> system/core/libutils/Looper.cpp

# TODO

C++源码调用图生成器实现: <https://segmentfault.com/a/1190000008314896>

Backtrace源码分析与使用

Native进程之Trace原理

<http://gityuan.com/2016/11/27/native-traces/>

可通过一条命令来获取指定Native进程的traces信息，例如输出pid=17529进程信息：

adb shell debuggerd -b 17529 //可指定进程pid

# Libutils

## 打印堆栈CallStack

### 为什么要打印函数调用堆栈

打印调用堆栈可以直接把问题发生时的函数调用关系打出来，非常有利于理解函数调用关系。比如函数A可能被B/C/D调用，如果只看代码，B/C/D谁调用A都有可能，如果打印出调用堆栈，直接就把谁调的打出来了。

不仅如此，打印函数调用堆栈还有另一个好处。在Android代码里，函数命名很多雷同的，虚函数调用，几个类里的函数名相同等，即使用source insight工具看也未必容易看清函数调用关系。如果用了堆栈打印，很容易看到函数调用逻辑。

那么一个问题来了，Android/kernel本身在发生问题(kernel panic, tombstone, …)时，都可以打出详细的堆栈信息，这里干嘛还要费劲研究打堆栈？答案是发生问题时的堆栈的确很详细，但这里研究的是不影响(准确说是基本不影响)系统运行的境况下，打印出某个情形下的堆栈信息，这个对源代码逻辑研究很有帮助。

### 如何调用

#### Linux Kernel

Kernel里最简单，直接有几现成的函数可以使用：

dump\_stack() 这个函数打出当前堆栈和函数调用backtrace后接着运行

WARN\_ON(x) 这个函数跟dump\_stack很像，它有个条件，如果条件满足了就把stack打出来。

打印出来的结果都在kernel log里，一般dmesg命令就可以看到了

#### Native C++

Android在新版(至少5.0, 6.0)里加入了CallStack类，这个类可以打出当前的backtrace。用法很简单：

* 前面确保包含头文件#include <utils/CallStack.h>
* Android.mk的库依赖列表(LOCAL\_SHARED\_LIBRARIES)里包含libutils，一般都已经包含了。
* 然后在要打印堆栈处加入android::CallStack cs("haha");

"haha"是在logcat输出的TAG，这里可以自己定义。如果上下文已经在android namespace里，”android::”前缀就不必加了。

Native C++的输出log可以在logcat里看到。

注意，错误用法，在新版Android里编译不过，在网上的一些文档里说要这么用：

CallStack stack;

stack.update();

stack.dump();

#### Native C

Android对C的堆栈打印支持不太好。一个简单方法是用C语言调C++的函数，对，就是extern “C”。

1.先在项目里加入一个c++文件，比如mycallstack.cpp，里面是：

#include <utils/CallStack.h>

extern "C" void dumping\_callstack(void);

void dumping\_callstack(void)

{

android::CallStack cs("haha");

}

2.在项目里再加入一个c++的头文件，比如mycallstack.h，里面是：

void dumping\_callstack(void);

3.在Android.mk里源文件列表LOCAL\_SRC\_FILES里加入callstack.cpp，确保libutils在依赖列表里。

4.在native C里include callstack.h后直接调用dumping\_callstack()就可以了。

这个log也可以在logcat里看到。

过时方法，新版Android上libcorkscrew已经被拿掉了，网上的加载libcorkscrew库的方法自然就不能用了。过去网上的文章一般是推荐libcorkscrew.so，并加入大段代码来unwind\_backtrace已经不可行了。。

#### Java

Java最简单，它的backtrace最详细，连文件名和行号都打出来了:

Exception e = new Exception("haha");

e.printStackTrace();

log在logcat里看以看到。

???

### 如何查看

Log级别

### 源码分析

#### CallStack

CallStack(const char\* logtag, int32\_t ignoreDepth=1);

this->update(ignoreDepth+1);

this->log(logtag);

#### update

void CallStack::update(int32\_t ignoreDepth, pid\_t tid) {

mFrameLines.clear();

std::unique\_ptr<Backtrace> backtrace(Backtrace::Create(BACKTRACE\_CURRENT\_PROCESS, tid));

if (!backtrace->Unwind(ignoreDepth)) {

ALOGW("%s: Failed to unwind callstack.", \_\_FUNCTION\_\_);

}

for (size\_t i = 0; i < backtrace->NumFrames(); i++) {

mFrameLines.push\_back(String8(backtrace->FormatFrameData(i).c\_str()));

}

#### log

void CallStack::log(const char\* logtag, android\_LogPriority priority, const char\* prefix) const {

LogPrinter printer(logtag, priority, prefix, /\*ignoreBlankLines\*/false);

print(printer);

}

在Android framework的C++层或其他C++服务中需要打印调用栈时，可以使用android中的CallStack类进行打印，该类的实现如下：

#### print

void CallStack::print(Printer& printer) const {

for (size\_t i = 0; i < mFrameLines.size(); i++) {

printer.printLine(mFrameLines[i]);

}

}

//TODO

[Android 平台 Native 代码的崩溃捕获机制及实现](https://my.oschina.net/bugly/blog/1354954)

[JNI中如何打印Call Stack](https://blog.csdn.net/xxooyc/article/details/51001345)

[Android下打印调试堆栈方法](https://blog.csdn.net/freshui/article/details/9456889)

### REF

[Android C++层打印调用](https://blog.csdn.net/smilefyx/article/details/54896758)栈

[Android下面打印进程函数调用堆栈(dump backtrace)的方法](https://www.cnblogs.com/CoderTian/p/6149332.html)

## 消息循环机制

# Fastboot

Gfd

在Android的SDK中自带了fastboot，路径为

${SDK\_HOME}/platform-tools/fastboot

查看help:

usage: fastboot [ <option> ] <command>

commands:

update <filename> Reflash device from update.zip.

Sets the flashed slot as active.

flashall Flash boot, system, vendor, and --

if found -- recovery. If the device

supports slots, the slot that has

been flashed to is set as active.

Secondary images may be flashed to

an inactive slot.

flash <partition> [ <filename> ] Write a file to a flash partition.

flashing lock Locks the device. Prevents flashing.

flashing unlock Unlocks the device. Allows flashing

any partition except

bootloader-related partitions.

flashing lock\_critical Prevents flashing bootloader-related

partitions.

flashing unlock\_critical Enables flashing bootloader-related

partitions.

flashing get\_unlock\_ability Queries bootloader to see if the

device is unlocked.

flashing get\_unlock\_bootloader\_nonce Queries the bootloader to get the

unlock nonce.

flashing unlock\_bootloader <request> Issue unlock bootloader using request.

flashing lock\_bootloader Locks the bootloader to prevent

bootloader version rollback.

erase <partition> Erase a flash partition.

format[:[<fs type>][:[<size>]] <partition>

Format a flash partition. Can

override the fs type and/or size

the bootloader reports.

getvar <variable> Display a bootloader variable.

set\_active <slot> Sets the active slot. If slots are

not supported, this does nothing.

boot <kernel> [ <ramdisk> [ <second> ] ] Download and boot kernel.

flash:raw <bootable-partition> <kernel> [ <ramdisk> [ <second> ] ]

Create bootimage and flash it.

devices [-l] List all connected devices [with

device paths].

continue Continue with autoboot.

reboot [bootloader|emergency] Reboot device [into bootloader or emergency mode].

reboot-bootloader Reboot device into bootloader.

oem <parameter1> ... <parameterN> Executes oem specific command.

stage <infile> Sends contents of <infile> to stage for

the next command. Supported only on

Android Things devices.

get\_staged <outfile> Receives data to <outfile> staged by the

last command. Supported only on Android

Things devices.

help Show this help message.

options:

-w Erase userdata and cache (and format

if supported by partition type).

-u Do not erase partition before

formatting.

-s <specific device> Specify a device. For USB, provide either

a serial number or path to device port.

For ethernet, provide an address in the

form <protocol>:<hostname>[:port] where

<protocol> is either tcp or udp.

-c <cmdline> Override kernel commandline.

-i <vendor id> Specify a custom USB vendor id.

-b, --base <base\_addr> Specify a custom kernel base

address (default: 0x10000000).

--kernel-offset Specify a custom kernel offset.

(default: 0x00008000)

--ramdisk-offset Specify a custom ramdisk offset.

(default: 0x01000000)

--tags-offset Specify a custom tags offset.

(default: 0x00000100)

-n, --page-size <page size> Specify the nand page size

(default: 2048).

-S <size>[K|M|G] Automatically sparse files greater

than 'size'. 0 to disable.

--slot <slot> Specify slot name to be used if the

device supports slots. All operations

on partitions that support slots will

be done on the slot specified.

'all' can be given to refer to all slots.

'other' can be given to refer to a

non-current slot. If this flag is not

used, slotted partitions will default

to the current active slot.

-a, --set-active[=<slot>] Sets the active slot. If no slot is

provided, this will default to the value

given by --slot. If slots are not

supported, this does nothing. This will

run after all non-reboot commands.

--skip-secondary Will not flash secondary slots when

performing a flashall or update. This

will preserve data on other slots.

--skip-reboot Will not reboot the device when

performing commands that normally

trigger a reboot.

--disable-verity Set the disable-verity flag in the

the vbmeta image being flashed.

--disable-verification Set the disable-verification flag in the vbmeta image being flashed.

--wipe-and-use-fbe On devices which support it,

erase userdata and cache, and

enable file-based encryption

--unbuffered Do not buffer input or output.

--version Display version.

-h, --help show this message.

够长的，好多参数解释看得也不是很懂。

自己编译Android源码也会产生fastboot，路径为：

## fastboot --version

C:\Users\10288>fastboot --version

fastboot version 28.0.1-4986621

Installed as C:\Users\10288\AppData\Local\Android\Sdk\platform-tools\fastboot.exe

${OUT}/host/darwin-x86/bin/fastboot //Mac的编译结果在darwin-x86下

查看help:

usage: fastboot [ <option> ] <command>

commands:

update <filename> reflash device from update.zip

flashall flash boot + recovery + system

flash <partition> [ <filename> ] write a file to a flash partition

erase <partition> erase a flash partition

format <partition> format a flash partition

getvar <variable> display a bootloader variable

boot <kernel> [ <ramdisk> ] download and boot kernel

flash:raw boot <kernel> [ <ramdisk> ] create bootimage and flash it

devices list all connected devices

continue continue with autoboot

reboot reboot device normally

reboot-bootloader reboot device into bootloader

help show this help message

options:

-w erase userdata and cache (and format

if supported by partition type)

-u do not first erase partition before

formatting

-s <specific device> specify device serial number

or path to device port

-l with "devices", lists device paths

-p <product> specify product name

-c <cmdline> override kernel commandline

-i <vendor id> specify a custom USB vendor id

-b <base\_addr> specify a custom kernel base address. default: 0x10000000

-n <page size> specify the nand page size. default: 2048

-S <size>[K|M|G] automatically sparse files greater than

size. 0 to disable

两者还不太一样，好像自己编译的在功能上是SDK自带的子集。在源码中有fastboot相关的代码，正好研究一下。

## fastboot devices

我们使用fastboot的第一个有效命令(不算 fastboot -h)通常是fastboot devices，我们来跟踪一下：

if (argc > 0 && !strcmp(\*argv, "devices")) {

skip(1);

list\_devices();

return 0;

}

list\_devices:

void list\_devices(void) {

// We don't actually open a USB device here,

// just getting our callback called so we can

// list all the connected devices.

usb\_open(list\_devices\_callback);

}

usb\_open，不同操作系统有不同的实现，为了方便理解，我们分析Linux版本的实现(system/core/fastboot/usb\_linux.c)：

usb\_handle \*usb\_open(ifc\_match\_func callback)

{

return find\_usb\_device("/dev/bus/usb", callback);

}

find\_usb\_device:

static usb\_handle \*find\_usb\_device(const char \*base, ifc\_match\_func callback)

{

usb\_handle \*usb = 0;

char busname[64], devname[64];

char desc[1024];

int n, in, out, ifc;

DIR \*busdir, \*devdir;

struct dirent \*de;

int fd;

int writable;

busdir = opendir(base);

if(busdir == 0) return 0;

while((de = readdir(busdir)) && (usb == 0)) {

if(badname(de->d\_name)) continue;

sprintf(busname, "%s/%s", base, de->d\_name);

devdir = opendir(busname);

if(devdir == 0) continue;

// DBG("[ scanning %s ]\n", busname);

while((de = readdir(devdir)) && (usb == 0)) {

if(badname(de->d\_name)) continue;

sprintf(devname, "%s/%s", busname, de->d\_name);

// DBG("[ scanning %s ]\n", devname);

writable = 1;

if((fd = open(devname, O\_RDWR)) < 0) {

// Check if we have read-only access, so we can give a helpful

// diagnostic like "adb devices" does.

writable = 0;

if((fd = open(devname, O\_RDONLY)) < 0) {

continue;

}

}

n = read(fd, desc, sizeof(desc));

if(filter\_usb\_device(fd, desc, n, writable, callback,

&in, &out, &ifc) == 0) {

usb = calloc(1, sizeof(usb\_handle));

strcpy(usb->fname, devname);

usb->ep\_in = in;

usb->ep\_out = out;

usb->desc = fd;

n = ioctl(fd, USBDEVFS\_CLAIMINTERFACE, &ifc);

if(n != 0) {

close(fd);

free(usb);

usb = 0;

continue;

}

} else {

close(fd);

}

}

closedir(devdir);

}

closedir(busdir);

return usb;

}

循环读取/dev/bus/usb目录下及子目录中的信息，解析并使用filter\_usb\_device过滤，然后对fastboot模式的usb进行callback调用，至于如何过滤fastboot模式的usb，这里面涉及到usb相关的知识，我也不是很了解，应该是通过usb信息中的某个标识来识别的，屌大的同学可以给我讲讲。

对callback的调用：

if(callback(&info) == 0) {

\*ept\_in\_id = in;

\*ept\_out\_id = out;

\*ifc\_id = ifc->bInterfaceNumber;

return 0;

}

回到list\_devices\_callback:

int list\_devices\_callback(usb\_ifc\_info \*info)

{

if (match\_fastboot\_with\_serial(info, NULL) == 0) {

char\* serial = info->serial\_number;

if (!info->writable) {

serial = "no permissions"; // like "adb devices"

}

if (!serial[0]) {

serial = "????????????";

}

// output compatible with "adb devices"

if (!long\_listing) {

printf("%s\tfastboot\n", serial);

} else if (!info->device\_path) {

printf("%-22s fastboot\n", serial);

} else {

printf("%-22s fastboot %s\n", serial, info->device\_path);

}

}

return -1;

}

其实就是输出连接的设备信息，假如是long\_listing，会把device\_path也输出来，long\_listing通过 -l指定：

$ fastboot devices -l

01d977445292ca8c fastboot usb:337838080X

## fastboot -w flashall

在刷机的时候，通常使用fastboot -w flashall，先看看-w:

case 'w':

wants\_wipe = 1;

break;

...

if (wants\_wipe) {

fb\_queue\_erase("userdata");

fb\_queue\_format("userdata", 1);

fb\_queue\_erase("cache");

fb\_queue\_format("cache", 1);

}

带上这个选项，会清除userdata和cache中的内容。  
再看看flashall:

else if(!strcmp(\*argv, "flashall")) {

skip(1);

do\_flashall(usb, erase\_first);

wants\_reboot = 1;

}

do\_flashall:

void do\_flashall(usb\_handle \*usb, int erase\_first)

{

char \*fname;

void \*data;

unsigned sz;

struct fastboot\_buffer buf;

int i;

queue\_info\_dump();

fb\_queue\_query\_save("product", cur\_product, sizeof(cur\_product));

fname = find\_item("info", product);

if (fname == 0) die("cannot find android-info.txt");

data = load\_file(fname, &sz);

if (data == 0) die("could not load android-info.txt: %s", strerror(errno));

setup\_requirements(data, sz);

for (i = 0; i < ARRAY\_SIZE(images); i++) {

fname = find\_item(images[i].part\_name, product);

if (load\_buf(usb, fname, &buf)) {

if (images[i].is\_optional)

continue;

die("could not load %s\n", images[i].img\_name);

}

do\_send\_signature(fname);

if (erase\_first && needs\_erase(images[i].part\_name)) {

fb\_queue\_erase(images[i].part\_name);

}

flash\_buf(images[i].part\_name, &buf);

}

}

find\_item:

char \*find\_item(const char \*item, const char \*product)

{

char \*dir;

char \*fn;

char path[PATH\_MAX + 128];

if(!strcmp(item,"boot")) {

fn = "boot.img";

} else if(!strcmp(item,"recovery")) {

fn = "recovery.img";

} else if(!strcmp(item,"system")) {

fn = "system.img";

} else if(!strcmp(item,"userdata")) {

fn = "userdata.img";

} else if(!strcmp(item,"cache")) {

fn = "cache.img";

} else if(!strcmp(item,"info")) {

fn = "android-info.txt";

} else {

fprintf(stderr,"unknown partition '%s'\n", item);

return 0;

}

if(product) {

get\_my\_path(path);

sprintf(path + strlen(path),

"../../../target/product/%s/%s", product, fn);

return strdup(path);

}

dir = getenv("ANDROID\_PRODUCT\_OUT");

if((dir == 0) || (dir[0] == 0)) {

die("neither -p product specified nor ANDROID\_PRODUCT\_OUT set");

return 0;

}

sprintf(path, "%s/%s", dir, fn);

return strdup(path);

}

会根据环境变量ANDROID\_PRODUCT\_OUT指定的目录下去找相关的文件，这里是android-info.txt。我们在刷机的时候，如果报找不到img文件时，需要设置ANDROID\_PRODUCT\_OUT就是这个原因。

接下来就是找相关的img文件，然后刷：

static struct {

char img\_name[13];

char sig\_name[13];

char part\_name[9];

bool is\_optional;

} images[3] = {

{"boot.img", "boot.sig", "boot", false},

{"recovery.img", "recovery.sig", "recovery", true},

{"system.img", "system.sig", "system", false},

};

...

for (i = 0; i < ARRAY\_SIZE(images); i++) {

fd = unzip\_to\_file(zip, images[i].img\_name);

if (fd < 0) {

if (images[i].is\_optional)

continue;

die("update package missing %s", images[i].img\_name);

}

rc = load\_buf\_fd(usb, fd, &buf);

if (rc) die("cannot load %s from flash", images[i].img\_name);

do\_update\_signature(zip, images[i].sig\_name);

if (erase\_first && needs\_erase(images[i].part\_name)) {

fb\_queue\_erase(images[i].part\_name);

}

flash\_buf(images[i].part\_name, &buf);

/\* not closing the fd here since the sparse code keeps the fd around

\* but hasn't mmaped data yet. The tmpfile will get cleaned up when the

\* program exits.

\*/

}

可以看到，flashall的时候，会批量刷入boot.img、recovery.img、system.img，其中recovery.img是可选刷入的。

其中load\_buf\_fd将解压的文件load到buf中。  
flash\_buf:

static void flash\_buf(const char \*pname, struct fastboot\_buffer \*buf)

{

struct sparse\_file \*\*s;

switch (buf->type) {

case FB\_BUFFER\_SPARSE:

s = buf->data;

while (\*s) {

int64\_t sz64 = sparse\_file\_len(\*s, true, false);

fb\_queue\_flash\_sparse(pname, \*s++, sz64);

}

break;

case FB\_BUFFER:

fb\_queue\_flash(pname, buf->data, buf->sz);

break;

default:

die("unknown buffer type: %d", buf->type);

}

}

fb\_queue\_flash和fb\_queue\_flash\_sparse：

void fb\_queue\_flash(const char \*ptn, void \*data, unsigned sz)

{

Action \*a;

a = queue\_action(OP\_DOWNLOAD, "");

a->data = data;

a->size = sz;

a->msg = mkmsg("sending '%s' (%d KB)", ptn, sz / 1024);

a = queue\_action(OP\_COMMAND, "flash:%s", ptn);

a->msg = mkmsg("writing '%s'", ptn);

}

void fb\_queue\_flash\_sparse(const char \*ptn, struct sparse\_file \*s, unsigned sz)

{

Action \*a;

a = queue\_action(OP\_DOWNLOAD\_SPARSE, "");

a->data = s;

a->size = 0;

a->msg = mkmsg("sending sparse '%s' (%d KB)", ptn, sz / 1024);

a = queue\_action(OP\_COMMAND, "flash:%s", ptn);

a->msg = mkmsg("writing '%s'", ptn);

}

queue\_action:

static Action \*queue\_action(unsigned op, const char \*fmt, ...)

{

Action \*a;

va\_list ap;

size\_t cmdsize;

a = calloc(1, sizeof(Action));

if (a == 0) die("out of memory");

va\_start(ap, fmt);

cmdsize = vsnprintf(a->cmd, sizeof(a->cmd), fmt, ap);

va\_end(ap);

if (cmdsize >= sizeof(a->cmd)) {

free(a);

die("Command length (%d) exceeds maximum size (%d)", cmdsize, sizeof(a->cmd));

}

if (action\_last) {

action\_last->next = a;

} else {

action\_list = a;

}

action\_last = a;

a->op = op;

a->func = cb\_default;

a->start = -1;

return a;

}

其实就是将对应命令和数据放入到action\_last链表中。那肯定有另外一个地方对这个链表进行真正的操作。  
fb\_execute\_queue：

int fb\_execute\_queue(usb\_handle \*usb)

{

Action \*a;

char resp[FB\_RESPONSE\_SZ+1];

int status = 0;

a = action\_list;

if (!a)

return status;

resp[FB\_RESPONSE\_SZ] = 0;

double start = -1;

for (a = action\_list; a; a = a->next) {

a->start = now();

if (start < 0) start = a->start;

if (a->msg) {

// fprintf(stderr,"%30s... ",a->msg);

fprintf(stderr,"%s...\n",a->msg);

}

if (a->op == OP\_DOWNLOAD) {

status = fb\_download\_data(usb, a->data, a->size);

status = a->func(a, status, status ? fb\_get\_error() : "");

if (status) break;

} else if (a->op == OP\_COMMAND) {

status = fb\_command(usb, a->cmd);

status = a->func(a, status, status ? fb\_get\_error() : "");

if (status) break;

} else if (a->op == OP\_QUERY) {

status = fb\_command\_response(usb, a->cmd, resp);

status = a->func(a, status, status ? fb\_get\_error() : resp);

if (status) break;

} else if (a->op == OP\_NOTICE) {

fprintf(stderr,"%s\n",(char\*)a->data);

} else if (a->op == OP\_FORMAT) {

status = fb\_format(a, usb, (int)a->data);

status = a->func(a, status, status ? fb\_get\_error() : "");

if (status) break;

} else if (a->op == OP\_DOWNLOAD\_SPARSE) {

status = fb\_download\_data\_sparse(usb, a->data);

status = a->func(a, status, status ? fb\_get\_error() : "");

if (status) break;

} else {

die("bogus action");

}

}

fprintf(stderr,"finished. total time: %.3fs\n", (now() - start));

return status;

}

fb\_download\_data:

int fb\_download\_data(usb\_handle \*usb, const void \*data, unsigned size)

{

char cmd[64];

int r;

sprintf(cmd, "download:%08x", size);

r = \_command\_send(usb, cmd, data, size, 0);

if(r < 0) {

return -1;

} else {

return 0;

}

}

\_command\_send:

static int \_command\_send(usb\_handle \*usb, const char \*cmd,

const void \*data, unsigned size,

char \*response)

{

int r;

if (size == 0) {

return -1;

}

r = \_command\_start(usb, cmd, size, response);

if (r < 0) {

return -1;

}

r = \_command\_data(usb, data, size);

if (r < 0) {

return -1;

}

r = \_command\_end(usb);

if(r < 0) {

return -1;

}

return size;

}

\_command\_start:

static int \_command\_start(usb\_handle \*usb, const char \*cmd, unsigned size,

char \*response)

{

int cmdsize = strlen(cmd);

int r;

if(response) {

response[0] = 0;

}

if(cmdsize > 64) {

sprintf(ERROR,"command too large");

return -1;

}

if(usb\_write(usb, cmd, cmdsize) != cmdsize) {

sprintf(ERROR,"command write failed (%s)", strerror(errno));

usb\_close(usb);

return -1;

}

return check\_response(usb, size, response);

}

usb\_write(system/core/fastboot/usb\_linux.c):

int usb\_write(usb\_handle \*h, const void \*\_data, int len)

{

unsigned char \*data = (unsigned char\*) \_data;

unsigned count = 0;

struct usbdevfs\_bulktransfer bulk;

int n;

if(h->ep\_out == 0) {

return -1;

}

if(len == 0) {

bulk.ep = h->ep\_out;

bulk.len = 0;

bulk.data = data;

bulk.timeout = 0;

n = ioctl(h->desc, USBDEVFS\_BULK, &bulk);

if(n != 0) {

fprintf(stderr,"ERROR: n = %d, errno = %d (%s)\n",

n, errno, strerror(errno));

return -1;

}

return 0;

}

while(len > 0) {

int xfer;

xfer = (len > MAX\_USBFS\_BULK\_SIZE) ? MAX\_USBFS\_BULK\_SIZE : len;

bulk.ep = h->ep\_out;

bulk.len = xfer;

bulk.data = data;

bulk.timeout = 0;

n = ioctl(h->desc, USBDEVFS\_BULK, &bulk);

if(n != xfer) {

DBG("ERROR: n = %d, errno = %d (%s)\n",

n, errno, strerror(errno));

return -1;

}

count += xfer;

len -= xfer;

data += xfer;

}

return count;

}

这里就是将内容写入到usb\_handle对应的usb设备中。这样我们的数据就通过usb写入到设备中了，而在设备中，有对应的usb驱动程序来处理写过去的数据。对于设备如何处理写过去的数据，我并没有搜索到相关的代码，也许在厂商的驱动程序里面，同样，屌大的同学可以给我讲讲。

## fastboot flash xxx xxx.img

fastboot flash vendor xxx.img

有时候我们只刷单个的img，使用的命令是fastboot flash xxx xxx.img，也来看看吧：

else if(!strcmp(\*argv, "flash")) {

char \*pname = argv[1];

char \*fname = 0;

require(2);

if (argc > 2) {

fname = argv[2];

skip(3);

} else {

fname = find\_item(pname, product);

skip(2);

}

if (fname == 0) die("cannot determine image filename for '%s'", pname);

if (erase\_first && needs\_erase(pname)) {

fb\_queue\_erase(pname);

}

do\_flash(usb, pname, fname);

}

do\_flash:

void do\_flash(usb\_handle \*usb, const char \*pname, const char \*fname)

{

struct fastboot\_buffer buf;

if (load\_buf(usb, fname, &buf)) {

die("cannot load '%s'", fname);

}

flash\_buf(pname, &buf);

}

同样的是将数据load到buf，然后调用flash\_buf，flash\_buf前面已经分析过了，就不再复述。

## 情景四：

在help中，我们看到这个：

boot <kernel> [ <ramdisk> ] download and boot kernel

貌似对boot镜像有特别的处理，跟踪一下：

else if(!strcmp(\*argv, "boot")) {

char \*kname = 0;

char \*rname = 0;

skip(1);

if (argc > 0) {

kname = argv[0];

skip(1);

}

if (argc > 0) {

rname = argv[0];

skip(1);

}

data = load\_bootable\_image(kname, rname, &sz, cmdline);

if (data == 0) return 1;

fb\_queue\_download("boot.img", data, sz);

fb\_queue\_command("boot", "booting");

}

其中，第一个参数为kernel文件名，第二个参数为ramdisk文件名(可选的)。

load\_bootable\_image:

void \*load\_bootable\_image(const char \*kernel, const char \*ramdisk,

unsigned \*sz, const char \*cmdline)

{

void \*kdata = 0, \*rdata = 0;

unsigned ksize = 0, rsize = 0;

void \*bdata;

unsigned bsize;

if(kernel == 0) {

fprintf(stderr, "no image specified\n");

return 0;

}

kdata = load\_file(kernel, &ksize);

if(kdata == 0) {

fprintf(stderr, "cannot load '%s': %s\n", kernel, strerror(errno));

return 0;

}

/\* is this actually a boot image? \*/

if(!memcmp(kdata, BOOT\_MAGIC, BOOT\_MAGIC\_SIZE)) {

if(cmdline) bootimg\_set\_cmdline((boot\_img\_hdr\*) kdata, cmdline);

if(ramdisk) {

fprintf(stderr, "cannot boot a boot.img \*and\* ramdisk\n");

return 0;

}

\*sz = ksize;

return kdata;

}

if(ramdisk) {

rdata = load\_file(ramdisk, &rsize);

if(rdata == 0) {

fprintf(stderr,"cannot load '%s': %s\n", ramdisk, strerror(errno));

return 0;

}

}

fprintf(stderr,"creating boot image...\n");

bdata = mkbootimg(kdata, ksize, kernel\_offset,

rdata, rsize, ramdisk\_offset,

0, 0, second\_offset,

page\_size, base\_addr, tags\_offset, &bsize);

if(bdata == 0) {

fprintf(stderr,"failed to create boot.img\n");

return 0;

}

if(cmdline) bootimg\_set\_cmdline((boot\_img\_hdr\*) bdata, cmdline);

fprintf(stderr,"creating boot image - %d bytes\n", bsize);

\*sz = bsize;

return bdata;

}

先通过load\_file将kernel加载到内存中，如果发现其魔数是ANDROID!，代表其已经是打包好的boot.img文件(包含了kernel和ramdisk)，这种情况下就忽略对ramdisk的处理。  
不然的话，就将ramdisk也加载到内存，并使用mkbootimg将二者打包成boot.img格式的数据。  
然后将这个打包好的数据，写入的usb中，指定要刷的目标为boot.img。

所以，这个命令可以有两种用法：

1.

fastboot boot boot.img

2.

fastboot boot kernel ramdisk //会先打包成boot.img

另外，可以看到，在使用mkbootimg时，用到了很多参数，很多参数是可以通过特定选项指定的，比如kernel\_offset用-k:

case 'k':

kernel\_offset = strtoul(optarg, 0, 16);

## 情景五

类似的flash:raw:

else if(!strcmp(\*argv, "flash:raw")) {

char \*pname = argv[1];

char \*kname = argv[2];

char \*rname = 0;

require(3);

if(argc > 3) {

rname = argv[3];

skip(4);

} else {

skip(3);

}

data = load\_bootable\_image(kname, rname, &sz, cmdline);

if (data == 0) die("cannot load bootable image");

fb\_queue\_flash(pname, data, sz);

}

和fastboot boot类似，先将kernel和ramdisk打包，在刷入，这里的不同是，你需要指定pname，即boot，使用方式如下：

fastboot flash:raw boot kernel ramdisk

## 总结

至此，重要的几个命令分析清楚了，即通过fastboot协议，将数据写入到usb中。感觉完整的整个过程，需要再分析一下设备上的usb驱动接收到数据后的处理过程。但暂时没有相关的代码可以分析，先到此为止。

## 参考

[Android Fastboot源码分析](https://www.jianshu.com/p/e3342b5e5581)

# Toolbox

## Getevent

源码路径：/system/core/toolbox/getevent.c

getevent 指令用于获取 input 输入事件，比如获取按键上报信息、获取触摸屏上报信息等。

getevent监听输入设备节点的内容，当输入事件被写入到节点中时，getevent会将其读出并打印在屏幕上。由于getevent不会对事件数据做任何加工，因此其输出的内容是由内核提供的最原始的事件

### 常用命令

#### bullhead:/dev/input # getevent -h

Usage: getevent [-t] [-n] [-s switchmask] [-S] [-v [mask]] [-d] [-p] [-i] [-l] [-q] [-c count] [-r] [device]

-t: show time stamps

-n: don't print newlines

-s: print switch states for given bits

-S: print all switch states

-v: verbosity mask (errs=1, dev=2, name=4, info=8, vers=16, pos. events=32, props=64)

-d: show HID descriptor, if available

-p: show possible events (errs, dev, name, pos. events)

-i: show all device info and possible events

-l: label event types and names in plain text

-q: quiet (clear verbosity mask)

-c: print given number of events then exit

-r: print rate events are received

-l：以文本形式输出事件类型和名称，比 -t 更清楚直观

Sdf

getevent /dev/input/event3

帮助信息

getevent -h

-t：显示时间戳

参数可以组合使用，一次性查看需要的触摸屏信息

#### getevent -tlr /dev/input/event3

输出是十六进制的。每条数据有五项信息：产生事件时的时间戳（[ 1262.443489]），产生事件的设备节点（/dev/input/event0），事件类型（0001），事件代码（0074）以及事件的值（00000001）。其中时间戳、类型、代码、值便是原始事件的四项基本元素

// 事件类型 事件码 事件值

bullhead:/dev/input # getevent -tlr /dev/input/event0

[ 66568.304633] EV\_SYN 0004 00010408

[ 66568.304633] EV\_SYN 0005 11ea975a

[ 66568.304633] EV\_ABS ABS\_MT\_TRACKING\_ID 000000b7

[ 66568.304633] EV\_ABS ABS\_MT\_POSITION\_X 00000000

[ 66568.304633] EV\_ABS ABS\_MT\_POSITION\_Y 00000771

[ 66568.304633] EV\_ABS ABS\_MT\_PRESSURE 0000001c

[ 66568.304633] EV\_ABS ABS\_MT\_TOUCH\_MAJOR 00000006

[ 66568.304633] EV\_ABS ABS\_MT\_TOUCH\_MINOR 00000006

[ 66568.304633] EV\_SYN SYN\_REPORT 00000000 rate 0

[ 66568.353456] EV\_SYN 0004 00010408

[ 66568.353456] EV\_SYN 0005 14dde349

[ 66568.353456] EV\_ABS ABS\_MT\_POSITION\_Y 00000772

[ 66568.353456] EV\_ABS ABS\_MT\_PRESSURE 00000013

[ 66568.353456] EV\_SYN SYN\_REPORT 00000000 rate 20

[ 66568.369257] EV\_SYN 0004 00010408

[ 66568.369257] EV\_SYN 0005 15cb5a4f

[ 66568.369257] EV\_ABS ABS\_MT\_TRACKING\_ID ffffffff

[ 66568.369257] EV\_SYN SYN\_REPORT 00000000 rate 63

点击一次屏幕日志：

参考：<https://www.cnblogs.com/lialong1st/p/9093851.html>

### 附录-查看输入设备信息

使用getevent获得/dev/input/eventX设备汇报的事件，这个命令还会输出所有event设备的基本信息，

bullhead:/dev/input # getevent

add device 1: /dev/input/event7

name: "uinput-folio"

add device 2: /dev/input/event6

name: "uinput-fpc"

add device 3: /dev/input/event5

name: "msm8994-tomtom-snd-card Headset Jack"

add device 4: /dev/input/event4

name: "msm8994-tomtom-snd-card Button Jack"

add device 5: /dev/input/event2==》键盘、按键keypad

name: "qpnp\_pon"

add device 6: /dev/input/event1

name: "STM VL6180 proximity sensor"

add device 7: /dev/input/event3

name: "gpio-keys"

add device 8: /dev/input/event0 ===》显示屏touchscreen

name: "synaptics\_rmi4\_i2c"

 /dev/input目录下的事件都是在驱动中调用input\_register\_device(struct input\_dev \*dev)产生的。

127|bullhead:/dev/input # ls -al

total 0

drwxr-xr-x 2 root root 200 1970-01-04 05:30 .

drwxr-xr-x 15 root root 3680 1970-01-04 05:30 ..

crw-rw---- 1 root input 13, 64 1970-01-04 05:30 event0

crw-rw---- 1 root input 13, 65 1970-01-04 05:30 event1

crw-rw---- 1 root input 13, 66 1970-01-04 05:30 event2

crw-rw---- 1 root input 13, 67 1970-01-04 05:30 event3

crw-rw---- 1 root input 13, 68 1970-01-04 05:30 event4

crw-rw---- 1 root input 13, 69 1970-01-04 05:30 event5

crw-rw---- 1 root input 13, 70 1970-01-04 05:30 event6

crw-rw---- 1 root input 13, 71 1970-01-04 05:30 event7

与event对应的相关设备信息位于/proc/bus/input/devices，

2|bullhead:/dev/input # cat /proc/bus/input/devices

I: Bus=0018 Vendor=0000 Product=0001 Version=1005

N: Name="synaptics\_rmi4\_i2c"

P: Phys=synaptics\_rmi4\_i2c/input0

S: Sysfs=/devices/soc.0/f9924000.i2c/i2c-2/2-0020/input/input0

U: Uniq=

H: Handlers=kgsl event0 cpufreq

B: PROP=2

B: EV=b

B: KEY=0

B: ABS=663800000000000

I: Bus=0000 Vendor=0000 Product=0000 Version=0000

N: Name="STM VL6180 proximity sensor"

P: Phys=

S: Sysfs=/devices/virtual/input/input1

U: Uniq=

H: Handlers=event1

B: PROP=0

B: EV=9

B: ABS=2ff0000

I: Bus=0000 Vendor=0000 Product=0000 Version=0000

N: Name="qpnp\_pon"

P: Phys=qpnp\_pon/input0

S: Sysfs=/devices/virtual/input/input2

U: Uniq=

H: Handlers=event2 cpufreq keychord

B: PROP=10

B: EV=3

B: KEY=18000000000000 0

I: Bus=0019 Vendor=0001 Product=0001 Version=0100

N: Name="gpio-keys"

P: Phys=gpio-keys/input0

S: Sysfs=/devices/soc.0/gpio\_keys.86/input/input3

U: Uniq=

H: Handlers=event3 cpufreq keychord

B: PROP=0

B: EV=3

B: KEY=4000000000000 0

I: Bus=0000 Vendor=0000 Product=0000 Version=0000

N: Name="msm8994-tomtom-snd-card Button Jack"

P: Phys=ALSA

S: Sysfs=/devices/soc.0/fe034000.sound/sound/card0/input4

U: Uniq=

H: Handlers=event4 cpufreq keychord

B: PROP=10

B: EV=3

B: KEY=40 0 0 0 0 f0 400000000 0 c000000000000 0

## sendevent

输入设备的节点不仅在用户空间可读，而且是可写的，因此可以将将原始事件写入到节点中，从而实现模拟用户输入的功能。sendevent工具的作用正是如此。其用法如下：

|  |  |
| --- | --- |
| 1 | sendevent <节点路径> <类型><代码> <值> |

可以看出，sendevent的输入参数与getevent的输出是对应的，只不过sendevent的参数为十进制。电源键的代码0x74的十进制为116，因此可以通过快速执行如下两条命令实现点击电源键的效果：

|  |  |
| --- | --- |
| 1 2 3 | adb shell sendevent /dev/input/event0 1 116 1 #按下电源键  adb shell sendevent /dev/input/event0 1 116 0 #抬起电源键 |