## TODO

Git log 看看installd的变更，和pm的变更？？

技术学习：

/data/**user/0/**dji.prof.mg2/lib是什么时候创建的？什么时候链接到**priv-app**目录的

**/system/priv-app/mg2/lib/arm**

1|ag411:/data/user/0/dji.prof.mg2 # ls -al | grep lib

lrwxrwxrwx 1 root root 28 2013-01-18 16:50 **lib -> /system/priv-app/mg2/lib/arm**

drwx------ 2 u0\_a43 u0\_a43 3488 2020-01-14 19:27 lib-main

/data/app-lib/目录目前为空，是为了兼容5.1么

nativeLibraryDir是什么时候复制的，为什么系统预装和用户安装都能能够准确更新

lib-main是啥，是进入界面才有的工作，为啥要这么设计？

## 版本差異

Android9.0采用binder通信了

Android7.1還在Local\_socket通信

# 测试入口

frameworks\native\cmds\installd\tests\installd\_utils\_test.cpp

# Installd

在应用程序的管理工作中，有时候需要对存储设备做一些操作，比如创建目录修改目录权限等，这些操作有的是需要特权级的权限。PackageMS存活在system\_server进程中，这个进程的用户为system，它是没有特权级root的权限，不能完成在文件系统中更改目录、复制、删除文件、APK文件优化等操作，出于安全的角度考虑，Android单独将一部分需要特权的工作，转交给installd进程去完成。因此这里引入了installd进程

installd是一个native的守护进程，在init.rc中定义

service installd /system/bin/installd

class main

socket installd stream 600 system system

所以installd进程创建的socket名为：/dev/socket/installd， 600表示socket的用户权限为可读可写，system表示用户和用户组。

frameworks/native/cmds/installd/



ps | grep installd

install 404 1 2264 704 00b57be4 8b895b18 S /system/bin/installd

在installd的入口函数中，首先是做一些初始化的工作，然后放弃自己的root用户身份，改变了自己的用户类型和组用户类型，但同时它保留了设置目录权限，以及修改目录属主的权限：

在PackageMS扫描安装应用的过程中，给installd先后发送了如下消息：

* install
* dexopt

这三个消息分别由do\_install，do\_dexopt函数来处理。

* do\_install函数为应用创建了以下目录，在install过程中，同时为创建的目录设置uid和gid。
  + 数据目录:/data/data/packageName/
  + lib目录：/data/app-lib/packageName
  + lib符号链接:/data/data/packageName/lib → /data/app-lib/packageName
* do\_dexopt中调用dexopt函数处理apk文件。主要流程为：
  + 首先判断apk所在目录下面是否存在同名的odex文件，如果存在就直接返回；
  + 以apk文件的路径为名创建dex路径。比如在debug版本中，/system/app/Settings.apk的dex文件将会保存在： /data/dalvik-cache/system@[app@Settings.apk](mailto:app@Settings.apk)@classes.dex文件中。dex文件命名规则是/data/dalvik-cache字符串拼接apk的路径，再拼“/classs.dex"。然后将”/data/dalvik-cache/“字符串后面的‘/’替换为@字符。
  + 最后调用系统工具/system/bin/dexopt来提取dex文件。





LocalSocket通信（右）：   


他是什么，有什么用，怎么用，

哪里来的，为什么会出来，存在的合理性，背景

是什么，现状

前景，哪里去，哪里会用到，怎么用

## 1 main()入口函数

installd.cpp

**int** main(**const int** argc, **char** \*argv[]) {  
 **return** android::installd::installd\_main(argc, argv);  
}

int main(const int argc, const char \*argv[]) {

char buf[BUFFER\_MAX];//BUFFER\_MAX:1024: input buffer for commands

struct sockaddr addr;

socklen\_t alen;

int lsocket, s, count;

int selinux\_enabled = (is\_selinux\_enabled() > 0);

ALOGI("installd firing up\n");

union selinux\_callback cb;

cb.func\_log = log\_callback;

selinux\_set\_callback(SELINUX\_CB\_LOG, cb);

if (initialize\_globals() < 0) {//初始化全局变量，创建目录

ALOGE("Could not initialize globals; exiting.\n");

exit(1);

}

if (initialize\_directories() < 0) {//初始化系统目录

ALOGE("Could not create directories; exiting.\n");

exit(1);

}

if (selinux\_enabled && selinux\_status\_open(true) < 0) {

ALOGE("Could not open selinux status; exiting.\n");

exit(1);

}

drop\_privileges();//变更installd进程的权限

lsocket = android\_get\_control\_socket(SOCKET\_PATH);//从环境变量ANDROID\_SOCKET\_INSTALLD中获取用于监听的本地socket，SOCKET\_PATH== installd

if (lsocket < 0) {

ALOGE("Failed to get socket from environment: %s\n", strerror(errno));

exit(1);

}

if (listen(lsocket, 5)) {//监听socket

ALOGE("Listen on socket failed: %s\n", strerror(errno));

exit(1);

}

fcntl(lsocket, F\_SETFD, FD\_CLOEXEC);

for (;;) {

alen = sizeof(addr);

s = accept(lsocket, &addr, &alen);//接收连接

if (s < 0) {

ALOGE("Accept failed: %s\n", strerror(errno));

continue;

}

fcntl(s, F\_SETFD, FD\_CLOEXEC);

ALOGI("new connection\n");

for (;;) {

unsigned short count;

if (readx(s, &count, sizeof(count))) {//读取命令的长度

ALOGE("failed to read size\n");

break;

}

if ((count < 1) || (count >= BUFFER\_MAX)) {//如果命令长度错误则停止处理

ALOGE("invalid size %d\n", count);

break;

}

if (readx(s, buf, count)) {

ALOGE("failed to read command\n");

break;

}

buf[count] = 0;

if (selinux\_enabled && selinux\_status\_updated() > 0) {

selinux\_android\_seapp\_context\_reload();

}

if (execute(s, buf)) break;//执行命令

}

ALOGI("closing connection\n");

close(s);//关闭连接

}

return 0;

}

installd就是监听一个本地的socket，这个socket通过在init.rc文件中指定服务属性的方式创建。如果有socket连接进来，则通过socket读取命令字符串，然后执行命令。

在main函数中，installd通过调用initialize\_globals()和initialize\_directories()来完成初始化的工作。initialize\_globals()函数将设置安装应用需要用到的目录名；initialize\_directories()则创建所有用户的安装目录。

在上面的方法中都使用到了dir\_rec\_t结构体，那么这个结构体是如何定义的呢？如下：

typedef struct {

char\* path;

size\_t len;

} dir\_rec\_t;

在这个结构体中有两个成员变量：path、len，分别表示文件路径和文件路径长度。

同样使用到了android\_system\_dirs，定义如下：

dir\_rec\_array\_t android\_system\_dirs;

typedef struct {

size\_t count;

dir\_rec\_t\* dirs;

} dir\_rec\_array\_t;

在这个结构体中有两个成员变量：count、dirs；其中dirs又是dir\_rec\_t的结构体类型。

## 2、初始化全局变量initialize\_globals

int initialize\_globals() {

// Get the android data directory.从环境变量中读取数据存储路径，android\_data\_dir=/data/

***// 数据目录/data/***

if (get\_path\_from\_env(&android\_data\_dir, "ANDROID\_DATA") < 0) {

return -1;

}

// Get the android app directory.得到应用程序安装目录：android\_app\_dir=/data/app/

***// app目录/data/app/***

if (copy\_and\_append(&android\_app\_dir, &android\_data\_dir, APP\_SUBDIR) < 0) {

return -1;

}

// Get the android protected app directory.得到应用程序私有目录：android\_app\_private\_dir=/data/app-private/

***// 受保护的app目录/data/priv-app/***

if (copy\_and\_append(&android\_app\_private\_dir, &android\_data\_dir, PRIVATE\_APP\_SUBDIR) < 0) {

return -1;

}

// Get the android app native library directory.android\_app\_lib\_dir=/data/app-lib/

***// app本地库目录/data/app-lib/***

if (copy\_and\_append(&android\_app\_lib\_dir, &android\_data\_dir, APP\_LIB\_SUBDIR) < 0) {

return -1;

}

// Get the sd-card ASEC mount point.从环境变量中取得sdcard ASEC的挂载点，android\_asec\_dir=/mnt/asec

***// sdcard挂载点/mnt/asec***

if (get\_path\_from\_env(&android\_asec\_dir, "ASEC\_MOUNTPOINT") < 0) {

return -1;

}

// Get the android media directory.android\_media\_dir=/data/media/

***// 多媒体目录/data/media***

if (copy\_and\_append(&android\_media\_dir, &android\_data\_dir, MEDIA\_SUBDIR) < 0) {

return -1;

}

***// 系统和厂商目录***

// Take note of the system and vendor directories.定义android\_system\_dirs变量，并分配存储空间

android\_system\_dirs.count = 4;

android\_system\_dirs.dirs = calloc(android\_system\_dirs.count, sizeof(dir\_rec\_t));

if (android\_system\_dirs.dirs == NULL) {

ALOGE("Couldn't allocate array for dirs; aborting\n");

return -1;

}

dir\_rec\_t android\_root\_dir;

if (get\_path\_from\_env(&android\_root\_dir, "ANDROID\_ROOT") < 0) {//android\_root\_dir=/system/

ALOGE("Missing ANDROID\_ROOT; aborting\n");

return -1;

}

android\_system\_dirs.dirs[0].path = build\_string2(android\_root\_dir.path, APP\_SUBDIR);// /system/app

android\_system\_dirs.dirs[0].len = strlen(android\_system\_dirs.dirs[0].path);

android\_system\_dirs.dirs[1].path = build\_string2(android\_root\_dir.path, PRIV\_APP\_SUBDIR);// /system/priv-app

android\_system\_dirs.dirs[1].len = strlen(android\_system\_dirs.dirs[1].path);

android\_system\_dirs.dirs[2].path = "/vendor/app/";

android\_system\_dirs.dirs[2].len = strlen(android\_system\_dirs.dirs[2].path);

android\_system\_dirs.dirs[3].path = "/oem/app/";

android\_system\_dirs.dirs[3].len = strlen(android\_system\_dirs.dirs[3].path);

return 0;

}

上面函数中使用到了ANDROID\_DATA、ASEC\_MOUNTPOINT、ANDROID\_ROOT，这些环境变量是在哪里定义的呢？在Android的启动脚本Init.environ.rc.in中配置了环境变量，如下：system/core/rootdir/init.environ.rc.in

# set up the global environment

on init

export PATH /sbin:/vendor/bin:/system/sbin:/system/bin:/system/xbin

export ANDROID\_BOOTLOGO 1

export ANDROID\_ROOT /system

export ANDROID\_ASSETS /system/app

export ANDROID\_DATA /data

export ANDROID\_STORAGE /storage

export ASEC\_MOUNTPOINT /mnt/asec

export LOOP\_MOUNTPOINT /mnt/obb

export BOOTCLASSPATH %BOOTCLASSPATH%

export SYSTEMSERVERCLASSPATH %SYSTEMSERVERCLASSPATH%

从代码中可以看出，initialize\_globals()函数主要是初始化一些全局变量，这些全局变量初始化为一些安装的路径。

## 3、初始化目录initialize\_directories

int initialize\_directories() {

int res = -1;

// Read current filesystem layout version to handle upgrade paths

char version\_path[PATH\_MAX];

snprintf(version\_path, PATH\_MAX, "%s.layout\_version", android\_data\_dir.path);

int oldVersion;// //读取当前文件系统版本

if (fs\_read\_atomic\_int(version\_path, &oldVersion) == -1) {

oldVersion = 0;

}

int version = oldVersion;

// **/data/user**

char \*user\_data\_dir = build\_string2(android\_data\_dir.path, SECONDARY\_USER\_PREFIX);

**// /data/data**

char \*legacy\_data\_dir = build\_string2(android\_data\_dir.path, PRIMARY\_USER\_PREFIX);

**// /data/user/0**

char \*primary\_data\_dir = build\_string3(android\_data\_dir.path, SECONDARY\_USER\_PREFIX, "0");

if (!user\_data\_dir || !legacy\_data\_dir || !primary\_data\_dir) {

goto fail;

}

// Make the /data/user directory if necessary

if (access(user\_data\_dir, R\_OK) < 0) {

if (mkdir(user\_data\_dir, 0711) < 0) {//如果目录不存在，创建/data/user/目录

goto fail;

}

if (chown(user\_data\_dir, AID\_SYSTEM, AID\_SYSTEM) < 0) {//设置属性组

goto fail;

}

if (chmod(user\_data\_dir, 0711) < 0) {//设置读写权限

goto fail;

}

}

// Make the /data/user/0 symlink to /data/data if necessary

//将/data/user/0链接到/data/data

if (access(primary\_data\_dir, R\_OK) < 0) {

if (symlink(legacy\_data\_dir, primary\_data\_dir)) {

goto fail;

}

}

//处理data/media 相关

if (version == 0) {

// Introducing multi-user, so migrate /data/media contents into /data/media/0

ALOGD("Upgrading /data/media for multi-user");

// Ensure /data/media。重新创建/data/media目录

if (fs\_prepare\_dir(android\_media\_dir.path, 0770, AID\_MEDIA\_RW, AID\_MEDIA\_RW) == -1) {

goto fail;

}

// /data/media.tmp

char media\_tmp\_dir[PATH\_MAX];

snprintf(media\_tmp\_dir, PATH\_MAX, "%smedia.tmp", android\_data\_dir.path);

// Only copy when upgrade not already in progress

if (access(media\_tmp\_dir, F\_OK) == -1) {

if (rename(android\_media\_dir.path, media\_tmp\_dir) == -1) {

ALOGE("Failed to move legacy media path: %s", strerror(errno));

goto fail;

}

}

// Create /data/media again

if (fs\_prepare\_dir(android\_media\_dir.path, 0770, AID\_MEDIA\_RW, AID\_MEDIA\_RW) == -1) {

goto fail;

}

if (selinux\_android\_restorecon(android\_media\_dir.path, 0)) {

goto fail;

}

// /data/media/0

char owner\_media\_dir[PATH\_MAX];

snprintf(owner\_media\_dir, PATH\_MAX, "%s0", android\_media\_dir.path);

// Move any owner data into place

if (access(media\_tmp\_dir, F\_OK) == 0) {

if (rename(media\_tmp\_dir, owner\_media\_dir) == -1) {

ALOGE("Failed to move owner media path: %s", strerror(errno));

goto fail;

}

}

// Ensure media directories for any existing users

DIR \*dir;

struct dirent \*dirent;

char user\_media\_dir[PATH\_MAX];

dir = opendir(user\_data\_dir);

if (dir != NULL) {

while ((dirent = readdir(dir))) {

if (dirent->d\_type == DT\_DIR) {

const char \*name = dirent->d\_name;

// skip "." and ".."

if (name[0] == '.') {

if (name[1] == 0) continue;

if ((name[1] == '.') && (name[2] == 0)) continue;

}

// /data/media/<user\_id>

snprintf(user\_media\_dir, PATH\_MAX, "%s%s", android\_media\_dir.path, name);

if (fs\_prepare\_dir(user\_media\_dir, 0770, AID\_MEDIA\_RW, AID\_MEDIA\_RW) == -1) {

goto fail;

}

}

}

closedir(dir);

}

version = 1;

}

// /data/media/obb

char media\_obb\_dir[PATH\_MAX];

snprintf(media\_obb\_dir, PATH\_MAX, "%sobb", android\_media\_dir.path);

if (version == 1) {

// Introducing /data/media/obb for sharing OBB across users; migrate

// any existing OBB files from owner.

ALOGD("Upgrading to shared /data/media/obb");

// /data/media/0/Android/obb

char owner\_obb\_path[PATH\_MAX];

snprintf(owner\_obb\_path, PATH\_MAX, "%s0/Android/obb", android\_media\_dir.path);

// Only move if target doesn't already exist

if (access(media\_obb\_dir, F\_OK) != 0 && access(owner\_obb\_path, F\_OK) == 0) {

if (rename(owner\_obb\_path, media\_obb\_dir) == -1) {

ALOGE("Failed to move OBB from owner: %s", strerror(errno));

goto fail;

}

}

version = 2;

}

if (ensure\_media\_user\_dirs(0) == -1) {

ALOGE("Failed to setup media for user 0");

goto fail;

}

if (fs\_prepare\_dir(media\_obb\_dir, 0770, AID\_MEDIA\_RW, AID\_MEDIA\_RW) == -1) {

goto fail;

}

if (ensure\_config\_user\_dirs(0) == -1) {

ALOGE("Failed to setup misc for user 0");

goto fail;

}

if (version == 2) {

ALOGD("Upgrading to /data/misc/user directories");

char misc\_dir[PATH\_MAX];

snprintf(misc\_dir, PATH\_MAX, "%smisc", android\_data\_dir.path);

char keychain\_added\_dir[PATH\_MAX];

snprintf(keychain\_added\_dir, PATH\_MAX, "%s/keychain/cacerts-added", misc\_dir);

char keychain\_removed\_dir[PATH\_MAX];

snprintf(keychain\_removed\_dir, PATH\_MAX, "%s/keychain/cacerts-removed", misc\_dir);

DIR \*dir;

struct dirent \*dirent;

dir = opendir(user\_data\_dir);

if (dir != NULL) {

while ((dirent = readdir(dir))) {

const char \*name = dirent->d\_name;

// skip "." and ".."

if (name[0] == '.') {

if (name[1] == 0) continue;

if ((name[1] == '.') && (name[2] == 0)) continue;

}

uint32\_t user\_id = atoi(name);

// /data/misc/user/<user\_id>

if (ensure\_config\_user\_dirs(user\_id) == -1) {

goto fail;

}

char misc\_added\_dir[PATH\_MAX];

snprintf(misc\_added\_dir, PATH\_MAX, "%s/user/%s/cacerts-added", misc\_dir, name);

char misc\_removed\_dir[PATH\_MAX];

snprintf(misc\_removed\_dir, PATH\_MAX, "%s/user/%s/cacerts-removed", misc\_dir, name);

uid\_t uid = multiuser\_get\_uid(user\_id, AID\_SYSTEM);

gid\_t gid = uid;

if (access(keychain\_added\_dir, F\_OK) == 0) {

if (copy\_dir\_files(keychain\_added\_dir, misc\_added\_dir, uid, gid) != 0) {

ALOGE("Some files failed to copy");

}

}

if (access(keychain\_removed\_dir, F\_OK) == 0) {

if (copy\_dir\_files(keychain\_removed\_dir, misc\_removed\_dir, uid, gid) != 0) {

ALOGE("Some files failed to copy");

}

}

}

closedir(dir);

if (access(keychain\_added\_dir, F\_OK) == 0) {

delete\_dir\_contents(keychain\_added\_dir, 1, 0);

}

if (access(keychain\_removed\_dir, F\_OK) == 0) {

delete\_dir\_contents(keychain\_removed\_dir, 1, 0);

}

}

version = 3;

}

// Persist layout version if changed

if (version != oldVersion) {

if (fs\_write\_atomic\_int(version\_path, version) == -1) {

ALOGE("Failed to save version to %s: %s", version\_path, strerror(errno));

goto fail;

}

}

// Success!

res = 0;

fail:

free(user\_data\_dir);

free(legacy\_data\_dir);

free(primary\_data\_dir);

return res;

}

## 4、变更installd进程的权限

我们看下drop\_privileges()，代码如下：

static void drop\_privileges() {

if (prctl(PR\_SET\_KEEPCAPS, 1) < 0) {//保留进程的权限

ALOGE("prctl(PR\_SET\_KEEPCAPS) failed: %s\n", strerror(errno));

exit(1);

}

if (setgid(AID\_INSTALL) < 0) {//设置进程的gid为“install”

ALOGE("setgid() can't drop privileges; exiting.\n");

exit(1);

}

if (setuid(AID\_INSTALL) < 0) {//设置进程的uid为"install"

ALOGE("setuid() can't drop privileges; exiting.\n");

exit(1);

}

struct \_\_user\_cap\_header\_struct capheader;

struct \_\_user\_cap\_data\_struct capdata[2];

memset(&capheader, 0, sizeof(capheader));

memset(&capdata, 0, sizeof(capdata));

capheader.version = \_LINUX\_CAPABILITY\_VERSION\_3;

capheader.pid = 0;

capdata[CAP\_TO\_INDEX(CAP\_DAC\_OVERRIDE)].permitted |= CAP\_TO\_MASK(CAP\_DAC\_OVERRIDE);

capdata[CAP\_TO\_INDEX(CAP\_CHOWN)].permitted |= CAP\_TO\_MASK(CAP\_CHOWN);

capdata[CAP\_TO\_INDEX(CAP\_SETUID)].permitted |= CAP\_TO\_MASK(CAP\_SETUID);

capdata[CAP\_TO\_INDEX(CAP\_SETGID)].permitted |= CAP\_TO\_MASK(CAP\_SETGID);

capdata[CAP\_TO\_INDEX(CAP\_FOWNER)].permitted |= CAP\_TO\_MASK(CAP\_FOWNER);

capdata[0].effective = capdata[0].permitted;

capdata[1].effective = capdata[1].permitted;

capdata[0].inheritable = 0;

capdata[1].inheritable = 0;

if (capset(&capheader, &capdata[0]) < 0) {//设置进程的权限

ALOGE("capset failed: %s\n", strerror(errno));

exit(1);

}

}

drop\_privileges()使用系统调用prctl来保留进程的能力。可参考Linux内核，虽然installd的uid和gid都变成了AID\_INSTALL，但是保留了5项能力，使得installd无需root用户身份也能够完成安装过程。

上面函数中用到了\_\_user\_cap\_data\_struct结构体，我们看下这个结构体是如何定义的，在android\_filesystem\_capability.h文件中定义如下：

typedef struct \_\_user\_cap\_data\_struct {

\_\_u32 effective;

\_\_u32 permitted;

\_\_u32 inheritable;

} \_\_user \*cap\_user\_data\_t;

typedef struct \_\_user\_cap\_header\_struct {

\_\_u32 version;

int pid;

} \_\_user \*cap\_user\_header\_t;

## 5、execute()

main()中通过调用execute()函数来执行命令。

static int execute(int s, char cmd[BUFFER\_MAX]) {

char reply[REPLY\_MAX];

char \*arg[TOKEN\_MAX+1];

unsigned i;

unsigned n = 0;

unsigned short count;

int ret = -1;

reply[0] = 0;

arg[0] = cmd;

while (\*cmd) {

if (isspace(\*cmd)) {

\*cmd++ = 0;

n++;

arg[n] = cmd;

if (n == TOKEN\_MAX) {

goto done;

}

}

if (\*cmd) {

cmd++; //计算参数个数

}

}

for (i = 0; i < sizeof(cmds) / sizeof(cmds[0]); i++) {

if (!strcmp(cmds[i].name,arg[0])) {

if (n != cmds[i].numargs) {

**//参数个数不匹配，直接返回**

ALOGE("%s requires %d arguments (%d given)\n",

cmds[i].name, cmds[i].numargs, n);

} else {

//执行相应的命令，func为函数指针

ret = cmds[i].func(arg + 1, reply);

}

goto done;

}

}

done:

if (reply[0]) {

n = snprintf(cmd, BUFFER\_MAX, "%d %s", ret, reply);

} else {

n = snprintf(cmd, BUFFER\_MAX, "%d", ret);

}

if (n > BUFFER\_MAX) n = BUFFER\_MAX;

count = n;

//将命令执行后的返回值写入socket套接字

if (writex(s, &count, sizeof(count))) return -1;

if (writex(s, cmd, count)) return -1;

return 0;

};

execute()函数执行过程就是查找下面并执行下面的cmds标准命令字符串对应的函数：

struct cmdinfo cmds[] = {

{ "ping", 0, do\_ping },

{ "install", 4, do\_install },

{ "dexopt", 6, do\_dexopt },

{ "markbootcomplete", 1, do\_mark\_boot\_complete },

{ "movedex", 3, do\_move\_dex },

{ "rmdex", 2, do\_rm\_dex },

{ "remove", 2, do\_remove },

{ "rename", 2, do\_rename },

{ "fixuid", 3, do\_fixuid },

{ "freecache", 1, do\_free\_cache },

{ "rmcache", 2, do\_rm\_cache },

{ "rmcodecache", 2, do\_rm\_code\_cache },

{ "getsize", 7, do\_get\_size },

{ "rmuserdata", 2, do\_rm\_user\_data },

{ "movefiles", 0, do\_movefiles },

{ "linklib", 3, do\_linklib },

{ "mkuserdata", 4, do\_mk\_user\_data },

{ "mkuserconfig", 1, do\_mk\_user\_config },

{ "rmuser", 1, do\_rm\_user },

{ "idmap", 3, do\_idmap },

{ "restorecondata", 3, do\_restorecon\_data },

{ "patchoat", 5, do\_patchoat },

};

dexopt：将应用优化为dex格式。

rmdex：删除apk文件。

remove：卸载应用。

rename：更改应用数据目录的名称。

fixuid：更改应用数据目录的uid。

freecache：清除/cache目录下的文件。

rmcache：删除/cache目录下的某个应用的目录。

getsize：计算一个应用占用的空间大小，包括apk大小、数据目录、cache目录等。

rmuserdata：删除一个user的所有安装的应用。

movefiles：执行/system/etc/updatecmds目录下的移动目录的脚本。

linklib：为动态库建立符合连接。

mkuserdata：为一个user创建目录。

mkuserconfig：为一个user创建配置文件。

rmuser：删除一个user的所有文件。

idmap：对两个apk进程执行idmap操作。

restorecondata：恢复目录的SELinux安全上下文。

patchoat：将应用优化为oat格式。

此命令表总共有25的命令，该表中第二列是指命令所需的参数个数，第三列是指命令所指向的函数。 不同的Android版本该表格都会有所不同

## 6、分析install命令

install命令执行的函数是install()，如下：

//commands.c

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

{

char pkgdir[PKG\_PATH\_MAX];// 256

char libsymlink[PKG\_PATH\_MAX];

char applibdir[PKG\_PATH\_MAX];

struct stat libStat;

Xandy:app\_install\_cnt

if ((uid < AID\_SYSTEM) || (gid < AID\_SYSTEM)) {//检查uid、gid

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

return -1;

}

//得到应用的数据目录名/data/data/<包名>

if (create\_pkg\_path(pkgdir, pkgname, PKG\_DIR\_POSTFIX, 0)) {

ALOGE("cannot create package path\n");

return -1;

}

//得到应用的动态库目录名/data/data/<包名>/lib

if (create\_pkg\_path(libsymlink, pkgname, PKG\_LIB\_POSTFIX, 0)) {

ALOGE("cannot create package **lib symlink origin** path\n");

return -1;

}

//得到/app-lib目录下的符号链接的名称/data/app-lib/<包名>

if (create\_pkg\_path\_in\_dir(applibdir, &android\_app\_lib\_dir, pkgname, PKG\_DIR\_POSTFIX)) {

ALOGE("cannot create package **lib symlink dest** path\n");

return -1;

}

//创建应用的数据目录

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

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

return -1;

}

if (chmod(pkgdir, 0751) < 0) {//修改目录权限

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

unlink(pkgdir);

return -1;

}

//检查符号链接是否已经存在

if (lstat(libsymlink, &libStat) < 0) {

if (errno != ENOENT) {

ALOGE("couldn't stat lib dir: %s\n", strerror(errno));

return -1;

}

} else {

if (S\_ISDIR(libStat.st\_mode)) {

if (delete\_dir\_contents(libsymlink, 1, NULL) < 0) {

ALOGE("couldn't delete lib directory during install for: %s", libsymlink);

return -1;

}

} else if (S\_ISLNK(libStat.st\_mode)) {

if (unlink(libsymlink) < 0) {

ALOGE("couldn't unlink lib directory during install for: %s", libsymlink);

return -1;

}

}

}

//为目录设置SELinux的安全上下文

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

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

unlink(libsymlink);

unlink(pkgdir);

return -errno;

}

//创建符号链接

if (symlink(applibdir, libsymlink) < 0) {

ALOGE("couldn't symlink directory '%s' -> '%s': %s\n", libsymlink, applibdir,

strerror(errno));

unlink(pkgdir);

return -1;

}

//修改目录的gid和uid

if (chown(pkgdir, uid, gid) < 0) {

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

unlink(libsymlink);

unlink(pkgdir);

return -1;

}

return 0;

}

上面多次用到create\_pkg\_path()函数，我们分析一下：

/\*\*

\* Create the package path name for a given package name with a postfix for

\* a certain userid. Returns 0 on success, and -1 on failure.

\*/

int create\_pkg\_path(char path[PKG\_PATH\_MAX],//PKG\_PATH\_MAX=256

const char \*pkgname,

const char \*postfix,//后缀

userid\_t userid)

{

size\_t userid\_len;

char\* userid\_prefix;

if (userid == 0) {

userid\_prefix = PRIMARY\_USER\_PREFIX;// 前缀 data/

userid\_len = 0;

} else {

userid\_prefix = SECONDARY\_USER\_PREFIX;// 前缀 user/

userid\_len = snprintf(NULL, 0, "%d", userid);// userid

}

const size\_t prefix\_len = android\_data\_dir.len + strlen(userid\_prefix)

+ userid\_len + 1 /\*slash\*/;// /data/data/ 计算前缀的长度

char prefix[prefix\_len + 1];

char \*dst = prefix;

size\_t dst\_size = sizeof(prefix);

if (append\_and\_increment(&dst, android\_data\_dir.path, &dst\_size) < 0 // /data/

|| append\_and\_increment(&dst, userid\_prefix, &dst\_size) < 0) {

ALOGE("Error building prefix for APK path");

return -1;

}

if (userid != 0) {

int ret = snprintf(dst, dst\_size, "%d/", userid);

if (ret < 0 || (size\_t) ret != userid\_len + 1) {

ALOGW("Error appending UID to APK path");

return -1;

}

}

dir\_rec\_t dir;

dir.path = prefix;

dir.len = prefix\_len;

return create\_pkg\_path\_in\_dir(path, &dir, pkgname, postfix);

}

上述函数中用到了append\_and\_increment函数，这个函数的作用就是调用strlcpy函数，将src源字符串复制到dst目标字符串。

int append\_and\_increment(char\*\* dst, const char\* src, size\_t\* dst\_size) {

ssize\_t ret = strlcpy(\*dst, src, \*dst\_size);//将src复制到dst中

if (ret < 0 || (size\_t) ret >= \*dst\_size) {

return -1;

}

\*dst += ret;

\*dst\_size -= ret;

return 0;

}

create\_pkg\_path函数最后调用了create\_pkg\_path\_in\_dir函数，如下：

int create\_pkg\_path\_in\_dir(char path[PKG\_PATH\_MAX],

const dir\_rec\_t\* dir,

const char\* pkgname,

const char\* postfix)

{

const size\_t postfix\_len = strlen(postfix);

const size\_t pkgname\_len = strlen(pkgname);

if (pkgname\_len > PKG\_NAME\_MAX) {

return -1;

}

if (is\_valid\_package\_name(pkgname) < 0) {

return -1;

}

if ((pkgname\_len + dir->len + postfix\_len) >= PKG\_PATH\_MAX) {

return -1;

}

char \*dst = path;

size\_t dst\_size = PKG\_PATH\_MAX;

if (append\_and\_increment(&dst, dir->path, &dst\_size) < 0

|| append\_and\_increment(&dst, pkgname, &dst\_size) < 0

|| append\_and\_increment(&dst, postfix, &dst\_size) < 0) {

ALOGE("Error building APK path");

return -1;

}

return 0;

}

install()命令创建了应用的数据目录，并把目录的uid和gid改成参数传递的值。同时在/data/app-lib目录下创建了一个符号链接，指向应用的本地动态库的安装路径。

751



<http://lib.csdn.net/article/android/63896>

# linklib

## 核心底层逻辑

C:/k/android-7.1.1\_r1/frameworks/native/cmds/installd/installd.cpp

{ **"linklib"**, 4, do\_linklib },

C:/k/android-7.1.1\_r1/frameworks/native/cmds/installd/commands. Linklib

## 输入

**final** String volumeUuid = pkg.volumeUuid;  
**final** String packageName = pkg.packageName;  
**final** ApplicationInfo app = pkg.applicationInfo;

**final** String **nativeLibPath** = app.nativeLibraryDir;

if(**!VMRuntime.is64BitAbi(app.primaryCpuAbi)**){//32位才有符号链接的问题？64位为啥不会有？？  
mInstaller.linkNativeLibraryDirectory(volumeUuid, packageName,  
 **nativeLibPath**, userId)

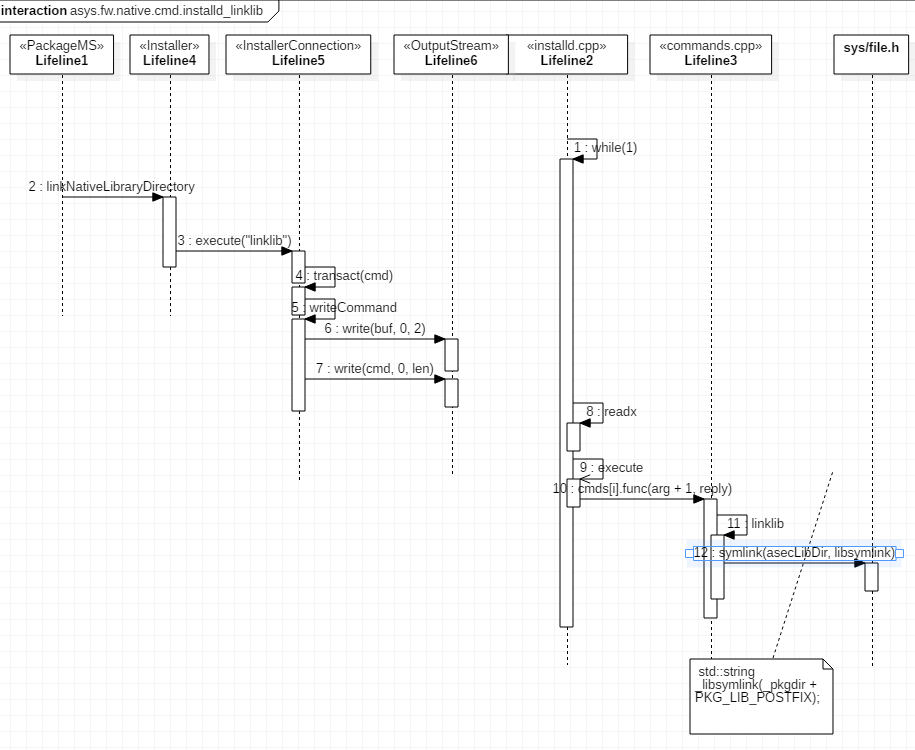
}

**核心参数是nativeLibPath，这个是在解析apk的时候添加的**

### nativeLibPath来自哪里？

## 上层调用时序

传递参数



## 创建用户lib目录

**PKG\_LIB\_POSTFIX = "/lib";**

std::string \_pkgdir(create\_data\_user\_ce\_package\_path(uuid, userId, pkgname));//  
std::string \_libsymlink(\_pkgdir + PKG\_LIB\_POSTFIX); **/data/user/0/com.kg.app/lib**

注：创建包目录**/data/user/0/com.kg.app**

frameworks/native/cmds/installd/utils.cpp

std::string create\_data\_user\_ce\_package\_path(**const char**\* volume\_uuid,  
 userid\_t user, **const char**\* package\_name) {  
 check\_package\_name(package\_name);  
 **return** StringPrintf(**"%s/%s"**,  
 create\_data\_user\_ce\_path(volume\_uuid, user).c\_str(), package\_name);  
}

## 链接lib

symlink(asecLibDir, libsymlink)

**const int** ret = link(from\_path, to\_path);

# link\_file

**int** link\_file(**const char**\* relative\_path, **const char**\* from\_base, **const char**\* to\_base)

# linkNativeLibraryDirectory情景分析

## PMS. prepareAppDataContentsLeafLIF

**private void** prepareAppDataContentsLeafLIF(PackageParser.Package pkg, **int** userId, **int** flags) {  
 **final** String volumeUuid = pkg.**volumeUuid**;  
 **final** String packageName = pkg.**packageName**;  
 **final** ApplicationInfo app = pkg.**applicationInfo**;

**final** String nativeLibPath = app.**nativeLibraryDir**;

**mInstaller**.linkNativeLibraryDirectory(volumeUuid, packageName,  
 nativeLibPath, userId);

**mInstalld**.linkNativeLibraryDirectory

frameworks\base\services\core\java\com\android\server\pm\Installer.java

**mInstalld**.linkNativeLibraryDirectory(uuid, packageName, nativeLibPath32, userId);

**void** linkNativeLibraryDirectory(@nullable @utf8InCpp String uuid,  
 @utf8InCpp String packageName, @utf8InCpp String nativeLibPath32, **int** userId);

frameworks\base\services\core\java\com\android\server\pm\PackageInstallerSession.java

linkFiles(fromFiles, toDir, **mInheritedFilesBase**);

**mPm**.**mInstaller**.linkFile(relativePath, fromDir.getAbsolutePath(),  
 toDir.getAbsolutePath());