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 2](#__RefHeading___Toc16451_1060533589)

[0.2 socketpair 2](#__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 6](#__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 7](#__RefHeading___Toc1499_1698680629)

[1.6 输入系统\_Reader线程\_核心类及配置文件\_分析 8](#__RefHeading___Toc1501_1698680629)

[1.7 输入系统\_Reader线程\_简单处理 9](#__RefHeading___Toc1503_1698680629)

[1.8 输入系统\_Dispatcher线程总体框架 10](#__RefHeading___Toc1505_1698680629)

[1.9 输入系统\_Dispatcher线程情景分析\_Reader线程传递事件 10](#__RefHeading___Toc2043_1712734693)

[1.9.1 interceptKeyBeforeQueueing 11](#__RefHeading___Toc2045_1712734693)

[1.9.2 enqueueInboundEventLocked 12](#__RefHeading___Toc2047_1712734693)

[1.10 输入系统\_Dispatcher线程情景分析\_dispatch前处理 12](#__RefHeading___Toc2179_1584671019)

[1.11 输入系统\_实战\_使用GlobalKey一键启动程序 12](#__RefHeading___Toc2313_1214300884)

[1.12 输入系统\_APP跟输入系统建立联系\_InputChannel和Connection 13](#__RefHeading___Toc2783_2051444420)

[1.13 输入系统\_Dispatcher线程\_分发dispatch 13](#__RefHeading___Toc2785_2051444420)

[1.13.1 findFocusedWindowTargetsLocked 13](#__RefHeading___Toc2787_2051444420)

[1.13.2 dispatchEventLocked 13](#__RefHeading___Toc2789_2051444420)

[1.14 输入系统\_APP获得并处理输入事件流程 15](#__RefHeading___Toc2791_2051444420)

[1.15 输入系统\_补充知识\_activity\_window\_decor\_view关系\_理论 15](#__RefHeading___Toc2793_2051444420)

[1.16 输入系统\_补充知识\_activity\_window\_decor\_view关系\_实验 15](#__RefHeading___Toc2795_2051444420)

[1.17 输入系统\_InputStage\_理论 16](#__RefHeading___Toc3426_47849975)

[1.17.1输入法之前的处理 16](#__RefHeading___Toc3428_47849975)

[1.17.2输入法之后的处理 17](#__RefHeading___Toc3430_47849975)

[1.17.3小结 19](#__RefHeading___Toc3432_47849975)

[1.18 输入系统\_InputStage\_实验 19](#__RefHeading___Toc3740_1828780931)

[1.19 输入系统\_多点触摸\_电容屏驱动程序\_理论框架 20](#__RefHeading___Toc3811_1636444319)

# 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

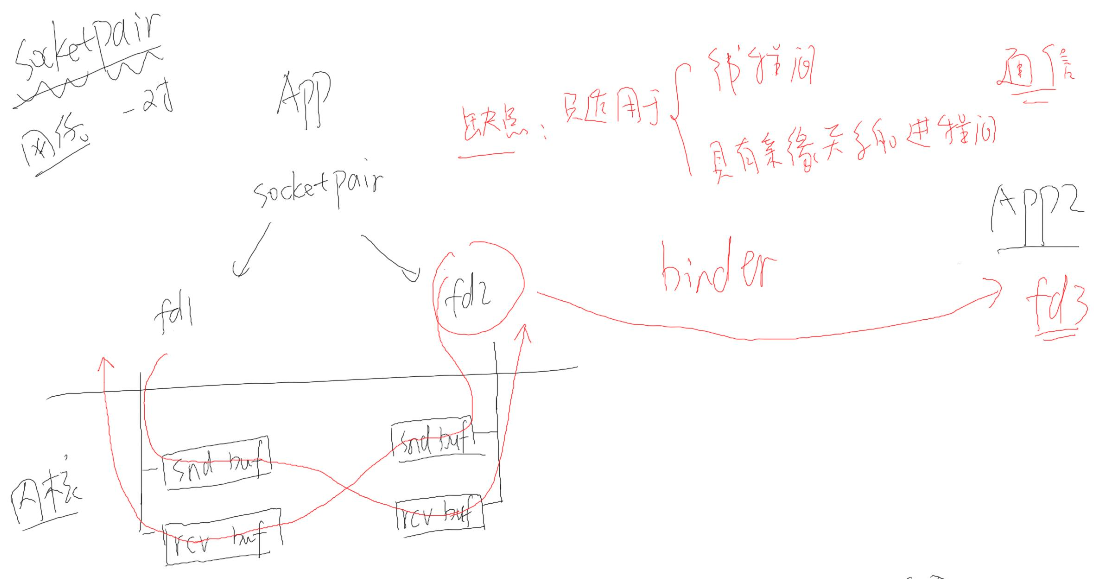
echo aaa > tmp/1

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

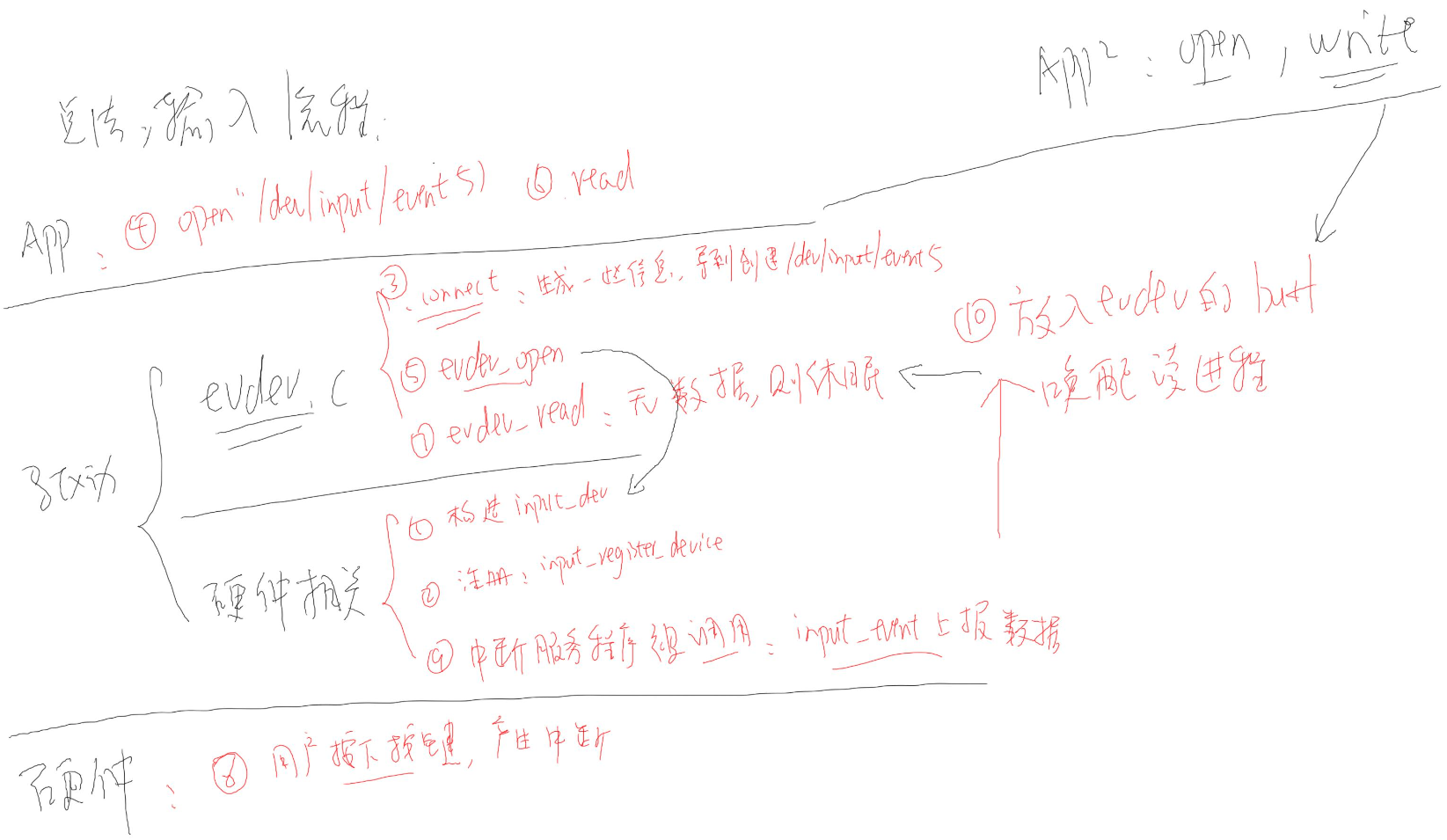
小节:

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

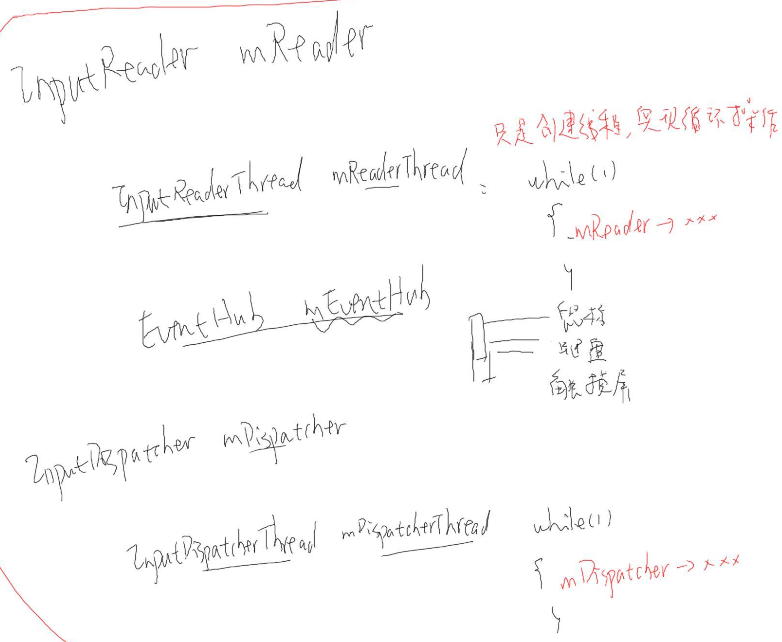
name: "InputEmulatorFrom100ask.net"

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

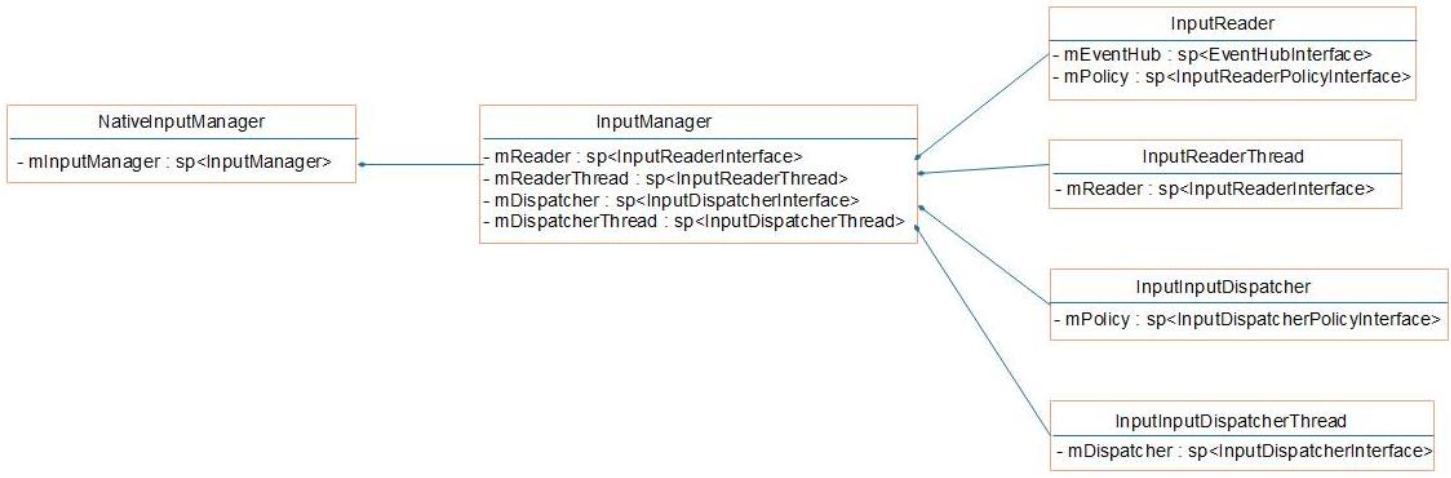
/dev/input/event1: EV\_KEY KEY\_1 UP

/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:

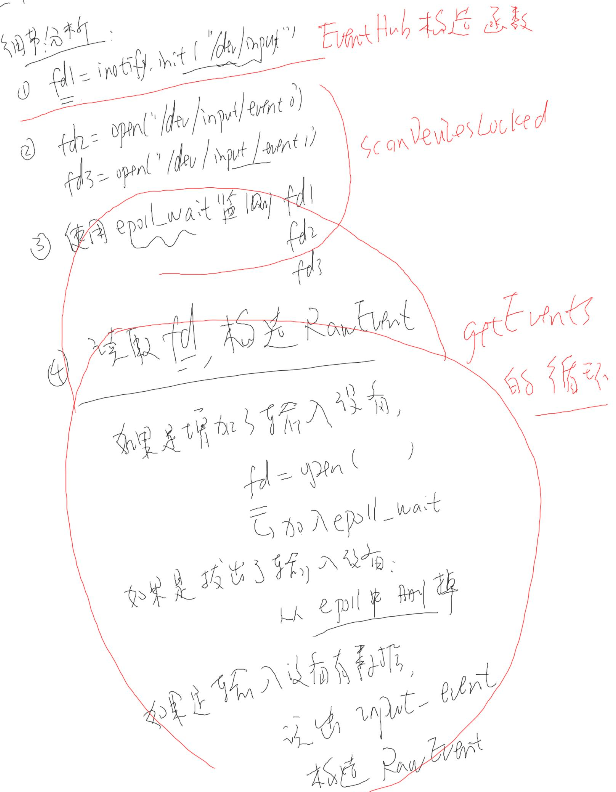
EV\_KEY

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

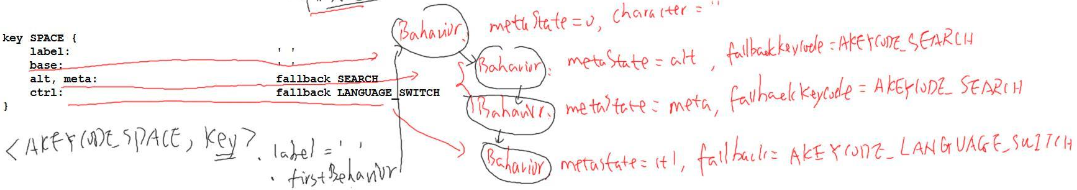
device.internal = 0 | 1

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\_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

...

}

## 1.10 输入系统\_Dispatcher线程情景分析\_dispatch前处理

1.9节提到的唤醒dispatcher线程

bool InputDispatcherThread::threadLoop() {

mDispatcher->dispatchOnce();

return true;

}

void InputDispatcher::dispatchOnce() {

nsecs\_t nextWakeupTime = LONG\_LONG\_MAX;

{ // acquire lock

AutoMutex \_l(mLock);

mDispatcherIsAliveCondition.broadcast();

if (!haveCommandsLocked()) {//如果没有命令的话，生成一个命令

dispatchOnceInnerLocked(&nextWakeupTime);

}

if (runCommandsLockedInterruptible()) {//运行命令

nextWakeupTime = LONG\_LONG\_MIN;

}

} // release lock

nsecs\_t currentTime = now();

int timeoutMillis = toMillisecondTimeoutDelay(currentTime, nextWakeupTime);

mLooper->pollOnce(timeoutMillis);//timeoutMillis 是LONG\_LONG\_MIN，表示不会立刻休眠，而是处理下一个命令

}

输入系统深入分析5\_Dispatcher线程分析.jpg，右下角的图片分析，最后的小结就是修订框图。

对于Reader线程读到的时间先进行解析，再过滤一下，没有过滤掉的，不管是否pass to user，都会放入mInBoundQueue队列中; 然后从这个队列中取出来，如果决定不pass to user的话，就直接release掉，否则就放入mCommandQueue中，再从mCommandQueue中取出来再次做解析，如果决定可以丢弃的话(比如global key、system key)，否则会放入mOutBoundQueue队列中，本节主要讲解的是第三步。

## 1.11 输入系统\_实战\_使用GlobalKey一键启动程序

**a. 对于global key, 系统会根据global\_keys.xml发送消息给某个组件**

修改 /work/android-5.0.2/frameworks/base/core/res/res/xml/global\_keys.xml

<key keyCode="KEYCODE\_TV" component="com.thisway.app\_0001\_leddemo/.MyBroadcastReceiver" />

编译:

mmm frameworks/base/core/res

它会生成 framework-res.apk, 复制到单板/system/framework/

**b. APP应该注册广播消息的接收者**

git clone https://github.com/weidongshan/APP\_0001\_LEDDemo.git

git checkout v9 //add BroadcastReceiver to start itself

b.1 编写BroadcastReceiver派生类, 实现消息处理函数

public class MyBroadcastReceiver extends BroadcastReceiver {

@Override

public void onReceive(Context context, Intent intent) {

Toast.makeText(context, "Get BroadcastReceiver", Toast.LENGTH\_SHORT).show();

Intent intentNewTask =new Intent(context, MainActivity.class);

intentNewTask.setFlags(Intent.FLAG\_ACTIVITY\_NEW\_TASK);

context.startActivity(intentNewTask);

}

}

b.2 注册派生类: 修改 AndroidManifest.xml

<receiver android:name=".MyBroadcastReceiver">

<intent-filter>

<action android:name="android.intent.action.GLOBAL\_BUTTON"/>

</intent-filter>

</receiver>

**c. 然后在该组件中启动app**

**实验:**

**a. 手工发广播**

am broadcast -a android.intent.action.GLOBAL\_BUTTON -n com.thisway.app\_0001\_leddemo/.MyBroadcastReceiver

**b. 用按键触发**

修改 /work/android-5.0.2/frameworks/base/core/res/res/xml/global\_keys.xml

添加:

<key keyCode="KEYCODE\_TV" component="com.thisway.app\_0001\_leddemo/.MyBroadcastReceiver" />

编译:

mmm frameworks/base/core/res

它会生成 framework-res.apk (out/target/product/tiny4412/system/framework/framework-res.apk)

修改驱动程序对应的kl文件(对于TV键不需要修改，scancode 377就对应TV键)

cp /system/usr/keylayout/Generic.kl /data/system/devices/keylayout/InputEmulatorFrom100ask\_net.kl

insmod InputEmulator.ko

模拟上报按键:

sendevent /dev/input/event5 1 377 1

sendevent /dev/input/event5 1 377 0

sendevent /dev/input/event5 0 0 0

也可以不使用驱动而使用以下命令模拟按键:

input keyevent TV

注意这里的TV指的是android的keycode

## 1.12 输入系统\_APP跟输入系统建立联系\_InputChannel和Connection

frameworks/base/services/core/java/com/android/server/wm/Session.java

输入系统深入分析6\_输入系统跟APP之间的联系.jpg

## 1.13 输入系统\_Dispatcher线程\_分发dispatch

输入系统深入分析7\_分发Dispatch.jpg

在前文我们分析过了Reader和Dispatcher线程针对global key和system key进行处理和过滤

bool InputDispatcher::dispatchKeyLocked(nsecs\_t currentTime, KeyEntry\* entry,

DropReason\* dropReason, nsecs\_t\* nextWakeupTime) {

…

int32\_t injectionResult = findFocusedWindowTargetsLocked(currentTime,

entry, inputTargets, nextWakeupTime);//查找目标窗口

…

dispatchEventLocked(currentTime, entry, inputTargets);//然后把它发送出去

}

### 1.13.1 findFocusedWindowTargetsLocked

int32\_t InputDispatcher::findFocusedWindowTargetsLocked(nsecs\_t currentTime,

const EventEntry\* entry, Vector<InputTarget>& inputTargets, nsecs\_t\* nextWakeupTime) {

if (mFocusedWindowHandle == NULL) {//在输入系统里面，它使用来表示目标窗口，这个变量是被WMS来设置的，本文不关心设置变量的这个过程，但是我们要知道它表示目标窗口

...权限检查等等

// Success! Output targets.

injectionResult = INPUT\_EVENT\_INJECTION\_SUCCEEDED;

addWindowTargetLocked(mFocusedWindowHandle,//使用mFocusedWindowHandle来构造出inputTarget然后加入vector里面，然后进行修改

InputTarget::FLAG\_FOREGROUND | InputTarget::FLAG\_DISPATCH\_AS\_IS, BitSet32(0),

inputTargets);

}

}

void InputDispatcher::addWindowTargetLocked(const sp<InputWindowHandle>& windowHandle,

int32\_t targetFlags, BitSet32 pointerIds, Vector<InputTarget>& inputTargets) {

inputTargets.push();

const InputWindowInfo\* windowInfo = windowHandle->getInfo();

InputTarget& target = inputTargets.editTop();

target.inputChannel = windowInfo->inputChannel;//核心在这里inputChannel

target.flags = targetFlags;

target.xOffset = - windowInfo->frameLeft;

target.yOffset = - windowInfo->frameTop;

target.scaleFactor = windowInfo->scaleFactor;

target.pointerIds = pointerIds;

}

### 1.13.2 dispatchEventLocked

void InputDispatcher::dispatchEventLocked(nsecs\_t currentTime,

EventEntry\* eventEntry, const Vector<InputTarget>& inputTargets) {

#if DEBUG\_DISPATCH\_CYCLE

ALOGD("dispatchEventToCurrentInputTargets");

#endif

ALOG\_ASSERT(eventEntry->dispatchInProgress); // should already have been set to true

pokeUserActivityLocked(eventEntry);

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

const InputTarget& inputTarget = inputTargets.itemAt(i);

ssize\_t connectionIndex = getConnectionIndexLocked(inputTarget.inputChannel);

if (connectionIndex >= 0) {// inputChannel有一个mFd，根据这个文件句柄在mConnectionsByFd 找到对应的connection

sp<Connection> connection = mConnectionsByFd.valueAt(connectionIndex);//把connection取出来

prepareDispatchCycleLocked(currentTime, connection, eventEntry, &inputTarget);//把输入事件放入connection的队列里面

} else {

#if DEBUG\_FOCUS

ALOGD("Dropping event delivery to target with channel '%s' because it "

"is no longer registered with the input dispatcher.",

inputTarget.inputChannel->getName().c\_str());

#endif

}

}

}

void InputDispatcher::prepareDispatchCycleLocked(nsecs\_t currentTime,

const sp<Connection>& connection, EventEntry\* eventEntry, const InputTarget\* inputTarget) {

...

// Not splitting. Enqueue dispatch entries for the event as is. 注意是复数，就是会分发多个Entry多个事件

enqueueDispatchEntriesLocked(currentTime, connection, eventEntry, inputTarget);

}

void InputDispatcher::enqueueDispatchEntriesLocked(nsecs\_t currentTime,

const sp<Connection>& connection, EventEntry\* eventEntry, const InputTarget\* inputTarget) {

bool wasEmpty = connection->outboundQueue.isEmpty();

// Enqueue dispatch entries for the requested modes.

enqueueDispatchEntryLocked(connection, eventEntry, inputTarget,

InputTarget::FLAG\_DISPATCH\_AS\_HOVER\_EXIT);

enqueueDispatchEntryLocked(connection, eventEntry, inputTarget,

InputTarget::FLAG\_DISPATCH\_AS\_OUTSIDE);

enqueueDispatchEntryLocked(connection, eventEntry, inputTarget,

InputTarget::FLAG\_DISPATCH\_AS\_HOVER\_ENTER);

enqueueDispatchEntryLocked(connection, eventEntry, inputTarget,

InputTarget::FLAG\_DISPATCH\_AS\_IS);

enqueueDispatchEntryLocked(connection, eventEntry, inputTarget,

InputTarget::FLAG\_DISPATCH\_AS\_SLIPPERY\_EXIT);

enqueueDispatchEntryLocked(connection, eventEntry, inputTarget,

InputTarget::FLAG\_DISPATCH\_AS\_SLIPPERY\_ENTER);

// If the outbound queue was previously empty, start the dispatch cycle going.

if (wasEmpty && !connection->outboundQueue.isEmpty()) {

startDispatchCycleLocked(currentTime, connection);//放入之后，把队列里的事件取出来写到文件句柄里面去

}

}

void InputDispatcher::enqueueDispatchEntryLocked(

const sp<Connection>& connection, EventEntry\* eventEntry, const InputTarget\* inputTarget,

int32\_t dispatchMode) {

int32\_t inputTargetFlags = inputTarget->flags;

if (!(inputTargetFlags & dispatchMode)) {//根据flag来决定是否把这个事件放到队列里面

return;

}

…

// Enqueue the dispatch entry.

connection->outboundQueue.enqueueAtTail(dispatchEntry);//从 connection得到这个队列，把 dispatchEntry放到这个队列outboundQueue 尾部，这个dispatchEntry就表示了某个输入事件

traceOutboundQueueLengthLocked(connection);

}

void InputDispatcher::startDispatchCycleLocked(nsecs\_t currentTime,

const sp<Connection>& connection) {

#if DEBUG\_DISPATCH\_CYCLE

ALOGD("channel '%s' ~ startDispatchCycle",

connection->getInputChannelName().c\_str());

#endif

while (connection->status == Connection::STATUS\_NORMAL

&& !connection->outboundQueue.isEmpty()) {

DispatchEntry\* dispatchEntry = connection->outboundQueue.head;//从connection 队列的头部取出一个dispatchEntry

case EventEntry::TYPE\_KEY://对于按键类事件的发送方法

status = connection->inputPublisher.publishKeyEvent(dispatchEntry->seq,

keyEntry->deviceId, keyEntry->source,

dispatchEntry->resolvedAction, dispatchEntry->resolvedFlags,

keyEntry->keyCode, keyEntry->scanCode,

keyEntry->metaState, keyEntry->repeatCount, keyEntry->downTime,

keyEntry->eventTime);

status\_t InputPublisher::publishKeyEvent(

uint32\_t seq,

int32\_t deviceId,

int32\_t source,

int32\_t action,

int32\_t flags,

int32\_t keyCode,

int32\_t scanCode,

int32\_t metaState,

int32\_t repeatCount,

nsecs\_t downTime,

nsecs\_t eventTime) {

…

InputMessage msg;//构造一个 InputMessage，使用sendMessage 发送

msg.header.type = InputMessage::TYPE\_KEY;

msg.body.key.seq = seq;

msg.body.key.deviceId = deviceId;

msg.body.key.source = source;

msg.body.key.action = action;

msg.body.key.flags = flags;

msg.body.key.keyCode = keyCode;

msg.body.key.scanCode = scanCode;

msg.body.key.metaState = metaState;

msg.body.key.repeatCount = repeatCount;

msg.body.key.downTime = downTime;

msg.body.key.eventTime = eventTime;

return mChannel->sendMessage(&msg); {

→ nWrite = ::send(mFd, msg, msgLength, MSG\_DONTWAIT | MSG\_NOSIGNAL);//把数据直接写到了文件句柄里，APP就可以从另外一个fd里面立刻读出来，下一步就是APP怎么处理这个 InputMessage

}

}

## 1.14 输入系统\_APP获得并处理输入事件流程

输入系统深入分析8\_APP获得及处理流程.jpg

## 1.15 输入系统\_补充知识\_activity\_window\_decor\_view关系\_理论

输入系统深入分析9\_Activity\_window\_decor\_view关系.jpg

android里:

1个application, 有1个或多个activity

1个activity, 有1个window

1个window, 有1个decor

1个decor, 有多个viewgroup/layout

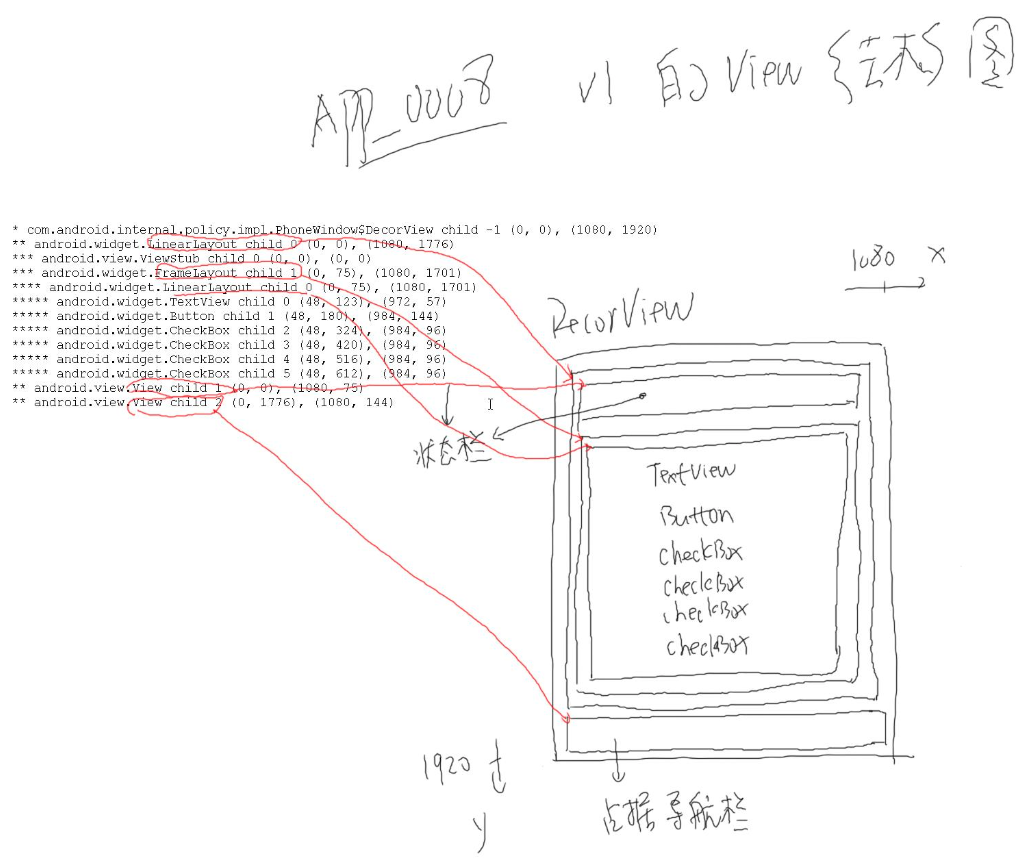
viewgroup/layout中, 有多个view

## 1.16 输入系统\_补充知识\_activity\_window\_decor\_view关系\_实验

git clone https://github.com/weidongshan/APP\_0008\_ViewHierarchy.git

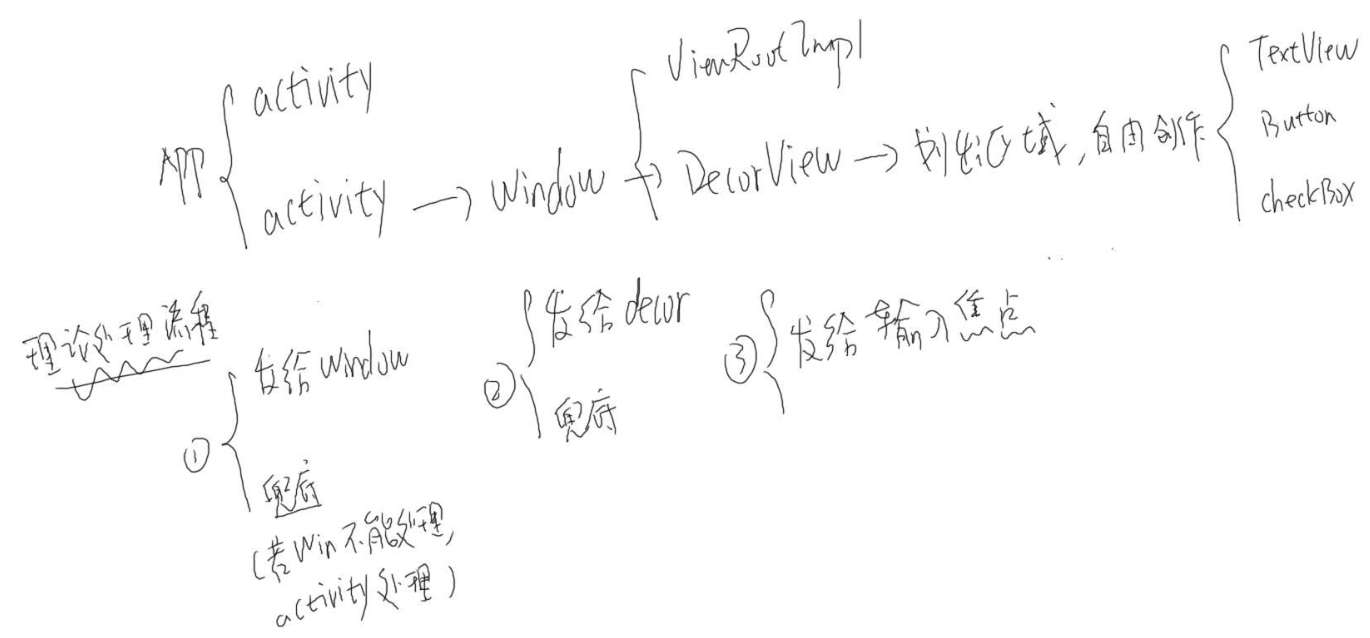
git pull origin

git checkout v1

起始坐标(x, y)，宽度高度(w, h)，状态栏、导航栏是system ui绘制的，我们可以通过api将decor的child 1和child 2隐藏起来，即状态栏和导航栏。

android:theme="@style/AppTheme" >，这是图片中提到的选择的decor样式，比如没有TitileBar+全屏

## 1.17 输入系统\_InputStage\_理论



### 1.17.1输入法之前的处理

frameworks/base/core/java/android/view/ViewRootImpl.java

final class ViewPreImeInputStage extends InputStage {

public ViewPreImeInputStage(InputStage next) {

super(next);

}

@Override

protected int onProcess(QueuedInputEvent q) {

if (q.mEvent instanceof KeyEvent) {//判断下输入事件是不是按键类事件

return processKeyEvent(q);

}

return FORWARD;

}

private int processKeyEvent(QueuedInputEvent q) {

final KeyEvent event = (KeyEvent)q.mEvent;

if (mView.dispatchKeyEventPreIme(event)) {//PreIme表示这个是输入法之前做的分发，mView就是之前说的DecorView，通过代码我们知道每个window还有一个ViewRootImpl

return FINISH\_HANDLED;

}

return FORWARD;

}

}

frameworks/base/core/java/com/android/internal/policy/DecorView.java

frameworks/base/core/java/android/widget/FrameLayout.java

frameworks/base/core/java/android/view/ViewGroup.java

@Override

public boolean dispatchKeyEventPreIme(KeyEvent event) {

if ((mPrivateFlags & (PFLAG\_FOCUSED | PFLAG\_HAS\_BOUNDS))

== (PFLAG\_FOCUSED | PFLAG\_HAS\_BOUNDS)) {

return super.dispatchKeyEventPreIme(event);

} else if (mFocused != null && (mFocused.mPrivateFlags & PFLAG\_HAS\_BOUNDS)

== PFLAG\_HAS\_BOUNDS) {

return mFocused.dispatchKeyEventPreIme(event);//把这个输入事件传给输入焦点mFocused，它也是一个控件

}

return false;

}

假设这个控件是 EditText

frameworks/base/core/java/android/widget/EditText.java

frameworks/base/core/java/android/widget/TextView.java

frameworks/base/core/java/android/view/View.java

public boolean dispatchKeyEventPreIme(KeyEvent event) {

return onKeyPreIme(event.getKeyCode(), event);//在输入法之前做按键的处理

}

public boolean onKeyPreIme(int keyCode, KeyEvent event) {//默认是什么处理都没有做，所以我们知道如果你APP想在输入法之前做一些处理，那么就要重写这个函数

return false;

}

### 1.17.2输入法之后的处理

ViewRootImpl.java

final class ViewPostImeInputStage extends InputStage {

private int processKeyEvent(QueuedInputEvent q) {

final KeyEvent event = (KeyEvent)q.mEvent;

if (mUnhandledKeyManager.preViewDispatch(event)) {

return FINISH\_HANDLED;

}

// Deliver the key to the view hierarchy.

if (mView.dispatchKeyEvent(event)) {

return FINISH\_HANDLED;

}

}

}

DecorView.java

@Override

public boolean dispatchKeyEvent(KeyEvent event) {

。。。

if (!mWindow.isDestroyed()) {

final Window.Callback cb = mWindow.getCallback();//获得一个回调，有 getCallback就会有 setCallback，在Activity.java里做的mWindow.setCallback(this)，这个cb得到的就是Acivity的实例化对象

final boolean handled = cb != null && mFeatureId < 0 ? cb.dispatchKeyEvent(event)

: super.dispatchKeyEvent(event);

if (handled) {

return true;

}

}

return isDown ? mWindow.onKeyDown(mFeatureId, event.getKeyCode(), event)**//如果转了一圈都不能处理的话，最终会调用到PhoneWindow的 onKeyDown、 onKeyUp**

: mWindow.onKeyUp(mFeatureId, event.getKeyCode(), event);

}

frameworks/base/core/java/android/app/Activity.java

public boolean dispatchKeyEvent(KeyEvent event) {

onUserInteraction();

。。。

Window win = getWindow();//获得window

if (win.superDispatchKeyEvent(event)) {

return true;

}

View decor = mDecor;

if (decor == null) decor = win.getDecorView();

return event.dispatch(this, decor != null //如果window不能处理的话，在调用这个函数

? decor.getKeyDispatcherState() : null, this);

}

frameworks/base/core/java/com/android/internal/policy/PhoneWindow.java

@Override

public boolean superDispatchKeyEvent(KeyEvent event) {

return mDecor.superDispatchKeyEvent(event);

}

DecorView.java

public boolean superDispatchKeyEvent(KeyEvent event) {

。。。

if (super.dispatchKeyEvent(event)) {

return true;

}

return (getViewRootImpl() != null) && getViewRootImpl().dispatchUnhandledKeyEvent(event);

}

ViewGroup.java

@Override

public boolean dispatchKeyEvent(KeyEvent event) {

if (mInputEventConsistencyVerifier != null) {

mInputEventConsistencyVerifier.onKeyEvent(event, 1);

}

if ((mPrivateFlags & (PFLAG\_FOCUSED | PFLAG\_HAS\_BOUNDS))

== (PFLAG\_FOCUSED | PFLAG\_HAS\_BOUNDS)) {

if (super.dispatchKeyEvent(event)) {

return true;

}

} else if (mFocused != null && (mFocused.mPrivateFlags & PFLAG\_HAS\_BOUNDS)

== PFLAG\_HAS\_BOUNDS) {

if (mFocused.dispatchKeyEvent(event)) {

return true;

}

}

if (mInputEventConsistencyVerifier != null) {

mInputEventConsistencyVerifier.onUnhandledEvent(event, 1);

}

return false;

}

我们仍然假设这个控件是 EditText

View.java

public boolean dispatchKeyEvent(KeyEvent event) {

if (mInputEventConsistencyVerifier != null) {

mInputEventConsistencyVerifier.onKeyEvent(event, 0);

}

// Give any attached key listener a first crack at the event.

//noinspection SimplifiableIfStatement

ListenerInfo li = mListenerInfo;//获得监听者的信息

if (li != null && li.mOnKeyListener != null && (mViewFlags & ENABLED\_MASK) == ENABLED

&& li.mOnKeyListener.onKey(this, event.getKeyCode(), event)) {//如果这个监听者里面有 mOnKeyListener的话，那么就调用onKey，取决于看控件是否设置了

return true;

}

if (event.dispatch(this, mAttachInfo != null //假如前面没有监听者，则调用这个

? mAttachInfo.mKeyDispatchState : null, this)) {

return true;

}

if (mInputEventConsistencyVerifier != null) {

mInputEventConsistencyVerifier.onUnhandledEvent(event, 0);

}

return false;

}

//如果你的控件调用过其函数，设置了这个监听者

public void setOnKeyListener(OnKeyListener l) {

getListenerInfo().mOnKeyListener = l;

}

frameworks/base/core/java/android/view/KeyEvent.java

public final boolean dispatch(Callback receiver, DispatcherState state,

Object target) {//做最终的处理

switch (mAction) {

case ACTION\_DOWN: {

mFlags &= ~FLAG\_START\_TRACKING;

if (DEBUG) Log.v(TAG, "Key down to " + target + " in " + state

+ ": " + this);

boolean res = receiver.onKeyDown(mKeyCode, this);

。。。

case ACTION\_UP:

if (DEBUG) Log.v(TAG, "Key up to " + target + " in " + state

+ ": " + this);

if (state != null) {

state.handleUpEvent(this);

}

return receiver.onKeyUp(mKeyCode, this);

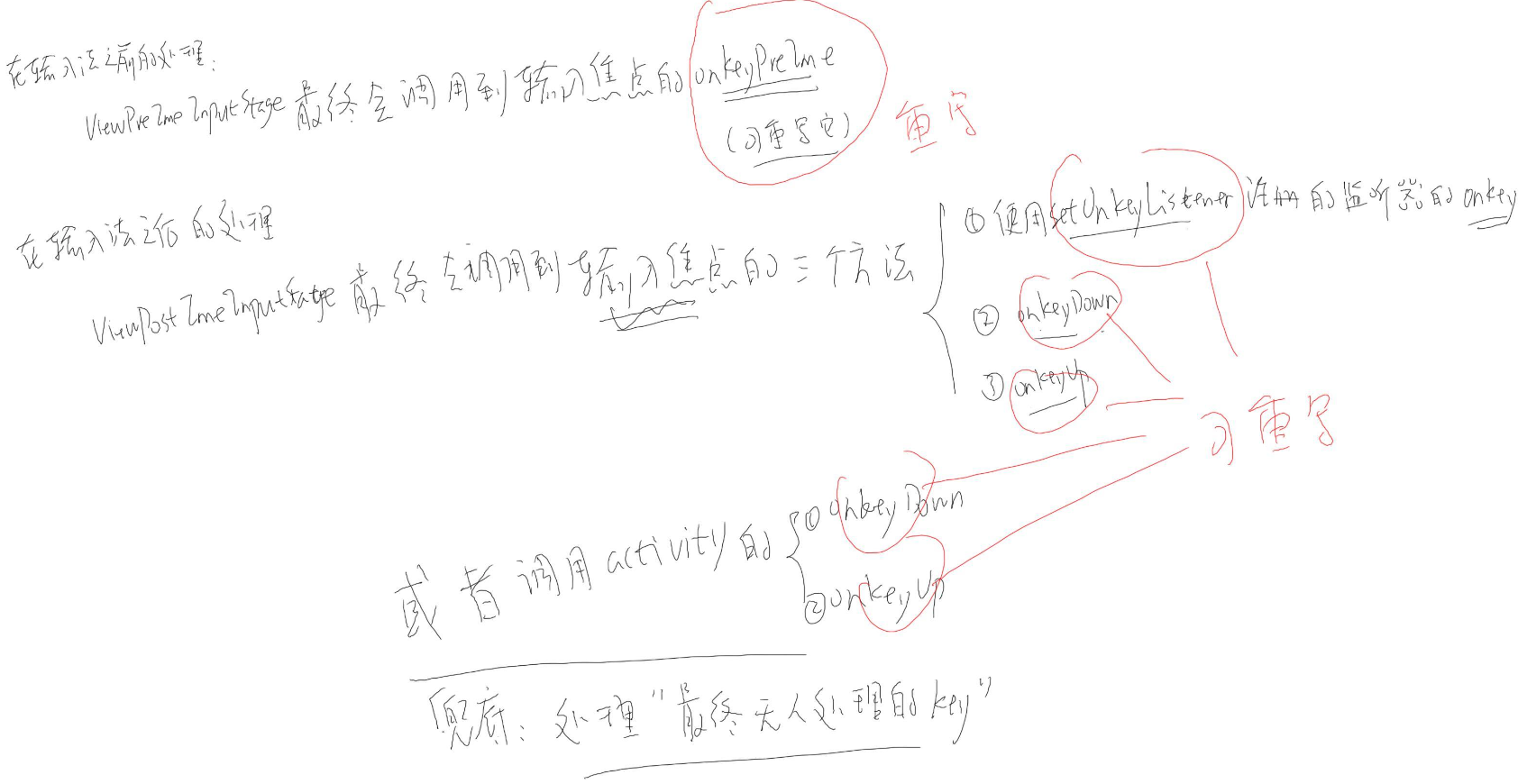
case ACTION\_MULTIPLE:

。。。

return false;

}

### 1.17.3小结

 输入系统深入分析10\_InputStage.jpg

## 1.18 输入系统\_InputStage\_实验

git clone https://github.com/weidongshan/APP\_0009\_InputStage.git

git pull origin

git checkout v1

目的: 验证上节的分析

**a. 在输入法之前添加自己的处理函数**

给某个控件重写onKeyPreIme

FramworkFromAndroidOin2018/Input/APP\_0009\_InputStage/app/src/main/java/com/thisway/app\_0001\_leddemo/MyEditText.java

public String getKeyStatus(KeyEvent event) {

if (event.getAction() == KeyEvent.ACTION\_DOWN)

return "down";

else

return "up";

}

@Override

public boolean onKeyPreIme(int keyCode, KeyEvent event) {//这里面可以改成你想要的功能

Log.d(TAG, "onKeyPreIme "+keyCode+" "+getKeyStatus(event));

//return true; //如果直接返回true的话，就标明在输入法之前的stage被处理(截取)了，就不会传给下一个stage，所以文本编辑框就不会显示出来

return super.onKeyPreIme(keyCode, event);

}

FramworkFromAndroidOin2018/Input/APP\_0009\_InputStage/app/src/main/res/layout/activity\_main.xml

<com.thisway.app\_0001\_leddemo.MyEditText

android:id="@+id/MYEDITTEXT1"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

/>

运行之后，输入字符a，log

onKeyPreIme 29 down

onKeyPreIme 29 up

**b. 在输入法之后添加自己的处理函数**

b.1 在显示字符之前添加处理函数

b.2 在显示字符之后添加处理函数

给某个控件注册: setOnKeyListener，或重写它的onKeyDown, onKeyUp函数

FramworkFromAndroidOin2018/Input/APP\_0009\_InputStage/app/src/main/java/com/thisway/app\_0001\_leddemo/MainActivity.java

View myEditText = (View) findViewById(R.id.MYEDITTEXT1);

myEditText.setOnKeyListener( new View.OnKeyListener() {

@Override

public boolean onKey(View v, int keyCode, KeyEvent event) {

Log.d(TAG, "OnKeyListener.onKey "+keyCode+" "+getKeyStatus(event));

return false;//返回false，标明没有处理这个按键，所以这个按键会继续传给后面的stage，如果返回true的话，那么也就不显示了

}

});

运行之后，输入字符a，log

onKeyPreIme 29 down

OnKeyListener.onKey 29 down

onKeyPreIme 29 up

OnKeyListener.onKey 29 up

**c. 添加善后处理函数(如果所有的View控件无法处理按键, 由Activity来处理)**

重写Activity的onKeyDown, onKeyUp函数

MyEditText.java

@Override

public boolean onKeyDown(int keyCode, KeyEvent event) {

Log.d(TAG, "MyEditText onKeyDown "+keyCode+" "+getKeyStatus(event));

return super.onKeyDown(keyCode, event);

//return super.onKeyDown(keyCode+1, createAnotherKeyEvent(event));//构造一个新的keyevent，按下a显示b

}

@Override

public boolean onKeyUp(int keyCode, KeyEvent event) {

Log.d(TAG, "MyEditText onKeyUp "+keyCode+" "+getKeyStatus(event));

return super.onKeyUp(keyCode, event);

}

MainActivity.java

@Override

public boolean onKeyDown(int keyCode, KeyEvent event) {

Log.d(TAG, "Activity onKeyDown "+keyCode+" "+getKeyStatus(event));

return super.onKeyDown(keyCode, event);

}

@Override

public boolean onKeyUp(int keyCode, KeyEvent event) {

Log.d(TAG, "Activity onKeyUp "+keyCode+" "+getKeyStatus(event));

return super.onKeyUp(keyCode, event);

}

运行之后，输入字符a，log

onKeyPreIme 29 down

OnKeyListener.onKey 29 down

MyEditText onKeyDown 29 down //MyEditText处理完了，所以没有 Activity的 onKeyDown

onKeyPreIme 29 up

OnKeyListener.onKey 29 up

MyEditText onKeyUp 29 up

Activity onKeyUp 29 up

下面的log是android P上的实验

2019-07-29 14:59:18.316 6279-6279/com.thisway.app\_0001\_leddemo D/LedDemo: onKeyPreIme 29 down

2019-07-29 14:59:18.325 6279-6279/com.thisway.app\_0001\_leddemo D/LedDemo: onKeyPreIme 29 up

2019-07-29 14:59:18.326 6279-6279/com.thisway.app\_0001\_leddemo D/LedDemo: OnKeyListener.onKey 29 up

2019-07-29 14:59:18.326 6279-6279/com.thisway.app\_0001\_leddemo D/LedDemo: MyEditText onKeyUp 29 up

2019-07-29 14:59:18.326 6279-6279/com.thisway.app\_0001\_leddemo D/LedDemo: Activity onKeyUp 29 up

导入别人的工程步骤

<https://blog.csdn.net/billy_chen_2013/article/details/81231083>

## 1.19 输入系统\_多点触摸\_电容屏驱动程序\_理论框架

在kernel里已经实现了读写等通用接口，我们只需要实现硬件操作相关的部分，比如有数据时给输入子系统上报数据。

kernel/msm-4.14/drivers/input/evdev.c

static const struct file\_operations evdev\_fops = {

.owner = THIS\_MODULE,

.read = evdev\_read,

.write = evdev\_write,

.poll = evdev\_poll,

.open = evdev\_open,

.release = evdev\_release,

.unlocked\_ioctl = evdev\_ioctl,

#ifdef CONFIG\_COMPAT

.compat\_ioctl = evdev\_ioctl\_compat,

#endif

.fasync = evdev\_fasync,

.flush = evdev\_flush,

.llseek = no\_llseek,

};

我们需要构造input\_device，设置，注册，有数据时调用input\_event函数上报数据，放入buffer中，交给evdev.c。

输入系统深入分析11\_多点触摸屏.jpg，来了解多点滑动时上报的数据，简单上报和复杂上报，其中复杂的上报需要上报ID，来区分哪些坐标属于一个手指。

## 1.20 输入系统\_多点触摸\_电容屏驱动程序\_编写框架

参考drivers\input\touchscreen\ft5x06\_ts.c

git clone https://github.com/weidongshan/DRV\_0005\_MultiTouchPanel.git

git pull origin

git checkout v1 // Demo driver for multi touch panel, it is only a Framework

git checkout v2 // There are some errors in v1

mtp\_input.c

static const unsigned short addr\_list[] = { MTP\_ADDR, I2C\_CLIENT\_END };

/\* 1. 分配/设置i2c\_driver \*/

static struct i2c\_driver mtp\_driver = {

.class = I2C\_CLASS\_HWMON, /\* 表示去哪些适配器上找设备 \*/

.driver = {

.name = "100ask",

.owner = THIS\_MODULE,

},

.probe = mtp\_probe,

.remove = \_\_devexit\_p(mtp\_remove),

.id\_table = mtp\_id\_table,

.detect = mtp\_detect, /\* 用这个函数来检测设备确实存在 \*/

.address\_list = addr\_list, /\* 这些设备的地址，addr\_list 表示它能够支持设备的地址，当我们注册上这个i2c driver的时候，它会去遍历所有的adapter，如果找到它能够支持的设备，那么就会调用mtp\_detect\*/

};

input\_set\_abs\_params(ts\_dev, ABS\_MT\_TRACKING\_ID, 0, 10, 0, 0);//最大支持十点触摸