# Linux input 子系统

## 一、输入子系统的介绍

### 1、为何引入 input system

以前我们写一些输入设备（键盘、鼠标等）的驱动都是采用字符设备处理的。问题由此而来，Linux 开源社区的大神们看到了这大量输入设备如此分散不堪，有木有可以实现一种机制，可以对分散的、不同类别的输入设备进行统一的驱动，所以才出现了输入子系统。

### 2、输入子系统引入的好处

1. 统一了物理形态各异的相似的输入设备的处理功能

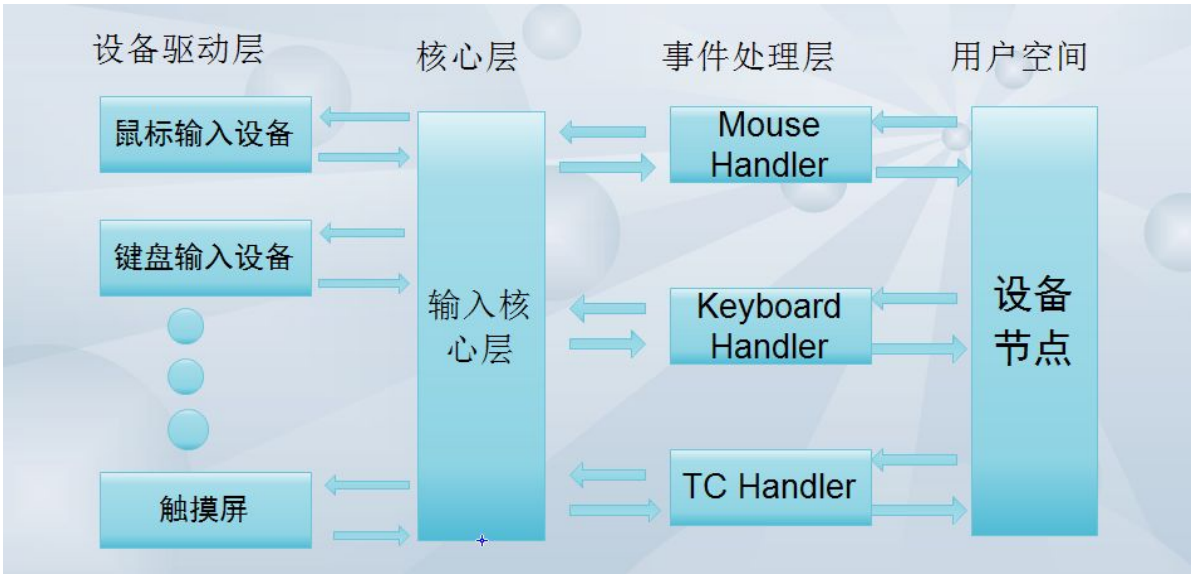
2. 提供了用于分发输入报告给用户应用程序的简单的事件接口

3. 抽取出了输入驱动程序的通用部分，简化了驱动，并引入了一致性。

### 3、输入子系统介绍

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底层在按键，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 通过去读取键值、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或者异步通知这个键值。

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。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。




如上图，input 子系统分三层，最上一层是 event handler，中间是 intput core，底层是 input driver。input driver 把 event report 到 input core 层。input core 对 event 进行分发，传到 event handler,相应的 event handler 层把 event 放到 event buffer 中，等待用户进程来取。

## 二、怎么描述三层

Linux 输入子系统包括三个层次，由上到下别是事件处理层（Event Handler）、核心层（Input Core）和驱动层（Input Driver）。

1. 事件处理层负责与用户程序打交道，将硬件驱动层传来的事件报告给用户程序。

2. 核心层是链接其他两个层之间的纽带与桥梁，向下提供驱动层的接口，向上提供事件处理层的接口。

3. 驱动层负责操作具体的硬件设备，这层的代码是针对具体的驱动程序的，键盘、鼠标、触摸屏等字符设备驱动功能的实现工作主要在这层。

在 Input 子系统三层框架中对应 3 个结构体。

1. 结构体 input_dev 表示底层硬件设备，是所有输入设备的抽象。

2. handle 是手柄的意思，结构体 input_handle 表示连接杆，连接底层硬件和上层事件处理层。

3. 结构体 input_handler 表示事件处理器，是对事件处理的抽象。

我们看到输入子系统中设备驱动层、核心层、事件处理层，当我们要为一个输入设备（如按键）的编写驱动的时候，我们是要编写 3 个吗？？

答案是否定的。在子系统中，事件核心层和事件处理层的驱动是**标准**的，对所有的输入类都是可以用的，所以你实现的仅仅是设备驱动层。你的设备可以利用一个已经存在的，合适的输入事件驱动通过输入核心和用户应用程序接口打交道。

## 三、相关 API 介绍

在输入子系统的设备驱动中，最重要的数据结构是 struct input_dev，如下所示。需要完成的大部分工作都是围绕着它来的，它是驱动的主体。每个 struct input_dev 代表一个输入设备。

/* include/linux/input.h */

**struct input_dev** {

const char ***name**; 设备名

const char *phys; //sys 目录下的设备路径，不需要用户设定

const char *uniq;

struct **input_id** id; //用于匹配事件处理层 handler（事件处理者），如果为空，表示可以匹配所有

**unsigned long evbit[BITS_TO_LONGS(EV_CNT)]; //用于记录支持的事件类型的位图**

**//例如：button_dev->evbit[0] = BIT_MASK(EV_KEY) | BIT_MASK(EV_REL) ; 表示这个设备支持按键事件和相对事件，一般情况下，应该使用 set_bit(EV_KEY, input_dev->evbit);set_bit(EV_REL, input_dev->evbit);**

**unsigned long keybit[BITS_TO_LONGS(KEY_CNT)]; //记录支持的按键值的位图**

**// 实际上就是存放所支持的具体哪些按键，例如：button_dev->keybit[BIT_WORD(BTN_0)] = BIT_MASK(BTN_0); 实际编程应该使用 set_bit(BTN_0, input_dev->keybit);**

unsigned long **relbit**[BITS_TO_LONGS(REL_CNT)];//记录支持的相对坐标的位图

//实际是支持哪些相对事件，如： set_bit(REL_X, input_dev->relbit);set_bit(REL_Y, input_dev->relbit);

unsigned long **absbit**[BITS_TO_LONGS(ABS_CNT)];//记录支持的绝对坐标的位图

//实际是支持哪些绝对事件，如： set_bit(ABS_X, input_dev->absbit);set_bit(ABS_Y, input_dev->absbit);

unsigned long mscbit[BITS_TO_LONGS(MSC_CNT)];

unsigned long ledbit[BITS_TO_LONGS(LED_CNT)];//led

unsigned long sndbit[BITS_TO_LONGS(SND_CNT)];//beep

unsigned long ffbit[BITS_TO_LONGS(FF_CNT)];

unsigned long swbit[BITS_TO_LONGS(SW_CNT)];

unsigned int keycodemax;//支持的按键值的个数

unsigned int keycodesize;//每个键值的字节数

void *keycode;//存储按键值的数组首地址

int (*setkeycode)(struct input_dev *dev,

unsigned int scancode, unsigned int keycode);//修改键值的函数，可选

int (*getkeycode)(struct input_dev *dev,

unsigned int scancode, unsigned int *keycode);//获取扫描码的键值，可选

struct ff_device *ff;

unsigned int repeat_key;//最近一次按键值，用于连击

struct timer_list timer; //自动连击计时器

int sync; //最后一次同步后没有新的事件置 1

int abs[ABS_CNT]; //当前各个坐标的值

int rep[REP_MAX + 1]; //自动连击的参数

unsigned long key[BITS_TO_LONGS(KEY_CNT)];//反映当前按键状态的位图

unsigned long led[BITS_TO_LONGS(LED_CNT)];//反映当前 led 状态的位图

unsigned long snd[BITS_TO_LONGS(SND_CNT)];//反映当前 beep 状态的位图

unsigned long sw[BITS_TO_LONGS(SW_CNT)];

**int absmax[ABS_CNT]; //记录各个坐标的最大值**

**int absmin[ABS_CNT]; //记录各个坐标的最小值**

int absfuzz[ABS_CNT]; //记录各个坐标的分辨率，

int absflat[ABS_CNT]; //记录各个坐标的基准值

int absres[ABS_CNT];

int (*open)(struct input_dev *dev); //打开函数，可以不实现，如果有就调用。

void (*close)(struct input_dev *dev);//关闭函数，可以不实现，如果有就调用。

int (*flush)(struct input_dev *dev, struct file *file);//断开连接时清空上报上去数据

int (*event)(struct input_dev *dev, unsigned int type, unsigned int code, int value);//回调函数，可选

struct input_handle *grab;

spinlock_t event_lock;

struct mutex mutex;

unsigned int users;

bool going_away;

struct device dev;

struct list_head h_list;//handle 链表

struct list_head node;//input_dev 链表

};

注册输入设备函数：

```
int input_register_device(struct input_dev *dev)
```

注销输入设备函数：

```
void input_unregister_device(struct input_dev *dev)
```

注册输入子系统前我们要有这个结构体类型的变量，怎么定义呢？我们可以直接定义也可以自动分配。

自动分配 struct input_dev 结构体函数：

```
struct input_dev *p=input_allocate_device();
```

分配完这个结构体以后，我们就要填充这个结构体，怎么填充呢？

1、填充名字---myinput->name = "mykey";

2、事件类型

```
#define EV_SYN          0x00        /* 同步事件                                */
#define EV_KEY          0x01         /*按键事件                                */
#define EV_REL          0x02         /*相对坐标                                */
#define EV_ABS          0x03         /*绝对坐标                                */
#define EV_MSC          0x04        /*其他*/
#define EV_SW           0x05
#define EV_LED          0x11         /*led                                     */
#define EV_SND        0x12          /*beep                                  */
#define EV_REP        0x14          /*连击事件                                */
#define EV_FF               0x15        /*压力*/
#define EV_PWR          0x16         /*电源事件*/
#define EV_FF_STATUS                    0x17        /*受压状态*/
#define EV_MAX                         0x1f
#define EV_CNT                         (EV_MAX+1)
```

**set_bit():位操作，**告诉输入子系统支持哪些事件，哪些按键。

注册完这些事件以后，我们就要在这些时候完成以后，通知上层以完成操作了。

上报事件函数：

```
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
```

以上就完成内核层的输入子系统编写

输入子系统编写完成以后，就会在/dev/input/下面自动生成节点，上层可以通过读取这个节点获取驱动报告的值。这个节点文件时自动创建的。我们可以看一下。

```
[root@XYD /]# ls /dev/input/
event0  event1  event2  mice    mouse0
[root@XYD /]# ls /dev/input/
event0  event1  event2  mice    mouse0
[root@XYD /]#
```

以上是没有加载驱动的时候，此时已经有 4 个文件了。

```
[root@XYD /]# ls /dev/input/event
event0  event1  event2  event3
[root@XYD /]#
```

加载完以后就多了一个，这个就是我们加载的。

还有在 pro/device 下面有一个类。主设备号默认都为 13。

```
  5 /dev/ptmx
  7 vcs
 10 misc
 13 input
 14 sound
 21 sg
```

报告的事件就存放在/dev/input/event3 里面我们就读取即可。

上层定义 struct input_event e;

```
 struct input_event {
  struct timeval time;//时间戳
 __u16 type;//事件类型
 __u16 code;//事件代码
 __s32 value;//事件值，如坐标的偏移值
};
```

fd=open(“/dev/input/event3”,O_RDWR);

ret = read(fd,&e,sizeof(struct input_event));

type --- 对应你内核注册的事件，code 对应你要反回的值