**安装**

**1.copy**

创建InstallParams使用PackageHandler发送INIT\_COPY消息给packagemanager线程进行处理。

final Message msg = mHandler.obtainMessage(INIT\_COPY);

msg.obj = new InstallParams(origin, null, observer, installFlags, installerPackageName,

null, verificationParams, user, packageAbiOverride, null);

mHandler.sendMessage(msg);

packagemanager线程进行处理：链接到DefaultContainerService服务中，返回该服务中的

IMediaContainerService binder node对象。同时将InstallParams 加入到mPendingInstalls容器中。

当IMediaContainerService binder node对象返回后发送MCS\_BOUND消息给packagemanager线程进行

处理。

调用HandlerParams的startCopy进行拷贝的操作。

private abstract class HandlerParams {

final boolean startCopy() {

boolean res;

try {

if (++mRetries > MAX\_RETRIES) {

} else {

handleStartCopy();

res = true;

}

} catch (RemoteException e) {

}

handleReturnCode();

return res;

}

调用子类InstallParams的handleStartCopy进行拷贝的操作：

创建PackageInfoLite对象：PackageParser来解析AndroidManifest.xml获取相应的信息。

public void **handleStartCopy**() throws RemoteException {

PackageInfoLite pkgLite = null;

pkgLite = mContainerService.getMinimalPackageInfo(origin.resolvedPath, installFlags,

packageAbiOverride);

}

public class DefaultContainerService extends IntentService {

@Override

public PackageInfoLite getMinimalPackageInfo(String packagePath, int flags,String abiOverride) {

PackageInfoLite ret = new PackageInfoLite();

pkg = PackageParser.parsePackageLite(packageFile, 0);

ret.**recommendedInstallLocation** = PackageHelper.**resolveInstallLocation**(context,

pkg.packageName, pkg.installLocation, sizeBytes, flags);

}

**resolveInstallLocation函数的解析：**

PackageInfoLite.recommendedInstallLocation检查当前安装设备的空间大小是否满足和安装在内存储还是外存储上。

安装路径：

1.如果有安装时指定的installFlags中是否赋值INSTALL\_EXTERNAL或者INSTALL\_INTERNAL使用该值来指定。

2.如果没有指定的installFlags中安装路径，使用AndroidManifest.xml中的installLocation来指定。

3.如果前面都没有指定使用内存储来进行安装。

检查安装的存储空间是否满足大小（apk+lib）：

1.如果大小满足，那么返回指定的安装存储设备。

2.如果不满足返回安装存储空间不足的错误。

返回到handleStartCopy函数中：

当pkgLite.recommendedInstallLocation为RECOMMEND\_FAILED\_INSUFFICIENT\_STORAGE时调用mInstaller.freeCache函数来清理缓存。

**创建InstallArgs进行拷贝：**

createInstallArgs使用params.move和params.installFlags来创建MoveInstallArgs、AsecInstallArgs、FileInstallArgs三个对象中的一个。

然后调用对象的copyApk函数进行拷贝的工作。

final InstallArgs args = createInstallArgs(this);

mArgs = args;

ret = args.copyApk(mContainerService, true);

以FileInstallArgs为示例：

int copyApk(IMediaContainerService imcs, boolean temp) throws RemoteException {

//创建tempDir

final File tempDir = mInstallerService.allocateStageDirLegacy(volumeUuid);

codeFile = tempDir;

resourceFile = tempDir;

//拷贝apk文件

int ret = PackageManager.INSTALL\_SUCCEEDED;

ret = imcs.copyPackage(origin.file.getAbsolutePath(), target);

//拷贝lib库文件

final File libraryRoot = new File(codeFile, LIB\_DIR\_NAME);

NativeLibraryHelper.Handle handle = null;

try {

handle = NativeLibraryHelper.Handle.create(codeFile);

ret = NativeLibraryHelper.copyNativeBinariesWithOverride(handle, libraryRoot,

abiOverride);

}

return ret;

}

copyApk函数的解析：

1. 创建tempDir

在PackageInstallerService的allocateStageDirLegacy函数中分配相应的sessionId，然后在buildStageDir函数中使用volumeUuid和sessionId来创建/data/app/vmdl+ sessionId + .tmp 或者 /mnt/expand/volumeUuid/vmdl+ sessionId + .tmp目录。

1. 拷贝apk文件

将apk文件拷贝到缓存目录中以base.apk来命名。

1. 拷贝lib库文件

三个abi属性：

Build.SUPPORTED\_64\_BIT\_ABIS：ro.product.cpu.abilist64

Build.SUPPORTED\_32\_BIT\_ABIS：ro.product.cpu.abilist32

Build.SUPPORTED\_ABIS：ro.product.cpu.abilist

通过Build.SUPPORTED\_ABIS属性获取device支持的cpu架构。

分app是否支持多架构cpu：

当支持多架构cpu：

不支持多架构cpu：

copyNativeBinariesForSupportedAbi通过支持的abi来拷贝相应的lib库。

1.在缓存目录下创建lib文件夹。

2.通过findSupportedAbi函数来查找apk中是否有支持的abi的so库，如果有支持的so库，在目录/data/app/vmdl1496164171.tmp/lib创建abi目录。Sample：/data/app/vmdl1496164171.tmp/lib/arm。

3.通过copyNativeBinaries函数将base.apk中找到的支持device的abi so库拷贝到该目录下。

拷贝完成后返回到InstallParams的handleReturnCode函数：

开始进行安装的流程。

void handleReturnCode() {

if (mArgs != null) {

processPendingInstall(mArgs, mRet);

}

}

**installPackageLI：**

1. **解析apk获取PackageParser.Package对象，该对象中包括AndroidManifest.xml**

pkg = pp.parsePackage(tmpPackageFile, parseFlags);

**2.搜集签名**

PackageParser pp = new PackageParser();

pp.collectCertificates(pkg, parseFlags);

pp.collectManifestDigest(pkg);

collectCertificates函数的解析：

private static void collectCertificates(Package pkg, File apkFile, int flags) throws PackageParserException {

final String apkPath = apkFile.getAbsolutePath();

StrictJarFile jarFile = null;

try {

**jarFile = new StrictJarFile(apkPath);**

final ZipEntry manifestEntry = jarFile.findEntry(ANDROID\_MANIFEST\_FILENAME);

final List<ZipEntry> toVerify = new ArrayList<>();

toVerify.add(manifestEntry);

if ((flags & PARSE\_IS\_SYSTEM) == 0) {

final Iterator<ZipEntry> i = jarFile.iterator();

while (i.hasNext()) {

final ZipEntry entry = i.next();

if (entry.isDirectory()) continue;

if (entry.getName().startsWith("META-INF/")) continue;

if (entry.getName().equals(ANDROID\_MANIFEST\_FILENAME)) continue;

**toVerify.add(entry);**

}

}

for (ZipEntry entry : toVerify) {

final Certificate[][] entryCerts = loadCertificates(jarFile, entry);

if (ArrayUtils.isEmpty(entryCerts)) {

throw new PackageParserException(INSTALL\_PARSE\_FAILED\_NO\_CERTIFICATES,

}

final Signature[] entrySignatures = convertToSignatures(entryCerts);

if (pkg.mCertificates == null) {

pkg.mCertificates = entryCerts;

pkg.mSignatures = entrySignatures;

pkg.mSigningKeys = new ArraySet<PublicKey>();

for (int i=0; i < entryCerts.length; i++) {

pkg.mSigningKeys.add(entryCerts[i][0].getPublicKey());

}

} else {

if (!Signature.areExactMatch(pkg.mSignatures, entrySignatures)) {

throw new PackageParserException(

}

}

}

} catch (GeneralSecurityException e) {

throw new PackageParserException(INSTALL\_PARSE\_FAILED\_CERTIFICATE\_ENCODING,

"Failed to collect certificates from " + apkPath, e);

} catch (IOException | RuntimeException e) {

throw new PackageParserException(INSTALL\_PARSE\_FAILED\_NO\_CERTIFICATES,

}

1. 创建StrictJarFile对象：

public StrictJarFile(String fileName) throws IOException, SecurityException {

this.nativeHandle = nativeOpenJarFile(fileName);

this.raf = new RandomAccessFile(fileName, "r");

HashMap<String, byte[]> **metaEntries** = getMetaEntries();

this.**manifest** = new Manifest(metaEntries.get(JarFile.MANIFEST\_NAME), true);

this.**verifier** = new JarVerifier(fileName, manifest, metaEntries);

Set<String> files = this.manifest.getEntries().keySet();

for (String file : files) {

if (findEntry(file) == null) {

throw new SecurityException(fileName + ": File " + file + " in manifest does not exist");

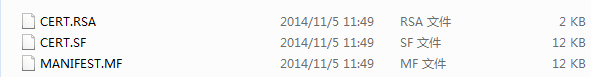
}

}

**isSigned** = verifier.readCertificates() && verifier.isSignedJar();

}

通过getMetaEntries函数将META-INF/文件夹下的文件读出保存在metaEntries 容器；



MANIFEST.MF中记录的是apk中所有文件的摘要值；CERT.SF中记录的是对MANIFEST.MF的摘要值，包括整个文件的摘要，还有文件中每一项的摘要；而CERT.RSA中记录的是对CERT.SF文件的签名，以及签名的公钥。

以META-INF/MANIFEST.MF文件来创建manifest对象。

通过调用readCertificates函数检查apk是否被签名，没有返回false,有签名进行签名验证.通过签名文件CERT.SF和证书文件CERT.RSA调用JarUtils.verifySignature验证CERT.SF文件签名是否正确。如果验证失败，则会抛出GeneralSecurityException异常；而如果验证成功，则会返回签名的证书链。

通过SHA1摘要算法验证MANIFEST.MF是否被修改过。

在StrictJarFile构造的过程中就已经完成了两步验证：**一是**通过在CERT.RSA文件中记录的签名信息，验证了CERT.SF没有被篡改过；**二是**通过CERT.SF文件中记录的摘要值，验证了MANIFEST.MF没有被修改过。

**三是**apk内文件的摘要值要与MANIFEST.MF文件中记录的一致验证.将除文件夹和META-INF/文件内容外所有的文件加入到toVerify容器中。

1. 通过installFlags & PackageManager.INSTALL\_REPLACE\_EXISTING来确认用户是否要替换已经安装的app，如果是的话检查mPackages容器中是否已经包含该app。
2. **调用derivePackageAbi函数：**

derivePackageAbi：  
 NativeLibraryHelper.Handle对象结构和创建：

public static class Handle implements Closeable {

final long[] apkHandles;

}

NativeLibraryHelper.Handle handle = null;

handle = NativeLibraryHelper.Handle.create(scanFile);

apkHandles保存扫描路径下所有apk文件的句柄。

调用nativeOpenApk函数来为每个apk文件创建ZipFileRO 对象的handle。

public static Handle create(PackageLite lite) throws IOException {

return create(lite.getAllCodePaths(), lite.multiArch, lite.extractNativeLibs);

}

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);

if (apkHandles[i] == 0) {

// Unwind everything we've opened so far

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

nativeClose(apkHandles[j]);

}

throw new IOException("Unable to open APK: " + path);

}

}

return new Handle(apkHandles, multiArch, extractNativeLibs);

}

指定ApplicationInfo的primaryCpuAbi、secondaryCpuAbi值为后面启动进程时来决定使用zygote32还是zygote64来启动。

1.isMultiArch（AndroidManifestApplication\_multiArch）

copyNativeBinariesForSupportedAbi

找到支持的abi，如果找到支持的abi同时拷贝so库。

findSupportedAbi

通过所有apk文件的handle查询app是否支持的abilist中的abi，就是遍历lib文件夹下的文件目录是否有abilist中的abi命名的文件夹，如果支持的话返回abilist中的abi。

if (isMultiArch(pkg.applicationInfo)) {

int abi32 = PackageManager.NO\_NATIVE\_LIBRARIES;

int abi64 = PackageManager.NO\_NATIVE\_LIBRARIES;

if (Build.SUPPORTED\_32\_BIT\_ABIS.length > 0) {

if (extractLibs) {

abi32 = NativeLibraryHelper.**copyNativeBinariesForSupportedAbi**(handle,

nativeLibraryRoot, Build.SUPPORTED\_32\_BIT\_ABIS,useIsaSpecificSubdirs);

} else {

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

}

if (Build.SUPPORTED\_64\_BIT\_ABIS.length > 0) {

if (extractLibs) {

abi64 = NativeLibraryHelper.copyNativeBinariesForSupportedAbi(handle,

nativeLibraryRoot, Build.SUPPORTED\_64\_BIT\_ABIS,useIsaSpecificSubdirs);

} else {

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

}

}

if (abi64 >= 0) {

pkg.applicationInfo.primaryCpuAbi = Build.SUPPORTED\_64\_BIT\_ABIS[abi64];

}

if (abi32 >= 0) {

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

if (abi64 >= 0) {

pkg.applicationInfo.secondaryCpuAbi = abi;

} else {

pkg.applicationInfo.primaryCpuAbi = abi;

}

}

}

2.notMultiArch

String[] abiList = (cpuAbiOverride != null) ?

new String[] { cpuAbiOverride } : Build.SUPPORTED\_ABIS;

if (extractLibs) {

copyRet = NativeLibraryHelper.copyNativeBinariesForSupportedAbi(handle,

nativeLibraryRoot, abiList, useIsaSpecificSubdirs);

} else {

copyRet = NativeLibraryHelper.findSupportedAbi(handle, abiList);

}

1. **mPackageDexOptimizer.performDexOpt**

1.获取base.apk等apk文件的路径。

2.DexFile.getDexOptNeeded来确认apk文件是否需要做dexopt。

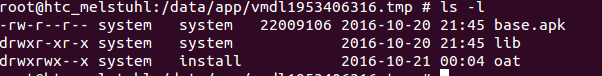
3.createoatdir

/oat 目录ui为1000、gid为1012

U 为：wrx

G为：wrx

Other: x



//frameworks\native\cmds\installd\commands.cpp

int create\_oat\_dir(const char\* oat\_dir, const char\* instruction\_set)

{

char oat\_instr\_dir[PKG\_PATH\_MAX];

if (validate\_apk\_path(oat\_dir)) {

ALOGE("invalid apk path '%s' (bad prefix)\n", oat\_dir);

return -1;

}

if (fs\_prepare\_dir(oat\_dir, S\_IRWXU | S\_IRWXG | S\_IXOTH, AID\_SYSTEM, AID\_INSTALL)) {

return -1;

}

if (selinux\_android\_restorecon(oat\_dir, 0)) {

ALOGE("cannot restorecon dir '%s': %s\n", oat\_dir, strerror(errno));

return -1;

}

snprintf(**oat\_instr\_dir**, PKG\_PATH\_MAX, "%s/%s", oat\_dir, **instruction\_set**);

if (fs\_prepare\_dir(oat\_instr\_dir, S\_IRWXU | S\_IRWXG | S\_IXOTH, **AID\_SYSTEM**, **AID\_INSTALL**)) {

return -1;

}

return 0;

}

1. 调用dexopt函数进行dex文件转oat文件的过程。

通过installd主线程fork一个运行/system/bin/dex2oat可执行文件的进程。

最终生成odex文件。

root@htc\_melstuhl:/data/app/vmdl1953406316.tmp/oat/arm # ls -l

-rw-r--r-- system u0\_a29999 21791152 2016-10-20 21:45 base.odex

主线程等待该子进程执行完毕。

基于LLVM架构开发的编译器执行过程



打包在APK里面的Dex字节码是通过LLVM翻译成本地机器指令的.

5.使用doRename函数重新以package name来命名安装的包名。

**installNewPackageLI**

1.scanPackageLI

**1.create the setting**

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

legacyNativeLibraryPathString, primaryCpuAbiString, secondaryCpuAbiString,

null /\* cpuAbiOverrideString \*/, vc, pkgFlags, pkgPrivateFlags);

p.setTimeStamp(codePath.lastModified());

p.sharedUser = sharedUser;

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

List<UserInfo> users = getAllUsers();

final int installUserId = installUser != null ? installUser.getIdentifier() : 0;

if (users != null && allowInstall) {

for (UserInfo user : users) {

final boolean installed = installUser == null

|| (installUserId == UserHandle.USER\_ALL

&& !isAdbInstallDisallowed(userManager, user.id))

|| installUserId == user.id;

p.setUserState(user.id, COMPONENT\_ENABLED\_STATE\_DEFAULT,

installed,

true, // stopped,

true, // notLaunched

false, // hidden

null, null, null,

false, // blockUninstall

INTENT\_FILTER\_DOMAIN\_VERIFICATION\_STATUS\_UNDEFINED, 0);

writePackageRestrictionsLPr(user.id);

}

}

}

**2.writePackageRestrictionsLPr：**

package-restrictions.xml

package-restrictions-backup.xml

1.将package-restrictions.xml文件进行备份。

2.遍历mPackages容器中所有的PackageSetting，写入以pkg为tag的apk的限制信息：

Installed：

Stopped：

notLaunched：

Hidden：

blockUninstall：

Enabled：

domainVerificationStatus：

app-link-generation：

enabled-components：

disabled-components：

3.写入以preferred-activities为tag的信息：

// The user's preferred activities associated with particular intent

// filters.

final SparseArray<PreferredIntentResolver> mPreferredActivities =

new SparseArray<PreferredIntentResolver>();

4.写入以persistent-preferred-activities为tag的信息：

// The persistent preferred activities of the user's profile/device owner

// associated with particular intent filters.

final SparseArray<PersistentPreferredIntentResolver> mPersistentPreferredActivities =

new SparseArray<PersistentPreferredIntentResolver>();

5.写入以crossProfile-intent-filters为tag的信息。

// For every user, it is used to find to which other users the intent can be forwarded.

final SparseArray<CrossProfileIntentResolver> mCrossProfileIntentResolvers =

new SparseArray<CrossProfileIntentResolver>();

6.写入以default-apps为tag的信息：

// For every user, it is used to find the package name of the default Browser App.

final SparseArray<String> mDefaultBrowserApp = new SparseArray<String>();

**3.创建appId：**

p.appId = newUserIdLPw(p);

1. **signatures验证：**

比较pkgSetting.signatures.mSignatures和pkg.mSignatures签名。

1. **创建data目录：**

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

if (ret < 0) {

// Error from installer

}

if (dataPath.exists()) {

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

} else {

Slog.w(TAG, "Unable to create data directory: " + dataPath);

pkg.applicationInfo.dataDir = null;

}

private int createDataDirsLI(String volumeUuid, String packageName, int uid, String seinfo) {

int[] users = sUserManager.getUserIds();

int res = mInstaller.install(volumeUuid, packageName, uid, uid, seinfo);

if (res < 0) {

return res;

}

for (int user : users) {

if (user != 0) {

res = mInstaller.createUserData(volumeUuid, packageName,

UserHandle.getUid(user, uid), user, seinfo);

if (res < 0) {

return res;

}

}

}

return res;

}

mInstaller.install：

创建一个属于ui和gid mode为0751下的一个data/user/0/package的路径。

1. **更新mSettings和mPackages**

// Add the new setting to mSettings

mSettings.insertPackageSettingLPw(pkgSetting, pkg);

// Add the new setting to mPackages

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

**7.更新PMS中的mReceivers、mServices、mProviders、mActivities、mInstrumentation**

1. **updateSettingsLI**

writing config files：

1.backup packages.xml文件。

2.写入mPermissionTrees tag。

3.写入permissions tag：

// Mapping from permission names to info about them.

final ArrayMap<String, BasePermission> **mPermissions** =

new ArrayMap<String, BasePermission>();

权限类型：

正常：final static int TYPE\_NORMAL = 0;

内置：final static int TYPE\_BUILTIN = 1;

动态：final static int TYPE\_DYNAMIC = 2;

mPermissions容器在开机的时候将/etc/permissions目录下的xml文件中的Permission写入到该容器中：TYPE\_BUILTIN（该类型permission不写入到packages.xml文件中）

在安装app的时候，会将app的mainfiest中的所有tag 为permission保存到mPermissions容器中

：TYPE\_NORMAL

在PermissionTree中的app可以通过addPermission/addPermissionAsync接口来进行Permissions的添加。

Permission：

定义的permisssion：sourcePackage为定义该权限的pkg。

定义在mianfest中：在定义activity，注册brodcast时添加权限控制。

<permission android:name="com.android.systemui.permission.SELF"

android:protectionLevel="signature" />

有12中保护等级：

public static final int PROTECTION\_NORMAL = 0;

public static final int PROTECTION\_DANGEROUS = 1;

public static final int PROTECTION\_SIGNATURE = 2;

。。。。。。

requestedPermissions：

用户请求时需要的权限，定义在mainfest中。

<uses-permission android:name="android.permission.SET\_WALLPAPER"/>

1. writePackage将容器mPackages写入到xml文件中。

final ArrayMap<String, PackageSetting> mPackages =

new ArrayMap<String, PackageSetting>();

包括app的application information、requestedpermission、etc。

1. writeDisabledSysPackageLPr 将容器mDisabledSysPackages 写入到xml文件中。
2. mSharedUsers
3. mPackagesToBeCleaned
4. mRenamedPackages
5. writePackageListLPr

更新packages.list文件。

1. writeAllUsersPackageRestrictionsLPr

更新package-restrictions.xml文件。

1. writeAllRuntimePermissionsLPr

更新runtime-permissions.xml文件。

1. updatePermissionsLPw：

将runtime 和install 权限更新到PermissionsState对象中。

1. write config：

将更新的权限分别更新到package.xml和runtime-permissions.xm文件中。

补充：

1. 当apk拷贝的时候进行install location 获取成功之后调用installLocationPolicy函数检查该app是否已经安装，如果已经安装需要检查该package是否降级安装。

// Check for downgrading.