**PackageManagerService**

**1.** **PackageManagerService的启动**

(1).Installer 启动：

创建Installer对象，同时检测和/system/bin/installd 进程的通信情况。

(2).使用mPackages来创建Settings对象。

final ArrayMap<String, PackageParser.Package> **mPackages** =

new ArrayMap<String, PackageParser.Package>();

mPackages 作为Settings对象中的一个锁对象，同时创建：

data/system 目录；

在data/system 目录下创建packages.xml、packages-backup.xml、packages.list、packages-stopped.xml、

packages-stopped-backup.xml文件。

(3).为Settings对象添加SharedUser：

public static final int SYSTEM\_UID = 1000; //Process.SYSTEM\_UID 为 system code runs。

public static final int FIRST\_APPLICATION\_UID = 10000; //第一个app的uid号。

public static final int LAST\_APPLICATION\_UID = 19999; //最后一个app类型的uid号。

mSettings.**addSharedUserLPw**("android.uid.system", Process.SYSTEM\_UID,

ApplicationInfo.FLAG\_SYSTEM, ApplicationInfo.PRIVATE\_FLAG\_PRIVILEGED);

**//Settings.java**

以android.uid.system 为索引以SharedUserSetting为value来存放在mSharedUsers容器中。

final ArrayMap<String, SharedUserSetting> **mSharedUsers** =

new ArrayMap<String, SharedUserSetting>();

SharedUserSetting **addSharedUserLPw**(String name, int uid, int pkgFlags, int pkgPrivateFlags) {

s = new SharedUserSetting(name, pkgFlags, pkgPrivateFlags);

s.userId = uid;

if (addUserIdLPw(uid, s, name)) {

**mSharedUsers**.put(name, s); ---->add成功后将useID保存在容器中。

return s;

}

return null;

}

addUserIdLPw函数：

当UID大于LAST\_APPLICATION\_UID时候addUserId失败。

当UID大于FIRST\_APPLICATION\_UID小于LAST\_APPLICATION\_UID的时候，将SharedUserSetting对象加入到mUserIds容器中。mUserIds容器以uid - Process.FIRST\_APPLICATION\_UID为索引插入到mUserIds容器中。

当UID小于FIRST\_APPLICATION\_UID的时候以UID为索引插入到mOtherUserIds容器中。

**private boolean addUserIdLPw(int uid, Object obj, Object name) {**

if (uid > Process.LAST\_APPLICATION\_UID) {

return false;

}

if (uid >= Process.FIRST\_APPLICATION\_UID) {

int N = **mUserIds**.size();

final int index = uid - Process.FIRST\_APPLICATION\_UID;

while (index >= N) {

**mUserIds**.add(null);

N++;

}

if (**mUserIds**.get(index) != null) {

PackageManagerService.reportSettingsProblem(Log.ERROR,

"Adding duplicate user id: " + uid

+ " name=" + name);

return false;

}

**mUserIds**.set(index, obj);

} else {

if (**mOtherUserIds**.get(uid) != null) {

PackageManagerService.reportSettingsProblem(Log.ERROR,

"Adding duplicate shared id: " + uid

+ " name=" + name);

return false;

}

**mOtherUserIds**.put(uid, obj);

}

return true;

}

1. .创建PackageDexOptimizer对象：

通过PMS来创建mDexoptWakeLock对象来保证在做dexopt的时候CPU运行。

1. .使用android.fg线程的loop来创建mMoveCallbacks和mOnPermissionChangeListeners对象。
2. .初始化SystemConfig相关参数：

SystemConfig systemConfig = SystemConfig.getInstance();

mGlobalGids = systemConfig.getGlobalGids();

mSystemPermissions = systemConfig.getSystemPermissions();

mAvailableFeatures = systemConfig.getAvailableFeatures();

1. .创建PackageManager线程，使用该线程的looper创建PackageHandler对象，同时将PackageManager线程线程加入到watchdog的监视队列中。
2. .在data目录下创建data、app等相应的目录。

File dataDir = Environment.getDataDirectory();

mAppDataDir = new File(dataDir, "data"); ------>保存app的使用数据，包括db、cache、shared\_prefs。

mAppInstallDir = new File(dataDir, "app"); ------->app安装的数据包括apk、lib、odex。

mAppLib32InstallDir = new File(dataDir, "app-lib");------->

mAsecInternalPath = new File(dataDir, "app-asec").getPath();-------->

mUserAppDataDir = new File(dataDir, "user"); -------->某个用户的app使用数据

mDrmAppPrivateInstallDir = new File(dataDir, "app-private");-------->

1. .创建UserManagerService对象：

//data/system/users/0/: 用户0的一些系统数据包括settings设置、插件数据、账户信息、壁纸数据。

root@htc\_m8tl:/data/system/users/0 # ls

accounts.db

accounts.db-journal

appwidgets.xml

package-restrictions.xml

registered\_services

runtime-permissions.xml

settings\_global.xml

settings\_secure.xml

settings\_system.xml

wallpaper\_info.xml

//data/system/users/userlist.xml （所有的user）

root@htc\_m8tl:/data/system/users # cat userlist.xml

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

<users nextSerialNumber="10" version="5">

<guestRestrictions>

**<restrictions no\_outgoing\_calls="true" no\_sms="true" />**

</guestRestrictions>

<user id="0" />

</users>

//data/system/users/0.xml （user的详细信息）

root@htc\_m8tl:/data/system/users # cat 0.xml

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

<user id="0" serialNumber="0" flags="19" created="0" lastLoggedIn="1471490229940">

<name>test65598@gmail.com</name>

<restrictions />

</user>

readUserListLocked 函数解析data/system/users/userlist.xml文件，通过user id来解析userid.xml文件

来创建UserInfo对象，将创建的UserInfo对象保存在mUsers容器中。

private final SparseArray<UserInfo> mUsers = new SparseArray<UserInfo>();

readUserListLocked 函数解析data/system/users/userlist.xml文件，将restrictions保存到

mGuestRestrictions对象中。

private final Bundle mGuestRestrictions = new Bundle();

private UserManagerService(Context context, PackageManagerService pm,Object installLock, Object packagesLock, File dataDir, File baseUserPath) {

mUsersDir = new File(dataDir, USER\_INFO\_DIR); //data/system/users/

mUsersDir.mkdirs();

File userZeroDir = new File(mUsersDir, "0"); //data/system/users/0/

userZeroDir.mkdirs();

mUserListFile = new File(mUsersDir, USER\_LIST\_FILENAME);//data/system/users/userlist.xml

initDefaultGuestRestrictions();

readUserListLocked();

sInstance = this;

}

1. .通过systemConfig中从etc/permissions/文件中中读取的Permissions、SharedLibraries创建BasePermission对象初始化mSettings.mPermissions容器和创建SharedLibraryEntry对象来初始化mSharedLibraries容器。

(etc/sysconfig/ 、etc/permissions/)

详细见PMS-permisssion-readLPw文档。

1. .packageManager的扫描流程：

扫描函数详解：

**private PackageParser.Package scanPackageLI(File scanFile, int parseFlags, int scanFlags,**

long currentTime, UserHandle user) throws PackageManagerException {

PackageParser pp = new PackageParser();

PackageParser.Package pkg = null;

pkg = **pp.parsePackage**(scanFile, parseFlags);

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

PackageParser.Package scannedPkg = **scanPackageLI**(pkg, parseFlags, scanFlags

| SCAN\_UPDATE\_SIGNATURE, currentTime, user);

}

**PackageParser.java**

PackageLite:单个轻量的package对象。

ApkLite:单个轻量的apk对象。

AssetManager.addAssetPath:将路径加入到assetmanager中。

Package:最终package解析生成的对象,返回给PMS。

**//PackageParser.java**

**public Package parsePackage(File packageFile, int flags) throws PackageParserException {**

if (packageFile.isDirectory()) {

return **parseClusterPackage**(packageFile, flags); ---->解析目录下所有apk文件。

} else {

return parseMonolithicPackage(packageFile, flags); ----->解析单个apk文件。

}

}

**private Package parseClusterPackage(File packageDir, int flags) {**

**final PackageLite lite = parseClusterPackageLite(packageDir, 0); (1)**

final AssetManager assets = new AssetManager();

**loadApkIntoAssetManager(assets, lite.baseCodePath, flags);**

if (!ArrayUtils.isEmpty(lite.splitCodePaths)) {

for (String path : lite.splitCodePaths) {

**loadApkIntoAssetManager(assets, path, flags); （2）**

}

}

final File baseApk = new File(lite.baseCodePath);

final Package pkg = **parseBaseApk(baseApk, assets, flags); (3)**

if (!ArrayUtils.isEmpty(lite.splitNames)) {

final int num = lite.splitNames.length;

pkg.splitNames = lite.splitNames;

pkg.splitCodePaths = lite.splitCodePaths;

pkg.splitRevisionCodes = lite.splitRevisionCodes;

pkg.splitFlags = new int[num];

pkg.splitPrivateFlags = new int[num];

for (int i = 0; i < num; i++) {

**parseSplitApk(pkg, i, assets, flags); （4）**

}

}

pkg.codePath = packageDir.getAbsolutePath();

return pkg;

}

(1)parseClusterPackageLite函数：

1.遍历packageDir路径下所有的apk文件，调用parseApkLite函数通过解析AndroidManifest.xml文件中manifest tag的package、coreApp等元素为每一个Apk文件创建ApkLite轻量级对象，将每个ApkLite对象保存在ArrayMap容器中，以第一个apk中的package name作为整个包名。

2.使用第一个创建的ApkLite对象来创建PackageLite对象。

(2)loadApkIntoAssetManager函数：

将所有的apk路径加入到assetmanager。

(3)parseBaseApk函数：

1.通过apk的path从 AssetManager中获取资源包的cookie，通过AssetManager和cookie创建AndroidManifest.xml文件的解析器。

2.调用parseBaseApk函数创建Package对象：

@解析manifest tag为coreApp、mSharedUserId等赋值。

@解析application tag 调用parseBaseApplication函数解析application tag下的配置信息包括Application\_name、allowBackup、hardwareAccelerated（4.0以上不管是否有该值ai.flags都设为ApplicationInfo.FLAG\_HARDWARE\_ACCELERATED）

@解析Activity/receiver/service/provider/activity-alias/meta-data/library/uses-library/uses-package

这些TAG来创建相应的对象保存在相应的容器中。

ai.className /ai.manageSpaceActivityName/ai.backupAgentName/ai.flags/

ai.fullBackupContent/ai.nonLocalizedLabel/ai.icon/ai.logo/ai.banner/ai.theme/

ai.descriptionRes/ai.permission/ai.taskAffinity/ai.processName/ai.enabled/

ai.privateFlags/ai.uiOptions

Activity/receiver/service/provider/activity-alias/meta-data/library/uses-library/uses-package

(4)parseSplitApk函数：对目录下其他apk文件进行解析。

通过Application tag下的组件的信息创建组件对象加入到Package相应的容器中。

scanPackageDirtyLI函数详细解析：

1.当Package.mSharedUserId != null时创建SharedUserId。

通过保存在Package对象中的mSharedUserId字符串（从mainfiest中解析获得），调用

mSettings.getSharedUserLPw函数创建SharedUserSetting对象和新的UserId，保存mUserIds和mSharedUsers容器当中。

2.创建PackageSetting对象：

mSettings.getPackageLPw：为PackageSetting创建UID。

final ArrayMap<String, PackageSetting> **mPackages** = new ArrayMap<String, PackageSetting>();

**private PackageSetting getPackageLPw(String name, PackageSetting origPackage,**

**String realName, SharedUserSetting sharedUser, File codePath, File resourcePath,) {**

PackageSetting p = mPackages.get(name);

if (p != null) {

}

if (p == null) {

p = new PackageSetting(name, realName, codePath, resourcePath,

legacyNativeLibraryPathString,primaryCpuAbiString,secondaryCpuAbiString,null,vc,pkgFlags, pkgPrivateFlags);

p.setTimeStamp(codePath.lastModified());

p.sharedUser = sharedUser;

if ((pkgFlags&ApplicationInfo.FLAG\_SYSTEM) == 0) {

}

if (sharedUser != null) {

p.appId = sharedUser.userId;

} else {

**p.appId = newUserIdLPw(p);**

}

} else {

}

1. 为package中的application对象赋值UID。

pkg.applicationInfo.uid = pkgSetting.appId;

1. 签名验证 **\*\*\*\*\*//TODO**

if (shouldCheckUpgradeKeySetLP(pkgSetting, scanFlags)) {

if (checkUpgradeKeySetLP(pkgSetting, pkg)) {

}

}else {

**verifySignaturesLP(pkgSetting, pkg);**

}

1. 检查/data/user/0目录：

File dataPath;

dataPath = Environment.getDataUserPackageDirectory(pkg.volumeUuid,

UserHandle.USER\_OWNER, pkg.packageName);

if (dataPath.exists()) {

StructStat stat = Os.stat(dataPath.getPath());

**currentUid = stat.st\_uid;**  **//读取文件目录的UID**

currentGid = stat.st\_gid; **//读取文件目录的GID**

**//data目录的uid和package中记录的uid不一致**

if (currentUid != pkg.applicationInfo.uid) {

**//currentUid 出现问题。**

if (currentUid == 0) {

int ret = **mInstaller.fixUid**(pkg.volumeUuid, pkgName,

pkg.applicationInfo.uid, pkg.applicationInfo.uid);

}

//如果installer修复失败用删除data目录 使用installer重新创建目录

int ret = removeDataDirsLI(pkg.volumeUuid, pkgName); **//mInstaller.remove**

if (ret >= 0)

ret = createDataDirsLI(pkg.volumeUuid, pkgName, pkg.applicationInfo.uid,

pkg.applicationInfo.seinfo); **//mInstaller.install/mInstaller.createUserData**

}

pkg.applicationInfo.dataDir = dataPath.getPath();

} else {

int ret = createDataDirsLI(pkg.volumeUuid, pkgName, pkg.applicationInfo.uid,

pkg.applicationInfo.seinfo);**//mInstaller.install/mInstaller.createUserData**

}

通过前面创建的uid使用Installed进程对package进行安装同时为该package创建相应的用户数据。

Installd

//installd.cpp

mkdir/chmod/selinux\_android\_setfilecon/chown

创建package 的路径。

static int do\_install(char \*\*arg, char reply[REPLY\_MAX] \_\_unused)

{

return install(parse\_null(arg[0]), arg[1], atoi(arg[2]), atoi(arg[3]), arg[4]); /\* uuid, pkgname, uid, gid, seinfo \*/

}

int install(const char \*uuid, const char \*pkgname, uid\_t uid, gid\_t gid, const char \*seinfo)

{

if ((uid < AID\_SYSTEM) || (gid < AID\_SYSTEM)) {

ALOGE("invalid uid/gid: %d %d\n", uid, gid);

return -1;

}

std::string \_pkgdir(create\_data\_user\_package\_path(uuid, 0, pkgname));

const char\* pkgdir = \_pkgdir.c\_str();

if (**mkdir**(pkgdir, 0751) < 0) {

ALOGE("cannot create dir '%s': %s\n", pkgdir, strerror(errno));

return -1;

}

if (**chmod**(pkgdir, 0751) < 0) {

ALOGE("cannot chmod dir '%s': %s\n", pkgdir, strerror(errno));

unlink(pkgdir);

return -1;

}

if (**selinux\_android\_setfilecon**(pkgdir, pkgname, seinfo, uid) < 0) {

ALOGE("cannot setfilecon dir '%s': %s\n", pkgdir, strerror(errno));

unlink(pkgdir);

return -errno;

}

if (**chown**(pkgdir, uid, gid) < 0) { **//改变文件指定的所有者。**

ALOGE("cannot chown dir '%s': %s\n", pkgdir, strerror(errno));

unlink(pkgdir);

return -1;

}

return 0;

}

1. 为Package指定32/64位属性：

if ((scanFlags & SCAN\_NEW\_INSTALL) == 0) {

derivePackageAbi(pkg, scanFile, cpuAbiOverride, true /\* extract libs \*/, pkgSetting, scanFlags);

if (isSystemApp(pkg) && !pkg.isUpdatedSystemApp() &&

pkg.applicationInfo.primaryCpuAbi == null) {

setBundledAppAbisAndRoots(pkg, pkgSetting);

setNativeLibraryPaths(pkg);

}

} else {

if ((scanFlags & SCAN\_MOVE) != 0) {

pkg.applicationInfo.primaryCpuAbi = pkgSetting.primaryCpuAbiString;

pkg.applicationInfo.secondaryCpuAbi = pkgSetting.secondaryCpuAbiString;

}

setNativeLibraryPaths(pkg);

}

当package为非安装的扫描，调用derivePackageAbi函数来为package指定abi属性和设置nativelibray 路径。

当为安装的package直接进行native libray 路径的设置。

derivePackageAbi函数：

@:setNativeLibraryPaths(pkg)为ApplicationInfo对象中的nativeLibraryDir路径赋值 codePath+lib；

@:创建NativeLibraryHelper.Handle 对象：

获取所有apk路径的apk包的代码路径创建相应的ZipFileRO对象进行对apk包的解压。

保存在Handle对象的final long[] apkHandles变量中。

private static Handle create(List<String> codePaths, boolean multiArch,boolean extractNativeLibs) throws IOException {

final int size = codePaths.size();

final long[] apkHandles = new long[size];

for (int i = 0; i < size; i++) {

final String path = codePaths.get(i);

apkHandles[i] = nativeOpenApk(path);

}

return new Handle(apkHandles, multiArch, extractNativeLibs);

}

static jlong com\_android\_internal\_content\_NativeLibraryHelper\_openApk(JNIEnv \*env, jclass, jstring apkPath)

{

ScopedUtfChars filePath(env, apkPath);

ZipFileRO\* zipFile = ZipFileRO::open(filePath.c\_str());

return reinterpret\_cast<jlong>(zipFile);

}

@：支持多框架的app（ps.pkgFlags包含ApplicationInfo.FLAG\_MULTIARCH标志）进行检查ABI的支持：

if (isMultiArch(pkg.applicationInfo)) {

int abi32 = PackageManager.NO\_NATIVE\_LIBRARIES;

int abi64 = PackageManager.NO\_NATIVE\_LIBRARIES;

if (Build.SUPPORTED\_32\_BIT\_ABIS.length > 0) { ----->手机是否支持32位运算

abi32 = NativeLibraryHelper.findSupportedAbi(handle, Build.SUPPORTED\_32\_BIT\_ABIS);

}

if (Build.SUPPORTED\_64\_BIT\_ABIS.length > 0) { ----->手机是否支持64位运算

abi64 = NativeLibraryHelper.findSupportedAbi(handle, Build.SUPPORTED\_64\_BIT\_ABIS);

}

if (abi64 >= 0) {

pkg.applicationInfo.**primaryCpuAbi** = Build.SUPPORTED\_64\_BIT\_ABIS[abi64]; ---->设置该app为64

}

if (abi32 >= 0) {

final String abi = Build.SUPPORTED\_32\_BIT\_ABIS[abi32];

if (abi64 >= 0) {

pkg.applicationInfo.**secondaryCpuAbi** = abi; --->设置该app的secondaryCpuAbi为32

} else {

pkg.applicationInfo.**primaryCpuAbi** = abi; --->设置该app的primaryCpuAbi 为32当abi64<0

}

}

}

NativeLibraryHelper.findSupportedAbi:

static int findSupportedAbi(JNIEnv \*env, jlong apkHandle, jobjectArray supportedAbisArray) {

const int numAbis = env->GetArrayLength(supportedAbisArray);//java层要检查的abi数组个数

Vector<ScopedUtfChars\*> supportedAbis.add(supportedAbisArray);//java层要检查的abi数组

ZipFileRO\* zipFile = reinterpret\_cast<ZipFileRO\*>(apkHandle);//apk文件的handle用来遍历压缩包

while ((entry = it->next()) != NULL) {

const char\* abiOffset = fileName + APK\_LIB\_LEN;//检查apk压缩包中lib路径下的文件夹名称

const size\_t abiSize = lastSlash - abiOffset;

for (int i = 0; i < numAbis; i++) {

const ScopedUtfChars\* abi = supportedAbis[i];

if (abi->size() == abiSize && !strncmp(abiOffset, abi->c\_str(), abiSize)) {//是否有java层传过来的

if (((i < status) && (status >= 0)) || (status < 0) ) { //abi命名的文件夹

status = i;

}

}

}

}

return status;

}

7.保存创建的PackageSetting到Settings和PackageParser.Package到mPackages容器中。

mSettings.insertPackageSettingLPw(pkgSetting, pkg);

mPackages.put(pkg.applicationInfo.packageName, pkg);

1. 改变PackageSetting中的firstInstallTime和lastUpdateTime时间。
2. 将PackageParser.Package中从mainfiest中解析出来的四大组件对象和Permission组件PermissionGroup组件保存PAMS中的组件容器中。

10.SYSTEM\_SCAN：

1.获取BOOTCLASSPATH和SYSTEMSERVERCLASSPATH环境变量下的路径，将所有的文件路径加入到alreadyDexOpted容器中，这是因为这些路径下的jar包都已经在zygote启动的时候做过dexopt。

2.将mSharedLibraries容器中的lib进行dexopt优化，同时加入到alreadyDexOpted容器中。

3.扫描所有/framework/路径下不在alreadyDexOpted容器中的所有apk和jar包进行dexopt优化。

4.

11.DATA\_SCAN

1. 调用writeLPr函数将settins内存中的对象写入到xml中。

(13)创建PackageInstallerService对象。

1.创建PackageInstaller线程和对应的handler

2.创建install\_sessions目录和install\_sessions.xml文件。

1. performBootDexOpt函数：

//是否第一次开机 mRestoredSettings package.list文件是否存在。

public boolean isFirstBoot() {

return !mRestoredSettings;

}

//是否fota升级

public boolean isUpgrade() {

return mIsUpgrade;

}

多个apk在同一个进程中运行考虑：

Apk的abi属性是否一致。

**3.卸载**

**4.移动**

Binder接口：

MoveCallbacks mMoveCallbacks = new MoveCallbacks(FgThread.get().getLooper());

IMediaContainerService mContainerService;

**5.PMS中使用的工具**

1. Installer

getsize；

**6.知识点**