内容目录

[零 输入系统\_必备Linux编程知识 1](#__RefHeading___Toc248_2054555503)

[0.1 inotify和epoll 1](#__RefHeading___Toc16445_1060533589)

[0.1.1 inotify 1](#__RefHeading___Toc16447_1060533589)

[0.1.2 epoll 1](#__RefHeading___Toc16449_1060533589)

[0.1.3 inotify + epoll 1](#__RefHeading___Toc16451_1060533589)

[0.2 socketpair 1](#__RefHeading___Toc16453_1060533589)

[0.3 任意进程双向通信(socketpair+binder) 2](#__RefHeading___Toc16455_1060533589)

[一 输入系统框架 4](#__RefHeading___Toc16458_1060533589)

[1.1 输入系统\_框架 4](#__RefHeading___Toc890_1404923994)

[1.2 输入系统\_模拟输入系统驱动 4](#__RefHeading___Toc892_1404923994)

[1.3 输入系统\_Reader\_Dispatcher线程启动分析 5](#__RefHeading___Toc894_1404923994)

[1.4 输入系统\_Reader线程\_使用EventHub读取事件 6](#__RefHeading___Toc902_1404923994)

[1.5 输入系统\_Reader线程\_核心类及配置文件\_实验 6](#__RefHeading___Toc1499_1698680629)

[1.6 输入系统\_Reader线程\_核心类及配置文件\_分析 7](#__RefHeading___Toc1501_1698680629)

[1.7 输入系统\_Reader线程\_简单处理 8](#__RefHeading___Toc1503_1698680629)

[1.8 输入系统\_Dispatcher线程总体框架 9](#__RefHeading___Toc1505_1698680629)

# 零 输入系统\_必备Linux编程知识

(1) 键盘即插即用=》怎么检测键盘的接入与拔出

1) hotplug: 内核发现键盘插入/拔出=》启动hotplug进程=》发送消息给输入系统；ps: android没有用这个方式

2) inotify: 输入系统使用inotify检测目录/dev/input

(2) 对于多键盘，使用的是epoll

## 0.1 inotify和epoll

### 0.1.1 inotify

监测目录 / 文件的变化，参考代码: frameworks\native\services\inputflinger\EventHub.cpp

APP\_0006\_inotify\_epoll/inotify.c

gcc -o inotify inotify.c

mkdir tmp

./inotify tmp &

echo > tmp/1

echo > tmp/2

rm tmp/1 tmp/2

### 0.1.2 epoll

检测多个文件有无数据供读出，有无空间供写入

APP\_0006\_inotify\_epoll/epoll.c

gcc -o epoll epoll.c

mkdir tmp

mkfifo tmp/1 tmp/2 tmp/3

./epoll tmp/1 tmp/2 tmp/3 &

echo aaa > tmp/1

echo bbb > tmp/2

使用fifo是，我们的epoll程序是reader

echo aa > tmp/1 是writer

a. int tmpFd = open(argv[i], O\_RDONLY|O\_NONBLOCK);//会导致无限循环

如果reader以 O\_RDONLY|O\_NONBLOCK打开FIFO文件，

当writer写入数据时，epoll\_wait会立刻返回；

当writer关闭FIFO之后，reader再次调用epoll\_wait，它也会立刻返回(原因是EPPLLHUP，描述符被挂断)

b. int tmpFd = open(argv[i], O\_RDWR);

如果reader以 O\_RDWR打开FIFO文件

当writer写入数据时，epoll\_wait会立刻返回；

当writer关闭FIFO之后，reader再次调用epoll\_wait，它并不会立刻返回，而是继续等待有数据

### 0.1.3 inotify + epoll

编写 inotify\_epoll.c，用它来监测tmp/目录: 有文件被创建/删除，有文件可读出数据

a. 当在tmp/下创建文件时，会立刻监测到，并且使用epoll监测该文件

b. 当文件有数据时，读出数据

c. 当tmp/下文件被删除时，会立刻监测到，并且把它从epoll中移除不再监测

inotify\_epoll.c

gcc -o inotify\_epoll inotify\_epoll.c

mkdir tmp

./inotify\_epoll tmp/ &

mkfifo tmp/1 tmp/2 tmp/3

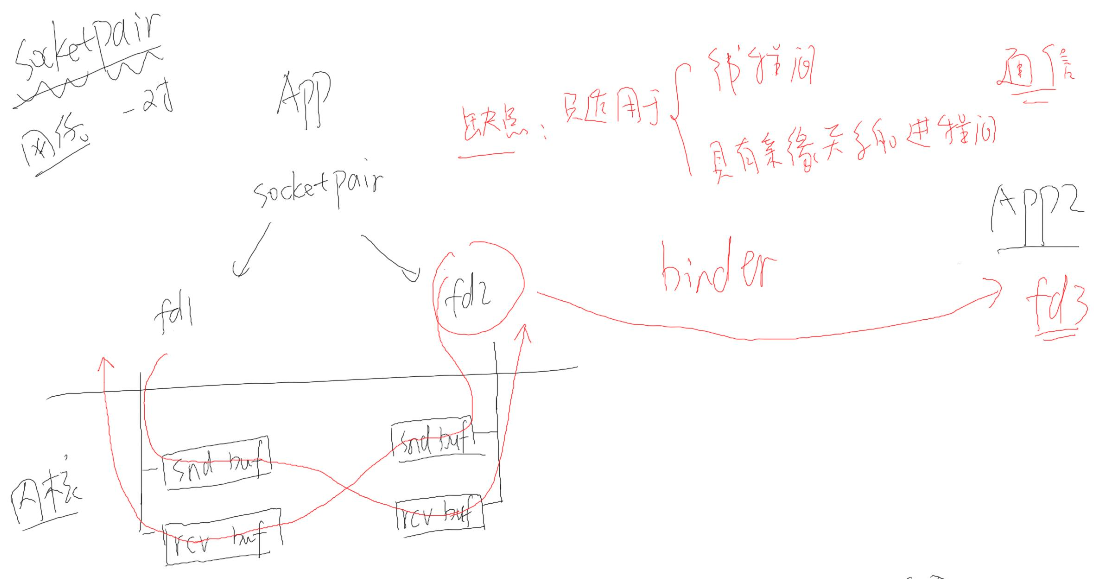
echo aaa > tmp/1

echo bbb > tmp/2

rm tmp/3

## 0.2 socketpair

APP\_0007\_socketpair\_binder/socketpair.c，亲缘关系是指fork



### 0.3 任意进程双向通信(socketpair+binder)

APP1打开1.txt得到句柄fd，在kernel中task\_struct来描述一个线程(进程)，

struct task\_struct {

struct files\_struct \*files;

{

struct fdtable \_\_rcu \*fdt;

{

struct file \_\_rcu \*\*fd; /\* current fd array \*/

}

}

}

使用binder传输文件句柄

(1) APP1 open file得到fd1

(2) 通过binder驱动，通过fd1得到file，file = APP1->files->fdt->fd[fd1]

(3) 从APP2的files->fdt->fd取出空项fd2，让它指向该file，APP2->files->fdt->fd[fd2] = file

(4) APP1通过fd1，APP2通过fd2可访问同一个文件

Input/APP\_0004\_Binder\_CPP\_App/

git pull origin

git checkout v5 // v5, use binder to transfer file descriptor

test\_server.cpp

/\* usage : test\_server <file> \*/

int main(int argc, char \*\*argv)

{

int fd;

if (argc == 2)

fd = open(argv[1], O\_RDWR);

…

sm->addService(String16("hello"), new BnHelloService(fd));

…

}

IHelloService.h

class BnHelloService: public BnInterface<IHelloService>

{

private:

int fd;

public:

virtual int get\_fd(void);

…

BnHelloService(int fd);

};

BnHelloService.cpp

status\_t BnHelloService::onTransact( uint32\_t code,

const Parcel& data,

Parcel\* reply,

uint32\_t flags)

{

…

case HELLO\_SVR\_CMD\_GET\_FD: {

int fd = this->get\_fd();

reply->writeInt32(0); /\* no exception \*/

/\* 参考:

\* frameworks\base\core\jni\android\_view\_InputChannel.cpp

\* android\_view\_InputChannel\_nativeWriteToParcel

\*/

reply->writeDupFileDescriptor(fd);

return NO\_ERROR;

} break;

…

}

int BnHelloService::get\_fd(void)

{

return fd;

}

BpHelloService.cpp

int get\_fd(void)

{

/\* 构造/发送数据 \*/

Parcel data, reply;

int exception;

data.writeInt32(0);

data.writeString16(String16("IHelloService"));

remote()->transact(HELLO\_SVR\_CMD\_GET\_FD, data, &reply);

exception = reply.readInt32();

if (exception)

return -1;

else

{

/\* 参考:

\* frameworks\base\core\jni\android\_view\_InputChannel.cpp

\* android\_view\_InputChannel\_nativeReadFromParcel

\*/

int rawFd = reply.readFileDescriptor();

return dup(rawFd);//rawFd被析构时，要用dup来保证fd不被关闭，而不是直接return rawFd

}

}

test\_client.cpp

/\* 调用Service的函数 \*/

int fd = service->get\_fd();

ALOGI("client call get\_fd = %d", fd);

//while (1) sleep(10);

lseek(fd, 0, SEEK\_SET);

char buf[500];

int len = read(fd, buf, 500);

buf[len] = '\0';

ALOGI("client read file: %s", buf);

编译

mmm frameworks/testing/APP\_0004\_Binder\_CPP\_App

执行test\_server

sm6150\_au:/proc/3287/fd # cat /sdcard/1.txt

hello, fd!

sm6150\_au:/sdcard # test\_server /sdcard/1.txt

sm6150\_au:/proc/3287/fd # ls -l

total 0

...

lrwx------ 1 root root 64 2019-07-03 21:01 3 -> /storage/emulated/0/1.txt

...

执行test\_client

sm6150\_au:/proc/3287/fd # test\_client readfile

sm6150\_au:/proc/3287/fd # logcat | grep TestService

07-03 21:03:47.162 3341 3341 I TestService: client call get\_fd = 5

07-03 21:03:47.162 3341 3341 I TestService: client read file: hello, fd!

sm6150\_au:/proc/3341/fd # ls -l

total 0

lrwx------ 1 root root 64 2019-07-03 21:10 0 -> /dev/pts/1

lrwx------ 1 root root 64 2019-07-03 21:10 1 -> /dev/pts/1

lrwx------ 1 root root 64 2019-07-03 21:10 2 -> /dev/pts/1

lrwx------ 1 root root 64 2019-07-03 21:10 3 -> /dev/binder

lrwx------ 1 root root 64 2019-07-03 21:10 4 -> socket:[66320]

lrwx------ 1 root root 64 2019-07-03 21:10 5 -> /storage/emulated/0/1.txt

原理

reply->writeDupFileDescriptor(fd);

→ obj.hdr.type = BINDER\_TYPE\_FD;

→ case BINDER\_TYPE\_FD: {

struct binder\_fd\_object \*fp = to\_binder\_fd\_object(hdr);

target\_fd = binder\_translate\_fd(fp->fd, t, thread, in\_reply\_to);

}

→ file = fget(fd);//从当前进程里面，根据fd得到一个file结构体

target\_fd = task\_get\_unused\_fd\_flags(target\_proc, O\_CLOEXEC);//从目标进程里面，获得一个没有使用的文件句柄

task\_fd\_install(target\_proc, target\_fd, file);//把file安装到目标进程的target\_fd上面去

课后作业: 支持双向通信的程序 v6

第一次:

git clone https://github.com/weidongshan/APP\_0004\_Binder\_CPP\_App.git

更新:

git pull origin

取出指定版本:

git checkout v6 // v6, use binder and socketpair for bidirectional transfer，将socketpair的fd通过binder传递给client

编译命令与v5相同

# 一 输入系统框架

<http://source.android.com/devices/input/index.html>

## 1.1 输入系统\_框架

输入系统深入分析1\_框架.jpg

## 1.2 输入系统\_模拟输入系统驱动

驱动操作硬件有多套open/read/write操作，

层1:

drivers/input/evdev.c

evdev\_open、evdev\_read、evdev\_write //得到输入事件的原始数据，/dev/input/event0、1、2

keyboard.c

mousedev.c，/dev/mouse0、1、2，得到经过加工后的鼠标数据，也可以通过/dev/input/event0、1、2得到鼠标的原始数据

层2:

硬件相关: 根据硬件的状态上报数据(原始数据)，比如鼠标上报给 mouse设备得到鼠标加工后的数据，也可以上报给event设备得到鼠标原始数据

层3:

硬件

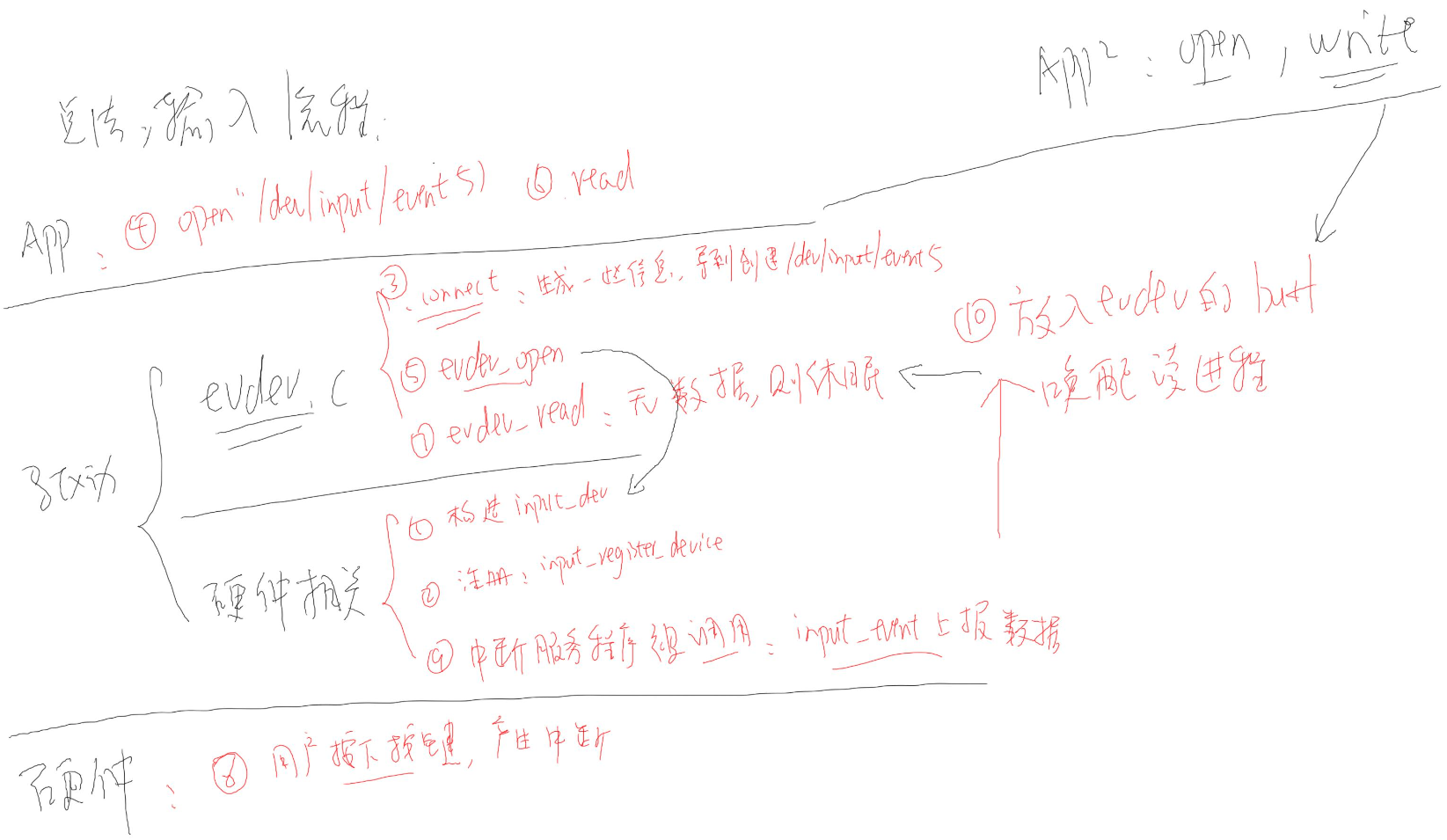
小节:

所以，我们需要写出硬件相关的驱动程序:

1)分配/构造input\_device结构体

2)注册input\_register\_device，可以跟evdev.c、keyboard.c、mousedev.c建立联系

3)有输入事件产生时，中断程序上报: input\_event(dev, type, code, value)，比如 type=EV\_KEY(按键类事件)，code=KEY\_1(按下了1键)，value=1(按下) / 0(松开)

 如何模拟输入驱动程序呢？如上图所示，用APP2来write数据，sendevent。

sm6150\_au:/ # sendevent --help

usage: sendevent DEVICE TYPE CODE VALUE

Sends a Linux input event.

编写一个万能模拟输入驱动程序，Input/DRV\_0004\_InputEmulator/InputEmulator.c

static int input\_emulator\_init(void)

{

int i;

/\* 1. 分配一个input\_dev结构体 \*/

input\_emulator\_dev = input\_allocate\_device();;

/\* 2. 设置 \*/

/\* 2.1 能产生哪类事件 \*/

set\_bit(EV\_KEY, input\_emulator\_dev->evbit);//按键类事件

set\_bit(EV\_REP, input\_emulator\_dev->evbit);//当你长按一个键时，重复上报事件

/\* 2.2 能产生所有的按键 \*/

for (i = 0; i < BITS\_TO\_LONGS(KEY\_CNT); i++)

input\_emulator\_dev->keybit[i] = ~0UL;//设置里面的每一项等于全ff，产生所有的按键类事件

/\* 2.3 为android构造一些设备信息 \*/

<https://source.android.com/devices/input/input-device-configuration-files>，android会查找根据.idc文件里面的配置信息，如果没有则使用缺省值，我们把这些信息设置上去

input\_emulator\_dev->name = "InputEmulatorFrom100ask.net";

input\_emulator\_dev->id.bustype = 1;

input\_emulator\_dev->id.vendor = 0x1234;

input\_emulator\_dev->id.product = 0x5678;

input\_emulator\_dev->id.version = 1;

/\* 3. 注册 \*/

input\_register\_device(input\_emulator\_dev);

return 0;

}

本地测试直接编译到kernel里

diff --git a/drivers/input/Makefile b/drivers/input/Makefile

index f0351af..c05df4c 100644

--- a/drivers/input/Makefile

+++ b/drivers/input/Makefile

@@ -31,3 +31,5 @@ obj-$(CONFIG\_INPUT\_KEYRESET) += keyreset.o

obj-$(CONFIG\_INPUT\_KEYCOMBO) += keycombo.o

obj-$(CONFIG\_RMI4\_CORE) += rmi4/

+

+obj-y += InputEmulator.o

sendevent /dev/input/event1 1 2 1 // 1 2 1 : EV\_KEY, KEY\_1, down

sendevent /dev/input/event1 1 2 0 // 1 2 0 : EV\_KEY, KEY\_1, up

sendevent /dev/input/event1 0 0 0 // sync，输入时间报完毕

sendevent /dev/input/event1 1 3 1

sendevent /dev/input/event1 1 3 0

sendevent /dev/input/event1 0 0 0

以下三行输入完之后

sm6150\_au:/ # sendevent /dev/input/event1 1 2 1

sm6150\_au:/ # sendevent /dev/input/event1 1 2 0

sm6150\_au:/ # sendevent /dev/input/event1 0 0 0

才会打印

sm6150\_au:/ $ getevent -l

add device 1: /dev/input/event0

name: "qpnp\_pon"

add device 2: /dev/input/event1

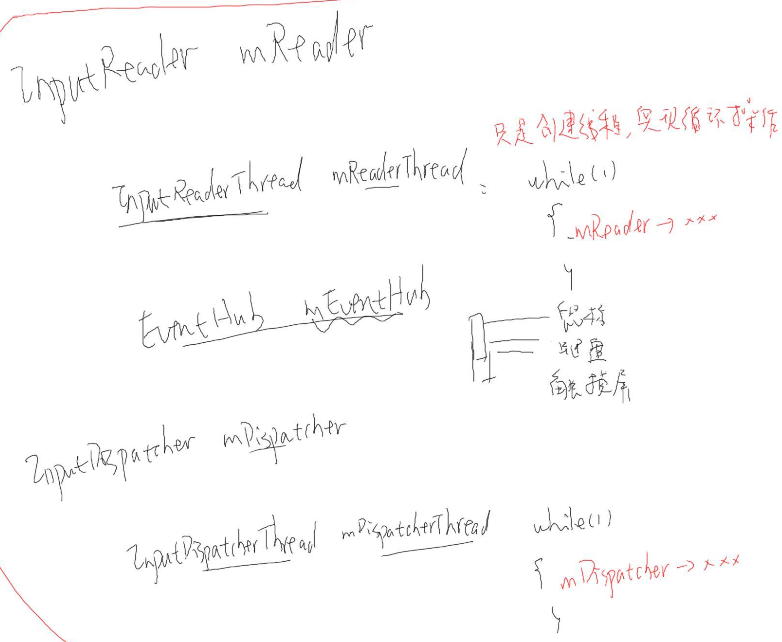
name: "InputEmulatorFrom100ask.net"

/dev/input/event1: EV\_KEY KEY\_1 DOWN

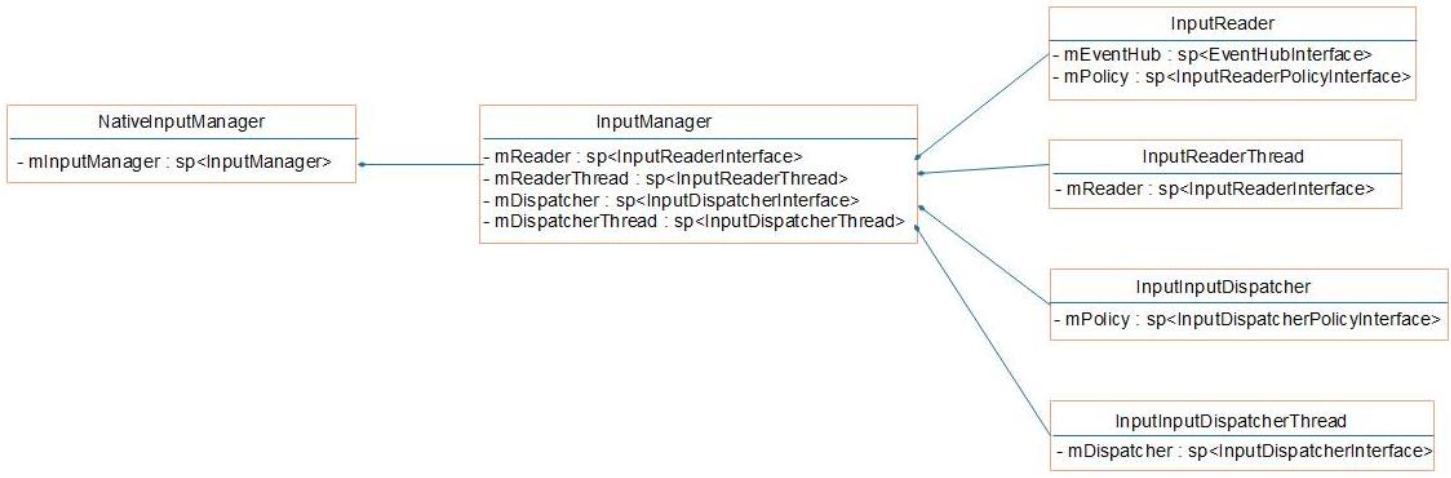
/dev/input/event1: EV\_KEY KEY\_1 UP

/dev/input/event1: EV\_SYN SYN\_REPORT 00000000

## 1.3 输入系统\_Reader\_Dispatcher线程启动分析



把上面用InputManager封装起来，C++实现的，Java通过NativeInputManager(实例化对象im)来访问，从SystemServer.java开始，创建并启动Reader/Dispatcher线程。

 InputReaderThread: 创建一个子线程，然后提供一个循环，在这个循环里面需要用到mReader来执行某些操作，比如去读取输入事件; mReader里面还有一个mEventHub，用这个mEventHub可以来监测多个输入设备。

InputDispatcherThread: 同样是创建一个线程，提供一个循环，最终要调用到mDispatcher提供的某些函数，来分发输入事件

图中最右边涉及到的四个类被实例化到InputManager对象里面统一管理; Java通过NativeInputManager来进行创建InputManager，最终导致这四个成员被创建。

具体code flow，wds老师已经画出了时序图。先启动Dispatcher线程，再启动Reader线程。

## 1.4 输入系统\_Reader线程\_使用EventHub读取事件

Reader线程: (1)获得事件、(2)简单处理、(3)传给Dispatcher线程

frameworks/native/services/inputflinger/InputReader.h

frameworks/native/services/inputflinger/InputReader.cpp

void InputReader::loopOnce() {

size\_t count = mEventHub->getEvents(timeoutMillis, mEventBuffer, EVENT\_BUFFER\_SIZE);//获得的事件存放在数组里面

}

RawEvent mEventBuffer[EVENT\_BUFFER\_SIZE];

frameworks/native/services/inputflinger/EventHub.h

struct RawEvent {

nsecs\_t when;

int32\_t deviceId;

int32\_t type;

int32\_t code;

int32\_t value;

};

type: //进行了扩展，可以表示更多的类型，比如输入设备的拔插

DEVICE\_ADDED

DEVICE\_REMOVED

FINISHED\_DEVICE\_SCAN

FIRST\_SYNTHETIC\_EVENT = DEVICE\_ADDED

EV\_KEY

EV\_ABS

EV\_REL

...

kernel/msm-4.14/include/linux/input.h

struct input\_value {

\_\_u16 type;

\_\_u16 code;

\_\_s32 value;

};

type:

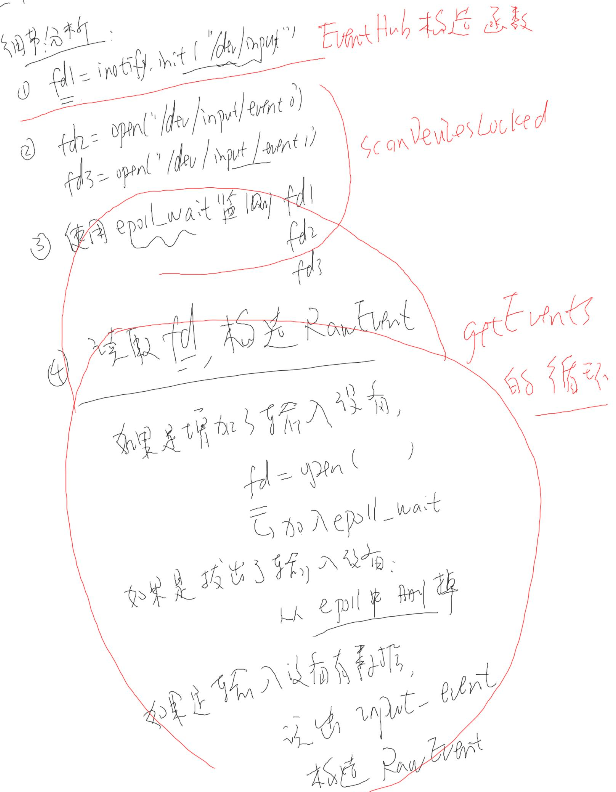
EV\_KEY

EV\_ABS

EV\_REL

...

如果输入设备有数据产生，驱动程序上报数据，读取数据构造成 RawEvent数据，使用inotify检测/dev/input/输入设备的拔插，使用epoll检测有无数据



## 1.5 输入系统\_Reader线程\_核心类及配置文件\_实验

**(1)kl(keylayout)文件**

根据命名规则，将/system/usr/keylayout/Generic.kl命名为InputEmulatorFrom100ask\_net.kl，添加这两行，\*和#

key 227 STAR

key 228 POUND

然后push到设备上

adb push InputEmulatorFrom100ask\_net.kl /data/system/devices/keylayout/InputEmulatorFrom100ask\_net.kl

sm6150\_au:/ # chmod 777 -R /data/system/devices/

sm6150\_au:/ # reboot

发送\*键

sendevent /dev/input/event1 1 227 1

sendevent /dev/input/event1 1 227 0

sendevent /dev/input/event1 0 0 0

发送#键

sendevent /dev/input/event1 1 228 1

sendevent /dev/input/event1 1 228 0

sendevent /dev/input/event1 0 0 0

**(2)kcm文件格式**

key B {

label: 'B' # 印在按键上的文字

base: 'b' # 如果没有其他按键(shift, ctrl等)同时按下，此按键对应的字符是'b'

shift, capslock: 'B'

}

frameworks/native/include/android/keycodes.h

B 表示 AKEYCODE\_B

实验:

mkdir -p /data/system/devices/keychars

cp /system/usr/keychars/Generic.kcm /data/system/devices/keychars/InputEmulatorFrom100ask\_net.kcm

修改:

key STAR {

label: '\*'

# base: '\*'

base: '1' #改为1，UI文本框会显示1

}

key POUND {

label: '#'

# base: '#'

base: '2' #改为2

}

busybox chmod 777 /data/system/devices -R

重启

insmod InputEmulator.ko

发送\*键, 得到1

sendevent /dev/input/event5 1 227 1

sendevent /dev/input/event5 1 227 0

sendevent /dev/input/event5 0 0 0

发送#键, 得到2

sendevent /dev/input/event5 1 228 1

sendevent /dev/input/event5 1 228 0

sendevent /dev/input/event5 0 0 0

**小结:**

keylayout: 只是用来表示驱动上报的scancode对应哪一个android按键(AKEYCODE\_x)

只是表示按键被按下

它对应哪一个字符，由kcm文件决定

kcm: 用来表示android按键(AKEYCODE\_x)对应哪一个字符

表示同时按下其他按键后，对应哪个字符

也可以用组合键 shilt + xx

kernel/msm-4.14/include/uapi/linux/input-event-codes.h

sendevent /dev/input/event5 1 42 1

sendevent /dev/input/event5 1 9 1

sendevent /dev/input/event5 1 9 0

sendevent /dev/input/event5 1 42 0

sendevent /dev/input/event5 0 0 0

sendevent /dev/input/event5 1 42 1

sendevent /dev/input/event5 1 4 1

sendevent /dev/input/event5 1 4 0

sendevent /dev/input/event5 1 42 0

sendevent /dev/input/event5 0 0 0

## 1.6 输入系统\_Reader线程\_核心类及配置文件\_分析

.idc文件一般不用，其格式

property = value

比如

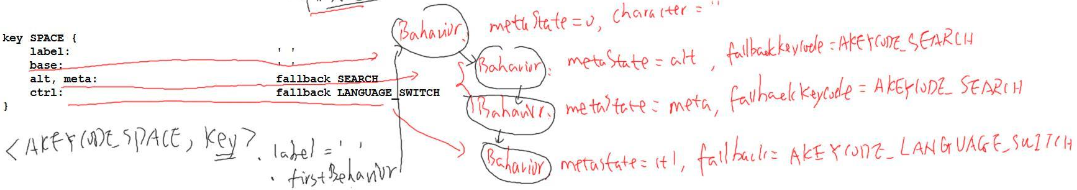
device.internal = 0 | 1

指定输入设备属于内置组件，还是外部连接（很可能可拆卸）的外围设备。

你还可以使用idc文件来指定使用哪个kl和kcm文件

流程框图见: 输入系统深入分析4\_Reader线程分析.jpg

kcm文件也有kl功能，但是一般不使用这个功能

 这里说一下fallback，在Android输入系统里，native收到键值之后，会向应用程序上报一个keycode，应用程序如果能处理这个keycode的话，处理完之后会回复一个已经处理的信号；如果不能处理的话，输入系统会再次上报一个值，这个值来自fallback。

比如当按下alt+空格的话，如果应用程序不能处理的话，它会反馈给输入系统，输入系统会再次上报一个AKEYCODE\_SEARCH给应用程序。

## 1.7 输入系统\_Reader线程\_简单处理

**Add Device/Remove Device**

EventHub有一个KeyedVector<int32\_t, Device\*> mDevices;维护具体的输入设备，读取事件，还有kl、ksm

同时，InputReader还有一个KeyedVector<int32\_t, InputDevice\*> mDevices 维护输入设备，InputDevice mDevices.mId可以在Device mDevices找到对应的device；

InputReader Vector<InputMapper\*> mMappers 处理上报的事件；对于键盘还有一个KeyboardInputMapper继承InputMapper 类来处理。

**真正的输入事件**

void InputReader::processEventsLocked(const RawEvent\* rawEvents, size\_t count)

→ void InputReader::processEventsForDeviceLocked(int32\_t deviceId, const RawEvent\* rawEvents, size\_t count)

→ void InputDevice::process(const RawEvent\* rawEvents, size\_t count)

void MultiTouchMotionAccumulator::process(const RawEvent\* rawEvent) //多点触摸屏

void SwitchInputMapper::process(const RawEvent\* rawEvent) //switch开关，滑盖、翻盖

void SingleTouchInputMapper::process(const RawEvent\* rawEvent) //单点触摸屏

//还有很多种类，等等，我们找到键盘类的

void KeyboardInputMapper::process(const RawEvent\* rawEvent)

void KeyboardInputMapper::process(const RawEvent\* rawEvent) {

switch (rawEvent->type) {

case EV\_KEY: {

int32\_t scanCode = rawEvent->code;

int32\_t usageCode = mCurrentHidUsage;

mCurrentHidUsage = 0;

if (isKeyboardOrGamepadKey(scanCode)) {

processKey(rawEvent->when, rawEvent->value != 0, scanCode, usageCode);

}

break;

}

case EV\_MSC: {

if (rawEvent->code == MSC\_SCAN) {

mCurrentHidUsage = rawEvent->value;

}

break;

}

case EV\_SYN: {

if (rawEvent->code == SYN\_REPORT) {

mCurrentHidUsage = 0;

}

}

}

}

void KeyboardInputMapper::processKey(nsecs\_t when, bool down, int32\_t scanCode,

int32\_t usageCode) {

int32\_t keyCode;

int32\_t keyMetaState;

uint32\_t policyFlags;

//对于驱动上报的 scanCode转换成安卓的keyCode

if (getEventHub()->mapKey(getDeviceId(), scanCode, usageCode, mMetaState,

&keyCode, &keyMetaState, &policyFlags)) {

keyCode = AKEYCODE\_UNKNOWN;

keyMetaState = mMetaState;

policyFlags = 0;

}

…

//假设映射成功了，通过keyCode、sacnCode还有metaState(shift等)、按下的时间来构造出一个args

NotifyKeyArgs args(when, getDeviceId(), mSource, policyFlags,

down ? AKEY\_EVENT\_ACTION\_DOWN : AKEY\_EVENT\_ACTION\_UP,

AKEY\_EVENT\_FLAG\_FROM\_SYSTEM, keyCode, scanCode, keyMetaState, downTime);

getListener()->notifyKey(&args);**//通知listener处理，猜测Dispatcher线程处理**

}

status\_t EventHub::mapKey(int32\_t deviceId,

int32\_t scanCode, int32\_t usageCode, int32\_t metaState,

int32\_t\* outKeycode, int32\_t\* outMetaState, uint32\_t\* outFlags) const {

AutoMutex \_l(mLock);

Device\* device = getDeviceLocked(deviceId);

status\_t status = NAME\_NOT\_FOUND;

if (device) {//先尝试用kcm文件转换

// Check the key character map first.

sp<KeyCharacterMap> kcm = device->getKeyCharacterMap();

if (kcm != NULL) {

if (!kcm->mapKey(scanCode, usageCode, outKeycode)) {

\*outFlags = 0;

status = NO\_ERROR;

}

}

// Check the key layout next.

if (status != NO\_ERROR && device->keyMap.haveKeyLayout()) {//再尝试用kl文件转换

if (!device->keyMap.keyLayoutMap->mapKey(

scanCode, usageCode, outKeycode, outFlags)) {

status = NO\_ERROR;

}

}

}

## 1.8 输入系统\_Dispatcher线程总体框架

输入系统深入分析5\_Dispatcher线程分析.jpg

## 1.9 输入系统\_Dispatcher线程情景分析值Reader线程传递事件

输入系统深入分析5\_Dispatcher线程分析.jpg

frameworks/base/services/core/jni/com\_android\_server\_input\_InputManagerService.cpp

static jlong nativeInit(JNIEnv\* env, jclass /\* clazz \*/,

jobject serviceObj, jobject contextObj, jobject messageQueueObj) {

NativeInputManager\* im = new NativeInputManager(contextObj, serviceObj,

}

NativeInputManager::NativeInputManager(jobject contextObj,

jobject serviceObj, const sp<Looper>& looper) :

mLooper(looper), mInteractive(true) {

mInputManager = new InputManager(eventHub, this, this);

}

frameworks/native/services/inputflinger/InputManager.cpp

InputManager::InputManager(

const sp<EventHubInterface>& eventHub,

const sp<InputReaderPolicyInterface>& readerPolicy,

const sp<InputDispatcherPolicyInterface>& dispatcherPolicy) {

mDispatcher = new InputDispatcher(dispatcherPolicy);

mReader = new InputReader(eventHub, readerPolicy, mDispatcher);

initialize();

}

frameworks/native/services/inputflinger/InputReader.cpp

InputReader::InputReader(const sp<EventHubInterface>& eventHub,

const sp<InputReaderPolicyInterface>& policy,

const sp<InputListenerInterface>& listener) :

mContext(this), mEventHub(eventHub), mPolicy(policy),

mGlobalMetaState(0), mGeneration(1),

mDisableVirtualKeysTimeout(LLONG\_MIN), mNextTimeout(LLONG\_MAX),

mConfigurationChangesToRefresh(0) {

mQueuedListener = new QueuedInputListener(listener);

{ // acquire lock

AutoMutex \_l(mLock);

refreshConfigurationLocked(0);

updateGlobalMetaStateLocked();

} // release lock

}

所以1.7节中getListener()->notifyKey(&args);中getListerner到是 InputDispatcher实例化对象，notifyKey 调用的是void InputDispatcher::notifyKey(const NotifyKeyArgs\* args)

一开始下面三个key处理入口是一样的，都是1.7节中的InputReader线程里getListener()->notifyKey(&args);

**global key:**

按下按键，启动某个APP

可以自己指定，修改frameworks\base\core\res\res\xml\global\_keys.xml

假设它是AKEYCODE\_TV

**system key:**

比如音量键 AKEYCODE\_VOLUME\_DOWN

**user key:**

其他按键，比如ABCD

AKEYCODE\_A

void InputDispatcher::notifyKey(const NotifyKeyArgs\* args) {//这个调用运行在InputReader线程里

…

KeyEvent event;

event.initialize(args->deviceId, args->source, args->action,

flags, keyCode, args->scanCode, metaState, 0,

args->downTime, args->eventTime);

android::base::Timer t;

mPolicy->interceptKeyBeforeQueueing(&event, /\*byref\*/ policyFlags);//放入队列之前稍加处理

//deviceId: 哪一个设备产生的输入事件，action: 按下还是松开，keyCode: Android的按键码

KeyEntry\* newEntry = new KeyEntry(args->eventTime,

args->deviceId, args->source, policyFlags, //interceptKeyBeforeQueueing 处理结果policyFlags 会放入 newEntry

args->action, flags, keyCode, args->scanCode,

metaState, repeatCount, args->downTime);

needWake = enqueueInboundEventLocked(newEntry);//放入mInboundQqueue队列

if (needWake) {

mLooper->wake();//有必要的话，唤醒Dispatcher线程，如果Dispatch线程运行的话就没必要唤醒它

}

}

### 1.9.1 interceptKeyBeforeQueueing

void NativeInputManager::interceptKeyBeforeQueueing(const KeyEvent\* keyEvent,

uint32\_t& policyFlags) {

ATRACE\_CALL();

bool interactive = mInteractive.load();

if (interactive) {

policyFlags |= POLICY\_FLAG\_INTERACTIVE;

}

if ((policyFlags & POLICY\_FLAG\_TRUSTED)) {

nsecs\_t when = keyEvent->getEventTime();

JNIEnv\* env = jniEnv();

jobject keyEventObj = android\_view\_KeyEvent\_fromNative(env, keyEvent);

jint wmActions;

if (keyEventObj) {

wmActions = env->CallIntMethod(mServiceObj,

gServiceClassInfo.interceptKeyBeforeQueueing, //调用java里面的同名函数 interceptKeyBeforeQueueing，PhoneWindowManager.java，返回结果放在wmActions

keyEventObj, policyFlags);

if (checkAndClearExceptionFromCallback(env, "interceptKeyBeforeQueueing")) {

wmActions = 0;

}

android\_view\_KeyEvent\_recycle(env, keyEventObj);

env->DeleteLocalRef(keyEventObj);

} else {

ALOGE("Failed to obtain key event object for interceptKeyBeforeQueueing.");

wmActions = 0;

}

handleInterceptActions(wmActions, when, /\*byref\*/ policyFlags); //根据来wmActions 设置 policyFlags

} else {

if (interactive) {

policyFlags |= POLICY\_FLAG\_PASS\_TO\_USER;

}

}

}

void NativeInputManager::handleInterceptActions(jint wmActions, nsecs\_t when,

uint32\_t& policyFlags) {

if (wmActions & WM\_ACTION\_PASS\_TO\_USER) {

policyFlags |= POLICY\_FLAG\_PASS\_TO\_USER;

} else {

#if DEBUG\_INPUT\_DISPATCHER\_POLICY

ALOGD("handleInterceptActions: Not passing key to user.");

#endif

}

}

frameworks/base/services/core/java/com/android/server/policy/PhoneWindowManager.java

public int interceptKeyBeforeQueueing(KeyEvent event, int policyFlags) {

final boolean interactive = (policyFlags & FLAG\_INTERACTIVE) != 0;//跟用户是否处于交互状态

if (interactive || (isInjected && !isWakeKey)) {

result = ACTION\_PASS\_TO\_USER;

}

// 如果按键是global key，直接返回result，ACTION\_PASS\_TO\_USER

if (isValidGlobalKey(keyCode)

&& mGlobalKeyManager.shouldHandleGlobalKey(keyCode, event)) {

if (isWakeKey) {

wakeUp(event.getEventTime(), mAllowTheaterModeWakeFromKey, "android.policy:KEY");

}

return result;

}

//对于system key分类处理，可以处理的话直接处理，不要PASS\_TO\_USER，否则传给下一步处理ACTION\_PASS\_TO\_USER，以音量按键为例

switch (keyCode) {

case KeyEvent.KEYCODE\_VOLUME\_DOWN:

case KeyEvent.KEYCODE\_VOLUME\_UP:

case KeyEvent.KEYCODE\_VOLUME\_MUTE:

interceptScreenshotChord();//是否需要截屏，该函数内部逻辑如果你同时按下电源键，就会截屏

if (telecomManager.isRinging()) {//如果按下音量键的同时，正有来电震铃

telecomManager.silenceRinger();//将来电震铃静音

result &= ~ACTION\_PASS\_TO\_USER;//按键不会传给用户

}

}

case KeyEvent.KEYCODE\_ENDCALL: {

result &= ~ACTION\_PASS\_TO\_USER;//挂机按键直接决定不传给用户，然后挂断电话之类的

}

...

return result;//对于普通按键直接ACTION\_PASS\_TO\_USER

}

frameworks/base/services/core/java/com/android/server/policy/GlobalKeyManager.java

boolean shouldHandleGlobalKey(int keyCode, KeyEvent event) {

return mKeyMapping.get(keyCode) != null;//依赖frameworks/base/core/res/res/xml/global\_keys.xml文件

}

global\_keys.xml，可以设置AKEYCODE\_TV发送广播给对应的组件

<global\_keys version="1">

<!-- Example format: keyCode = keycode to handle globally. component = component which will handle this key. -->

<!-- <key keyCode="KEYCODE\_VOLUME\_UP" component="com.android.example.keys/.VolumeKeyHandler" /> -->

</global\_keys>

### 1.9.2 enqueueInboundEventLocked

bool InputDispatcher::enqueueInboundEventLocked(EventEntry\* entry) {

bool needWake = mInboundQueue.isEmpty();

mInboundQueue.enqueueAtTail(entry);//放入队列的尾部

...

}