cannot calculate installed size " + file + ". Try only
apk size.");
                params.setSize(file.length());
            } catch (IOException e) {
                Log.e(LOG_TAG,
                        "Cannot calculate installed size " + file + ". Try only
apk size.");
                params.setSize(file.length());
            }

            try {
                // 第五步.向InstallEventReceiver注册一个观察者返回一个新的mInstallId,
                //其中InstallEventReceiver继承自BroadcastReceiver，用于接收安装事件并
回调给EventResultPersister。
                mInstallId = InstallEventReceiver
                        .addObserver(this, EventResultPersister.GENERATE_NEW_ID,
                                this::launchFinishBasedOnResult);
            } catch (EventResultPersister.OutOfIdsException e) {
                launchFailure(PackageManager.INSTALL_FAILED_INTERNAL_ERROR,
null);
            }

            try {
                // 第六步.PackageInstaller的createSession方法内部会通过
IPackageInstaller与PackageInstallerService进行进程间通信，
                //最终调用的是PackageInstallerService的createSession方法来创建并返回
mSessionId
                mSessionId =
getPackageManager().getPackageInstaller().createSession(params);
            } catch (IOException e) {
                launchFailure(PackageManager.INSTALL_FAILED_INTERNAL_ERROR,
null);
            }
        }

        mCancelButton = mAlert.getButton(DialogInterface.BUTTON_NEGATIVE);

        mSessionCallback = new InstallSessionCallback();
    }
}
```

同学们注意: 以上**第六步**是重点 PackageInstaller 的 createSession()内部会通过IPackageInstaller与PackageInstallerService进行进程间通信，最终调用的是PackageInstallerService的createSession方法来创建并返回mSessionId

InstallInstalling.onResume方法中，调用onPostExecute()方法，将APK的信息通过IO流的形式写入到PackageInstaller.Session中

```
protected void onResume() {
    super.onResume();
    // This is the first onResume in a single life of the activity
    if (mInstallingTask == null) {
        PackageInstaller installer = getPackageManager().getPackageInstaller();
        // 获取sessionInfo
        PackageInstaller.SessionInfo sessionInfo =
installer.getSessionInfo(mSessionId);

        if (sessionInfo != null && !sessionInfo.isActive()) {
            // 【同学们注意】 最终执行onPostExecute() 下面来分析
            // 创建内部类InstallingAsyncTask的对象，调用execute()，最终进入
onPostExecute()
            mInstallingTask = new InstallingAsyncTask();
            mInstallingTask.execute();
        } else {
            // we will receive a broadcast when the install is finished
            mCancelButton.setEnabled(false);
            setFinishOnTouchOutside(false);
        }
    }
}
```

Installinstalling.**InstallingAsyncTask**：关注 第一步 和 第二步

```
 private final class InstallingAsyncTask extends AsyncTask<Void, Void,
PackageInstaller.Session> {
    volatile boolean isDone;

    // 第一步： doInBackground()会根据包(APK)的Uri，将APK的信息通过IO流的形式写入到
PackageInstaller.Session中
    @Override
    protected PackageInstaller.Session doInBackground(Void... params) {
        PackageInstaller.Session session;
        try {
            session =
getPackageManager().getPackageInstaller().openSession(mSessionId);
        } catch (IOException e) {
            return null;
        }

        session.setStagingProgress(0);

        try {
            File file = new File(mPackageURI.getPath());

            try (InputStream in = new FileInputStream(file)) {
                long sizeBytes = file.length();
                try (OutputStream out = session
                        .openWrite("PackageInstaller", 0, sizeBytes)) {
                    byte[] buffer = new byte[1024 * 1024];
                    while (true) {
                        int numRead = in.read(buffer);

                        if (numRead == -1) {
                            session.fsync(out);
                            break;
                        }
                        if (isCancelled()) {
                            session.close();
                            break;
                        }
                        //将APK的信息通过IO流的形式写入到PackageInstaller.Session中
                        out.write(buffer, 0, numRead);
                        if (sizeBytes > 0) {
                            float fraction = ((float) numRead / (float)
sizeBytes);
                            session.addProgress(fraction);
                        }
                    }
                }
            }

            return session;
        } catch (IOException | SecurityException e) {
            Log.e(LOG_TAG, "Could not write package", e);

            session.close();

            return null;
        } finally {
            synchronized (this) {
                isDone = true;
                notifyAll();
            }
        }
    }

    // 第二步：最后在onPostExecute()中 调用PackageInstaller.Session的commit方法，进行
安装
    @Override
    protected void onPostExecute(PackageInstaller.Session session) {
        if (session != null) {
            Intent broadcastIntent = new Intent(BROADCAST_ACTION);
            broadcastIntent.setFlags(Intent.FLAG_RECEIVER_FOREGROUND);
            broadcastIntent.setPackage(getPackageName());
            broadcastIntent.putExtra(EventResultPersister.EXTRA_ID, mInstallId);

            PendingIntent pendingIntent = PendingIntent.getBroadcast(
                    InstallInstalling.this,
                    mInstallId,
                    broadcastIntent,
                    PendingIntent.FLAG_UPDATE_CURRENT);

            // 【同学们注意】commit 下面会分析
            // 调用PackageInstaller.Session的commit方法，进行安装
            session.commit(pendingIntent.getIntentSender());
            mCancelButton.setEnabled(false);
            setFinishOnTouchOutside(false);
        } else {

 getPackageManager().getPackageInstaller().abandonSession(mSessionId);

            if (!isCancelled()) {
                launchFailure(PackageManager.INSTALL_FAILED_INVALID_APK, null);
            }
        }
    }
}
```

PackageInstaller的**commit**()

```
[PackageInstaller.java] commit
public void commit(@NonNull IntentSender statusReceiver) {
    try {
        //  mSession的类型为IPackageInstallerSession，这说明要通过
IPackageInstallerSession来进行进程间的通信，最终会调用PackageInstallerSession的commit
方法，这样代码逻辑就到了Java框架层的。
        // 调用IPackageInstallerSession的commit方法, 跨进程调用到
PackageInstallerSession.commit()
        mSession.commit(statusReceiver, false);
    } catch (RemoteException e) {
        throw e.rethrowFromSystemServer();
    }
}
```

PackageInstallerSession.**commit()**中

```
[PackageInstallerSession.java] commit()
public void commit(@NonNull IntentSender statusReceiver, boolean forTransfer) {
    if (mIsPerfLockAcquired && mPerfBoostInstall != null) {
        mPerfBoostInstall.perfLockRelease();
        mIsPerfLockAcquired = false;
    }
    ...
    // 调用markAsCommitted()
    if (!markAsCommitted(statusReceiver, forTransfer)) {
        return;
    }
    ...
    // 【同学们注意】向Handler发送一个类型为MSG_COMMIT的消息 ，下面会分析
    mHandler.obtainMessage(MSG_COMMIT).sendToTarget();
}
```

MSG_COMMIT在handler中进行处理，进入handleCommit()

```
public boolean handleMessage(Message msg) {
    switch (msg.what) {
        case MSG_COMMIT:
            handleCommit();
            break;
    }
}

private void handleCommit() {
    ...
    List<PackageInstallerSession> childSessions = getChildSessions();

    try {
        synchronized (mLock) {
            //最终调用installStage()，进入PKMS
            commitNonStagedLocked(childSessions);
        }
    } catch (PackageManagerException e) {
        final String completeMsg = ExceptionUtils.getCompleteMessage(e);
        Slog.e(TAG, "Commit of session " + sessionId + " failed: " +
completeMsg);
        destroyInternal();
        dispatchSessionFinished(e.error, completeMsg, null);
    }
}
```

最终调用 **mPm.installStage()**，进入PKMS 【经过千辛万苦，终于要进入PKMS了】

```
private void commitNonStagedLocked(...)throws PackageManagerException {
    if (isMultiPackage()) {
        ...
        mPm.installStage(activeChildSessions); // 【同学们注意】跨越进程 进入
PKMS.installStage了
    } else {
        mPm.installStage(committingSession);
    }
}
```

PKMS.installStage

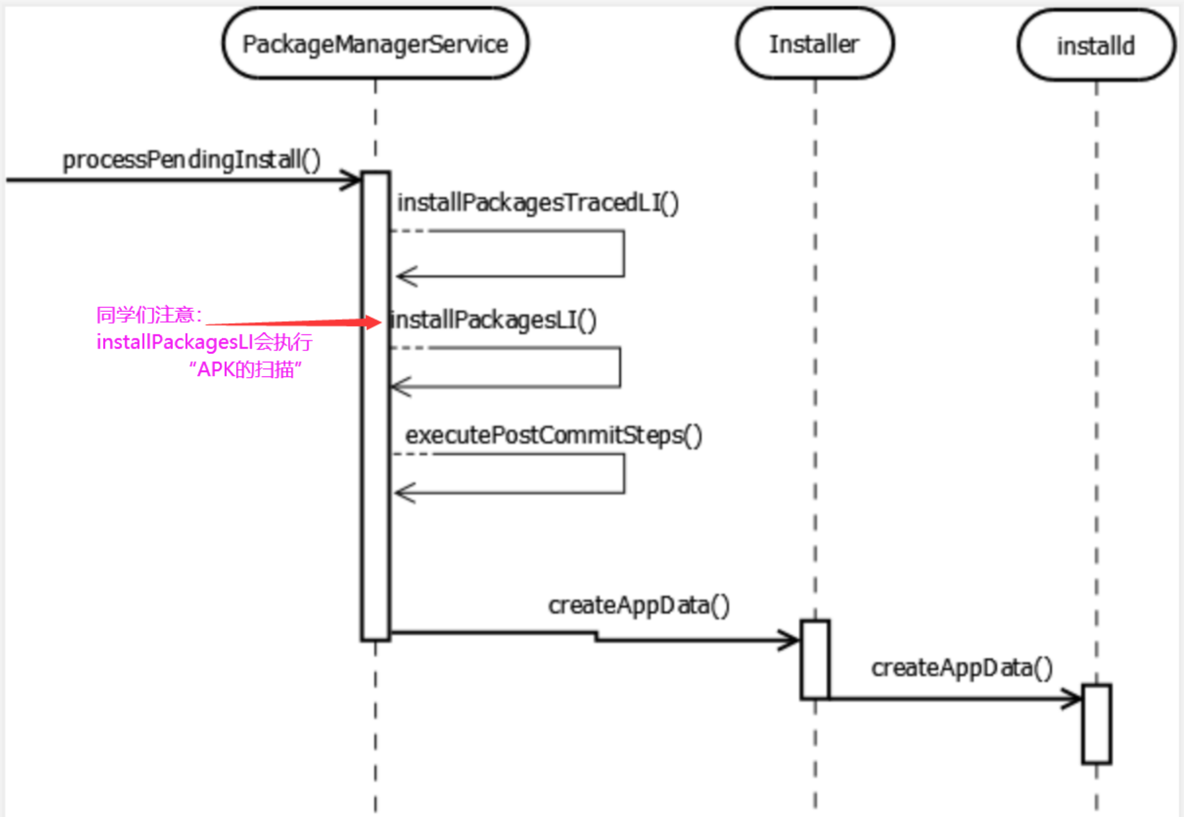
```
[PackageManagerService.java]
void installStage(ActiveInstallSession activeInstallSession) {
    if (DEBUG_INSTANT) {
        if ((activeInstallSession.getSessionParams().installFlags
                & PackageManager.INSTALL_INSTANT_APP) != 0) {
            Slog.d(TAG, "Ephemeral install of " +
activeInstallSession.getPackageName());
        }
    }
    // 第一步.创建了类型为INIT_COPY的消息
    final Message msg = mHandler.obtainMessage(INIT_COPY);

    // 第二步.创建InstallParams，它对应于包的安装数据
    final InstallParams params = new InstallParams(activeInstallSession);

 params.setTraceMethod("installStage").setTraceCookie(System.identityHashCode(pa
rams));
    msg.obj = params;

    Trace.asyncTraceBegin(TRACE_TAG_PACKAGE_MANAGER, "installStage",
            System.identityHashCode(msg.obj));
    Trace.asyncTraceBegin(TRACE_TAG_PACKAGE_MANAGER, "queueInstall",
            System.identityHashCode(msg.obj));
    // 第三步.将InstallParams通过消息发送出去。
    mHandler.sendMessage(msg);
}

对INIT_COPY的消息的处理
[PackageManagerService.java]
void doHandleMessage(Message msg) {
    switch (msg.what) {
        case INIT_COPY: {
            HandlerParams params = (HandlerParams) msg.obj;
            if (params != null) {
                if (DEBUG_INSTALL) Slog.i(TAG, "init_copy: " + params);
                Trace.asyncTraceEnd(TRACE_TAG_PACKAGE_MANAGER, "queueInstall",
                        System.identityHashCode(params));
                Trace.traceBegin(TRACE_TAG_PACKAGE_MANAGER, "startCopy");
                // 【同学们注意】执行APK拷贝动作，这里会执行到  final void startCopy()
                params.startCopy();
                Trace.traceEnd(TRACE_TAG_PACKAGE_MANAGER);
            }
            break;
        }
    }
}


        [PKMS.HandlerParams]
        final void startCopy() {
            if (DEBUG_INSTALL) Slog.i(TAG, "startCopy " + mUser + ": " + this);
            handleStartCopy();
            handleReturnCode();  // 调用到下面 handleReturnCode
        }

        [PKMS.MultiPackageInstallParams]
        void handleReturnCode() {
            if (mVerificationCompleted && mEnableRollbackCompleted) {
                .....
                if (mRet == PackageManager.INSTALL_SUCCEEDED) {
                    mRet = mArgs.copyApk(); // 【同学们注意】 下面会说到 copyApk
                }
                .....
            }
}
```

APK 拷贝 方法调用步骤如下：

```
PKMS
    copyApk()
    doCopyApk()


                PackageManagerServiceUtils
                    copyPacakge()
                    copyFile()

// TODO 通过文件流的操作，把APK拷贝到/data/app等目录
```

```
private static void copyFile(String sourcePath, File targetDir, String
targetName)
        throws ErrnoException, IOException {
    if (!FileUtils.isValidExtFilename(targetName)) {
        throw new IllegalArgumentException("Invalid filename: " + targetName);
    }
    Slog.d(TAG, "Copying " + sourcePath + " to " + targetName);

    final File targetFile = new File(targetDir, targetName);
    final FileDescriptor targetFd = Os.open(targetFile.getAbsolutePath(),
            O_RDWR | O_CREAT, 0644);
    Os.chmod(targetFile.getAbsolutePath(), 0644);
    FileInputStream source = null;
    try {
        source = new FileInputStream(sourcePath);
        FileUtils.copy(source.getFD(), targetFd);
    } finally {
        IoUtils.closeQuietly(source);
    }
}
```

进入 Android 10.0 核心安装环节：



processPendingInstall：

```
private void processPendingInstall(final InstallArgs args, final int
currentStatus) {
    if (args.mMultiPackageInstallParams != null) {
        args.mMultiPackageInstallParams.tryProcessInstallRequest(args,
currentStatus);
    } else {
        //1.设置安装参数
        PackageInstalledInfo res = createPackageInstalledInfo(currentStatus);
        //2.创建一个新线程，处理安装参数，进行安装
        processInstallRequestsAsync(
                res.returnCode == PackageManager.INSTALL_SUCCEEDED,
                Collections.singletonList(new InstallRequest(args, res)));
    }
}

private void processInstallRequestsAsync(boolean success,
        List<InstallRequest> installRequests) {
    mHandler.post(() -> {
        if (success) {
            for (InstallRequest request : installRequests) {
                //1.如果之前安装失败，清除无用信息
                request.args.doPreInstall(request.installResult.returnCode);
            }
            synchronized (mInstallLock) {
                //2. installPackagesTracedLI 是安装过程的核心方法，然后调用
installPackagesLI 进行安装。
                // 【同学们注意】下面会分析此函数 installPackagesTracedLI
                installPackagesTracedLI(installRequests);
            }
            for (InstallRequest request : installRequests) {
                //3.如果之前安装失败，清除无用信息
                request.args.doPostInstall(
                        request.installResult.returnCode,
request.installResult.uid);
            }
        }
        for (InstallRequest request : installRequests) {
            restoreAndPostInstall(request.args.user.getIdentifier(),
request.installResult,
                    new PostInstallData(request.args, request.installResult,
null));
        }
    });
}
```

installPackagesTracedLI

```
private void installPackagesLI(List<InstallRequest> requests) {
    ...
    // 环节一.Prepare 准备：分析任何当前安装状态，分析包并对其进行初始验证。
    prepareResult = preparePackageLI(request.args, request.installResult);
    ...
    // 环节二.Scan  扫描：考虑到prepare中收集的上下文，询问已分析的包。
    final List<ScanResult> scanResults = scanPackageTracedLI(
                            prepareResult.packageToScan,
prepareResult.parseFlags,
                            prepareResult.scanFlags, System.currentTimeMillis(),
                            request.args.user);
    ...
    // 环节三.Reconcile 调和：在彼此的上下文和当前系统状态中验证扫描的包，以确保安装成功。
    ReconcileRequest reconcileRequest = new ReconcileRequest(preparedScans,
installArgs,
            installResults,
            prepareResults,
            mSharedLibraries,
            Collections.unmodifiableMap(mPackages), versionInfos,
            lastStaticSharedLibSettings);
    ...
    // 环节四.Commit 提交：提交所有扫描的包并更新系统状态。这是安装流中唯一可以修改系统状态的
地方，必须在此阶段之前确定所有可预测的错误。
    commitPackagesLocked(commitRequest);
    ...
    // 环节五.完成APK的安装【同学们注意：下面会分析这个操作】
    executePostCommitSteps(commitRequest);
}
```

executePostCommitSteps 安装APK,并为新的代码路径准备应用程序配置文件,并再次检查是否需要**dex优化**

如果是直接安装新包，会为新的代码路径准备应用程序配置文件

如果是替换安装：其主要过程为更新设置，清除原有的某些APP数据，重新生成相关的app数据目录等步骤，同时要区分系统应用替换和非系统应用替换。而安装新包：则直接更新设置，生成APP数据即可。

```
[PackageManagerService.java] executePostCommitSteps()
private void executePostCommitSteps(CommitRequest commitRequest) {
    for (ReconciledPackage reconciledPkg :
commitRequest.reconciledPackages.values()) {
        ...
        //1)进行安装
        prepareAppDataAfterInstallLIF(pkg);
        //2)如果需要替换安装，则需要清楚原有的APP数据
        if (reconciledPkg.prepareResult.clearCodeCache) {
            clearAppDataLIF(pkg, UserHandle.USER_ALL, FLAG_STORAGE_DE |
FLAG_STORAGE_CE
                    | FLAG_STORAGE_EXTERNAL |
Installer.FLAG_CLEAR_CODE_CACHE_ONLY);
        }


        //3)为新的代码路径准备应用程序配置文件。这需要在调用dexopt之前完成，以便任何安装时配
置文件都可以用于优化。
        mArtManagerService.prepareAppProfiles(
                pkg,
                resolveUserIds(reconciledPkg.installArgs.user.getIdentifier()),
                /* updateReferenceProfileContent= */ true);


        final boolean performDexopt =
                (!instantApp || Global.getInt(mContext.getContentResolver(),
                Global.INSTANT_APP_DEXOPT_ENABLED, 0) != 0)
                && ((pkg.applicationInfo.flags &
ApplicationInfo.FLAG_DEBUGGABLE) == 0);

        if (performDexopt) {
            ...
            //4)执行dex优化
            mPackageDexOptimizer.performDexOpt(pkg,
                    null /* instructionSets */,
                    getOrCreateCompilerPackageStats(pkg),
                    mDexManager.getPackageUseInfoOrDefault(packageName),
                    dexoptOptions);
        }

        BackgroundDexOptService.notifyPackageChanged(packageName);
    }
}
```

prepareAppDataAfterInstallLIF:

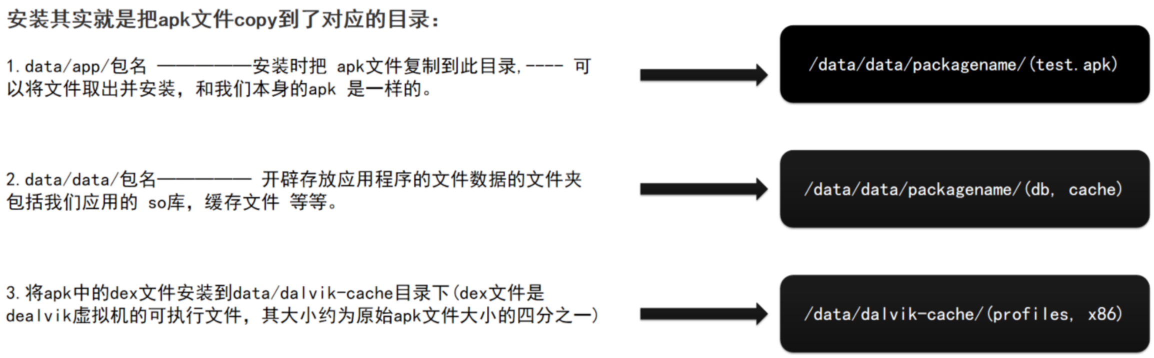
```
通过一系列的调用，最终会调用到[Installer.java] createAppData()进行安装，交给installed进程
进行APK的安装
调用栈如下：

prepareAppDataAfterInstallLIF()
        |
prepareAppDataLIF()
        |
prepareAppDataLeafLIF()
        |
[Installer.java]
        createAppData()
```

```
private void prepareAppDataAfterInstallLIF(PackageParser.Package pkg) {
    ...
    for (UserInfo user : um.getUsers()) {
        ...
        if (ps.getInstalled(user.id)) {
            // TODO: when user data is locked, mark that we're still dirty
            prepareAppDataLIF(pkg, user.id, flags);
        }
    }
}

private void prepareAppDataLIF(PackageParser.Package pkg, int userId, int flags)
{
    if (pkg == null) {
        Slog.wtf(TAG, "Package was null!", new Throwable());
        return;
    }
    prepareAppDataLeafLIF(pkg, userId, flags);
    final int childCount = (pkg.childPackages != null) ?
pkg.childPackages.size() : 0;
    for (int i = 0; i < childCount; i++) {
        prepareAppDataLeafLIF(pkg.childPackages.get(i), userId, flags);
    }
}

private void prepareAppDataLeafLIF(PackageParser.Package pkg, int userId, int
flags) {
    ...
    try {
        // 调用Installd守护进程的入口
        ceDataInode = mInstaller.createAppData(volumeUuid, packageName, userId,
flags,
                appId, seInfo, app.targetSdkVersion);
    } catch (InstallerException e) {
        if (app.isSystemApp()) {
            destroyAppDataLeafLIF(pkg, userId, flags);
            try {
                ceDataInode = mInstaller.createAppData(volumeUuid, packageName,
userId, flags,
                        appId, seInfo, app.targetSdkVersion);
            } catch (InstallerException e2) {
                ...
            }
        }
    }
}
```

Installer.**createAppData** 收尾工作，安装完成后，更新设置，更新安装锁等：

```
[Installer.java]
public long createAppData(String uuid, String packageName, int userId, int
flags, int appId,
        String seInfo, int targetSdkVersion) throws InstallerException {
    if (!checkBeforeRemote()) return -1;
    try {
        // mInstalld 为IInstalld的对象，即通过Binder调用到 进程installd，最终调用
installd的createAppData()
        // 【同学们注意】 mInstalld是一个aidl文件，通过此aidl文件调用到 Binder机制的服务
端，服务端哪里要操控底层....
        return mInstalld.createAppData(uuid, packageName, userId, flags, appId,
seInfo,
                targetSdkVersion);
    } catch (Exception e) {
        throw InstallerException.from(e);
    }
}
```

总结：安装的原理：



安装其实就是把apk文件copy到了对应的目录：

1. data/app/包名 ————————安装时把 apk文件复制到此目录，---- 可以将文件取出并安装，和我们本身的apk 是一样的。

/data/data/packagename/(test.apk)

2. data/data/包名———————— 开辟存放应用程序的文件数据的文件夹 包括我们应用的 so库，缓存文件 等等。

/data/data/packagename/(db, cache)

3. 将apk中的dex文件安装到data/dalvik-cache目录下(dex文件是dealvik虚拟机的可执行文件，其大小约为原始apk文件大小的四分之一)

/data/dalvik-cache/(profiles, x86)

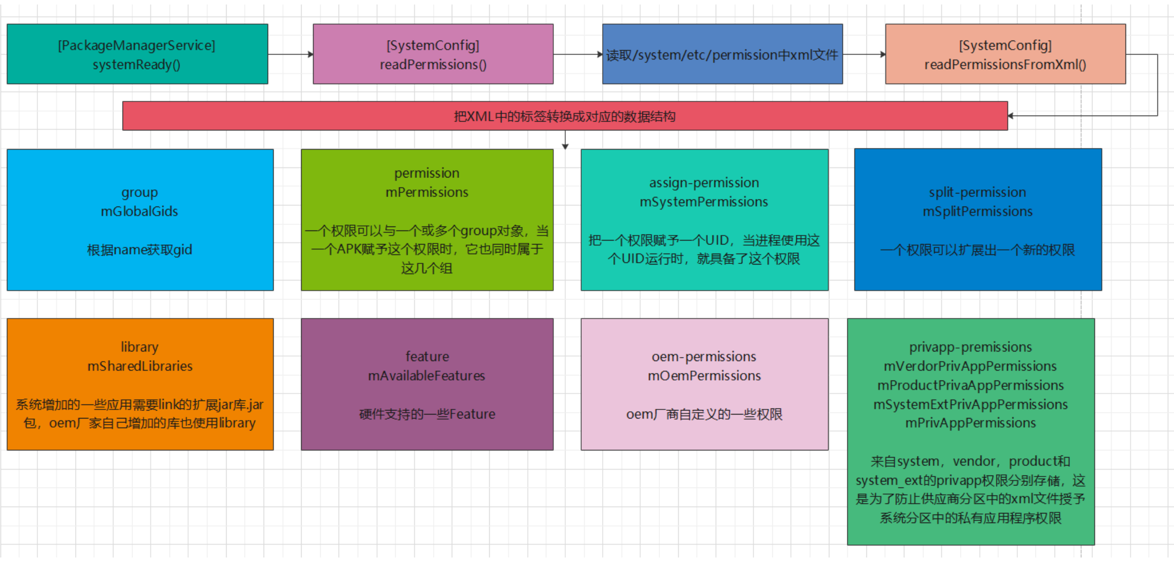
五部曲 - PMS之权限扫描

此 “PMS之权限扫描” 学习的目标是： PackageManagerService中执行systemReady()后，需求对/system/etc/permissions中的各种xml进行扫描，进行相应的权限存储，让以后可以使用，这就是本次“PMS只权限扫描”学习的目的

权限扫描:

PackageManagerService执行systemReady()时，通过SystemConfig的readPermissionsFromXml()来扫描读取/system/etc/permissions中的xml文件,包括platform.xml和系统支持的各种硬件模块的feature主要工作:

整体图:


[PackageManagerService]
systemReady()
[SystemConfig]
readPermissions()
读取/system/etc/permission中xml文件
[SystemConfig]
readPermissionsFromXml()
把XML中的标签转换成对应的数据结构
group
mGlobalGids
根据name获取gid
permission
mPermissions
一个权限可以与一个或多个group对象，当一个APK赋予这个权限时，它也同时属于这几个组
assign-permission
mSystemPermissions
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
split-permission
mSplitPermissions
一个权限可以扩展出一个新的权限
library
mSharedLibraries
系统增加的一些应用需要link的扩展jar库.jar包，oem厂家自己增加的库也使用library
feature
mAvailableFeatures
硬件支持的一些Feature
oem-permissions
mOemPermissions
oem厂商自定义的一些权限
privapp-premissions
mVerdorPrivAppPermissions
mProductPrivaAppPermissions
mSystemExtPrivAppPermissions
mPrivAppPermissions
来自system，vendor，product和system_ext的privapp权限分别存储，这是为了防止供应商分区中的xml文件授予系统分区中的私有应用程序权限

SystemConfig 的 readPermissions函数:

此函数目的： （扫描/system/etc/permissions中文件，调用readPermissionsFromXml()进行解析，存入SsytemConfig相应的成员数组变量中)

```
void readPermissions(File libraryDir, int permissionFlag) {
    ...
    // Iterate over the files in the directory and scan .xml files
    File platformFile = null;
    for (File f : libraryDir.listFiles()) {
        if (!f.isFile()) {
            continue;
        }

        // 最后读取platform.xml
        if (f.getPath().endsWith("etc/permissions/platform.xml")) {
            continue;
            platformFile = f;
        }
        ...
        readPermissionsFromXml(f, permissionFlag);
    }
    // Read platform permissions last so it will take precedence
    if (platformFile != null) {
        readPermissionsFromXml(platformFile, permissionFlag);
    }
}
```

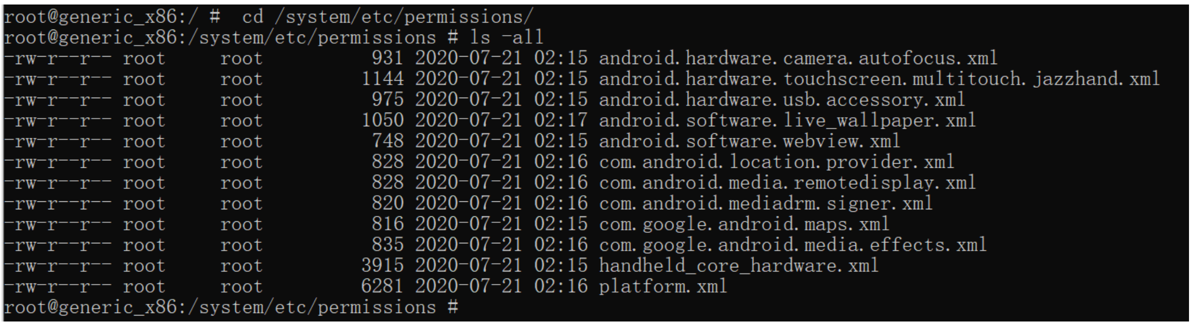
解析xml的标签节点，存入mGlobalGids、mPermissions、mSystemPermissions等成员变量中，供其他进行调用

```
private void readPermissionsFromXml(File permFile, int permissionFlag) {
    FileReader permReader = null;
    permReader = new FileReader(permFile);
    ...
    XmlPullParser parser = Xml.newPullParser();
    parser.setInput(permReader);

    while (true) {
        ...
        String name = parser.getName();
         switch (name) {
             //解析 group 标签，前面介绍的 XML 文件中没有单独使用该标签的地方
            case "group": {
                String gidStr = parser.getAttributeValue(null, "gid");
                if (gidStr != null) {
                    int gid = android.os.Process.getGidForName(gidStr);
                    //转换 XML 中的 gid字符串为整型，并保存到 mGlobalGids 中
                    mGlobalGids = appendInt(mGlobalGids, gid);
                } else {
                    Slog.w(TAG, "<" + name + "> without gid in " + permFile + "
at "  + parser.getPositionDescription());
                }
                ...
            }
            break;
            case "permission": { //解析 permission 标签
                if (allowPermissions) {
                    String perm = parser.getAttributeValue(null, "name");
                    if (perm == null) {
                        Slog.w(TAG, "<" + name + "> without name in " + permFile
+ " at " + parser.getPositionDescription());
                        XmlUtils.skipCurrentTag(parser);
                        break;
                    }
                    perm = perm.intern();
                    readPermission(parser, perm); //调用 readPermission 处理,存入
mPermissions
                } else {
                    logNotAllowedInPartition(name, permFile, parser);
                    XmlUtils.skipCurrentTag(parser);
                }
            } break;
         }
    }
}
```

查看 XML文件：

adb devices

adb shell



```
root@generic_x86:/ #  cd /system/etc/permissions/
root@generic_x86:/system/etc/permissions # ls -all
-rw-r--r-- root     root          931 2020-07-21 02:15 android.hardware.camera.autofocus.xml
-rw-r--r-- root     root         1144 2020-07-21 02:15 android.hardware.touchscreen.multitouch.jazzhand.xml
-rw-r--r-- root     root          975 2020-07-21 02:15 android.hardware.usb.accessory.xml
-rw-r--r-- root     root         1050 2020-07-21 02:17 android.software.live_wallpaper.xml
-rw-r--r-- root     root          748 2020-07-21 02:15 android.software.webview.xml
-rw-r--r-- root     root          828 2020-07-21 02:16 com.android.location.provider.xml
-rw-r--r-- root     root          828 2020-07-21 02:16 com.android.media.remotedisplay.xml
-rw-r--r-- root     root          820 2020-07-21 02:16 com.android.mediadrm.signer.xml
-rw-r--r-- root     root          816 2020-07-21 02:15 com.google.android.maps.xml
-rw-r--r-- root     root          835 2020-07-21 02:16 com.google.android.media.effects.xml
-rw-r--r-- root     root         3915 2020-07-21 02:15 handheld_core_hardware.xml
-rw-r--r-- root     root         6281 2020-07-21 02:16 platform.xml
root@generic_x86:/system/etc/permissions #
```

然后在导出去：

```
adb pull /system/etc/permissions
```

/system/etc/permissions中会存在很多的xml文件，例如我们看下 android.software.webview.xml的文件，内容如下：

里面只只有一个feature "android.software.webview",大部分的xml都是类似的定义方式

```
<?xml version="1.0" encoding="utf-8"?>
<permissions>
    <feature name="android.software.webview" />
</permissions>
```

让我们来简单的看下/system/etc/permissions/platform.xml的内容

```
<?xml version="1.0" encoding="utf-8"?>
<permissions>
    <permission name="android.permission.BLUETOOTH_ADMIN" >
        <group gid="net_bt_admin" />
    </permission>
    <permission name="android.permission.INTERNET" >
        <group gid="inet" />
    </permission>
    <permission name="android.permission.READ_LOGS" >
        <group gid="log" />
    </permission>
  ...
    <assign-permission name="android.permission.MODIFY_AUDIO_SETTINGS"
uid="media" />
    <assign-permission name="android.permission.ACCESS_SURFACE_FLINGER"
uid="media" />
    <assign-permission name="android.permission.WAKE_LOCK" uid="media" />
  ...
```

```
   <split-permission name="android.permission.ACCESS_FINE_LOCATION">
        <new-permission name="android.permission.ACCESS_COARSE_LOCATION" />
    </split-permission>
    <split-permission name="android.permission.WRITE_EXTERNAL_STORAGE">
        <new-permission name="android.permission.READ_EXTERNAL_STORAGE" />
    </split-permission>
    <split-permission name="android.permission.READ_CONTACTS"
                      targetSdk="16">
        <new-permission name="android.permission.READ_CALL_LOG" />
    </split-permission>
  ...
    <library name="android.test.base"
            file="/system/framework/android.test.base.jar" />
    <library name="android.test.mock"
            file="/system/framework/android.test.mock.jar"
            dependency="android.test.base" />
    <library name="android.test.runner"
            file="/system/framework/android.test.runner.jar"
            dependency="android.test.base:android.test.mock" />

    <!-- In BOOT_JARS historically, and now added to legacy applications. -->
    <library name="android.hidl.base-V1.0-java"
            file="/system/framework/android.hidl.base-V1.0-java.jar" />
    <library name="android.hidl.manager-V1.0-java"
            file="/system/framework/android.hidl.manager-V1.0-java.jar"
            dependency="android.hidl.base-V1.0-java" />
  ...
</permissions>
```

以上platform.xml中出现的标签种类则较为多样，它们的含义分别是：

```
platform.xml中出现的标签种类则较为多样，它们的含义分别是：

<group>:根据name获取gid

<permission >标签：把属性name所描述的权限赋予给<group>标签中属性gid所表示的用户组，一个权限可以有一个或多个group对象，当一个APK授权于这个这个权限时，它同时属于这几个组

<assign-permission>标签：把属性name所描述的权限赋予给uid属性所表示的用户

<split-permission>标签：一个权限可以扩展出一个新的权限

<library>标签：除framework中动态库以外的，所有系统会自动加载的动态库

<feature>标签：硬件支持的一些feature

<oem-permission>标签：oem厂商自己定义的一些权限

<privapp-permission>标签：来自system、vendor、product、system_ext的privapp权限分别存储，这是防止供应商分区中的xml授权于系统分区中的私有应用权限



最后将上面xml解析出来的数据做如下存储：

<group>标签gid属性的值会存放在mGlobalGids数组中；
```

```
<permission>标签,解析得到的值会存放在mPermissions集合中;

<assign-permission>标签解析得到的值会存放在mSystemPermissions中;

<split-permission>存储在mSplitPermissions

<library>标签解析得到的值会存放在mSharedLibraries中;

<feature>存储在mAvaliableFeatures

<oem-permission>存储在mOemPermissions

<privapp-permission>会根据不同的存储路径，分别存储在mVendorPrivAppPermissions、
mProductPrivAppPermissions、mSystemExtPrivAppPermissions、mPrivAppPermissions
```

总结：权限扫描，扫描/system/etc/permissions中的xml，存入相应的结构体中，供之后权限管理使用

===

同学们注意：此次 ”PackageManagerService大综合笔记-6.0“，只是给同学们写出来，不是授课内容哦

PackageManagerService大综合笔记-6.0

同学们我们开始进入前戏步骤:

首先 PackageManagerService 简称 PKMS

PKMS 是什么东西?

PackageManagerService（简称PMS）
安装
卸载
应用信息查询

答: PackageManagerService（简称 PKMS），是 Android 系统中核心服务之一，负责应用程序的**安装**，**卸载**，**信息查询**，等工作。

PKMS 概述信息:

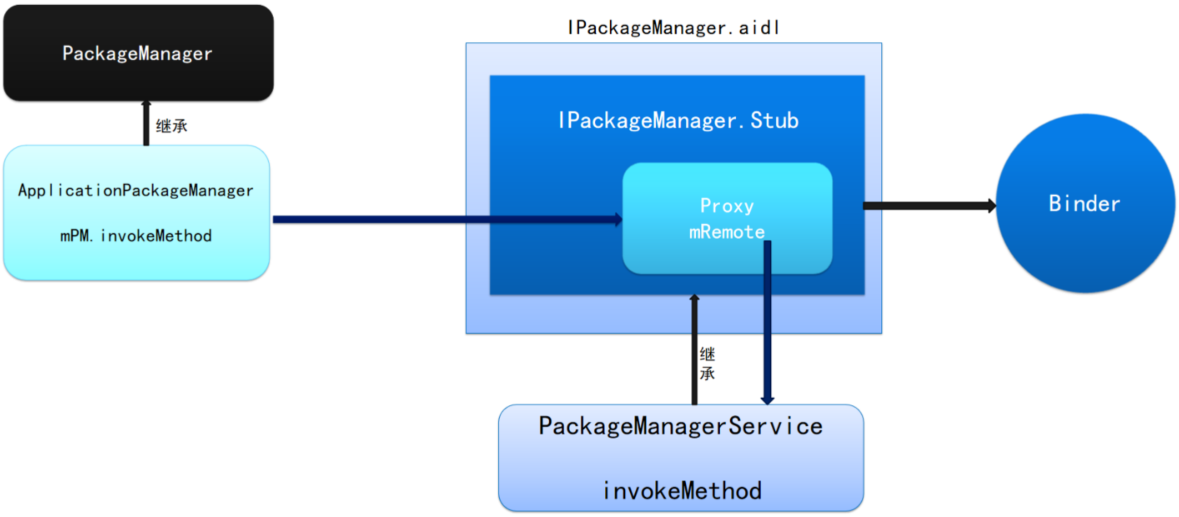
Android系统启动时，会启动（应用程序管理服务器PKMS），此服务负责扫描系统中特定的目录，寻找里面的APK格式的文件，并对这些文件进行解析，然后得到应用程序相关信息，最后完成应用程序的安装

PKMS在安装应用过程中, 会全面解析应用程序的AndroidManifest.xml文件, 来得到Activity, Service, BroadcastReceiver, ContextProvider 等信息, 在结合PKMS服务就可以在OS中正常的使用应用程序了

在Android系统中, 系统启动时由SystemServer启动PKMS服务, 启动该服务后会执行应用程序的安装过程,

接下来后就会重点的介绍 (SystemServer启动PKMS服务的过程, 讲解在Android系统中安装应用程序的过程)

一部曲 - PKMS角色位置：



同学们注意：客户端可通过Context.getPackageManager()获得ApplicationPackageManager对象, 而mPM指向的是Proxy代理，当调用到mPM.方法后，将会调用到IPackageManager的Proxy代理方法，然后通过Binder机制中的mRemote与服务端PackageManagerService通信 并调用到PackageManagerService的方法;

自我总结：PKMS是属于Binder机制的服务端角色

接下来，我们就自己来手写一个简单的PKMS，同学们好不好

二部曲 - PKMS之前启动流程源码分析：

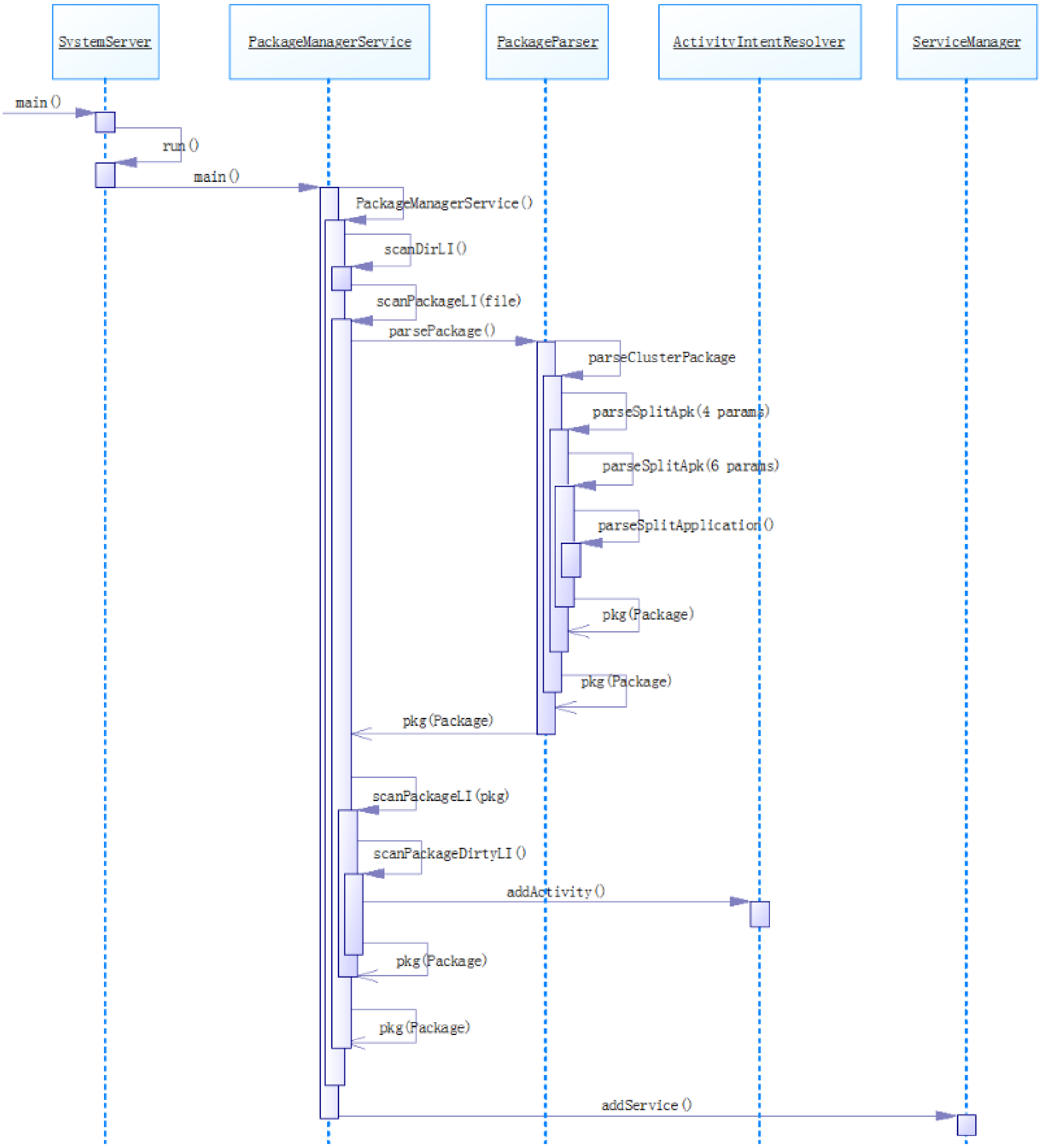
一、概述

目标：本文主要分析PKMS的初始化过程；

版本：Android 23

PKMS启动流程：Zygote --> SystemServer --> PackageManagerService(PMS)

二、PKMS初始化时序图



三、PKMS源码

在Android系统中, 通过Zygote进程启动 SystemServer组件时会调用主函数main

此类目录: **frameworks/base/services/java/com/android/server/SystemServer.java**

```
public static void main(String[] args) {
   new SystemServer().run();
}

private void run() {
    // ...省略大段代码...
    // TOOD 同学们注意: 除了启动PKMS服务之外,还启动了其他很多的服务, 例如:
ActivityManagerService等
    startBootstrapServices(); // 同学们 这是 引导服务
    startCoreServices(); // 同学们 这是 核心服务
    startOtherServices(); // 同学们 这是 其他服务
}

private void startBootstrapServices() {
     /*
      * 创建ActivityManagerService实例
      * 同学们注意：mActivityManagerService是被添加进SystemServiceManager维护起来的
      */
     mActivityManagerService = mSystemServiceManager.startService(
             ActivityManagerService.Lifecycle.class).getService();

     /*
      * 创建PackageManagerService实例;
      * 同学们注意：PackageManagerService没有被SystemServiceManager维护起来;
      * 思考：那它到底被谁维护起来了呢？
      */
     mPackageManagerService = PackageManagerService.main(mSystemContext,
installer,
             mFactoryTestMode != FactoryTest.FACTORY_TEST_OFF, mOnlyCore);
}
```

创建一个PKMS服务实例, 然后把这个服务实力添加到 ServiceManager中去, ServiceManager是Android系统Binder进程间通信机制的进程, 负责管理系统中的Binder对象

此类目录：**/frameworks/base/services/core/java/com/android/server/pm/PackageManagerService.java**

```java
public static PackageManagerService main(Context context, Installer installer,
            boolean factoryTest, boolean onlyCore) {
    .....

    PackageManagerService m = new PackageManagerService(context, installer,
            factoryTest, onlyCore);
    m.enableSystemUserPackages();

    // 在这里将PackageManagerService实例添加到ServiceManager中
    ServiceManager.addService("package", m);
    return m;
}
```

自我总结： 在SystemServer中

```
1. PackageManagerService    实例是被 ServiceManager    存储起来；
2. ActivityManagerService   实例是被 SystemServiceManager  存储起来；
```

PKMS 构造方法八件核心任务：

1. 创建 `Settings` 对象，其内部创建了 `packages.xml、packages-backup.xml、packages.list` 等文件，用于存储应用信息；
2. 在 `SystemConfig` 对象中，读取系统配置来初始化 `mGlobalGids、mSystemPermissions、mAvailableFeatures`；
3. 指定 `/data` 目录下的一系列的文件夹目录，如 `/data/data、/data/app` 等；
4. 如果 `packages.xml` 文件存在，则读取并解析该文件信息，然后保存到Settings相应的字段中；
5. 开始扫描指定目录下的apk文件，解析其Manifest文件，并将值赋值给Package对应的属性字段(这个步骤是重点)；
6. 更新所有的共享库；
7. 更新应用权限；
8. 将数据写入packages.xml中；

```java
/*
 * 该PKMS构造方法主要的作用：
 *
 * 扫描指定文件路径下的文件：
 * 1.vendorOverlayDir = "/vendor/overlay"
 * 2.frameworkDir = "/system/framework"
 * 3.privilegedAppDir = "/system/priv-app"
 * 4.systemAppDir = "/system/app"
 * 5.vendorAppDir = "/vendor/app"
 * 6.mAppInstallDir = "/data/app"
 * 7.mDrmAppPrivateInstallDir = "/data/app-private"
 */
public PackageManagerService(Context context, Installer installer, boolean
factoryTest, boolean onlyCore) {

    // ...代码省略...

    // 第一件事情.添加SharedUserSetting对象到mSettings中。
    mSettings = new Settings(mPackages);
    mSettings.addSharedUserLPw("android.uid.system", Process.SYSTEM_UID, ...);
    mSettings.addSharedUserLPw("android.uid.phone", RADIO_UID, ...);
    mSettings.addSharedUserLPw("android.uid.log", LOG_UID, ...);
    mSettings.addSharedUserLPw("android.uid.nfc", NFC_UID, ...);
    mSettings.addSharedUserLPw("android.uid.bluetooth", BLUETOOTH_UID, ...);
    mSettings.addSharedUserLPw("android.uid.shell", SHELL_UID, ...);

    // ...代码省略...

    // 第二件事情.读取系统配置来初始化mGlobalGids、mSystemPermissions、
mAvailableFeatures
    SystemConfig systemConfig = SystemConfig.getInstance();
    mGlobalGids = systemConfig.getGlobalGids();
    mSystemPermissions = systemConfig.getSystemPermissions();
    mAvailableFeatures = systemConfig.getAvailableFeatures();

    synchronized (mInstallLock) {
        // writer
        synchronized (mPackages) {
            mHandlerThread = new ServiceThread(TAG,
                    Process.THREAD_PRIORITY_BACKGROUND, true /*allowIo*/);
            mHandlerThread.start();
            mHandler = new PackageHandler(mHandlerThread.getLooper());

            /*
             * 第三件事情.指定 "/data" 目录下的文件夹，便于使用；
             * dataDir = "/data"
             * mAppDataDir = "/data/data" 应用数据存储目录
             * mAppInstallDir = "/data/app" 应用安装目录
             * mAppLib32InstallDir = "/data/app-lib"
             * mAsecInternalPath = "/data/app-asec"
             * mUserAppDataDir = "/data/user"
             * mDrmAppPrivateInstallDir = "/data/app-private"
             */
            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");

            // 第四件事情.从packages.xml文件中解析出信息(如果该文件存在)，并保存到
Settings相应的字段中;
            mRestoredSettings = mSettings.readLPw(this,
sUserManager.getUsers(false),
                    mSdkVersion, mOnlyCore);

            // ...省略大段代码...

            // 第五件事情.使用scanDirLI()扫描指定目录下的apk文件，解析其Manifest文件，并
将值赋值给Package对应的属性字段;
            // vendorOverlayDir = "/vendor/overlay"
            File vendorOverlayDir = new File(VENDOR_OVERLAY_DIR);
            // 同学们注意啦：scanDirLI()是重点，留一下。
            scanDirLI(vendorOverlayDir, PackageParser.PARSE_IS_SYSTEM
                    | PackageParser.PARSE_IS_SYSTEM_DIR, scanFlags |
SCAN_TRUSTED_OVERLAY, 0);
```

```
            /*
             * Find base frameworks (resource packages without code).
             * frameworkDir = "/system/framework"
             * framework包里都是jar包和apk
             */
            File frameworkDir = new File(Environment.getRootDirectory(),
"framework");
            scanDirLI(frameworkDir, PackageParser.PARSE_IS_SYSTEM
                    | PackageParser.PARSE_IS_SYSTEM_DIR
                    | PackageParser.PARSE_IS_PRIVILEGED,
                    scanFlags | SCAN_NO_DEX, 0);

            // Collected privileged system packages.
            // privilegedAppDir = "/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);

            // Collect ordinary system packages.
            // systemAppDir = "/system/app"
            final File systemAppDir = new File(Environment.getRootDirectory(),
"app");
            scanDirLI(systemAppDir, PackageParser.PARSE_IS_SYSTEM
                    | PackageParser.PARSE_IS_SYSTEM_DIR, scanFlags, 0);

            // Collect all vendor packages.
            // vendorAppDir = "/vendor/app"
            File vendorAppDir = new File("/vendor/app");
            try {
                vendorAppDir = vendorAppDir.getCanonicalFile();
            } catch (IOException e) {
                // failed to look up canonical path, continue with original one
            }
            scanDirLI(vendorAppDir, PackageParser.PARSE_IS_SYSTEM
                    | PackageParser.PARSE_IS_SYSTEM_DIR, scanFlags, 0);

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


            // mOnlyCore = true表示系统package
            if (!mOnlyCore) {
                // 这里扫描用户应用：mAppInstallDir = "/data/app"
                scanDirLI(mAppInstallDir, 0, scanFlags | SCAN_REQUIRE_KNOWN, 0);
                // 这里扫描用户应用：mDrmAppPrivateInstallDir = "/data/app-private"
                scanDirLI(mDrmAppPrivateInstallDir,
PackageParser.PARSE_FORWARD_LOCK,
                        scanFlags | SCAN_REQUIRE_KNOWN, 0);

                // ...省略大段代码...
            }
            // ...省略大段代码...

            // Now that we know all of the shared libraries, update all clients
to have the correct library paths.
            // 第六件事情.更新所有的共享库;
            updateAllSharedLibrariesLPw();

            // ...省略代码...
            // 第七件事情.更新应用权限;
            updatePermissionsLPw(null, null, updateFlags);
            // ...省略代码...

            // 第八件事情.将数据写入packages.xml中;
            mSettings.writeLPr();
        } // synchronized (mPackages)
    } // synchronized (mInstallLock)

    // ...省略代码...
}
```

第一件事情细节：创建`Settings`对象，其内部创建了`packages.xml、packages-backup.xml、packages.list`等文件，用于存储应用信息；

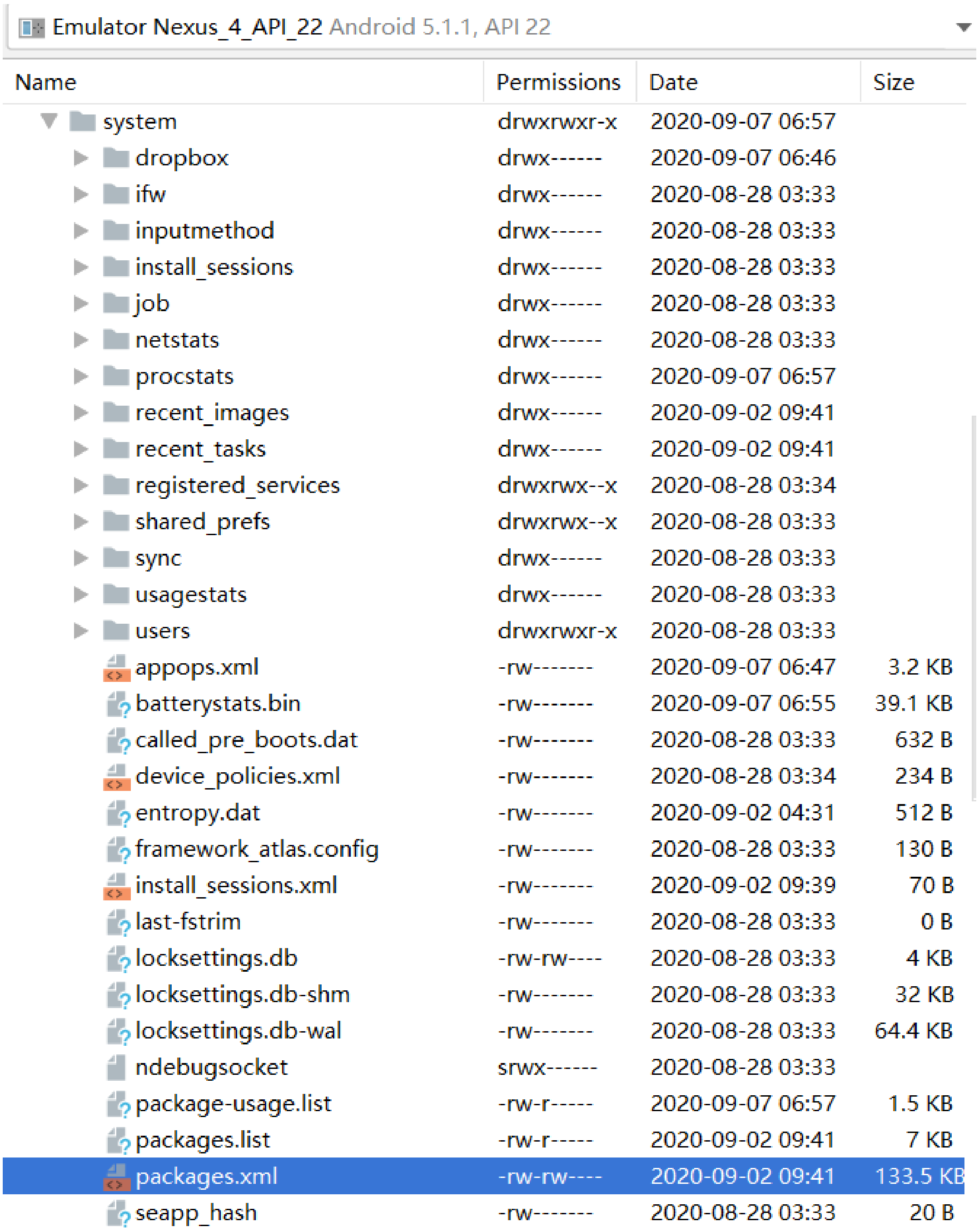
此类目录：**/frameworks/base/services/core/java/com/android/server/pm/Settings.java**

```
Settings(Context context) {
    this(context, Environment.getDataDirectory());
}

/*
 * 在 "/data/system/" 目录下创建一系列的文件;
 */
Settings(File dataDir, Object lock) {
    // 创建一个"data/system"目录
    mSystemDir = new File(dataDir, "system");
    mSystemDir.mkdirs();

    // 同学们注意：会新建几个文件
    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);

    // Deprecated: Needed for migration
    mStoppedPackagesFilename = new File(mSystemDir, "packages-stopped.xml");
    mBackupStoppedPackagesFilename = new File(mSystemDir, "packages-stopped-
backup.xml");
}
```

自我总结：

所在目录：`/data/system/`

```
packages.xml                                  记录系统中所有安装的应用信息，包括
基本信息、签名和权限
packages-backup.xml packages.xml              文件的备份
packages.list                                 保存普通应用的数据目录和uid等信息
packages-stopped.xml                          记录系统中被强制停止运行的应用信
息。
packages-stopped-backup.xml pacakges-stopped.xml    文件的备份
```



Emulator Nexus_4_API_22 Android 5.1.1, API 22

Name	Permissions	Date	Size
▼ system	drwxrwxr-x	2020-09-07 06:57	
▶ dropbox	drwx------	2020-09-07 06:46	
▶ ifw	drwx------	2020-08-28 03:33	
▶ inputmethod	drwx------	2020-08-28 03:33	
▶ install_sessions	drwx------	2020-08-28 03:33	
▶ job	drwx------	2020-08-28 03:33	
▶ netstats	drwx------	2020-08-28 03:33	
▶ procstats	drwx------	2020-09-07 06:57	
▶ recent_images	drwx------	2020-09-02 09:41	
▶ recent_tasks	drwx------	2020-09-02 09:41	
▶ registered_services	drwxrwx--x	2020-08-28 03:34	
▶ shared_prefs	drwxrwx--x	2020-08-28 03:33	
▶ sync	drwx------	2020-08-28 03:33	
▶ usagestats	drwx------	2020-08-28 03:33	
▶ users	drwxrwxr-x	2020-08-28 03:33	
appops.xml	-rw-------	2020-09-07 06:47	3.2 KB
batterystats.bin	-rw-------	2020-09-07 06:55	39.1 KB
called_pre_boots.dat	-rw-------	2020-08-28 03:33	632 B
device_policies.xml	-rw-------	2020-08-28 03:34	234 B
entropy.dat	-rw-------	2020-09-02 04:31	512 B
framework_atlas.config	-rw-------	2020-08-28 03:33	130 B
install_sessions.xml	-rw-------	2020-09-02 09:39	70 B
last-fstrim	-rw-------	2020-08-28 03:33	0 B
locksettings.db	-rw-rw----	2020-08-28 03:33	4 KB
locksettings.db-shm	-rw-------	2020-08-28 03:33	32 KB
locksettings.db-wal	-rw-------	2020-08-28 03:33	64.4 KB
ndebugsocket	srwx------	2020-08-28 03:33	
package-usage.list	-rw-r-----	2020-09-07 06:57	1.5 KB
packages.list	-rw-r-----	2020-09-02 09:41	7 KB
packages.xml	-rw-rw----	2020-09-02 09:41	133.5 KB
seapp_hash	-rw-------	2020-08-28 03:33	20 B

同学们这里是UID，留意一下啊：Settings.addSharedUserLPw(String name, int uid, int pkgFlags, int pkgPrivateFlags)

```
ArrayMap<String, SharedUserSetting> mSharedUsers = new ArrayMap<String,
SharedUserSetting>();

SharedUserSetting addSharedUserLPw(String name, int uid, int pkgFlags, int
pkgPrivateFlags) {
    SharedUserSetting s = mSharedUsers.get(name);
    // ...代码省略...
    s = new SharedUserSetting(name, pkgFlags, pkgPrivateFlags);
    s.userId = uid;
    if (addUserIdLPw(uid, s, name)) {
        // 将SharedUserSetting添加到mSharedUsers中;
        mSharedUsers.put(name, s);
        return s;
    }
    return null;
}
```

同学们注意：第四件事情细节：如果`packages.xml`文件存在，则读取并解析该文件信息，然后保存到Settings相应的字段中;

`packages.xml` 等文件的读取及解析

本方法实际就是对packages.xml、packages-backup.xml等文件进行操作，从中获取到对应的值;

```
/*
 * 本方法实际就是对packages.xml、packages-backup.xml等文件进行操作，从中获取到对应的值;
 */
boolean readLPw(PackageManagerService service, List<UserInfo> users,
        int sdkVersion, boolean onlyCore) {
    FileInputStream str = null;
    // 如果packages-backup.xml存在，则解析备份文件;
    if (mBackupSettingsFilename.exists()) {
        str = new FileInputStream(mBackupSettingsFilename);
        // ...代码省略...
        if (mSettingsFilename.exists()) {
            mSettingsFilename.delete();
        }
    }
    // ...代码省略...

    //  如果packages-backup.xml不存在，则解析packages.xml
    if (str == null) {
        if (!mSettingsFilename.exists()) {
            return false;
        }
        str = new FileInputStream(mSettingsFilename);
    }
    XmlPullParser parser = Xml.newPullParser();
    parser.setInput(str, StandardCharsets.UTF_8.name());

    // ...代码省略...

    int type;
    int outerDepth = parser.getDepth();
    while ((type = parser.next()) != XmlPullParser.END_DOCUMENT
            && (type != XmlPullParser.END_TAG || parser.getDepth() >
outerDepth)) {
        // 接下去就是xml文件解析
        String tagName = parser.getName();
        if (tagName.equals("package")) {
            readPackageLPw(parser);
        } else if (tagName.equals("permissions")) {
            readPermissionsLPw(mPermissions, parser);
        }
        // ...省略很多xml节点解析代码...
    }
    str.close();

    // ...代码省略...

    if (mBackupStoppedPackagesFilename.exists()
            || mStoppedPackagesFilename.exists()) {
        // 这个方法的逻辑和readLPw()方法类似，只是这里读取的是packages-stopped.xml和packages-stopped-backup.xml文件；
        readStoppedLPw();
        mBackupStoppedPackagesFilename.delete();
        mStoppedPackagesFilename.delete();
        // Migrate to new file format
        writePackageRestrictionsLPr(0);
    } else {
        // ...代码省略...
    }
    // ...代码省略...
    return true;
}
```

同学们注意：第五件事情细节：开始扫描指定目录下的apk文件，解析其Manifest文件，并将值赋值给Package对应的属性字段(这个步骤是重点)；

`Manifest` 文件的解析

> 扫描指定目录下的apk文件，解析其Manifest文件，并将值赋值给Package对应的属性字段(这个步骤是重点)；

```
/*
 * 扫描指定的文件夹，如果是该文件夹内的文件是apk文件或是文件夹，则继续扫描；
 */
private void scanDirLI(File dir, int parseFlags, int scanFlags, long currentTime) {
    final File[] files = dir.listFiles();
    if (ArrayUtils.isEmpty(files)) return;

    for (File file : files) {
        // isApkFile(file):文件后缀是否为apk
        final boolean isPackage = (isApkFile(file) || file.isDirectory())
                && !PackageInstallerService.isStageName(file.getName());
        if (!isPackage) {// 如果是常规文件(非文件夹、非Apk文件)，则跳过；
            continue;
        }
        // 扫描指定的文件目录或apk
        scanPackageLI(file, parseFlags | PackageParser.PARSE_MUST_BE_APK,
                    scanFlags, currentTime, null);
    }
}
```

```java
/*
*  解析Apk包中的Manifest.xml文件
*
 * 返回的PackageParser.Package对象，表示一个Apk文件对应的数据结构；
 */
private PackageParser.Package scanPackageLI(File scanFile, int parseFlags, int scanFlags,
                                            long currentTime, UserHandle user)
throws PackageManagerException {
    PackageParser pp = new PackageParser();

    // 解析Apk包中的清单文件，将对应节点设置到Package对应的属性字段上；
    final PackageParser.Package pkg = pp.parsePackage(scanFile, parseFlags);

    // ...省略大段代码...

    // Verify certificates against what was last scanned.
    // 对安装的程序进行签名验证(这里不展开)
    collectCertificatesLI(pp, ps, pkg, scanFile, parseFlags);

    // ...省略大段代码...

    // Note that we invoke the following method only if we are about to unpack an application
    PackageParser.Package scannedPkg = scanPackageLI(pkg, parseFlags, scanFlags
            | SCAN_UPDATE_SIGNATURE, currentTime, user);

    // ...省略代码...
    return scannedPkg;
}
```

类 `PackageParser` 主要功能就是解析Apk文件中的 `Manifest.xml` 文件

pp.parsePackage(scanFile, parseFlags);

```java
/*
 * `parseClusterPackage()、parseMonolithicPackage()`
 *  这两个方法内部都会执行parseBaseApk();
 */
public Package parsePackage(File packageFile, int flags) throws PackageParserException {
    if (packageFile.isDirectory()) { // 同学们注意：packageFile是文件夹
        return parseClusterPackage(packageFile, flags);
    } else { // 同学们注意：packageFile是Apk文件
        return parseMonolithicPackage(packageFile, flags);
    }
}

/*
 * 解析Apk文件
 */
public Package parseMonolithicPackage(File apkFile, int flags) throws PackageParserException {
    // ...代码省略...
    final Package pkg = parseBaseApk(apkFile, assets, flags);
    // ...代码省略...
    return pkg;
}

private Package parseClusterPackage(File packageDir, int flags) throws
PackageParserException {
    final PackageLite lite = parseClusterPackageLite(packageDir, 0);
    final AssetManager assets = new AssetManager();
    try {
        final File baseApk = new File(lite.baseCodePath);
        // 返回一个Package对象，该对象包含了Manifest文件中的所有节点信息；
        final Package pkg = parseBaseApk(baseApk, assets, flags);
        // ...代码省略...
        if (!ArrayUtils.isEmpty(lite.splitNames)) {
            final int num = lite.splitNames.length;
            // ...代码省略...
            for (int i = 0; i < num; i++) {
                // 这个方法和上面的parseBaseApk()执行逻辑类似，不再重复；
                parseSplitApk(pkg, i, assets, flags);
            }
        }
        return pkg;
    } finally {
        IoUtils.closeQuietly(assets);
    }
}

/*
 * 解析BaseApk的清单文件，并返回一个Package对象；
 */
private Package parseBaseApk(File apkFile, AssetManager assets, int flags)
            throws PackageParserException {
    final String apkPath = apkFile.getAbsolutePath();
    // ...代码省略...

    Resources res = null;
    XmlResourceParser parser = null;
    try {
        res = new Resources(assets, mMetrics, null);
        assets.setConfiguration(0, 0, null, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0,
                Build.VERSION.RESOURCES_SDK_INT);
        // 解析 ANDROID_MANIFEST_FILENAME 文件
        parser = assets.openXmlResourceParser(cookie,
ANDROID_MANIFEST_FILENAME);

        // 解析Manifest文件下的所有节点，将他们存储到Package对应的字段中，并返回Package对
象；
        final Package pkg = parseBaseApk(res, parser, flags, outError);
        // ...代码省略...
        return pkg;
    } catch (Exception e) {
        // ...代码省略...
    }
}

/*
 * 解析Manifest文件下的所有节点，将他们存储到Package对应的字段中，并返回Package对象；
 */
```

```
private Package parseBaseApk(Resources res, XmlResourceParser parser, int flags,
            String[] outError) throws XmlPullParserException, IOException {
    final boolean trustedOverlay = (flags & PARSE_TRUSTED_OVERLAY) != 0;
    // ...代码省略...

    // 创建一个Package对象，用于存储从Manifest文件中解析出来的节点信息；
    final Package pkg = new Package(pkgName);

    // ...代码省略...
    int outerDepth = parser.getDepth();
    while ((type = parser.next()) != XmlPullParser.END_DOCUMENT
            && (type != XmlPullParser.END_TAG || parser.getDepth() >
outerDepth)) {
        if (type == XmlPullParser.END_TAG || type == XmlPullParser.TEXT) {
            continue;
        }

        if (tagName.equals("application")) {
            // 解析Manifest文件中application节点之间的所有节点信息；
            if (!parseBaseApplication(pkg, res, parser, attrs, flags, outError))
{
                return null;
            }
        } else if (tagName.equals("uses-permission")) {
            // 解析Manifest文件中uses-permission节点信息，并将它存储到
Package.requestedPermissions字段中；
            if (!parseUsesPermission(pkg, res, parser, attrs)) {
                return null;
            }
        }
        // ...省略大段代码...
    }
    // ...省略大段代码...
    return pkg;
}

/**
 * 解析manifest文件中Application节点下的节点，将对应的节点分别保存在Package对象的对应字段属
性中；
 *
 * 如四大组件标签存放的字段：
 * List<Activity> activities;
 * List<Activity> receivers;
 * List<Provider> providers;
 * List<Service> services;
 */
private boolean parseBaseApplication(Package owner, Resources res,
        XmlPullParser parser, AttributeSet attrs, int flags, String[] outError)
    throws XmlPullParserException, IOException {

    // ...省略大段代码...

    // 开始解析
    final int innerDepth = parser.getDepth();
    int type;
    while ((type = parser.next()) != XmlPullParser.END_DOCUMENT
            && (type != XmlPullParser.END_TAG || parser.getDepth() >
innerDepth)) {
        // ...省略代码...

        String tagName = parser.getName();
        if (tagName.equals("activity")) {
            // 解析Manifest文件中的Activity节点
            Activity a = parseActivity(owner, res, parser, attrs,
                flags, outError, false, owner.baseHardwareAccelerated);
            // ...省略代码...
            // 将该Activity添加到Packagearser.Package.activities中;
            owner.activities.add(a);

        } else if (tagName.equals("receiver")) {
            // 将该Receiver添加到Packagearser.Package.receivers中;
            owner.receivers.add(a);

        } else if (tagName.equals("service")) {
            // 将该Service添加到Packagearser.Package.services中;
            owner.services.add(s);

        } else if (tagName.equals("provider")) {
            // 将该Provider添加到Packagearser.Package.providers中;
            owner.providers.add(p);
        }
        // ...省略大段代码...
    }
    // ...省略代码...
    return true;
}
```

自我总结： 同学们 到这里为止，这个Apk文件的AndroidManifest.xml文件的解析就结束了；

同学们注意：第八件事情 将数据写入packages.xml中；

```
void writeLPr() {
    /*
     * 前提：packages.xml存在
     * 1.备份不存在，就将packages.xml重命名成备份名；
     * 2.备份存在，就删除packages.xml文件；
     * 总结：只要有一份文件即可；
     */
    if (mSettingsFilename.exists()) {
        if (!mBackupSettingsFilename.exists()) {
            if (!mSettingsFilename.renameTo(mBackupSettingsFilename)) {
                return;
            }
        } else {
            mSettingsFilename.delete();
        }
    }
    // 开始向packages.xml文件中写入应用相关的数据；
    try {
        FileOutputStream fstr = new FileOutputStream(mSettingsFilename);
        BufferedOutputStream str = new BufferedOutputStream(fstr);

        //XmlSerializer serializer = XmlUtils.serializerInstance();
        XmlSerializer serializer = new FastXmlSerializer();
        serializer.setOutput(str, StandardCharsets.UTF_8.name());
        serializer.startDocument(null, true);
        serializer.setFeature("http://xmlpull.org/v1/doc/features.html#indent-
output", true);
        serializer.startTag(null, "packages");

        // ...省略很多节点插入的代码...

        // 写入权限
        serializer.startTag(null, "permissions");
        for (BasePermission bp : mPermissions.values()) {
            writePermissionLPr(serializer, bp);
        }
        serializer.endTag(null, "permissions");

        // 将Package数据写入
        for (final PackageSetting pkg : mPackages.values()) {
            writePackageLPr(serializer, pkg);
        }
        // ...省略很多节点插入的代码...

        serializer.endTag(null, "packages");
        serializer.endDocument();

        str.flush();
        FileUtils.sync(fstr);
        str.close();

        // 到这里，packages.xml已经写入了最新的数据，所以需要删除packages-backup.xml文件
        mBackupSettingsFilename.delete();
        // 设置packages.xml文件的权限；
        FileUtils.setPermissions(mSettingsFilename.toString(),
                FileUtils.S_IRUSR|FileUtils.S_IWUSR
                |FileUtils.S_IRGRP|FileUtils.S_IWGRP,
                -1, -1);

        // 向packages.list文件中写入数据
        writePackageListLPr();
        writeAllUsersPackageRestrictionsLPr();
        writeAllRuntimePermissionsLPr();
        return;

    } catch(Exception e) {
        //...
    }
    // Clean up partially written files
    if (mSettingsFilename.exists()) {
        if (!mSettingsFilename.delete()) {
            //...
        }
    }
}
```

大总结：

1. 在 `"/data/system"` 目录中创建一系列的文件用于存储应用的信息；
2. 在 `SystemConfig` 对象中读取系统配置来初始化 `mGlobalGids、mSystemPermissions、mAvailableFeatures`；
3. 指定 `"/data"` 目录下一些文件夹`，一部分是用于第4步的扫描目录的指定；
4. 读取 `packages.xml` 文件的数据，解析并存储到Settings中；
5. 扫描指定文件目录下的Apk包，解析其内部的Manifest.xml文件，将值赋值给PMS和Settings、Package中；
6. 将扫描出来的应用信息重新写入 第八件事情的 `packages.xml` 文件中；

三部曲 - 有界面安装 与 无界面安装

1、概述

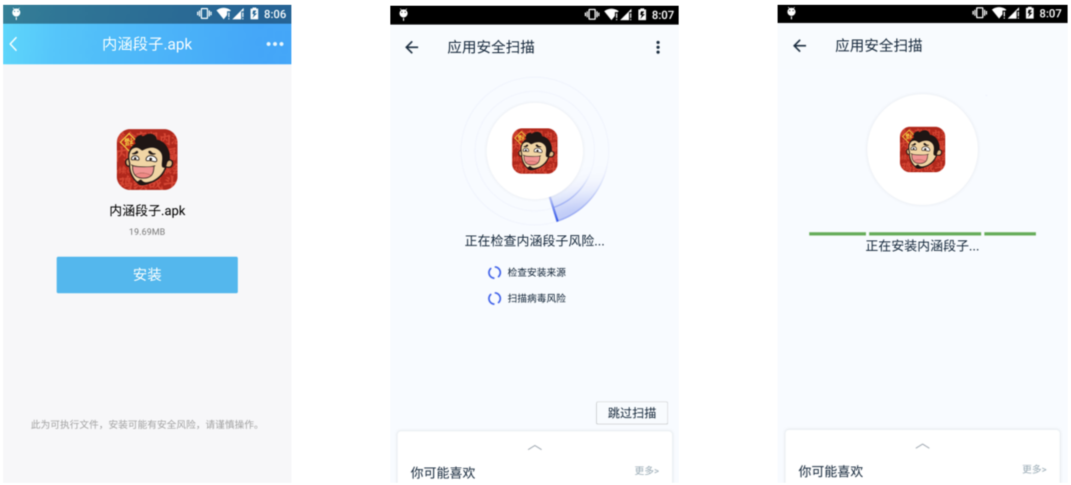
apk的安装有几种方式，整体可分为2类，一类是有界面安装，一类是无界面安装。无界面安装又可分为内置apk开机安装和命令安装，而命令安装又可分为2类，一类是通过电脑的adb安装，另一类是通过手机安装的pm命令。

今天我们主要介绍两种，一种是有界面安装，另一种就是通过电脑端的adb无界面安装。

2、有界面安装

有界面的安装方式相信同学们平时接触的都比较多了吧，比如从网络上下载一个apk之后就会弹出安装界面，那就是有界面安装方式。

有界面安装



此为可执行文件，安装可能有安全风险，请谨慎操作。

正在检查内涵段子风险...

正在安装内涵段子...

首先我们要知道有界面安装方式的那个界面是从哪里来的，还有就是如何启动那个安装界面的。那个界面其实就是PackageInstallerActivity类，这是**packages/app/PackageInstaller应用**中的主界面，我们可以看一下该应用的Manifest文件。

```
<activity
    android:name=".PackageInstallerActivity"
    android:configChanges="orientation|keyboardHidden|screenSize"
    android:theme="@android:style/Theme.DeviceDefault.Light.NoActionBar"
    android:excludeFromRecents="true">
        <intent-filter>
            <action android:name="android.intent.action.VIEW" />
            <action android:name="android.intent.action.INSTALL_PACKAGE" />
            <category android:name="android.intent.category.DEFAULT" />
            <data android:scheme="file" />
            //同学们注意：这里配置了application/vnd.android.package-archive这个字符串
的mime类型
            <data android:mimeType="application/vnd.android.package-archive" />
        </intent-filter>
        <intent-filter>
            <action android:name="android.intent.action.INSTALL_PACKAGE" />
            <category android:name="android.intent.category.DEFAULT" />
            <data android:scheme="file" />
            <data android:scheme="package" />
        </intent-filter>
        <intent-filter>
            <action android:name="android.content.pm.action.CONFIRM_PERMISSIONS"
/>
            <category android:name="android.intent.category.DEFAULT" />
        </intent-filter>
</activity>
```

接下来我们直接进入PackageInstallerActivity类中。

```
public class PackageInstallerActivity extends Activity implements
OnCancelListener, OnClickListener {

    protected void onCreate(Bundle icicle) {
        super.onCreate(icicle);
    }

    // 同学们注意：在其OnResume方法中主要是加载布局和对apk包进行相关解析，将权限信息显示在界
面上，而apk相关信息将保存下来，以  便后期安装时使用
    protected void onResume() {
        super.onResume();

        mPm = getPackageManager();
        mInstaller = mPm.getPackageInstaller();
        mUserManager = (UserManager) getSystemService(Context.USER_SERVICE);

        //加载界面布局
        setContentView(R.layout.install_start);

        //读取apk包相关信息
        PackageParser.Package parsed = PackageUtil.getPackageInfo(sourceFile);
        //读取权限相关信息
        //......
    }
}
```

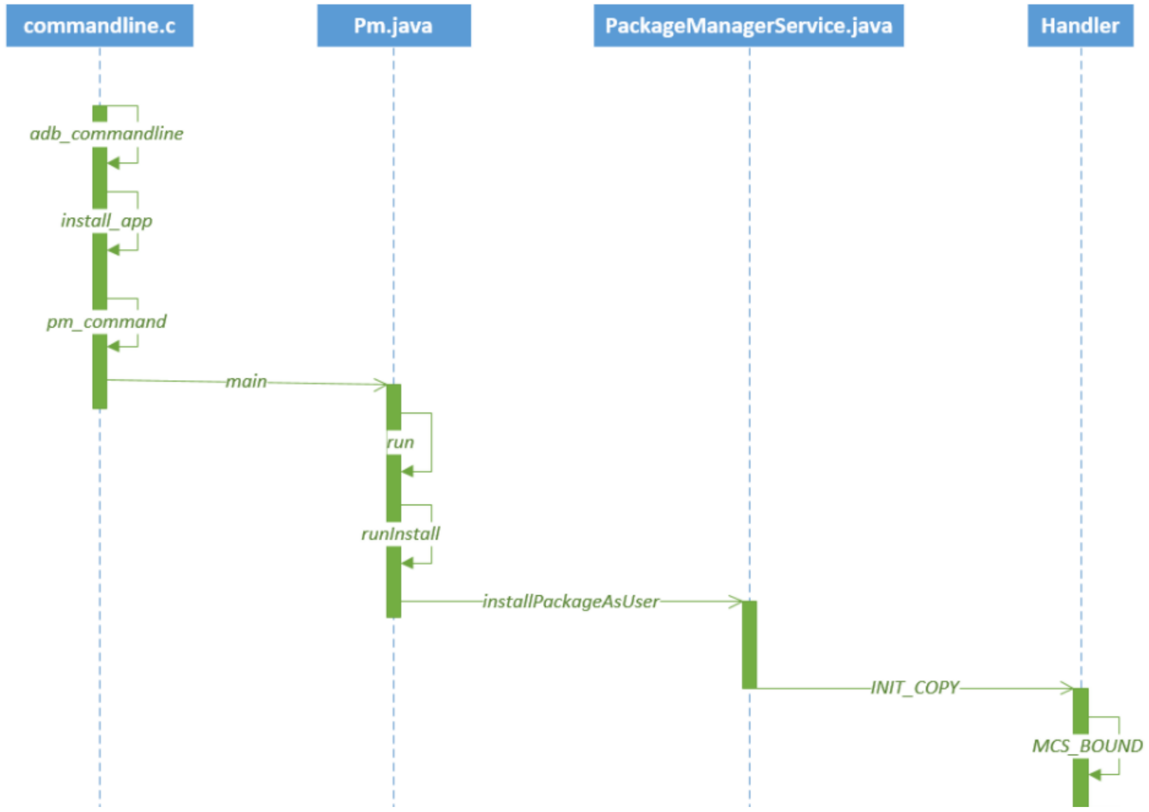
当用户点击安装的时候，调用startInstall方法跳转到InstallAppProgress，并把当前需要安装的应用信息给传递过去，其中有应用安装的完整包名:

```
private void startInstall() {
        // Start subactivity to actually install the application
        Intent newIntent = new Intent();
        newIntent.putExtra(PackageUtil.INTENT_ATTR_APPLICATION_INFO,
                mPkgInfo.applicationInfo);
        newIntent.setData(mPackageURI);
        newIntent.setClass(this, InstallAppProgress.class);
        newIntent.putExtra(InstallAppProgress.EXTRA_MANIFEST_DIGEST,
mPkgDigest);
        newIntent.putExtra(
                InstallAppProgress.EXTRA_INSTALL_FLOW_ANALYTICS,
mInstallFlowAnalytics);
        String installerPackageName = getIntent().getStringExtra(
                Intent.EXTRA_INSTALLER_PACKAGE_NAME);
        if (mOriginatingURI != null) {
            newIntent.putExtra(Intent.EXTRA_ORIGINATING_URI, mOriginatingURI);
        }
        if (mReferrerURI != null) {
            newIntent.putExtra(Intent.EXTRA_REFERRER, mReferrerURI);
        }
        if (mOriginatingUid != VerificationParams.NO_UID) {
            newIntent.putExtra(Intent.EXTRA_ORIGINATING_UID, mOriginatingUid);
        }
        if (installerPackageName != null) {
            newIntent.putExtra(Intent.EXTRA_INSTALLER_PACKAGE_NAME,
                    installerPackageName);
        }
        if (getIntent().getBooleanExtra(Intent.EXTRA_RETURN_RESULT, false)) {
            newIntent.putExtra(Intent.EXTRA_RETURN_RESULT, true);
            newIntent.addFlags(Intent.FLAG_ACTIVITY_FORWARD_RESULT);
        }
        if(localLOGV) Log.i(TAG, "downloaded app uri="+mPackageURI);
        startActivity(newIntent);
        finish();
    }
```

同学们注意：此Activity就是在安装过程中....

```
public class InstallAppProgress extends Activity implements
View.OnClickListener, OnCancelListener {

    public void initView() {

        setContentView(R.layout.op_progress);
        PackageManager pm = getPackageManager();
        //......
        //创建安装完成后的监听接口
        PackageInstallObserver observer = new PackageInstallObserver();

        if ("package".equals(mPackageURI.getScheme())) {
            try {
                pm.installExistingPackage(mAppInfo.packageName);
                //监听安装结果
                observer.packageInstalled(mAppInfo.packageName,
                        PackageManager.INSTALL_SUCCEEDED);
            } catch (PackageManager.NameNotFoundException e) {
                observer.packageInstalled(mAppInfo.packageName,
                        PackageManager.INSTALL_FAILED_INVALID_APK);
            }
        } else {
            pm.installPackageWithVerificationAndEncryption(mPackageURI,
observer, installFlags,
                    installerPackageName, verificationParams, null);
        }
    }
}
```

3、无界面安装



都知道，adb install有很多的参数，但是我们今天就分析最简单的adb install qq.apk，adb是一个命令而install是其参数，这里我们直接进入处理install的代码逻辑（**system/core/adb/commandline.c**）。

```
int adb_commandline(int argc, char **argv)
{
    //......
    if (!strcmp(argv[0], "install")) {
        if (argc < 2) return usage();
        //调用install_app函数进行处理
        return install_app(ttype, serial, argc, argv);
    }
}
```

我们进入install_app函数，看其细节。

```
int install_app(transport_type transport, char* serial, int argc, char** argv)
{
    //手机内部存储路径
    static const char *const DATA_DEST = "/data/local/tmp/%s";
    //SD卡路径
    static const char *const SD_DEST = "/sdcard/tmp/%s";
    const char* where = DATA_DEST;

    for (i = 1; i < argc; i++) {
        //表示安装在SD卡上
        if (!strcmp(argv[i], "-s")) {
            where = SD_DEST;
        }
    }

    char* apk_file = argv[last_apk];
    //安装路径
    char apk_dest[PATH_MAX];
    snprintf(apk_dest, sizeof apk_dest, where, get_basename(apk_file));
    //调用do_sync_push将apk文件push到手机中
    int err = do_sync_push(apk_file, apk_dest, 0 /* no show progress */);
    if (err) {
        goto cleanup_apk;
    } else {
        argv[last_apk] = apk_dest; /* destination name, not source location */
    }
    //调用shell pm命令去安装
    pm_command(transport, serial, argc, argv);

cleanup_apk:
    //在手机中执行shell rm来删除刚刚推入的apk文件
    delete_file(transport, serial, apk_dest);
    return err;
}
```

这个方法主要的功能就是找到apk安装的路径，然后执行pm命令去安装，并在最终通过rm命令将apk进行删除，我们在来看一下pm_command函数的功能吧。

```
static int pm_command(transport_type transport, char* serial, int argc, char**
argv)
```

```
{
    char buf[4096];
    //通过pm命令去执行安装操作
    snprintf(buf, sizeof(buf), "shell:pm");

    while(argc-- > 0) {
        char *quoted = escape_arg(*argv++);
        strncat(buf, " ", sizeof(buf) - 1);
        strncat(buf, quoted, sizeof(buf) - 1);
        free(quoted);
    }

    send_shellcommand(transport, serial, buf);
    return 0;
}
```

那什么是pm呢？其实pm只是一个脚本，其源码所在路径（frameworks/base/cmds/pm/pm）。

```
# Script to start "pm" on the device, which has a very rudimentary
# shell.
#
base=/system
export CLASSPATH=$base/framework/pm.jar
exec app_process $base/bin com.android.commands.pm.Pm "$@"
```

可以发现其执行的是pm.jar包的main函数，我们进入Pm.java类。

```
    public static void main(String[] args) {
        int exitCode = 1;
        try {
            exitCode = new Pm().run(args);
        } catch (Exception e) {
            Log.e(TAG, "Error", e);
            System.err.println("Error: " + e);
            if (e instanceof RemoteException) {
                System.err.println(PM_NOT_RUNNING_ERR);
            }
        }
        System.exit(exitCode);
    }
```

这里直接创建了Pm对象并调用其run方法，我们进入其run方法。

```
public int run(String[] args) throws IOException, RemoteException {
    mUm = IUserManager.Stub.asInterface(ServiceManager.getService("user"));
    //这里获得PKMS的代理对象
    mPm =
IPackageManager.Stub.asInterface(ServiceManager.getService("package"));
    if (mPm == null) {
        System.err.println(PM_NOT_RUNNING_ERR);
        return 1;
    }
    mInstaller = mPm.getPackageInstaller();

    //处理install参数，还有很多其他参数
    if ("install".equals(op)) {
        return runInstall();
    }
    //......
}
```

可以发现这里又将安装的操作交给了runInstall这个方法，我们再次进入该方法。

```
private int runInstall() {
    while ((opt=nextOption()) != null) {
        //处理很多的参数命令
        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");
                    return 1;
                }
            } 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;
            }
            //......
    }

    //多用户手机时将所有用户都安装
    if (userId == UserHandle.USER_ALL) {
       userId = UserHandle.USER_OWNER;
       installFlags |= PackageManager.INSTALL_ALL_USERS;
    }

    //监听安装结果
    LocalPackageInstallObserver obs = new LocalPackageInstallObserver();

    //调用PKMS的installPackageAsUser方法进行安装操作
    mPm.installPackageAsUser(apkFilePath, obs.getBinder(),
installFlags,installerPackageName, verificationParams, abi, userId);

    synchronized (obs) {
        while (!obs.finished) {
            obs.wait();
        }
        //当安装成功后打印Success
        if (obs.result == PackageManager.INSTALL_SUCCEEDED) {
             System.out.println("Success");
             return 0;
        } else {
             System.err.println("Failure ["
                  + installFailureToString(obs)
                  + "]");
             return 1;
        }
    }
}
```

以上就是runInstall方法的主要内容，首先根据安装参数来设置installFlags属性值，然后创建LocalPackageInstallObserver对象来监听安装结果，最后调用PKMS对象的installPackageAsUser来执行安装操作，当然最后无论安装成功还是失败都需返回一个结果以供PC的命令行进行展示，接下来我们将进入PKMS的installPackageAsUser方法。

```
// originPath表示apk路径，observer是LocalPackageInstallObserver对象，用于监听apk安装结果，installFlags是安装参数
public void installPackageAsUser(String originPath, IPackageInstallObserver2 observer,int installFlags,...) {
    //......
    //将操作通过发送Handler来处理
     final Message msg = mHandler.obtainMessage(INIT_COPY);
     //注意这里创建了InstallParams对象并将其传递给msg.obj对象，后面我们会详细分析这个对象
     msg.obj = new InstallParams(origin, observer, installFlags,installerPackageName, verificationParams, user, packageAbiOverride);
     mHandler.sendMessage(msg);
}
```

可以看到installPackageAsUser方法还是蛮简单的，只是创建一个Message对象，然后通过Handler来发送INIT_COPY的消息，不过这里大家要注意参数的传递，我们进入Handler的处理消息的代码。

```
public void handleMessage(Message msg) {

    switch (msg.what) {
        case INIT_COPY: {
            //（1）获得传递过来的params对象，其实际值是InstallParams
            HandlerParams params = (HandlerParams) msg.obj;
            //mPendingInstalls用于存储待安装应用，idx表示当前待安装个数
            int idx = mPendingInstalls.size();
            if (!mBound) {
                //（2）通过bindService来启动另外一个服务
                if (!connectToService()) {
                    params.serviceError();
                    return;
                } else {
                    //如果另外一个服务已启动，将其添加到mPendingInstalls中
                    mPendingInstalls.add(idx, params);
                }
             } else {
                    mPendingInstalls.add(idx, params);
                    if (idx == 0) {
                    //（3）表示要启动安装
                    mHandler.sendEmptyMessage(MCS_BOUND);
                    }
             }
             break;
        }
    }
}
```

重点分析第三步来处理MCS_BOUND消息

```
public void handleMessage(Message msg) {

    switch (msg.what) {
        case MCS_BOUND: {
            if (msg.obj != null) {
                mContainerService = (IMediaContainerService) msg.obj;
            }
            //如果Service没有启动则不能安装程序
            if (mContainerService == null) {
                for (HandlerParams params : mPendingInstalls) {
                    params.serviceError();
                }
                mPendingInstalls.clear();
            }else if (mPendingInstalls.size() > 0) {
                HandlerParams params = mPendingInstalls.get(0);
                if (params != null) {
                    // 同学们注意：调用params对象的startCopy方法，该方法有基类HandlerParams定义
                    if (params.startCopy()) {
                       if (mPendingInstalls.size() > 0) {
                           //删除队列头
                           mPendingInstalls.remove(0);
                       }
                       if (mPendingInstalls.size() == 0) {
                           if (mBound) {
                               //如果安装请求完成了，在通过调用unbindService方法来解绑服务

                               removeMessages(MCS_UNBIND);
                               Message ubmsg = obtainMessage(MCS_UNBIND);
                               sendMessageDelayed(ubmsg, 10000);
                           }
                       } else {
                           //如果还有待安装的事件，将继续发送MCS_BOUND消息来完成安装
                           mHandler.sendEmptyMessage(MCS_BOUND);
                       }
                    }
                }
            }
            break;
        }
    }
}
```

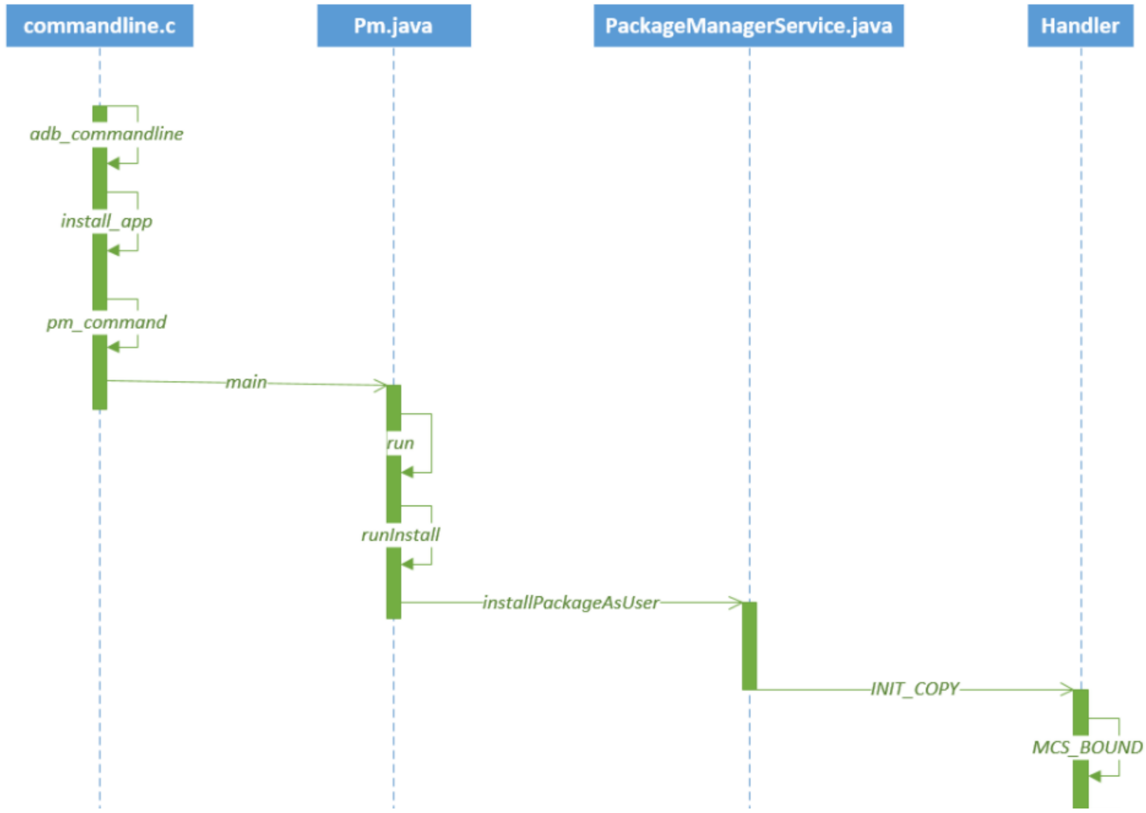
进入params.startCopy()方法进行分析，看其具体是如何进行安装的：

```
final boolean startCopy() {
     boolean res;
     try {
         //MAX_RETRIES的值是4，表示默认安装4次，如果还不成功就表示安装失败
          if (++mRetries > MAX_RETRIES) {
               mHandler.sendEmptyMessage(MCS_GIVE_UP);
               handleServiceError();
               return false;
           } else {
               //调用HandlerParams子类的handleStartCopy方法  【同学们注意：这是第一步，会先执行这个】
               handleStartCopy();
               res = true;
           }
      } catch (RemoteException e) {
           mHandler.sendEmptyMessage(MCS_RECONNECT);
           res = false;
      }
      //调用HandlerParams子类的handleReturnCode方法，将处理结果返回 【同学们注意：这是第而步，后执行这个】
      handleReturnCode();
      return res;
}
```

同学们上面的startCopy方法，发现其主要就执行了两步，先执行HandlerParams子类的handleStartCopy方法，然后在执行其handleReturnCode方法将结果返回，我们进入该对象的**handleStartCopy**方法。

```
public void handleStartCopy() throws RemoteException {

    //根据adb install的参数来判断安装位置
    final boolean onSd = (installFlags & PackageManager.INSTALL_EXTERNAL) != 0;
    final boolean onInt = (installFlags & PackageManager.INSTALL_INTERNAL) != 0;

    //内部存储和SD卡不能同时安装
    if (onInt && onSd) {
        ret = PackageManager.INSTALL_FAILED_INVALID_INSTALL_LOCATION;
    } else {

        //通过StorageManager对象并查询内部存储空间最小余量
        final StorageManager storage = StorageManager.from(mContext);
        final long lowThreshold = storage.getStorageLowBytes(
                            Environment.getDataDirectory());
        //......
        //创建安装参数对象
        final InstallArgs args = createInstallArgs(this);
            //......
            //调用InstallArgs对象的copyApk方法完成apk的拷贝
            ret = args.copyApk(mContainerService, true);
        }
    }
```

```
}
```



handleReturnCode()函数:

```
        @Override
        void handleReturnCode() {
            // If mArgs is null, then MCS couldn't be reached. When it
            // reconnects, it will try again to install. At that point, this
            // will succeed.
            if (mArgs != null) {
                processPendingInstall(mArgs, mRet);
            }
        }
```

processPendingInstall()函数:

```
private void processPendingInstall(final InstallArgs args, final int
currentStatus) {
        mHandler.post(new Runnable() {
            public void run() {
                mHandler.removeCallbacks(this);
                 // Result object to be returned
                PackageInstalledInfo res = new PackageInstalledInfo();
                res.returnCode = currentStatus;
                res.uid = -1;
                res.pkg = null;
                res.removedInfo = new PackageRemovedInfo();
                if (res.returnCode == PackageManager.INSTALL_SUCCEEDED) {
                    args.doPreInstall(res.returnCode);
                    synchronized (mInstallLock) {
                        // 同学们注意看这个函数
                        installPackageLI(args, res);
                    }
                    args.doPostInstall(res.returnCode, res.uid);
                }
                ........
            }
        });
    }
```

installPackageLI() 函数:

```
private void installPackageLI(InstallArgs args, PackageInstalledInfo res) {
        .....
        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 {
            pkg = pp.parsePackage(tmpPackageFile, parseFlags);
        } catch (PackageParserException e) {
            res.setError("Failed parse during installPackageLI", e);
            return;
        }

        ....... 又省略一万行代码，代码真多

        startIntentFilterVerifications(args.user.getIdentifier(), replace, pkg);

        if (replace) {
            replacePackageLI(pkg, parseFlags, scanFlags | SCAN_REPLACING,
args.user,
                    installerPackageName, volumeUuid, res);
        } else {
            // 【同学们注意：此函数内部会调用到 scanPackageLI() 】
            installNewPackageLI(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) {
                res.newUsers = ps.queryInstalledUsers(sUserManager.getUserIds(),
true);
            }
        }
    }
```

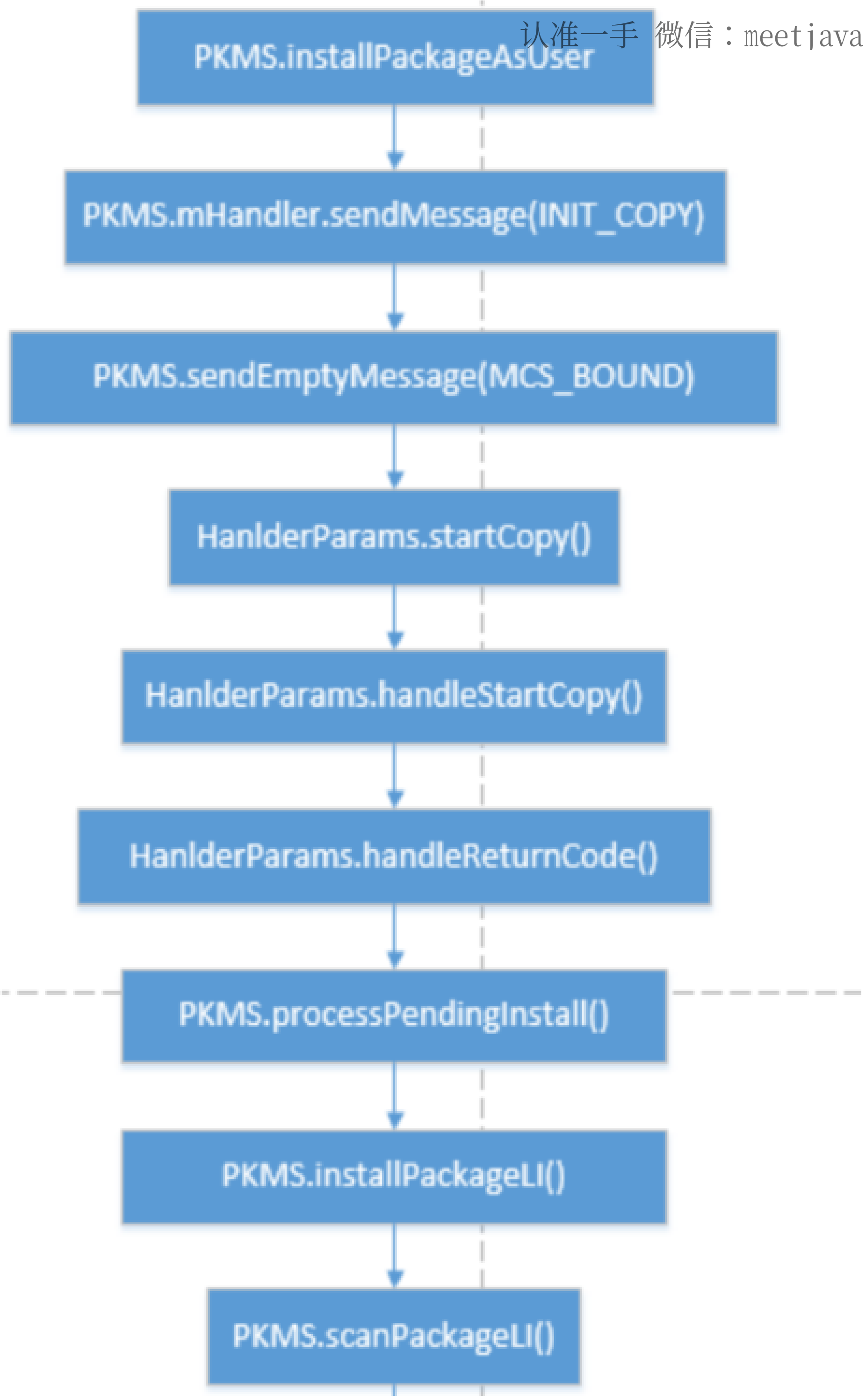
installNewPackageLI函数 调用到 ----> scanPackageLI() 函数：

```
private void installNewPackageLI(PackageParser.Package pkg, int parseFlags, int
scanFlags,
            UserHandle user, String installerPackageName, String volumeUuid,
            PackageInstalledInfo res) {
        // Remember this for later, in case we need to rollback this install
        String pkgName = pkg.packageName;

        .............

        try {
            // 【同学们注意：会执行 scanPackageLI 函数】
            PackageParser.Package newPackage = scanPackageLI(pkg, parseFlags,
scanFlags,
                    System.currentTimeMillis(), user);

            updateSettingsLI(newPackage, installerPackageName, volumeUuid, null,
null, res, user);
            // delete the partially installed application. the data directory
will have to be
            // restored if it was already existing
            if (res.returnCode != PackageManager.INSTALL_SUCCEEDED) {
                // remove package from internal structures.  Note that we want
deletePackageX to
                // delete the package data and cache directories that it created
in
                // scanPackageLocked, unless those directories existed before we
even tried to
                // install.
                deletePackageLI(pkgName, UserHandle.ALL, false, null, null,
                        dataDirExists ? PackageManager.DELETE_KEEP_DATA : 0,
                                res.removedInfo, true);
            }

        } catch (PackageManagerException e) {
            res.setError("Package couldn't be installed in " + pkg.codePath, e);
        }
    }
```

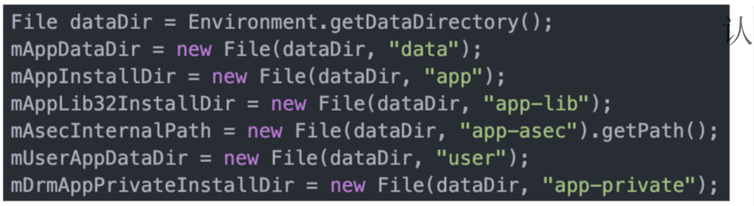
scanPackageLI() 函数：

```
private PackageParser.Package scanPackageLI(File scanFile, int parseFlags, int
scanFlags,
            long currentTime, UserHandle user) throws PackageManagerException {
        if (DEBUG_INSTALL) Slog.d(TAG, "Parsing: " + scanFile);
        parseFlags |= mDefParseFlags;
        PackageParser pp = new PackageParser();
        pp.setSeparateProcesses(mSeparateProcesses);
        pp.setOnlyCoreApps(mOnlyCore);
        pp.setDisplayMetrics(mMetrics);

        if ((scanFlags & SCAN_TRUSTED_OVERLAY) != 0) {
            parseFlags |= PackageParser.PARSE_TRUSTED_OVERLAY;
        }

        final PackageParser.Package pkg;
        try {
            pkg = pp.parsePackage(scanFile, parseFlags);
        } catch (PackageParserException e) {
            throw PackageManagerException.from(e);
        }

        PackageSetting ps = null;
        PackageSetting updatedPkg;
        // reader
        synchronized (mPackages) {
            // Look to see if we already know about this package.
            String oldName = mSettings.mRenamedPackages.get(pkg.packageName);
            if (pkg.mOriginalPackages != null &&
pkg.mOriginalPackages.contains(oldName)) {
                // This package has been renamed to its original name.  Let's
                // use that.
                ps = mSettings.peekPackageLPr(oldName);
            }
            // If there was no original package, see one for the real package
name.
            if (ps == null) {
                ps = mSettings.peekPackageLPr(pkg.packageName);
            }
            // Check to see if this package could be hiding/updating a system
            // package.  Must look for it either under the original or real
            // package name depending on our state.
            updatedPkg = mSettings.getDisabledSystemPkgLPr(ps != null ? ps.name
: pkg.packageName);
            if (DEBUG_INSTALL && updatedPkg != null) Slog.d(TAG, "updatedPkg = "
+ updatedPkg);
        }
        boolean updatedPkgBetter = false;
        // First check if this is a system package that may involve an update
        if (updatedPkg != null && (parseFlags&PackageParser.PARSE_IS_SYSTEM) !=
0) {
            // If new package is not located in "/system/priv-app" (e.g. due to
an OTA),
            // it needs to drop FLAG_PRIVILEGED.
            if (locationIsPrivileged(scanFile)) {
                updatedPkg.pkgPrivateFlags |=
ApplicationInfo.PRIVATE_FLAG_PRIVILEGED;
            } else {
                updatedPkg.pkgPrivateFlags &=
~ApplicationInfo.PRIVATE_FLAG_PRIVILEGED;
            }

            if (ps != null && !ps.codePath.equals(scanFile)) {
                // The path has changed from what was last scanned...  check the
                // version of the new path against what we have stored to
determine
                // what to do.
                if (DEBUG_INSTALL) Slog.d(TAG, "Path changing from " +
ps.codePath);
                if (pkg.mVersionCode <= ps.versionCode) {
                    // The system package has been updated and the code path
does not match
                    // Ignore entry. Skip it.
                    if (DEBUG_INSTALL) Slog.i(TAG, "Package " + ps.name + " at "
+ scanFile
                            + " ignored: updated version " + ps.versionCode
                            + " better than this " + pkg.mVersionCode);
                    if (!updatedPkg.codePath.equals(scanFile)) {
                        Slog.w(PackageManagerService.TAG, "Code path for hidden
system pkg : "
                                + ps.name + " changing from " +
updatedPkg.codePathString
                                + " to " + scanFile);
                        updatedPkg.codePath = scanFile;
                        updatedPkg.codePathString = scanFile.toString();
                        updatedPkg.resourcePath = scanFile;
                        updatedPkg.resourcePathString = scanFile.toString();
                    }
                    updatedPkg.pkg = pkg;
                    throw new
PackageManagerException(INSTALL_FAILED_DUPLICATE_PACKAGE,
                            "Package " + ps.name + " at " + scanFile
                                    + " ignored: updated version " +
ps.versionCode
                                    + " better than this " + pkg.mVersionCode);
                } else {
                    // The current app on the system partition is better than
                    // what we have updated to on the data partition; switch
                    // back to the system partition version.
                    // At this point, its safely assumed that package
installation for
                    // apps in system partition will go through. If not there
won't be a working
                    // version of the app
                    // writer
                    synchronized (mPackages) {
                        // Just remove the loaded entries from package lists.
                        mPackages.remove(ps.name);
                    }

                    logCriticalInfo(Log.WARN, "Package " + ps.name + " at " +
scanFile
                            + " reverting from " + ps.codePathString
                            + ": new version " + pkg.mVersionCode
                            + " better than installed " + ps.versionCode);

                    InstallArgs args =
createInstallArgsForExisting(packageFlagsToInstallFlags(ps),
                            ps.codePathString, ps.resourcePathString,
getAppDexInstructionSets(ps));
                    synchronized (mInstallLock) {
                        args.cleanUpResourcesLI();
                    }
                    synchronized (mPackages) {
                        mSettings.enableSystemPackageLPw(ps.name);
                    }
                    updatedPkgBetter = true;
                }
            }
        }

        if (updatedPkg != null) {
            // An updated system app will not have the PARSE_IS_SYSTEM flag set
            // initially
            parseFlags |= PackageParser.PARSE_IS_SYSTEM;

            // An updated privileged app will not have the PARSE_IS_PRIVILEGED
            // flag set initially
            if ((updatedPkg.pkgPrivateFlags &
ApplicationInfo.PRIVATE_FLAG_PRIVILEGED) != 0) {
                parseFlags |= PackageParser.PARSE_IS_PRIVILEGED;
            }
        }

        // 同学们 这里是：签名校验
        // Verify certificates against what was last scanned
        collectCertificatesLI(pp, ps, pkg, scanFile, parseFlags);

        /*
         * A new system app appeared, but we already had a non-system one of the
         * same name installed earlier.
         */
        boolean shouldHideSystemApp = false;
        if (updatedPkg == null && ps != null
                && (parseFlags & PackageParser.PARSE_IS_SYSTEM_DIR) != 0 &&
!isSystemApp(ps)) {
            /*
             * Check to make sure the signatures match first. If they don't,
             * wipe the installed application and its data.
             */
            if (compareSignatures(ps.signatures.mSignatures, pkg.mSignatures)
                    != PackageManager.SIGNATURE_MATCH) {
                logCriticalInfo(Log.WARN, "Package " + ps.name + " appeared on
system, but"
                        + " signatures don't match existing userdata copy;
removing");
                deletePackageLI(pkg.packageName, null, true, null, null, 0,
null, false);
                ps = null;
            } else {
                /*
                 * If the newly-added system app is an older version than the
                 * already installed version, hide it. It will be scanned later
                 * and re-added like an update.
                 */
                if (pkg.mVersionCode <= ps.versionCode) {
                    shouldHideSystemApp = true;
                    logCriticalInfo(Log.INFO, "Package " + ps.name + " appeared
at " + scanFile
                            + " but new version " + pkg.mVersionCode + " better
than installed "
                            + ps.versionCode + "; hiding system");
                } else {
                    /*
                     * The newly found system app is a newer version that the
                     * one previously installed. Simply remove the
                     * already-installed application and replace it with our own
                     * while keeping the application data.
                     */
                    logCriticalInfo(Log.WARN, "Package " + ps.name + " at " +
scanFile
                            + " reverting from " + ps.codePathString + ": new
version "
                            + pkg.mVersionCode + " better than installed " +
ps.versionCode);
                    InstallArgs args =
createInstallArgsForExisting(packageFlagsToInstallFlags(ps),
                            ps.codePathString, ps.resourcePathString,
getAppDexInstructionSets(ps));
                    synchronized (mInstallLock) {
                        args.cleanUpResourcesLI();
                    }
                }
            }
        }

        // The apk is forward locked (not public) if its code and resources
        // are kept in different files. (except for app in either system or
        // vendor path).
        // TODO grab this value from PackageSettings
        if ((parseFlags & PackageParser.PARSE_IS_SYSTEM_DIR) == 0) {
            if (ps != null && !ps.codePath.equals(ps.resourcePath)) {
                parseFlags |= PackageParser.PARSE_FORWARD_LOCK;
            }
        }

        // TODO: extend to support forward-locked splits
        String resourcePath = null;
        String baseResourcePath = null;
        if ((parseFlags & PackageParser.PARSE_FORWARD_LOCK) != 0 &&
!updatedPkgBetter) {
            if (ps != null && ps.resourcePathString != null) {
                resourcePath = ps.resourcePathString;
                baseResourcePath = ps.resourcePathString;
            } else {
                // Should not happen at all. Just log an error.
                Slog.e(TAG, "Resource path not set for pkg : " +
pkg.packageName);
            }
        } else {
            resourcePath = pkg.codePath;
            baseResourcePath = pkg.baseCodePath;
        }

        // Set application objects path explicitly.
        pkg.applicationInfo.volumeUuid = pkg.volumeUuid;
        pkg.applicationInfo.setCodePath(pkg.codePath);
        pkg.applicationInfo.setBaseCodePath(pkg.baseCodePath);
        pkg.applicationInfo.setSplitCodePaths(pkg.splitCodePaths);
        pkg.applicationInfo.setResourcePath(resourcePath);
        pkg.applicationInfo.setBaseResourcePath(baseResourcePath);
        pkg.applicationInfo.setSplitResourcePaths(pkg.splitCodePaths);

        // Note that we invoke the following method only if we are about to
unpack an application
        PackageParser.Package scannedPkg = scanPackageLI(pkg, parseFlags,
scanFlags
                | SCAN_UPDATE_SIGNATURE, currentTime, user);

        /*
         * If the system app should be overridden by a previously installed
         * data, hide the system app now and let the /data/app scan pick it up
         * again.
         */
        if (shouldHideSystemApp) {
            synchronized (mPackages) {
                /*
                 * We have to grant systems permissions before we hide, because
                 * grantPermissions will assume the package update is trying to
                 * expand its permissions.
                 */
                grantPermissionsLPw(pkg, true, pkg.packageName);
                mSettings.disableSystemPackageLPw(pkg.packageName);
            }
        }

        return scannedPkg;
    }
```

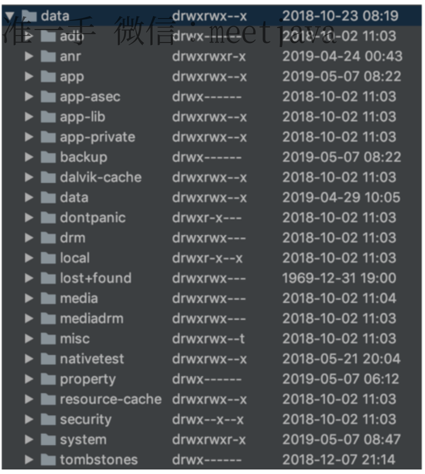
安装流程：


PKMS.installPackageAsUser
PKMS.mHandler.sendMessage(INIT_COPY)
PKMS.sendEmptyMessage(MCS_BOUND)
HanlderParams.startCopy()
HanlderParams.handleStartCopy()
HanlderParams.handleReturnCode()
PKMS.processPendingInstall()
PKMS.installPackageLI()
PKMS.scanPackageLI()

在PKMS构造函数中解析所需要的文件目录:



```
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");
```



```
▼ data            drwxrwx--x    2018-10-23 08:19
  ▶ adb           drwx------    2018-10-02 11:03
  ▶ anr           drwxrwxr-x    2019-04-24 00:43
  ▶ app           drwxrwx--x    2019-05-07 08:22
  ▶ app-asec      drwx------    2018-10-02 11:03
  ▶ app-lib       drwxrwx--x    2018-10-02 11:03
  ▶ app-private   drwxrwx--x    2018-10-02 11:03
  ▶ backup        drwx------    2019-05-07 08:22
  ▶ dalvik-cache  drwxrwx--x    2018-10-02 11:03
  ▶ data          drwxrwx--x    2019-04-29 10:05
  ▶ dontpanic     drwxr-x---    2018-10-02 11:03
  ▶ drm           drwxrwx---    2018-10-02 11:03
  ▶ local         drwxr-x--x    2018-10-02 11:03
  ▶ lost+found    drwxrwx---    1969-12-31 19:00
  ▶ media         drwxrwx---    2018-10-02 11:04
  ▶ mediadrm      drwxrwx---    2018-10-02 11:03
  ▶ misc          drwxrwx--t    2018-10-02 11:03
  ▶ nativetest    drwxrwx--x    2018-05-21 20:04
  ▶ property      drwx------    2019-05-07 06:12
  ▶ resource-cache drwxrwx--x   2018-10-02 11:03
  ▶ security      drwx--x--x    2018-10-02 11:03
  ▶ system        drwxrwxr-x    2019-05-07 08:47
  ▶ tombstones    drwx------    2018-12-07 21:14
```

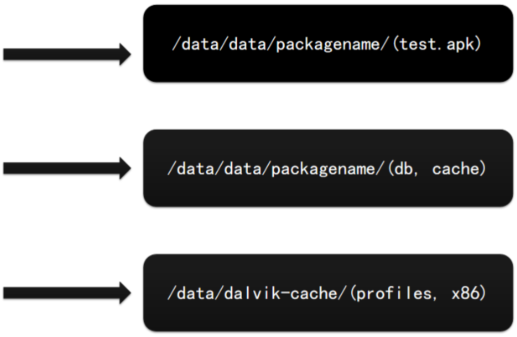
总结：安装的原理：

安装其实就是把apk文件copy到了对应的目录：

1. data/app/包名 ————安装时把 apk文件复制到此目录，---- 可以将文件取出并安装，和我们本身的apk 是一样的。

2. data/data/包名———— 开辟存放应用程序的文件数据的文件夹包括我们应用的 so库，缓存文件 等等。

3. 将apk中的dex文件安装到data/dalvik-cache目录下(dex文件是dealvik虚拟机的可执行文件，其大小约为原始apk文件大小的四分之一)



/data/data/packagename/(test.apk)

/data/data/packagename/(db, cache)

/data/dalvik-cache/(profiles, x86)