# Input子系统与多点触摸技术

## 简介

Linux 输入子系统支持鼠标、键盘等,支持蜂鸣器、触摸屏等设备，又叫 input 子系统。

输入设备(如按键,键盘，触摸屏，鼠标等)是典型的字符设备，其一般的工作机制是低层在按键，触摸等动作发生时产生一个中断(或驱动通过timer定时查询)，然后cpu通过SPI，I2C或者外部存储器总线读取键值，坐标等数据，放一个缓冲区，字符设备驱动管理该缓冲区，而驱动的read()接口让用户可以读取键值，坐标等数据。

在Linux中，输入子系统是由输入子系统**设备驱动层**、输入子系统**核心层(Input Core)**和输入子系统**事件处理层(Event Handler)**组成。

[Android](http://lib.csdn.net/base/android)层操作input子系统由**Native层、**[Java](http://lib.csdn.net/base/javase)**框架层、应用程序**三部分组成。

其中**设备驱动层**提供对硬件各寄存器的读写访问和将底层硬件对用户输入访问的响应转换为标准的输入事件，再**通过核心层**提交给**事件处理层**；

而**核心层**对下提供了设备驱动层的编程接口，对上又提供了事件处理层的编程接口；

而**事件处理层**就为我们用户空间的应用程序提供了统一访问设备的接口和驱动层提交来的事件处理。

所以这使得我们输入设备的驱动部分不在用关心对设备文件的操作，而是要关心对各硬件寄存器的操作和提交的输入事件。

下面是输入子系统结构图：

## ****设备驱动层****

**输入子系统设备驱动层实现原理**

在Linux中，Input设备用input\_dev结构体描述，定义在input.h中。设备的驱动只需按照如下步骤就可实现了。

1).在驱动模块加载函数[probe]中设置Input设备支持input子系统的哪些事件；

2).将Input设备注册到input子系统中；

3).在Input设备发生输入操作时(如：键盘被按下/抬起、触摸屏被触摸/抬起/移动、鼠标被移动/单击/抬起时等)，提交所发生的事件及对应的键值/坐标等状态。

linux中输入设备驱动的分层：



## 换个角度看input子系统

 整体来看，Input子系统有一个主线，称为三方关系，input\_dev对应于实际的device端，input\_handler从名字也可以猜出来是对device的处理。“处理”这个词语不单单指的是对device数据的处理，比如report等；它其实可以包括系统在该device事件发生时想做的任何动作。至于input\_handle，它是连接input\_dev与input\_handler。

struct input\_handle {

    ...

struct input\_dev \*dev;//对应dev结构体

struct input\_handler \*handler;//对应handler结构体

    struct list\_head    d\_node;//链入input\_dev的h\_list代表的链表

    struct list\_head    h\_node;//链入input\_handler的h\_list代表的链表

};

至此，三方关系形成完毕。我们实现了最终的目的，通过input\_dev，可以遍历所有与它有关的input\_handler；通过input\_handler，也可以遍历所有与它有关的input\_dev。

图解如下：图中单向箭头表示指针，双向箭头表示list\_head。可以看出，从任何一个双向箭头出发，通过handle的过度，完全实现了我们的最终目标。掌握了这点，再看input\_report那些流程的时候就非常容易了，dev想要report数据的时候无非是调用了handler的event函数指针指向的函数，我们可以在这个函数里定义任何想让系统去做的任务，比如cpu调频等，而不仅限于数据上报。



## 软件设计流程

[](http://blog.chinaunix.net/link.php?url=http://blog.chinaunix.net/link.php?url%3Dhttp://blog.chinaunix.net/attachment/201108/1/23770712_1312213167mtdZ.gif)

图 input子系统软件设计流程

## 与软件设计有关的API函数

**分配一个输入设备**

Struct input\_dev \*input\_allocate\_device\*(void);

**注册一个输入设备**

Int input\_register\_device (struct input\_dev \*dev);

**驱动实现-事件支持**

set\_bit(int nr,long \* addr)：**将addr的第nr位，置为1**

set\_bit告诉in/out子系统它支持哪些事件

set\_bit (EV\_KEY, button\_dev.evbit)

**(其中button\_dev是struct input\_dev类型)**

struct input\_dev中有两个成员，一个是evbit;一个是keybit.分别用来表示设备所支持的事件类型和按键类型,有两种等效的方法:  
set\_bit(EV\_KEY, input\_dev->evbit); set\_bit(EV\_REL, input\_dev->evbit);  
input\_dev->evbit  = BIT\_MASK(EV\_KEY) | BIT\_MASK(EV\_REL);

**按键类型**

当事件类型为EV\_KEY时包括：

BTN\_LEFT,

BTN\_0,

BTN\_1,

BTN\_MIDDLE等

**事件类型**

<http://lxr.free-electrons.com/source/include/dt-bindings/input/linux-event-codes.h>[事件类型的定义]

Linux中输入设备的事件类型有(这里只列出了常用的一些，更多请看linux/input.h中)：

EV\_SYN 0x00 同步事件

EV\_KEY 0x01 按键事件

EV\_REL 0x02 相对坐标

EV\_ABS 0x03 绝对坐标

EV\_MSC 0x04 其它

EV\_LED 0x11 LED

EV\_SND 0x12 声音

EV\_REP 0x14 Repeat

EV\_FF 0x15 力反馈

EV\_PWR     0x16  电源管理事件

驱动实现-报告事件

Void input\_event(struct input\_dev \* dev, unsigned int type, unsigned int code, int value);//报告指定type ,code的输入事件

Void input\_report\_key(struct input\_dev \*dev, unsigned int code, int value);//报告键值

Void input\_report\_rel(struct input\_dev \*dev, unsigned int code, int value);//报告相对坐标

Void input\_report\_abs(struct input\_dev \*dev, unsigned int code, int value);//报告绝对坐标

Void input\_sync(struct input\_dev \*dev);//报告同步事件

**在触摸屏驱动设计中，一次坐标及按下状态的整个报告过程如下：**

**Input\_report\_abs(input\_dev,ABS\_X,x);//X坐标**

**Input\_report\_abs(input\_dev,ABS\_Y,y);//Y坐标**

**Input\_report\_abs(input\_dev,ABS\_PRESSURE,pres);//压力**

**input\_sync(struct input\_dev \*dev);//同步**

1.6.5释放与注销设备

Void input\_free\_device(struct input\_dev \*dev);

Void input\_unregister\_device(struct input\_dev \*);

**Ps :Input core是没有缓存队列的，如果应用程序没有及时取事件，则事件被丢弃。**

Input 驱动的子系统已经控制I/O，换句话read/write不需要驱动编写  
驱动只需要input\_report\_xxx()上传信息  
input\_report\_key()上传按键  
input\_report\_abs() 绝对坐标  
它们最终调用input\_event来向input core上传信息，并最后转交给应用程序.  
Input core没有缓存事件信息，这样在应用程序开始read前的信息全部被丢弃.

## 多点触摸技术A/B(Slot)协议

### A/B协议究竟是如何划分

B协议又称为slot协议，slot直译为位置、槽，有两层含义，一层是位置，另一层是容器。在Input子系统中，它扮演的就是这两个角色。它产生于这样一个背景：

如果从Device获取的当前数据与上一个数据相同，我们有必要再上报当前数据吗？如果我们不管两次数据是否一致都上报，那就是A协议；如果我们选择不上报，那么既然需要比较，总需要把上一次数据存起来吧，slot就是做这个事情的，显然这就是Slot(B)协议。

### A协议实现方式

A协议不会使用slot，多指处理中，它的报点序列如下（每一个序列都以input\_report\_\*\*\*函数实现）：

手指按下的动作：

**ABS\_MT\_POSITION\_X x[0]**

**ABS\_MT\_POSITION\_Y y[0]**

**SYN\_MT\_REPORT**

**ABS\_MT\_POSITION\_X x[1]**

**ABS\_MT\_POSITION\_Y y[1]**

**SYN\_MT\_REPORT**

**…**

**SYN\_REPORT**

上面的序列中需要说明的是系统以SYN\_MT\_REPORT为一个点的信息的结尾，以SYN\_REPORT为一次事件的结尾。也就是说多指触摸的时候，android的中间件部分每收到一次SYN\_MT\_REPORT就形成一个点信息，收到一个点之后并不会立即处理，而是一个事件完成之后才会处理，SYN\_REPORT就是这个事件的标志。A协议比较简单，我们也可以发现在上面的序列中根本就没有轨迹跟踪的信息，有的只是点坐标等信息

#### 系统如果去判断当前的多个点各属于哪一条线

我们假设前一次事件共有5个点，本次触摸也有5个点，系统会分别计算前一次5个点与本次5个点的距离，distance[prev\_i, curr\_j] (i=0,1,...,4; j=0,1,...4)，这样会产生总共5\*5=25个数字。然后对这25个数字进行排序，android用的是堆排序。（我们在系统上如果用多指，一般最多也是双值，也就是4个数据，这里采用了堆排序，不知是出于什么情况考虑，感觉换个方法可能更实用些。）下面的任务就是判断哪些当前点与前一次的点最近，那么赋予它们相同的id，应用收到这个信息后，就可以知道当前点属于哪条线了。

#### 手抬起来的时候又用什么样的序列来通知系统

SYN\_MT\_REPORT

SYN\_REPORT

只有SYNC，没有其它任何信息，系统就会认为此次事件为UP。

### B协议实现方式

B协议使用了slot，还有一个新面孔TRACKING\_ID.

手指按下的动作：

ABS\_MT\_SLOT 0

ABS\_MT\_TRACKING\_ID \*\*

ABS\_MT\_POSITION\_X x[0]

ABS\_MT\_POSITION\_Y y[0]

ABS\_MT\_SLOT 1

ABS\_MT\_TRACKING\_ID \*\*

ABS\_MT\_POSITION\_X x[1]

ABS\_MT\_POSITION\_Y y[1]

SYN\_REPORT

没有SYN\_MT\_REPORT，那么它用什么来跟踪当前点属于哪一条线呢，用的就是ABS\_MT\_TRACKING\_ID，当前序列中某点的ID值，如果与前一次序列中某点的ID值相等，那么他们就属于同一条线。既然如此，那么android系统中还需要做排序等运算吗？当然不需要。

#### 手指全部抬起的时候序列又是怎样的

ABS\_MT\_SLOT 0

ABS\_MT\_TRACKING\_ID -1

SYN\_REPORT

ABS\_MT\_SLOT 1

ABS\_MT\_TRACKING\_ID -1

SYN\_REPORT

这里上报的ABS\_MT\_TRACKING\_ID为-1，也只有这里该值才可以小于零，收到该值，系统就会清除对应的ID。看似简单的两个协议内容到这里就分析完毕了。

#### 附上android 4.0上报方式（linux必须2.6.38以上）

1. #include <linux/input/mt.h>
2. //down
3. input\_mt\_slot(ts->input\_dev, id);
4. //input\_report\_abs(ts->input\_dev, ABS\_MT\_TRACKING\_ID, id);
5. input\_mt\_report\_slot\_state(data->input\_dev, MT\_TOOL\_FINGER, true);
6. input\_report\_abs(ts->input\_dev, ABS\_MT\_TOUCH\_MAJOR, w);
7. input\_report\_abs(ts->input\_dev, ABS\_MT\_POSITION\_X, x);
8. input\_report\_abs(ts->input\_dev, ABS\_MT\_POSITION\_Y, y);

ps：个人感觉注释的那一行是在初次手指摁下才会执行，移动的时候就不执行这句话了  
//up

1. input\_mt\_slot(ts->input\_dev, id);
2. //input\_report\_abs(ts->input\_dev, ABS\_MT\_TRACKING\_ID, -1);
3. input\_mt\_report\_slot\_state(ts->input\_dev, MT\_TOOL\_FINGER, false);

ps：同理，注释的这句应该是在slot为0的时候不执行

1. //init
2. //\_\_set\_bit(INPUT\_PROP\_DIRECT, ts->input\_dev->propbit);
3. input\_mt\_init\_slots(ts->input\_dev, 255);

**上面两条ps是在本人看了内核文档介绍多触点以后的感觉，不一定对**

## 总结

看了上面的分析，明显可以看出B协议要优于A协议，但事实上并不如此简单。B协议需要硬件上的支持，ID值并不是随便赋值的，而是硬件上跟踪了点的轨迹；另外，B协议的复杂性如果掌握不好往往会带来一些莫名其妙的问题，比如如果因为某些因素（同步等），在UP的时候少清除了一个slot的信息，那么下次单击的时候你也会惊奇地发现竟然有两个点（采用了B协议，slot已经保存了点信息，除非明确清除）。

Ps：

事件报告完毕后，设备驱动需要使用input\_sync函数告诉输入子系统一个完整的报告已经发送。

void input\_sync(struct input\_dev \*dev)

{

input\_event(dev,EV\_SYN,SYN\_REPORT,0);

}

**这一点在鼠标移动处理中很重要，因为鼠标坐标的X分量和Y分量是分开传送的，需要利用input\_sync函数来同步。**



### **input子系统最终做了几件事情[内核input子系统]**

**Evdev.c文件**

**event代表数据包数据**

**上报事件最终调用函数input\_event();最终通过handler->event()调用**

**在input\_handler结构体中.event=evdev\_event：最后通过处理函数\_\_pass\_event做1，2两件事**

**1：事件最终被放入到了客户端的input\_event[]数组中；[**/\*将event装入client的buffer中，buffer是一个环形缓存区\*/ client->buffer[client->head++] = \***event**;**]**

这里的操作很简单.就是将event(上传数据)保存到client->buffer中.而client->head就是当前的数据位置.注意这里是一个环形缓存区.写数据是从client->head写.而读数据则是从client->tail中读.

**2：并且通过异步通知机制****kill\_fasync向应用层发送消息，应用层会执行对应的消息处理函数[**kill\_fasync(&client->fasync, SIGIO, POLL\_IN);**]**

  fasync这个东西就是为了使驱动的读写和application的读写分开，使得application可以在驱动读写时去做别的事[并且用来唤醒用户进程空间那个睡着的读进程（他是因为读的时候没有数据而睡着的。阻塞）]

&client->fasync可用来区分向谁发；哪个进程调用这个fasync\_helper就向谁发[谁调用flie\_operations里面的fasync]

**3：应用层调用open/read等函数，input子系统的file\_operations会被调用；操作写在Evdev.c文件里面**

**节点操作：open，read函数等等。节点是：/dev/input/eventX**

### 内核层Event流向

input子系统是驱动上报数据，暂存在内存中，应用程序需要时去读这片内存就可以取到数据，至于为什么采用这个模式，因为知道应用程序需要数据的时机和硬件采集到的数据是很难同步的，采用异步的方式更为合适。

### 用户层Event流向

一个事件的流程是从 Eventhub ---> InputReader ---> InputDispatcher ---> InputPublisher ---> InputChannel ---> InputConsumer ---> WindowManager or Application.这就是整个事件分发的大致流程

如果是第一次调用getEvents方法,则会调用scanDevicesLocked方法来扫描设备：

EventHub::getEvents--->EventHub::scanDevicesLocked--->EventHub::scanDirLocked---->EventHub::openDeviceLocked -> EventHub:: addDeviceLocked

### 1.1  从EventHub获取事件

**EvenHub，这个类的主要功能就是主动监视Input驱动变化，一旦有事件产生，就从产生事件相应的驱动中读取出这个事件**。是通过Linux提供的epoll机制来实现这个监视驱动功能。epoll机制简单地说就是高效地I/O多路复用机制，使用epoll\_wait来监听所需要的文件描述符的变化，关于epoll的介绍有很多文章，man中也有详细的介绍。EventHub的主要功能是通过epoll\_wait来实现的，所以EventHub所在的线程应该会阻塞在epoll\_wait方法中，一直等到epoll\_wait设置的超时时间。现在我们开始看看EventHub的实现，在EventHub的构造函数中，建立了一个管道，并把这个管道的读端和写端的文件描述符添加到epoll的监视之下，以便于其他的线程或者进程能够使EventHub所在的线程从epoll\_wait的阻塞中返回。EventHub在创建完成之后，第一个被调用的方法就是getEvents，而且这个方法也是EventHub的主要功能，对于这个方法需要仔细分析，**getEvents方法分成了三个部分去解析，分别是：打开设备部分；事件读取部分；等待部分**。这三个部分中，以事件的**读取部分为重点**。**设备打开部分一般发生在Input系统建立的时候调用**，所以在系统启动完成，稳定之后，这部分内容应该不会再被执行的；而等待部分较为简单。不过这些作为系统必不可少的部分，还是要一一说明的，先说设备打开部分吧，代码如下：

1 size\_t EventHub::getEvents(int timeoutMillis, RawEvent\* buffer, size\_t bufferSize) {

2 ...

3 struct input\_event readBuffer[bufferSize];

4 //这是元事件指针，可以指向一系列的事件，这些事件按照数组的方式存放的

5 RawEvent\* event = buffer;

6 size\_t capacity = bufferSize;

7 bool awoken = false;

8 for (;;) {

9 nsecs\_t now = systemTime(SYSTEM\_TIME\_MONOTONIC);

10 //mNeedToReopenDevices = false; mClosingDevices = 0;mNeedToSendFinishedDeviceScan = false;mOpeningDevices = 0

11 //mNeedToScanDevices = true

12 if (mNeedToScanDevices) {

13 mNeedToScanDevices = false;

14 scanDevicesLocked();

15 mNeedToSendFinishedDeviceScan = true;

16 }

17 　　　　...

scanDevicesLocked方法。在这个方法执行之后，肯定会产生一些设备添加，移除之类的事件

1. void EventHub::scanDevicesLocked() {
2. status\_t res = scanDirLocked(DEVICE\_PATH);
3. if(res **<** **0**) {
4. ALOGE("scan dir failed for %s\n", DEVICE\_PATH);
5. }
6. if (mDevices.indexOfKey(VIRTUAL\_KEYBOARD\_ID) **<** **0**) {
7. createVirtualKeyboardLocked();
8. }
9. }
10. static const char \*WAKE\_LOCK\_ID = "KeyEvents";
11. static const char \*DEVICE\_PATH = "/dev/input";

接着就是跑scanDirLocked

1. status\_t EventHub::scanDirLocked(const char \*dirname)
2. {
3. char devname[PATH\_MAX];
4. char \*filename;
5. DIR \*dir;
6. struct dirent \*de;
7. dir = opendir(dirname);
8. if(dir == NULL)
9. return -1;
10. strcpy(devname, dirname);
11. filename = devname + strlen(devname);
12. \*filename++ = '/';
13. while((de = readdir(dir))) {
14. if(de-**>**d\_name[0] == '.' &&
15. (de-**>**d\_name[1] == '\0' ||
16. (de-**>**d\_name[1] == '.' && de-**>**d\_name[2] == '\0')))
17. continue;
18. strcpy(filename, de-**>**d\_name);
19. openDeviceLocked(devname);
20. }
21. closedir(dir);
22. return 0;
23. }

scanDirLocked打开目录/dev/input 目录后,对目录下的每个文件都调用

struct RawEvent {

nsecs\_t when;//事件发生的时间，在getEvents中对于事件时间的处理也是值得关注的

int32\_t deviceId;//产生这个事件对应的设备的ID，与具体的硬件无关，其数值和设备打开的顺序有关

int32\_t type;//事件的类型

int32\_t code;//事件对应的事件码

int32\_t value;//事件的内容

};//运行命令getevent –tl可显示这些数据

RawEvent来自两种，一种是在打开设备时自己赋值，不如设备的添加，移除等，这些事件对应的RawEvent都是getEvents自己赋值的，便于InputReader处理；还有一种是来自驱动的产生的事件，由驱动产生的这类事件，在内容中有其自己的定义的类型，就是input\_event。 getEvents可以根据input\_event产生相应的RawEvent便于InputReader处理。

EventHub是通过扫描/dev/input/目录下所有可用的设备，然后逐一打开这些设备，打开这些设备过程中，EventHub又做了一些Input系统必要的工作，比如构造Device对象，把这些设备加入到epoll的监视队列中等，时间戳的设定等。在构造Device对象的时候，是通过InputDeviceIdentifier来构造的，主要思路就是通过ioctl函数从内容中读取出一些必要的信息，然后把这些信息经过InputDeviceIdentifier存入Device中，然后再通过ioctl函数测试设备的属性，把这些属性信息也存入Device中。代码如下：

1. status\_t EventHub::openDeviceLocked(**const** **char** \*devicePath) {
2. **char** buffer[80];
4. ALOGV("Opening device: %s", devicePath);
6. **int** fd = open(devicePath, O\_RDWR | O\_CLOEXEC);/\*open()系统调用返回文件描述符，O\_RDWR是指以读写方式打开，O\_CLOEXEC的作用是百度来的。Linux中，文件描述符有一个属性：CLOEXEC，即当调用exec()函数成功后，文件描述符会自动关闭。在以往的内核版本（2.6.23以前）中，需要调用 fcntl(fd, F\_SETFD, FD\_CLOEXEC)来设置这个属性。而新版本（2.6.23开始）中，可以在调用open函数的时候，通过 flags 参数设置CLOEXEC 功能，如open(filename, O\_CLOEXEC)。\*/
7. **if**(fd < 0) {
8. ALOGE("could not open %s, %s\n", devicePath, strerror(errno));
9. **return** -1;
10. }
12. InputDeviceIdentifier identifier;//input设备标识符
13. **struct** InputDeviceIdentifier {
14. **inline** InputDeviceIdentifier() :
15. bus(0), vendor(0), product(0), version(0) {
16. }
18. // Information provided by the kernel.
19. String8 name;
20. String8 location;
21. String8 uniqueId;//唯一的ID
22. uint16\_t bus;
23. uint16\_t vendor;
24. uint16\_t product;
25. uint16\_t version;
27. // A composite input device descriptor string that uniquely identifies the device
28. // even across reboots or reconnections.  The value of this field is used by
29. // upper layers of the input system to associate settings with individual devices.
30. // It is hashed from whatever kernel provided information is available.
31. // Ideally, the way this value is computed should not change between Android releases
32. // because that would invalidate persistent settings that rely on it.
33. String8 descriptor;
34. };
36. // Get device name.
37. **if**(ioctl(fd, EVIOCGNAME(**sizeof**(buffer) - 1), &buffer) < 1) {//input\_dev的name，”ft5x06”
38. //fprintf(stderr, "could not get device name for %s, %s\n", devicePath, strerror(errno));
39. } **else** {
40. buffer[**sizeof**(buffer) - 1] = '\0';
41. identifier.name.setTo(buffer);
42. }
44. // Check to see if the device is on our excluded list
45. //检查是不是要排除的device，一般都不是
46. **for** (**size\_t** i = 0; i < mExcludedDevices.size(); i++) {
47. **const** String8& item = mExcludedDevices.itemAt(i);
48. **if** (identifier.name == item) {
49. ALOGI("ignoring event id %s driver %s\n", devicePath, item.string());
50. close(fd);
51. **return** -1;
52. }
53. }
55. // Get device driver version.
56. **int** driverVersion;
57. **if**(ioctl(fd, EVIOCGVERSION, &driverVersion)) {// 得到EV\_VERSION 0x010001
58. ALOGE("could not get driver version for %s, %s\n", devicePath, strerror(errno));
59. close(fd);
60. **return** -1;
61. }
63. // Get device identifier.
64. **struct** input\_id inputId;
65. **if**(ioctl(fd, EVIOCGID, &inputId)) {//返回struct input\_id
66. ALOGE("could not get device input id for %s, %s\n", devicePath, strerror(errno));
67. close(fd);
68. **return** -1;
69. }
70. identifier.bus = inputId.bustype;
71. identifier.product = inputId.product;
72. identifier.vendor = inputId.vendor;
73. identifier.version = inputId.version;
75. // Get device physical location.
76. **if**(ioctl(fd, EVIOCGPHYS(**sizeof**(buffer) - 1), &buffer) < 1) {//物理位置，字符串
77. //fprintf(stderr, "could not get location for %s, %s\n", devicePath, strerror(errno));
78. } **else** {
79. buffer[**sizeof**(buffer) - 1] = '\0';
80. identifier.location.setTo(buffer);
81. }
83. // Get device unique id.
84. **if**(ioctl(fd, EVIOCGUNIQ(**sizeof**(buffer) - 1), &buffer) < 1) {//唯一ID，字符串
85. //fprintf(stderr, "could not get idstring for %s, %s\n", devicePath, strerror(errno));
86. } **else** {
87. buffer[**sizeof**(buffer) - 1] = '\0';
88. identifier.uniqueId.setTo(buffer);
89. }
91. // Fill in the descriptor.
92. setDescriptor(identifier);//设置identifier.descriptor
94. // Make file descriptor non-blocking for use with poll().
95. **if** (fcntl(fd, F\_SETFL, O\_NONBLOCK)) {//设置fd非阻塞
96. ALOGE("Error %d making device file descriptor non-blocking.", errno);
97. close(fd);
98. **return** -1;
99. }
101. // Allocate device.  (The device object takes ownership of the fd at this point.)
102. int32\_t deviceId = mNextDeviceId++;//deviceId=1，mNextDeviceId=2
103. Device\* device = **new** Device(fd, deviceId, String8(devicePath), identifier);//一个input device
105. ALOGV("add device %d: %s\n", deviceId, devicePath);
106. ALOGV("  bus:        %04x\n"
107. "  vendor      %04x\n"
108. "  product     %04x\n"
109. "  version     %04x\n",
110. identifier.bus, identifier.vendor, identifier.product, identifier.version);
111. ALOGV("  name:       \"%s\"\n", identifier.name.string());
112. ALOGV("  location:   \"%s\"\n", identifier.location.string());
113. ALOGV("  unique id:  \"%s\"\n", identifier.uniqueId.string());
114. ALOGV("  descriptor: \"%s\"\n", identifier.descriptor.string());
115. ALOGV("  driver:     v%d.%d.%d\n",
116. driverVersion >> 16, (driverVersion >> 8) & 0xff, driverVersion & 0xff);
118. // Load the configuration file for the device.
119. loadConfigurationLocked(device);
120. /\*获取输入设备配置文件，有几个路径，依次找，找到为止
121. // Figure out the kinds of events the device reports.
122. /\*使用EVIOCGBIT ioctl可以获取设备的能力和特性。它告知你设备是否有key或者button。EVIOCGBIT ioctl处理4个参数( ioctl(fd, EVIOCGBIT(ev\_type, max\_bytes), bitfield))。 ev\_type是返回的 type feature( 0是个特殊 case，表示返回设备支持的所有的 type features)。 max\_bytes表示返回的最大字节数。bitfield域是指向保存结果的内存指针。return value表示保存结果的实际字节数，如果调用失败，则返回负值。\*/
123. ioctl(fd, EVIOCGBIT(EV\_KEY, **sizeof**(device->keyBitmask)), device->keyBitmask);
124. ioctl(fd, EVIOCGBIT(EV\_ABS, **sizeof**(device->absBitmask)), device->absBitmask);
125. ioctl(fd, EVIOCGBIT(EV\_REL, **sizeof**(device->relBitmask)), device->relBitmask);
126. ioctl(fd, EVIOCGBIT(EV\_SW, **sizeof**(device->swBitmask)), device->swBitmask);
127. ioctl(fd, EVIOCGBIT(EV\_LED, **sizeof**(device->ledBitmask)), device->ledBitmask);
128. ioctl(fd, EVIOCGBIT(EV\_FF, **sizeof**(device->ffBitmask)), device->ffBitmask);
129. ioctl(fd, EVIOCGPROP(**sizeof**(device->propBitmask)), device->propBitmask);
131. // See if this is a keyboard.  Ignore everything in the button range except for
132. // joystick and gamepad buttons which are handled like keyboards for the most part.
133. /\*如果是键盘，除了操纵杆和大部分像键盘一样处理的游戏手柄按钮之外，可以忽略所以的按钮范围\*/
134. **bool** haveKeyboardKeys = containsNonZeroByte(device->keyBitmask, 0, sizeof\_bit\_array(BTN\_MISC))
135. || containsNonZeroByte(device->keyBitmask, sizeof\_bit\_array(KEY\_OK),
136. sizeof\_bit\_array(KEY\_MAX + 1));
137. /\*
138. haveKeyboardKeys：上报0~BTN\_MISC(100)-1或KEY\_OK(160)~KEY\_MAX(0x2ff)之间的type。
139. haveGamepadButtons：上报BTN\_MISC~BTN\_MOUSE(0x110)-1或BTN\_JOYSTICK(0x120)~BTN\_DIGI(0x140)的type。
140. \*/
141. **bool** haveGamepadButtons = containsNonZeroByte(device->keyBitmask, sizeof\_bit\_array(BTN\_MISC),
142. sizeof\_bit\_array(BTN\_MOUSE))
143. || containsNonZeroByte(device->keyBitmask, sizeof\_bit\_array(BTN\_JOYSTICK),
144. sizeof\_bit\_array(BTN\_DIGI));
145. **if** (haveKeyboardKeys || haveGamepadButtons) {
146. device->classes |= INPUT\_DEVICE\_CLASS\_KEYBOARD;/\*input外设是一个键盘或者按钮\*/
147. }
149. // See if this is a cursor device such as a trackball or mouse.
150. **if** (test\_bit(BTN\_MOUSE, device->keyBitmask)
151. && test\_bit(REL\_X, device->relBitmask)
152. && test\_bit(REL\_Y, device->relBitmask)) {
153. device->classes |= INPUT\_DEVICE\_CLASS\_CURSOR;//是一个光标，比如轨迹球或鼠标
154. }
156. // See if this is a touch pad.
157. // Is this a new modern multi-touch driver?
158. **if** (test\_bit(ABS\_MT\_POSITION\_X, device->absBitmask)
159. && test\_bit(ABS\_MT\_POSITION\_Y, device->absBitmask)) {
160. // Some joysticks such as the PS3 controller report axes that conflict
161. // with the ABS\_MT range.  Try to confirm that the device really is
162. // a touch screen.
163. **if** (test\_bit(BTN\_TOUCH, device->keyBitmask) || !haveGamepadButtons) {
164. device->classes |= INPUT\_DEVICE\_CLASS\_TOUCH | INPUT\_DEVICE\_CLASS\_TOUCH\_MT;
165. }
166. // Is this an old style single-touch driver?
167. } **else** **if** (test\_bit(BTN\_TOUCH, device->keyBitmask)
168. && test\_bit(ABS\_X, device->absBitmask)
169. && test\_bit(ABS\_Y, device->absBitmask)) {
170. device->classes |= INPUT\_DEVICE\_CLASS\_TOUCH;
171. }
172. /\*
173. (1) 如果是一个touch pad(不透明的触摸板)，还要看它是不是现代的多点触摸driver，所以要看一下有没有report ABS\_MT\_POSITION\_X和ABS\_MT\_POSITION\_Y；多点协议要求的。
174. (2) 如果是多点上报协议，还要看下是不是操纵杆。比如PS3控制器也会上报坐标轴，这与多点上报 ABS\_MT范围是冲突的，所以还要确认一下这个外设确实是触摸屏。怎么看呢？如果上报了BTN\_TOUCH那就是touch，如果没有上报BTN\_TOUCH，也不是游戏手柄按钮，那也是touch。device->classes |= INPUT\_DEVICE\_CLASS\_TOUCH | INPUT\_DEVICE\_CLASS\_TOUCH\_MT。
175. (3) 如果是老式的单点上报，device->classes |= INPUT\_DEVICE\_CLASS\_TOUCH。
176. \*/
177. // See if this device is a joystick.
178. // Assumes that joysticks always have gamepad buttons in order to distinguish them
179. // from other devices such as accelerometers that also have absolute axes.
180. //如果这个外设是一个操纵杆。假设它总是有游戏手柄按钮，为了与同样上报绝对坐标的其他外设，比如感应器区分开来，还需要设置INPUT\_DEVICE\_CLASS\_JOYSTICK。
181. **if** (haveGamepadButtons) {
182. uint32\_t assumedClasses = device->classes | INPUT\_DEVICE\_CLASS\_JOYSTICK;
183. **for** (**int** i = 0; i <= ABS\_MAX; i++) {
184. **if** (test\_bit(i, device->absBitmask)
185. && (getAbsAxisUsage(i, assumedClasses) & INPUT\_DEVICE\_CLASS\_JOYSTICK)) {
186. device->classes = assumedClasses;
187. **break**;
188. }
189. }
190. }
192. // Check whether this device has switches.开关
193. **for** (**int** i = 0; i <= SW\_MAX; i++) {
194. **if** (test\_bit(i, device->swBitmask)) {
195. device->classes |= INPUT\_DEVICE\_CLASS\_SWITCH;
196. **break**;
197. }
198. }
200. // Check whether this device supports the vibrator.振荡器
201. **if** (test\_bit(FF\_RUMBLE, device->ffBitmask)) {
202. device->classes |= INPUT\_DEVICE\_CLASS\_VIBRATOR;
203. }
205. // Configure virtual keys.
206. **if** ((device->classes & INPUT\_DEVICE\_CLASS\_TOUCH)) {
207. // Load the virtual keys for the touch screen, if any.
208. // We do this now so that we can make sure to load the keymap if necessary.
209. /\*如果有的话，为触摸屏load虚拟按键。我们现在这样做，所以可以确保load键映射，如果需要的话。一般虚拟按键都是利用触摸屏的边缘坐标模拟的按键。配置文件名就是/sys/board\_properties/virtualkeys.{devicename}，格式为：0x1:扫描码:X:Y:W:H:0x1: ……例如：
210. 0x01:158:55:835:90:55:0x01:139:172:835:125:55:0x01:102:298:835:115:55:0x01:217:412:835:95:55。如果定义了这个配置文件就可以自动把RawInputEvent(原始输入事件)转换为KeyEvent(按键事件)。base/core/java/android/view/inputDevice.java负责处理虚拟按键。要实现虚拟按键还可以在driver中用input\_event发送按键消息，往往是这种方式较为常用，尤其是需要校准的电阻屏。
211. 注意：使用虚拟按键转换成为的是按键的扫描码，不是按键码，因此依然需要经过按键布局文件的转化才能得到按键码。我们driver中所用的也是扫描码,例如：KEY\_MENU、KEY\_BACK。
212. \*/
213. status\_t status = loadVirtualKeyMapLocked(device);//load虚拟按键配置文件
214. **if** (!status) {
215. device->classes |= INPUT\_DEVICE\_CLASS\_KEYBOARD;//支持键盘
216. }
217. }
219. // Load the key map.
220. // We need to do this for joysticks too because the key layout may specify axes.
221. //Load按键映射，我们还需要为操纵杆做这个是因为键盘布局可能是一个指定轴。
222. status\_t keyMapStatus = NAME\_NOT\_FOUND;
223. **if** (device->classes & (INPUT\_DEVICE\_CLASS\_KEYBOARD | INPUT\_DEVICE\_CLASS\_JOYSTICK)) {
224. // Load the keymap for the device.先找\*.kl，再找\*.kcm，查找顺序同
225. keyMapStatus = loadKeyMapLocked(device);
226. }
228. // Configure the keyboard, gamepad or virtual keyboard.
229. **if** (device->classes & INPUT\_DEVICE\_CLASS\_KEYBOARD) {
230. // Register the keyboard as a built-in keyboard if it is eligible.
231. //如果有资格注册一个键盘作为嵌入键盘，什么是有资格，就是if的条件了
232. **if** (!keyMapStatus//上一节load keymap失败了
233. && mBuiltInKeyboardId == NO\_BUILT\_IN\_KEYBOARD(构造函数是这样初始化的)
234. && isEligibleBuiltInKeyboard(device->identifier,
235. device->configuration, &device->keyMap)) {
236. mBuiltInKeyboardId = device->id;
237. }
238. /\*isEligibleBuiltInKeyboard()成立的条件是：
239. (1) \*.kcm有，type不是SPECIAL\_FUNCTION。
240. (2) 如果idc文件中设置了keyboard.builtIn = true，那(1)+(2)条件成立。
241. (3) 如果input device的name中含有"-keypad"，那(1)+(3)条件也成立。
242. \*/
243. // 'Q' key support = cheap test of whether this is an alpha-capable kbd
244. //简单测试下是否有字母功能的键盘文本
245. **if** (hasKeycodeLocked(device, AKEYCODE\_Q)) {
246. device->classes |= INPUT\_DEVICE\_CLASS\_ALPHAKEY;
247. }
249. // See if this device has a DPAD.//D-Pad( directional pad)方向键
250. **if** (hasKeycodeLocked(device, AKEYCODE\_DPAD\_UP) &&
251. hasKeycodeLocked(device, AKEYCODE\_DPAD\_DOWN) &&
252. hasKeycodeLocked(device, AKEYCODE\_DPAD\_LEFT) &&
253. hasKeycodeLocked(device, AKEYCODE\_DPAD\_RIGHT) &&
254. hasKeycodeLocked(device, AKEYCODE\_DPAD\_CENTER)) {
255. device->classes |= INPUT\_DEVICE\_CLASS\_DPAD;
256. }
258. // See if this device has a gamepad.
259. **for** (**size\_t** i = 0; i < **sizeof**(GAMEPAD\_KEYCODES)/**sizeof**(GAMEPAD\_KEYCODES[0]); i++) {
260. **if** (hasKeycodeLocked(device, GAMEPAD\_KEYCODES[i])) {
261. device->classes |= INPUT\_DEVICE\_CLASS\_GAMEPAD;
262. **break**;
263. }
264. }
266. // Disable kernel key repeat since we handle it ourselves
267. //失能 kernel key repeat，因为我们除了它
268. unsigned **int** repeatRate[] = {0,0};
269. **if** (ioctl(fd, EVIOCSREP, repeatRate)) {
270. ALOGW("Unable to disable kernel key repeat for %s: %s", devicePath, strerror(errno));
271. }
272. }
274. // If the device isn't recognized as something we handle, don't monitor it.
275. //如果device没有被识别为我们可以处理的东西，就不要监视它了
276. **if** (device->classes == 0) {
277. ALOGV("Dropping device: id=%d, path='%s', name='%s'",
278. deviceId, devicePath, device->identifier.name.string());
279. **delete** device;
280. **return** -1;
281. }
283. // Determine whether the device is external or internal.
284. //确定是内部设备还是外部设备
285. **if** (isExternalDeviceLocked(device)) {
286. device->classes |= INPUT\_DEVICE\_CLASS\_EXTERNAL;
287. }
288. (1) 如果idc配置文件中，device.internal = **true**，就直接是内部设备了。
289. (2) 如果device.internal 没有写，要看input device的bus，如果是BUS\_USB或者BUS\_BLUETOOTH就是外部设备。
290. // Register with epoll.
291. **struct** epoll\_event eventItem;
292. memset(&eventItem, 0, **sizeof**(eventItem));
293. eventItem.events = EPOLLIN;
294. eventItem.data.u32 = deviceId;
295. **if** (epoll\_ctl(mEpollFd, EPOLL\_CTL\_ADD, fd, &eventItem)) {
296. ALOGE("Could not add device fd to epoll instance.  errno=%d", errno);
297. **delete** device;
298. **return** -1;
299. }
300. /\*又添加了一个epoll事件，这次是要查询/dev/input/eventx是否可读。
301. \*/
302. // Enable wake-lock behavior on kernels that support it.
303. // TODO: Only need this for devices that can really wake the system.
304. **bool** usingSuspendBlockIoctl;
305. **char** value[8];
306. property\_get("ro.platform.has.mbxuimode", value, "false");
307. **if**(strcmp(value, "true") == 0) {
308. usingSuspendBlockIoctl = !ioctl(fd, EVIOCSSUSPENDBLOCK, 0);//失能
309. } **else** {
310. usingSuspendBlockIoctl = !ioctl(fd, EVIOCSSUSPENDBLOCK, 1);//使能
311. }
312. /\*int property\_get(const char \*key, char \*value, const char \*default\_value);
313. 失能时，到kernel调用evdev\_disable\_suspend\_block()->
314. client->use\_wake\_lock = false;
315. wake\_lock\_destroy(&client->wake\_lock);
316. 使能时调用evdev\_enable\_suspend\_block()->
317. wake\_lock\_init(&client->wake\_lock, WAKE\_LOCK\_SUSPEND, client->name);
318. client->use\_wake\_lock = true;
319. if (client->packet\_head != client->tail)
320. 这时候是上锁，什么时候解锁呢？循环buffer首尾相接的时候。
321. if (unlikely(client->head == client->tail)) {
322. if (client->use\_wake\_lock)
323. wake\_unlock(&client->wake\_lock);
324. }
325. if (client->use\_wake\_lock &&
326. client->packet\_head == client->tail)
327. wake\_unlock(&client->wake\_lock);
328. \*/
329. // Tell the kernel that we want to use the monotonic clock for reporting timestamps
330. // associated with input events.  This is important because the input system
331. // uses the timestamps extensively and assumes they were recorded using the monotonic
332. // clock.
333. /\*通知kernel我们想用monotonic(单调递增)时钟作为input events的报告时间戳，这是非常重要的，假设input system用monotonic时钟记录时间戳，时间戳的应用非常广泛。
334. // In older kernel, before Linux 3.4, there was no way to tell the kernel which
335. // clock to use to input event timestamps.  The standard kernel behavior was to
336. // record a real time timestamp, which isn't what we want.  Android kernels therefore
337. // contained a patch to the evdev\_event() function in drivers/input/evdev.c to
338. // replace the call to do\_gettimeofday() with ktime\_get\_ts() to cause the monotonic
339. // clock to be used instead of the real time clock.
340. /\*在Linux 3.4之前的内核中，没有办法通知kernel用哪种时钟作为input系统的时间戳。标准内核行为是记录real(实时)时间的时间戳，这个时间并不是我们想要的。因此，android内核包含一个对drivers/input/evdev.c中evdev\_event()函数的patch，用ktime\_get\_ts()取代 do\_gettimeofday()，从而实现monotonic时钟代替real time时钟。
341. \*/
342. // As of Linux 3.4, there is a new EVIOCSCLOCKID ioctl to set the desired clock.
343. // Therefore, we no longer require the Android-specific kernel patch described above
344. // as long as we make sure to set select the monotonic clock.  We do that here.
345. /\*从Linux 3.4开始，出现了新的EVIOCSCLOCKID EVIOCSCLOCKID来设置期望的时钟。因此，我们不再需要android特殊的内核patch，综上所述，只有我们确定需要设置 monotonic clock，就执行下列代码。
346. \*/
347. **int** clockId = CLOCK\_MONOTONIC;
348. **bool** usingClockIoctl = !ioctl(fd, EVIOCSCLOCKID, &clockId);
349. ALOGI("New device: id=%d, fd=%d, path='%s', name='%s', classes=0x%x, "
350. "configuration='%s', keyLayout='%s', keyCharacterMap='%s', builtinKeyboard=%s, "
351. "usingSuspendBlockIoctl=%s, usingClockIoctl=%s",
352. deviceId, fd, devicePath, device->identifier.name.string(),
353. device->classes,
354. device->configurationFile.string(),
355. device->keyMap.keyLayoutFile.string(),
356. device->keyMap.keyCharacterMapFile.string(),
357. toString(mBuiltInKeyboardId == deviceId),
358. toString(usingSuspendBlockIoctl), toString(usingClockIoctl));
360. addDeviceLocked(device);
361. **return** 0;
362. }
363. **void** EventHub::addDeviceLocked(Device\* device) {
364. mDevices.add(device->id, device);
365. device->next = mOpeningDevices;
366. mOpeningDevices = device;
367. }
368. //KeyedVector.add()添加一个键值对，最后通过device->id就能找到device。
369. //通过mOpeningDevices可以找到我们第一open的设备，一直next下去，所以open的都找到了。

关于如何确认设备类型。在hal层里面是在eventhub.cpp 1160左右的ioctl赋值判断得到；最后是通过test\_bit[主要是判断这个设备支持哪些功能] 而这个东西的测试依据，是我们在driver里面写好的set\_bit;最终的判断结果放在device->classes里面,后面的时候对load mapper这件事也是依据的device->classes，整体上来说这个device->classes是非常重要的。

### InputReader

**InputReaderThread::threadLoop()**

**InputReader::loopOnce()—>InputReader::processEventsLocked🡪InputReader::addDeviceLocked🡪 InputReader::createDeviceLocked**

**InputReader::loopOnce()—>InputReader::processEventsLocked🡪InputReader::processEventsForDeviceLocked 🡪 Inputdevice::process**

1. **void** InputReader::loopOnce() {
2. ......
3. **size\_t** count = mEventHub->getEvents(timeoutMillis, mEventBuffer, EVENT\_BUFFER\_SIZE);  //从eventHub中获取事件
5. { // acquire lock
6. AutoMutex \_l(mLock);
7. mReaderIsAliveCondition.broadcast();
9. **if** (count) {
10. processEventsLocked(mEventBuffer, count);  //调用这个函数来处理
11. }

先来分析InputReaderThread，在一直调用InputReader的loopOnce函数；

**1先从EventHub中获取事件;**

**2然后调用processEventsLocked函数对事件进行处理：**

1. **void** InputReader::processEventsLocked(**const** RawEvent\* rawEvents, **size\_t** count) {
2. **for** (**const** RawEvent\* rawEvent = rawEvents; count;) {
3. int32\_t type = rawEvent->type;
4. **size\_t** batchSize = 1;
5. **if** (type < EventHubInterface::FIRST\_SYNTHETIC\_EVENT) {
6. int32\_t deviceId = rawEvent->deviceId;
7. **while** (batchSize < count) {
8. **if** (rawEvent[batchSize].type >= EventHubInterface::FIRST\_SYNTHETIC\_EVENT
9. || rawEvent[batchSize].deviceId != deviceId) {
10. **break**;
11. }
12. batchSize += 1;
13. }
14. #if DEBUG\_RAW\_EVENTS
15. ALOGD("BatchSize: %d Count: %d", batchSize, count);
16. #endif
17. processEventsForDeviceLocked(deviceId, rawEvent, batchSize);//设备事件
18. } **else** {
19. **switch** (rawEvent->type) {
20. **case** EventHubInterface::DEVICE\_ADDED://设备增加
21. addDeviceLocked(rawEvent->when, rawEvent->deviceId);
22. **break**;
23. **case** EventHubInterface::DEVICE\_REMOVED://设备移除
24. removeDeviceLocked(rawEvent->when, rawEvent->deviceId);
25. **break**;
26. **case** EventHubInterface::FINISHED\_DEVICE\_SCAN:
27. handleConfigurationChangedLocked(rawEvent->when);
28. **break**;
29. **default**:
30. ALOG\_ASSERT(**false**); // can't happen
31. **break**;
32. }
33. }
34. count -= batchSize;
35. rawEvent += batchSize;
36. }
37. }

会在mDevices中增加一个设备; 这里出现了一个mDevices变量，存储的是InputDevice类型，其实大家可以认为和前面EventHub中的Device是差不多的东西，都是代表一个输入设备。

注意:EventHub中添加的类型为Device, 这儿把Device转为了InputDevice ;即要筛选出非InputDevice,只将InputDevice添加到device列表中

1. **void** InputReader::addDeviceLocked(nsecs\_t when, int32\_t deviceId) {
2. ssize\_t deviceIndex = mDevices.indexOfKey(deviceId);
3. **if** (deviceIndex >= 0) {
4. ALOGW("Ignoring spurious device added event for deviceId %d.", deviceId);
5. **return**;
6. }
8. InputDeviceIdentifier identifier = mEventHub->getDeviceIdentifier(deviceId);
9. uint32\_t classes = mEventHub->getDeviceClasses(deviceId);
10. int32\_t controllerNumber = mEventHub->getDeviceControllerNumber(deviceId);
12. InputDevice\* device = createDeviceLocked(deviceId, controllerNumber, identifier, classes);//创建一个InputDevice
13. device->configure(when, &mConfig, 0);
14. device->reset(when);
16. **if** (device->isIgnored()) {
17. ALOGI("Device added: id=%d, name='%s' (ignored non-input device)", deviceId,
18. identifier.name.string());
19. } **else** {
20. ALOGI("Device added: id=%d, name='%s', sources=0x%08x", deviceId,
21. identifier.name.string(), device->getSources());
22. }
24. mDevices.add(deviceId, device);
25. bumpGenerationLocked();
27. **if** (device->getClasses() & INPUT\_DEVICE\_CLASS\_EXTERNAL\_STYLUS) {
28. notifyExternalStylusPresenceChanged();
29. }
30. }

那我们继续分析createDeviceLocked函数：

1. InputDevice\* InputReader::createDeviceLocked(int32\_t deviceId, int32\_t controllerNumber,
2. **const** InputDeviceIdentifier& identifier, uint32\_t classes) {
3. InputDevice\* device = **new** InputDevice(&mContext, deviceId, bumpGenerationLocked(),
4. controllerNumber, identifier, classes);
6. // External devices.
7. **if** (classes & INPUT\_DEVICE\_CLASS\_EXTERNAL) {
8. device->setExternal(**true**);
9. }
11. // Devices with mics.
12. **if** (classes & INPUT\_DEVICE\_CLASS\_MIC) {
13. device->setMic(**true**);
14. }
16. // Switch-like devices.
17. **if** (classes & INPUT\_DEVICE\_CLASS\_SWITCH) {
18. device->addMapper(**new** SwitchInputMapper(device));
19. }
21. // Vibrator-like devices.
22. **if** (classes & INPUT\_DEVICE\_CLASS\_VIBRATOR) {
23. device->addMapper(**new** VibratorInputMapper(device));
24. }
26. // Keyboard-like devices.
27. uint32\_t keyboardSource = 0;
28. int32\_t keyboardType = AINPUT\_KEYBOARD\_TYPE\_NON\_ALPHABETIC;
29. **if** (classes & INPUT\_DEVICE\_CLASS\_KEYBOARD) {
30. keyboardSource |= AINPUT\_SOURCE\_KEYBOARD;
31. }
32. **if** (classes & INPUT\_DEVICE\_CLASS\_ALPHAKEY) {
33. keyboardType = AINPUT\_KEYBOARD\_TYPE\_ALPHABETIC;
34. }
35. **if** (classes & INPUT\_DEVICE\_CLASS\_DPAD) {
36. keyboardSource |= AINPUT\_SOURCE\_DPAD;
37. }
38. **if** (classes & INPUT\_DEVICE\_CLASS\_GAMEPAD) {
39. keyboardSource |= AINPUT\_SOURCE\_GAMEPAD;
40. }
42. **if** (keyboardSource != 0) {
43. device->addMapper(**new** KeyboardInputMapper(device, keyboardSource, keyboardType));
44. }
46. // Cursor-like devices.
47. **if** (classes & INPUT\_DEVICE\_CLASS\_CURSOR) {
48. device->addMapper(**new** CursorInputMapper(device));
49. }
51. // Touchscreens and touchpad devices.
52. **if** (classes & INPUT\_DEVICE\_CLASS\_TOUCH\_MT) {
53. device->addMapper(**new** MultiTouchInputMapper(device));
54. } **else** **if** (classes & INPUT\_DEVICE\_CLASS\_TOUCH) {
55. device->addMapper(**new** SingleTouchInputMapper(device));
56. }
58. // Joystick-like devices.
59. **if** (classes & INPUT\_DEVICE\_CLASS\_JOYSTICK) {
60. device->addMapper(**new** JoystickInputMapper(device));
61. }
63. // External stylus-like devices.
64. **if** (classes & INPUT\_DEVICE\_CLASS\_EXTERNAL\_STYLUS) {
65. device->addMapper(**new** ExternalStylusInputMapper(device));
66. }
68. **return** device;
69. }

这个函数又会根据classes不同，增加各种Mapper。

//添加设备的mapper,即具体设备的处理函数

//例如tp的设备,需要用到MultiTouchInputMapper

当我们在InputReader增加设备后，再来分析下processEventsForDeviceLocked函数，处理设备事件的：

1. **void** InputReader::processEventsForDeviceLocked(int32\_t deviceId,
2. **const** RawEvent\* rawEvents, **size\_t** count) {
3. ssize\_t deviceIndex = mDevices.indexOfKey(deviceId);
4. **if** (deviceIndex < 0) {
5. ALOGW("Discarding event for unknown deviceId %d.", deviceId);
6. **return**;
7. }
9. InputDevice\* device = mDevices.valueAt(deviceIndex);
10. **if** (device->isIgnored()) {
11. //ALOGD("Discarding event for ignored deviceId %d.", deviceId);
12. **return**;
13. }
15. device->process(rawEvents, count);
16. }

这个函数是获取相对应的device执行process函数：

1. **void** InputDevice::process(**const** RawEvent\* rawEvents, **size\_t** count) {
2. // Process all of the events in order for each mapper.
3. // We cannot simply ask each mapper to process them in bulk because mappers may
4. // have side-effects that must be interleaved.  For example, joystick movement events and
5. // gamepad button presses are handled by different mappers but they should be dispatched
6. // in the order received.
7. **size\_t** numMappers = mMappers.size();
8. **for** (**const** RawEvent\* rawEvent = rawEvents; count--; rawEvent++) {
10. **if** (mDropUntilNextSync) {
11. **if** (rawEvent->type == EV\_SYN && rawEvent->code == SYN\_REPORT) {
12. mDropUntilNextSync = **false**;
13. } **else** {
14. }
15. } **else** **if** (rawEvent->type == EV\_SYN && rawEvent->code == SYN\_DROPPED) {
16. mDropUntilNextSync = **true**;
17. reset(rawEvent->when);
18. } **else** {
19. **for** (**size\_t** i = 0; i < numMappers; i++) {
20. InputMapper\* mapper = mMappers[i];
21. mapper->process(rawEvent);
22. }
23. }
24. }
25. }

最后遍历各个Event，然后遍历每个Device的InputMapper去执行其process函数。

### KeyboardInputMapper

Input设备有很多种类，其消息格式各不相同，因此就有很多InputMapper对各个不同的RawEvent进行处理。

我们今天主要从按键的KeyboardInputMapper来讲解。

下面我们主要从按键，比如音量键、power键的这个InputMapper说起：

1. **void** KeyboardInputMapper::process(**const** RawEvent\* rawEvent) {
2. **switch** (rawEvent->type) {
4. **case** EV\_KEY: {
5. int32\_t scanCode = rawEvent->code;
6. int32\_t usageCode = mCurrentHidUsage;
7. mCurrentHidUsage = 0;
9. **if** (isKeyboardOrGamepadKey(scanCode)) {
10. int32\_t keyCode;
11. uint32\_t flags;
12. **if** (getEventHub()->mapKey(getDeviceId(), scanCode, usageCode, &keyCode, &flags)) {//扫描码对应成按键码
13. keyCode = AKEYCODE\_UNKNOWN;
14. flags = 0;
15. }
16. processKey(rawEvent->when, rawEvent->value != 0, keyCode, scanCode, flags);
17. }
18. **break**;
19. }
20. **case** EV\_MSC: {
21. **if** (rawEvent->code == MSC\_SCAN) {
22. mCurrentHidUsage = rawEvent->value;
23. }
24. **break**;
25. }
26. **case** EV\_SYN: {
27. **if** (rawEvent->code == SYN\_REPORT) {
28. mCurrentHidUsage = 0;
29. }
30. }
31. }
32. }

我们先来看下EventHub的mapKey函数，这个函数中outKeycode是入参，但是传的是指针，会往里面写值。

1. status\_t EventHub::mapKey(int32\_t deviceId, int32\_t scanCode, int32\_t usageCode,
2. int32\_t\* outKeycode, uint32\_t\* outFlags) **const** {
3. AutoMutex \_l(mLock);
4. Device\* device = getDeviceLocked(deviceId);
6. **if** (device) {
7. // Check the key character map first.
8. sp<KeyCharacterMap> kcm = device->getKeyCharacterMap();
9. **if** (kcm != NULL) {
10. **if** (!kcm->mapKey(scanCode, usageCode, outKeycode)) {
11. scanCode, \*outKeycode);
12. \*outFlags = 0;
13. **return** NO\_ERROR;
14. }
15. }
17. // Check the key layout next.
18. **if** (device->keyMap.haveKeyLayout()) {
19. **if** (!device->keyMap.keyLayoutMap->mapKey(
20. scanCode, usageCode, outKeycode, outFlags)) {
21. **return** NO\_ERROR;
22. }
23. }
24. }
26. \*outKeycode = 0;
27. \*outFlags = 0;
28. **return** NAME\_NOT\_FOUND;
29. }
30. //扫描码对应成按键码,这个函数的作用

**KeyboardInputMapper的processKey函数**

1. **void** KeyboardInputMapper::processKey(nsecs\_t when, **bool** down, int32\_t keyCode,
2. int32\_t scanCode, uint32\_t policyFlags) {
4. **if** (down) {
5. // Rotate key codes according to orientation if needed.
6. **if** (mParameters.orientationAware && mParameters.hasAssociatedDisplay) {
7. keyCode = rotateKeyCode(keyCode, mOrientation);
8. }
10. // Add key down.
11. ssize\_t keyDownIndex = findKeyDown(scanCode);
12. **if** (keyDownIndex >= 0) {
13. // key repeat, be sure to use same keycode as before in case of rotation
14. keyCode = mKeyDowns.itemAt(keyDownIndex).keyCode;
15. } **else** {
16. // key down
17. **if** ((policyFlags & POLICY\_FLAG\_VIRTUAL)
18. && mContext->shouldDropVirtualKey(when,
19. getDevice(), keyCode, scanCode)) {
20. **return**;
21. }
22. **if** (policyFlags & POLICY\_FLAG\_GESTURE) {
23. mDevice->cancelTouch(when);
24. }
26. mKeyDowns.push();
27. KeyDown& keyDown = mKeyDowns.editTop();
28. keyDown.keyCode = keyCode;
29. keyDown.scanCode = scanCode;
30. }
32. mDownTime = when;
33. } **else** {
34. // Remove key down.
35. ssize\_t keyDownIndex = findKeyDown(scanCode);
36. **if** (keyDownIndex >= 0) {
37. // key up, be sure to use same keycode as before in case of rotation
38. keyCode = mKeyDowns.itemAt(keyDownIndex).keyCode;
39. mKeyDowns.removeAt(**size\_t**(keyDownIndex));
40. } **else** {
41. // key was not actually down
42. ALOGI("Dropping key up from device %s because the key was not down.  "
43. "keyCode=%d, scanCode=%d",
44. getDeviceName().string(), keyCode, scanCode);
45. **return**;
46. }
47. }
49. int32\_t oldMetaState = mMetaState;
50. int32\_t newMetaState = updateMetaState(keyCode, down, oldMetaState);
51. **bool** metaStateChanged = oldMetaState != newMetaState;
52. **if** (metaStateChanged) {
53. mMetaState = newMetaState;
54. updateLedState(**false**);
55. }
57. nsecs\_t downTime = mDownTime;
59. // Key down on external an keyboard should wake the device.
60. // We don't do this for internal keyboards to prevent them from waking up in your pocket.
61. // For internal keyboards, the key layout file should specify the policy flags for
62. // each wake key individually.
63. // TODO: Use the input device configuration to control this behavior more finely.
64. **if** (down && getDevice()->isExternal()) {
65. policyFlags |= POLICY\_FLAG\_WAKE;
66. }
68. **if** (mParameters.handlesKeyRepeat) {
69. policyFlags |= POLICY\_FLAG\_DISABLE\_KEY\_REPEAT;
70. }
72. **if** (metaStateChanged) {
73. getContext()->updateGlobalMetaState();
74. }
76. **if** (down && !isMetaKey(keyCode)) {
77. getContext()->fadePointer();
78. }
80. NotifyKeyArgs args(when, getDeviceId(), mSource, policyFlags,
81. down ? AKEY\_EVENT\_ACTION\_DOWN : AKEY\_EVENT\_ACTION\_UP,
82. AKEY\_EVENT\_FLAG\_FROM\_SYSTEM, keyCode, scanCode, newMetaState, downTime);
83. getListener()->notifyKey(&args);
84. }

**这个函数，一开始还是再把扫描码转成键盘码，后面主要调用了notifyKey函数。**

**我们再回到InputReader的loopOnce函数，我们主要看最后mQueuedListener->flush函数。**

**QueuedInputListener::flush函数在文件InputListener.cpp中。**

1. **void** QueuedInputListener::flush() {
2. **size\_t** count = mArgsQueue.size();
3. **for** (**size\_t** i = 0; i < count; i++) {
4. NotifyArgs\* args = mArgsQueue[i];
5. args->notify(mInnerListener);
6. **delete** args;
7. }
8. mArgsQueue.clear();
9. }

**这个函数中遍历之前我们在每个InputMapper存入的NotifyArgs对象，最后调用了NotifyArgs对象的notify函数**

**之前我们的NotifyArgs对象是NotifyKeyArgs对象，这个notify是个虚函数，就到NotifyKeyArgs::notify**

1. **void** NotifyKeyArgs::notify(**const** sp<InputListenerInterface>& listener) **const** {
2. listener->notifyKey(**this**);
3. }

**而这个listener，是新建QueuedInputListener的时候传进来的**

1. mQueuedListener = **new** QueuedInputListener(listener);

**listener是InputReader里面传过来的，所以我们知道InputReader是在InputManager中创建的**

1. InputReader::InputReader(**const** sp<EventHubInterface>& eventHub,
2. **const** sp<InputReaderPolicyInterface>& policy,
3. **const** sp<InputListenerInterface>& listener) :

**我们看下InputManager的构造函数，传进来的InputDispatcher**

1. InputManager::InputManager(
2. **const** sp<EventHubInterface>& eventHub,
3. **const** sp<InputReaderPolicyInterface>& readerPolicy,
4. **const** sp<InputDispatcherPolicyInterface>& dispatcherPolicy) {
5. mDispatcher = **new** InputDispatcher(dispatcherPolicy);
6. mReader = **new** InputReader(eventHub, readerPolicy, mDispatcher);
7. initialize();
8. }

**所以最后这个listener是InputDispatcher，因此最后就是调用的InputDispatcher的notifyKey函数。**

**InputReader中扫描码与键盘码的转化**

Ps：input设备的主设备号是13.这个是在input.c文件中

## 参考资料

<http://blog.csdn.net/wangkaiblog/article/details/12085183> android input

http://blog.csdn.net/droidphone/article/details/8434768 多点触控的input TP

http://blog.csdn.net/yaozhenguo2006/article/details/6775751 input\_event 函数

http://blog.csdn.net/loongembedded/article/details/51166888

http://blog.csdn.net/loongembedded/article/details/51167111 input事件类型

http://blog.csdn.net/tianruxishui/article/details/7173045 写得不错，

https://www.kernel.org/doc/Documentation/input/multi-touch-protocol.txt 源码的文档

<http://www.arm9home.net/read.php?tid=24754> 对源码文档的翻译

http://blog.chinaunix.net/uid-20776117-id-3212095.html 内部函数剖析

<http://blog.chinaunix.net/xmlrpc.php?r=blog/article&uid=29151914&id=3921536>

<http://blog.chinaunix.net/xmlrpc.php?r=blog/article&uid=29151914&id=3887032>

ps：其实源码的说明文档介绍的很清楚，但是不细节，所以需要在了解一定的基础上去看看说明文档

<http://wenku.baidu.com/link?url=FYMYA6KI4vCRGpDohcMIO6C5XgaxQ7uXwXlqah7JdQUJV3RdLJR2ZeAodChULrlHQnfUJNkavBoQ7LBp_e9wZIdcS_yeI4OLfHLojuwuInO> 讲述的比较详细。上层到下层

<http://wenku.baidu.com/view/ef3ccbe4ed630b1c58eeb5a4.html?re=view> 图解详细且清晰

<http://blog.csdn.net/zjngogo/article/details/46363571>和上面的一起看，Android层

<http://blog.chinaunix.net/uid-20672257-id-3149668.html> 内核层的讲解，比较到位

http://blog.csdn.net/coldsnow33/article/details/15809965上层好文

<http://www.cnblogs.com/lcw/p/3294356.html> 全解析

http://www.cnblogs.com/jason-lu/articles/3156411.html

<http://blog.csdn.net/tankai19880619/article/details/17019085>

http://blog.csdn.net/kc58236582/article/details/50610311

# Android调试方法

## getevent，sendevent，input三命令

### Getevent

**Usage**: getevent [-t] [-n] [-sswitchmask] [-**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

键入getevent后，我们能够看到设备中的一些列输入硬件驱动信息，同样下面会出现很多输入指令信号，通常情况下，这些信号量都在刷屏

#### 格式说明

root@android:/ # getevent

**/dev/input/event5: 0005 0002 00000001**

**device的名字： 事件类型 键码类别 具体的数值**

**/dev/input/event5: 0000 0000 00000000**

**表示一次输入结束（全为0表示一次输入的结束）**

其实这些Lable已经在其input.h头文件中定义好，其中type的定义如下：

/\*

\* **Event types**

\*/

**#define EV\_SYN 0x00**

**#define EV\_KEY 0x01**

**#define EV\_REL 0x02**

**#define EV\_ABS 0x03**

**#define EV\_MSC 0x04**

**#define EV\_SW 0x05**

**#define EV\_LED 0x11**

**#define EV\_SND 0x12**

**#define EV\_REP 0x14**

**#define EV\_FF 0x15**

**#define EV\_PWR 0x16**

**#define EV\_FF\_STATUS 0x17**

**#define EV\_MAX 0x1f**

**#define EV\_CNT (EV\_MAX+1)**

一般来说，常用的是EV\_KEY、EV\_REL、EV\_ABS、EV\_SYN，分别对应键盘按键、相对坐标、绝对坐标、同步事件。EV\_SYN则表示一组完整事件已经完成，需要处理，EV\_SYN的code定义事件分发的类型。

在触摸事件上的几个常见的Label说明如下表所示：

| **标签名** | **说明** |
| --- | --- |
| ABS\_X | 对应触摸屏的X坐标 |
| ABS\_Y | 对应触摸屏的Y坐标 |
| ABS\_PRESSURE | 压力值，一般触摸屏也只是区分是否有按下去，按下去的话值会大于多少，没有按的话值小于多少。 |
| ABS\_TOOL\_WIDTH | 触摸工具的宽度 |
| ABS\_MT\_POSITION\_X | 接触面的形心的X坐标值 |
| ABS\_MT\_POSITION\_Y | 接触面的形心的Y坐标值 |
| ABS\_MT\_TOUCH\_MAJOR | 触摸手指大小 |
| ABS\_MT\_WIDTH\_MAJOR | 触摸面积大小 |

了解了这些Label的含义我们再看看信号量就简单多了，如我们列举几个常见的事件与信号，如下表所示：

| **操作** | **输出信号** |
| --- | --- |
| 按下电源键 | /dev/input/event0: EV\_KEY KEY\_POWER DOWN /dev/input/event0: EV\_SYN SYN\_REPORT 000000 /dev/input/event0: EV\_KEY KEY\_POWER UP /dev/input/event0: EV\_SYN SYN\_REPORT 000000 |
| 音量键下 | /dev/input/event8: EV\_KEY KEY\_VOLUMEDOWN DOWN /dev/input/event8: EV\_SYN SYN\_REPORT 00000000 /dev/input/event8: EV\_KEY KEY\_VOLUMEDOWN UP /dev/input/event8: EV\_SYN SYN\_REPORT 00000000 |
| 音量键上 | /dev/input/event8: EV\_KEY KEY\_VOLUMEUP DOWN /dev/input/event8: EV\_SYN SYN\_REPORT 00000000 /dev/input/event8: EV\_KEY KEY\_VOLUMEUP UP /dev/input/event8: EV\_SYN SYN\_REPORT 00000000 |
| 按下物理按键“1” | /dev/input/event0: EV\_KEY KEY\_1 DOWN /dev/input/event0: EV\_KEY KEY\_1 UP |
| 按下物理按键“q” | /dev/input/event0: EV\_KEY KEY\_Q DOWN /dev/input/event0: EV\_KEY KEY\_Q UP |
| 按下软键盘上的“q”字母 | /dev/input/event0: EV\_ABS ABS\_X 0000001b /dev/input/event0: EV\_ABS ABS\_Y 000001d5 /dev/input/event0: EV\_KEY BTN\_TOUCH DOWN /dev/input/event0: EV\_SYN SYN\_REPORT 00000000 /dev/input/event0: EV\_KEY BTN\_TOUCH UP /dev/input/event0: EV\_SYN SYN\_REPORT 00000000 |
| 按下软件键盘上的的“1”按键 | /dev/input/event0: EV\_ABS ABS\_X 00000019 /dev/input/event0: EV\_ABS ABS\_Y 000001d7 /dev/input/event0: EV\_KEY BTN\_TOUCH DOWN /dev/input/event0: EV\_SYN SYN\_REPORT 00000000 /dev/input/event0: EV\_KEY BTN\_TOUCH UP /dev/input/event0: EV\_SYN SYN\_REPORT 00000000 |

从上表中，我们发现要是按下的是物理按键，其输入出来的信息我们很容易读懂，如果按下的是软键盘中的按键，给出的信号信息就是一些位置坐标信息。我们无法直接读懂，当然，我们可以根据这些位置坐标信息，再拿到Android设备的屏幕尺寸，计算比例也能够直接获得按键的具体内容。

当然，输出条件不会是想我们表格中的这么规范，中间会夹杂则各式各样的信息，有些可能是你不关心的。这里我们把一些无关的信号量过滤去掉了。实际查看上对应信息条件比较多，大家可以将Android设备连接如自己的电脑进行调试，这里我们就不做一一的解释了。

Ps:以上结果仅供参考。实际消息会有一些偏差

The getevent tool runs on the device and provides information about input devices and a live dump of kernel input events.

This tool is useful for ensuring device drivers are reporting the expected set of capabilities for each input device and are generating the desired stream of input events.

Showing device capabilities

Use the -p option to see all of the keys and axes a device reports. The following example lists the Linux key codes and other events a particular keyboard says it supports.

$ adb shell su -- getevent -p  
  
  name:     "Motorola Bluetooth Wireless Keyboard"  
  events:  
    KEY (0001): 0001  0002  0003  0004  0005  0006  0007  0008  
                0009  000a  000b  000c  000d  000e  000f  0010  
                0011  0012  0013  0014  0015  0016  0017  0018  
                0019  001a  001b  001c  001d  001e  001f  0020  
                0021  0022  0023  0024  0025  0026  0027  0028  
                0029  002a  002b  002c  002d  002e  002f  0030  
                0031  0032  0033  0034  0035  0036  0037  0038  
                0039  003a  003b  003c  003d  003e  003f  0040  
                0041  0042  0043  0044  0045  0046  0047  0048  
                0049  004a  004b  004c  004d  004e  004f  0050  
                0051  0052  0053  0055  0056  0057  0058  0059  
                005a  005b  005c  005d  005e  005f  0060  0061  
                0062  0063  0064  0066  0067  0068  0069  006a  
                006b  006c  006d  006e  006f  0071  0072  0073  
                0074  0075  0077  0079  007a  007b  007c  007d  
                007e  007f  0080  0081  0082  0083  0084  0085  
                0086  0087  0088  0089  008a  008c  008e  0090  
                0096  0098  009b  009c  009e  009f  00a1  00a3  
                00a4  00a5  00a6  00ab  00ac  00ad  00b0  00b1  
                00b2  00b3  00b4  00b7  00b8  00b9  00ba  00bb  
                00bc  00bd  00be  00bf  00c0  00c1  00c2  00d9  
                00f0  0110  0111  0112  01ba  
    REL (0002): 0000  0001  0008  
    ABS (0003): 0028  : value 223, min 0, max 255, fuzz 0, flat 0, resolution 0  
                0029  : value 0, min 0, max 255, fuzz 0, flat 0, resolution 0  
                002a  : value 0, min 0, max 255, fuzz 0, flat 0, resolution 0  
                002b  : value 0, min 0, max 255, fuzz 0, flat 0, resolution 0  
    MSC (0004): 0004  
    LED (0011): 0000  0001  0002  0003  0004  
  input props:  
    <none>

Use the -i option to get more information, including HID mapping tables and debugging information.

Use the -l option to display textual labels for all event codes. Example:

$ adb shell su -- getevent -lp /dev/input/event1  
  
  name:     "Melfas MMSxxx Touchscreen"  
  events:  
    ABS (0003): ABS\_MT\_SLOT           : value 0, min 0, max 9, fuzz 0, flat 0, resolution 0  
                ABS\_MT\_TOUCH\_MAJOR    : value 0, min 0, max 30, fuzz 0, flat 0, resolution 0  
                ABS\_MT\_POSITION\_X     : value 0, min 0, max 720, fuzz 0, flat 0, resolution 0  
                ABS\_MT\_POSITION\_Y     : value 0, min 0, max 1280, fuzz 0, flat 0, resolution 0  
                ABS\_MT\_TRACKING\_ID    : value 0, min 0, max 65535, fuzz 0, flat 0, resolution 0  
                ABS\_MT\_PRESSURE       : value 0, min 0, max 255, fuzz 0, flat 0, resolution 0  
  input props:  
    INPUT\_PROP\_DIRECT

Showing live events

The following example shows a two-finger multi-touch gesture for a touchscreen using the Linux multi-touch input protocol "B". The -l option displays textual labels and the -t option displays timestamps.

$ adb shell su -- getevent -lt /dev/input/event1  
  
[   78826.389007] EV\_ABS       ABS\_MT\_TRACKING\_ID   0000001f  
[   78826.389038] EV\_ABS       ABS\_MT\_PRESSURE      000000ab  
[   78826.389038] EV\_ABS       ABS\_MT\_POSITION\_X    000000ab  
[   78826.389068] EV\_ABS       ABS\_MT\_POSITION\_Y    0000025b  
[   78826.389068] EV\_ABS       ABS\_MT\_SLOT          00000001  
[   78826.389068] EV\_ABS       ABS\_MT\_TRACKING\_ID   00000020  
[   78826.389068] EV\_ABS       ABS\_MT\_PRESSURE      000000b9  
[   78826.389099] EV\_ABS       ABS\_MT\_POSITION\_X    0000019e  
[   78826.389099] EV\_ABS       ABS\_MT\_POSITION\_Y    00000361  
[   78826.389099] EV\_SYN       SYN\_REPORT           00000000  
[   78826.468688] EV\_ABS       ABS\_MT\_SLOT          00000000  
[   78826.468688] EV\_ABS       ABS\_MT\_TRACKING\_ID   ffffffff  
[   78826.468719] EV\_ABS       ABS\_MT\_SLOT          00000001  
[   78826.468719] EV\_ABS       ABS\_MT\_TRACKING\_ID   ffffffff  
[   78826.468719] EV\_SYN       SYN\_REPORT           00000000

**Note:** getevent timestamps use the format $SECONDS.$MICROSECONDS in the CLOCK\_MONOTONIC timebase. For details, refer to getevent.c.

Ps：getevent –l参数可以看到事件类型，以及键码类别

### Sendevent

### Input

**Usage: input [<source>] <command> [<arg>...]**

**The sources are:**

**mouse**

**keyboard**

**joystick**

**touchnavigation**

**touchpad**

**trackball**

**stylus**

**dpad**

**touchscreen**

**gamepad**

**The commands and default sources are:**

**text <string> (Default: touchscreen)**

**keyevent [--longpress] <key code number or name> ... (Default: keyboard)**

**tap <x> <y> (Default: touchscreen)**

**swipe <x1> <y1> <x2> <y2> [duration(ms)] (Default: touchscreen)**

**press (Default: trackball)**

**roll <dx> <dy> (Default: trackball)**

#### 基本用法

input keyevent [key code or name]//用来对应按键的触发

Keycode需要参考:$ROOT\_DIR/frameworks/base/core/java/android/view/KEYEVENT.java

Input tap x y //对应触摸点

Input swipe x1 y1 x2 y2 //对应滑动触摸

Input text “text” //输入text内容

## 参考资料

<http://blog.csdn.net/shift_wwx/article/details/49760735> **input**

<http://blog.csdn.net/hudan2714/article/details/8003585> **input**

http://blog.csdn.net/shift\_wwx/article/details/11481797 **input**

<http://myeyeofjava.iteye.com/blog/1999615> **getevent**

<http://www.codeceo.com/article/android-keyboard-monitor.html> **getevent**