涛少&

昵称：涛少&
园龄：1年11个月
粉丝：12
关注：0
+加关注

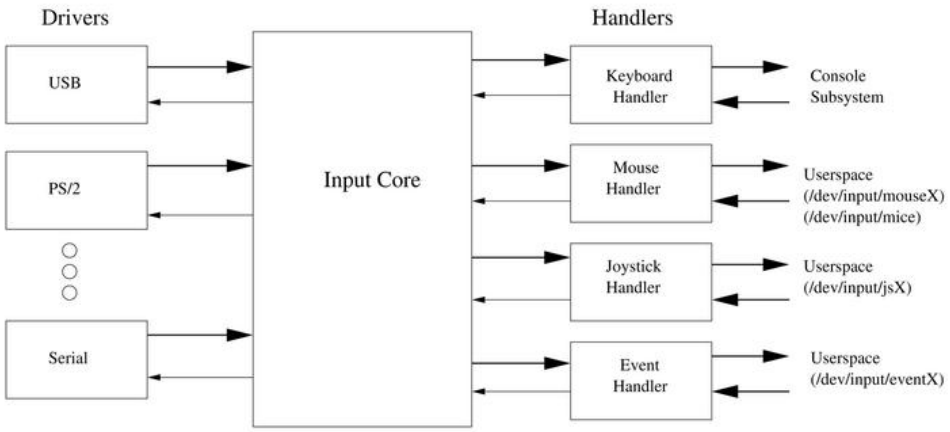
INPUT输入子系统

一、什么是input输入子系统？

1、Linux系统支持的输入设备繁多，例如键盘、鼠标、触摸屏、手柄或者是一些输入设备像体感输入等等，Linux系统是如何管理如此之多的不同类型、不同原理、不同的输入信息的

输入设备的呢？其实就是通过input输入子系统这套软件体系来完成的。从整体上来说，input输入子系统分为3层：上层（输入事件驱动层）、中层（输入核心层）、

下层（输入设备驱动层），如下图所示：



联系之前学过的驱动框架做对比，input输入子系统其实就是input输入设备的驱动框架，与之前的学过的驱动框架不同的是，input输入子系统分为3层：上、中、下，所以他的复杂度

要高于之前讲的lcd、misc、fb等的驱动框架。

2、图中Drivers对应的就是下层设备驱动层，对应各种各样不同的输入设备，Input Core对应的就是中层核心层，Handlers对应的就是上层输入事件驱动层，最右边的代表的是用户空间。

(1)从图中可以看出，系统中可以注册多个输入设备，每个输入设备的可以是不同的，例如一台电脑上可以带有鼠标，键盘.....。

(2)上层中的各个handler（Keyboard/Mouse/Joystick/Event）是属于平行关系，他们都是属于上层。不同的handler下对应的输入设备在应用层中的接口命名方式不一样，例如

Mouse下的输入设备在应用层的接口是 /dev/input/mousen （n代表0、1、2...），Joystick下的输入设备在应用层的接口是 /dev/input/jsn（n代表0、1、2...），

Event下的输入设备在应用层的接口是 /dev/input/eventn（n代表0、1、2...），这个是在input输入子系统中实现的，下面会分析其中的原由。

(3)输入核心层其实是负责协调上层和下层，使得上层和下层之间能够完成数据传递。当下层发生输入事件的时候，整个系统就被激活了，事件就会通过核心层传递到上层对应的一个/多个

handler中，最终会传递到应用空间。

3、输入子系统解决了什么问题？

(1)在GUI界面中，用户的自由度太大了，可以做的事情太多了，可以响应不同的输入类设备，而且还能够对不同的输入类设备的输入做出不同的动作。例如window中的一个软

件既可以响应鼠标输入事件，也可以相应键盘输入事件，而且这些事件都是预先不知道的。

(2)input子系统解决了不同的输入类设备的输入事件与应用层之间的数据传输，使得应用层能够获取到各种不同的输入设备的输入事件，input输入子系统能够囊括所有的不同种

类的输入设备，在应用层都能够感知到所有发生的输入事件。

4、input输入子系统如何工作？

例如以一次鼠标按下事件为例子来说明我们的input输入子系统的工作过程：

当我们按下鼠标左键的时候就会触发中断（中断是早就注册好的），就会去执行中断所绑定的处理函数，在函数中就会去读取硬件寄存器来判断按下的是哪个按键和状态 ---->

将按键信息上报给input core层 ---> input core层处理好了之后就会上报给input event层，在这里会将我们的输入事件封装成一个input_event结构体放入一个缓冲区中 --->

应用层read就会将缓冲区中的数据读取出去。

5、相关的数据结构

```
struct input_dev {
    const char *name;              //  input设备的名字
    const char *phys;               //
    const char *uniq;               //
    struct input_id id;              //

//  这些是用来表示该input设备能够上报的事件类型有哪些    是用位的方式来表示的
    unsigned long evbit[BITS_TO_LONGS(EV_CNT)];
    unsigned long keybit[BITS_TO_LONGS(KEY_CNT)];
    unsigned long relbit[BITS_TO_LONGS(REL_CNT)];
    unsigned long absbit[BITS_TO_LONGS(ABS_CNT)];
    unsigned long mscbit[BITS_TO_LONGS(MSC_CNT)];
    unsigned long ledbit[BITS_TO_LONGS(LED_CNT)];
    unsigned long sndbit[BITS_TO_LONGS(SND_CNT)];
    unsigned long ffbit[BITS_TO_LONGS(FF_CNT)];
    unsigned long swbit[BITS_TO_LONGS(SW_CNT)];

    unsigned int keycodemax;
    unsigned int keycodesize;
    void *keycode;
    int (*setkeycode)(struct input_dev *dev,
              unsigned int scancode, unsigned int keycode);
    int (*getkeycode)(struct input_dev *dev,
              unsigned int scancode, unsigned int *keycode);

    struct ff_device *ff;

    unsigned int repeat_key;
    struct timer_list timer;

    int sync;

    int abs[ABS_CNT];
    int rep[REP_MAX + 1];

    unsigned long key[BITS_TO_LONGS(KEY_CNT)];
    unsigned long led[BITS_TO_LONGS(LED_CNT)];
    unsigned long snd[BITS_TO_LONGS(SND_CNT)];
    unsigned long sw[BITS_TO_LONGS(SW_CNT)];

    int absmax[ABS_CNT];
    int absmin[ABS_CNT];
    int absfuzz[ABS_CNT];
    int absflat[ABS_CNT];
    int absres[ABS_CNT];

    int (*open)(struct input_dev *dev);              //     设备的open函数
    void (*close)(struct input_dev *dev);
    int (*flush)(struct input_dev *dev, struct file *file);
    int (*event)(struct input_dev *dev, unsigned int type, unsigned int code, int value);     //  上报事件

    struct input_handle *grab;

    spinlock_t event_lock;
```

```
    struct mutex mutex;

    unsigned int users;
    bool going_away;

    struct device dev;                  //  内置的device结构体变量

    struct list_head    h_list;     //  用来挂接input_dev 设备连接的所有handle 的一个链表头
    struct list_head    node;    //  作为链表节点挂接到  input_dev_list 链表上  (input_dev_list链表是input核心层维护的一个用来挂接所有input设备的一个链表头)
};
```

```
struct input_handler {

    void *private;            //  私有数据

    void (*event)(struct input_handle *handle, unsigned int type, unsigned int code, int value);   //  handler用于向上层上报输入事件的函数
    bool (*filter)(struct input_handle *handle, unsigned int type, unsigned int code, int value);
     bool (*match)(struct input_handler *handler, struct input_dev *dev);              //   match 函数用来匹配handler 与input_dev 设备
     int (*connect)(struct input_handler *handler, struct input_dev *dev, const struct input_device_id *id);    //  当handler 与 input_dev 匹配成功之后用来连接
    void (*disconnect)(struct input_handle *handle);          //  断开handler 与 input_dev 之间的连接
    void (*start)(struct input_handle *handle);

    const struct file_operations *fops;             //  一个file_operations 指针
    int minor;                                      //  该handler 的编号 (在input_table 数组中用来计算数组下标) input_table数组就是input子系统用来管理注册的handler的一个数据结构
    const char *name;                               //  handler的名字

    const struct input_device_id *id_table;      //  指向一个 input_device_id  类型的数组，用来进行与input设备匹配时用到的信息

    struct list_head    h_list;       //  用来挂接handler 上连接的所有handle 的一个链表头
     struct list_head    node;        //  作为一个链表节点挂接到 input_handler_list 链表上(input_handler_list 链表是一个由上层handler参维护的一个用来挂接所有注册的handler的链表头)
};
```

```
struct input_handle {

    void *private;              //   handle  的私有数据

    int open;                    //  这个也是用来做打开计数的
    const char *name;       //   该handle 的名字

    struct input_dev *dev;               //  用来指向该handle 绑定的input_dev 结构体
    struct input_handler *handler;    //  用来指向该handle 绑定的 handler 结构体

    struct list_head    d_node;      //  作为一个链表节点挂接到与他绑定的input_dev ->hlist 链表上
    struct list_head    h_node;      //  作为一个链表节点挂接到与他绑定的handler->hlist 链表上
};
```

```
struct input_device_id {

    kernel_ulong_t flags;    //  这个flag 表示我们的这个 input_device_id 是用来匹配下面的4个情况的哪一项
                                //  flag == 1表示匹配总线  2表示匹配供应商   4表示匹配产品  8表示匹配版本
    __u16 bustype;
    __u16 vendor;
    __u16 product;
    __u16 version;

    kernel_ulong_t evbit[INPUT_DEVICE_ID_EV_MAX / BITS_PER_LONG + 1];
    kernel_ulong_t keybit[INPUT_DEVICE_ID_KEY_MAX / BITS_PER_LONG + 1];
    kernel_ulong_t relbit[INPUT_DEVICE_ID_REL_MAX / BITS_PER_LONG + 1];
    kernel_ulong_t absbit[INPUT_DEVICE_ID_ABS_MAX / BITS_PER_LONG + 1];
    kernel_ulong_t mscbit[INPUT_DEVICE_ID_MSC_MAX / BITS_PER_LONG + 1];
    kernel_ulong_t ledbit[INPUT_DEVICE_ID_LED_MAX / BITS_PER_LONG + 1];
    kernel_ulong_t sndbit[INPUT_DEVICE_ID_SND_MAX / BITS_PER_LONG + 1];
    kernel_ulong_t ffbit[INPUT_DEVICE_ID_FF_MAX / BITS_PER_LONG + 1];
    kernel_ulong_t swbit[INPUT_DEVICE_ID_SW_MAX / BITS_PER_LONG + 1];
```

```
19
20      kernel_ulong_t driver_info;
21  };
```

二、输入核心层源码分析（内核版本：2.6.35.7）

input输入子系统中的所有源码都放在 drivers\input 这个目录中，input.c文件就是核心层的源代码文件。在input目录中还可以看到一些文件夹，例如gameport、joystick

keyboard、misc、mouse....，这些文件夹里面存放的就是属于这类的input输入设备的设备驱动源代码，可以理解为input输入子系统的下层。

input目录下的evdev.c、joydev.c、mousedev.c..分别对应上层的各个不同的handler的源代码。

1、输入核心层模块注册函数input_init

在Linux中实现为一个模块的方法，所以可以在内核配置的进行动态的加载和卸载，这样做的原由是，存在有些系统中不需要任何

的输入类设备，这样就可以将input输入子系统这个模块去掉（上层也是实现为模块的），使得内核尽量变得更小。

```
1  static int __init input_init(void)
2  {
3      int err;
4
5      input_init_abs_bypass();
6
7      err = class_register(&input_class);                //  创建设备类     /sys/class/input
8      if (err) {
9          printk(KERN_ERR "input: unable to register input_dev class\n");
10         return err;
11     }
12
13     err = input_proc_init();           //     proc文件系统相关的初始化
14     if (err)
15         goto fail1;
16
17     err = register_chrdev(INPUT_MAJOR, "input", &input_fops);       //    注册字符设备驱动    主设备号13    input_fops 中只实现了open函数，所以他的原理其实和misc其实是一样的
18     if (err) {
19         printk(KERN_ERR "input: unable to register char major %d", INPUT_MAJOR);
20         goto fail2;
21     }
22
23     return 0;
24
25  fail2:    input_proc_exit();
26  fail1:    class_unregister(&input_class);
27     return err;
28 }
```

(1)input_proc_init函数

```
1  static int __init input_proc_init(void)
2  {
3      struct proc_dir_entry *entry;
4
5      proc_bus_input_dir = proc_mkdir("bus/input", NULL);     /* 在/proc/bus/目录下创建input目录 */
6      if (!proc_bus_input_dir)
7          return -ENOMEM;
8
9      entry = proc_create("devices", 0, proc_bus_input_dir,  /* 在/proc/bus/input/目录下创建devices文件 */
10                 &input_devices_fileops);
11     if (!entry)
12         goto fail1;
13
14     entry = proc_create("handlers", 0, proc_bus_input_dir, /* 在/proc/bus/input/目录下创建handlers文件 */
15                 &input_handlers_fileops);
16     if (!entry)
17         goto fail2;
18
19     return 0;
```

```
20
21  fail2:    remove_proc_entry("devices", proc_bus_input_dir);
22  fail1: remove_proc_entry("bus/input", NULL);
23     return -ENOMEM;
24 }
```

当我们启动系统之后进入到proc文件系统中，确实可以看到在/proc/bus/input/目录下有两个文件devices和handlers，这两个文件就是在这里被创建的。我们cat devices 和 cat handlers

时对应的操作方法(show)就被封装在input_devices_fileops和input_handlers_fileops结构体中。

(2)input_fops变量

```
static const struct file_operations input_fops = {
    .owner = THIS_MODULE,
    .open = input_open_file,
};
```

```
1 static int input_open_file(struct inode *inode, struct file *file)
2 {
3     struct input_handler *handler;                                          //  定义一个input_handler指针
4     const struct file_operations *old_fops, *new_fops = NULL;   //  定义两个file_operations指针
5     int err;
6
7     err = mutex_lock_interruptible(&input_mutex);
8     if (err)
9         return err;
10
11    /* No load-on-demand here? */
12    handler = input_table[iminor(inode) >> 5];         //  通过次设备号在 input_table  数组中找到对应的 handler
13    if (handler)
14        new_fops = fops_get(handler->fops);           //  将handler 中的fops 指针赋值给 new_fops
15
16    mutex_unlock(&input_mutex);
17
18    /*
19     * That's _really_ odd. Usually NULL ->open means "nothing special",
20     * not "no device". Oh, well...
21     */
22    if (!new_fops || !new_fops->open) {
23        fops_put(new_fops);
24        err = -ENODEV;
25        goto out;
26    }
27
28    old_fops = file->f_op;           //   将 file->fops 先保存到 old_fops 中，以便出错时能够恢复
29    file->f_op = new_fops;          //   用new_fops 替换 file 中 fops
30
31    err = new_fops->open(inode, file);       //  执行 file->open  函数
32    if (err) {
33        fops_put(file->f_op);
34        file->f_op = fops_get(old_fops);
35    }
36    fops_put(old_fops);
37 out:
38    return err;
39 }
```

2、核心层提供给设备驱动层的接口函数

input设备驱动框架留给设备驱动层的接口函数主要有3个：

input_allocate_device。分配一块input_dev结构体类型大小的内存

input_set_capability。设置输入设备可以上报哪些输入事件

input_register_device。向input核心层注册设备

(1)input_allocate_device函数

```
1 struct input_dev *input_allocate_device(void)
2 {
3     struct input_dev *dev;                //   定义一个 input_dev  指针
```

```
4
5     dev = kzalloc(sizeof(struct input_dev), GFP_KERNEL);   //  申请分配内存
6     if (dev) {
7         dev->dev.type = &input_dev_type;          //  确定input设备的 设备类型      input_dev_type
8         dev->dev.class = &input_class;              //  确定input设备所属的设备类    class
9         device_initialize(&dev->dev);                 //  input设备的初始化
10        mutex_init(&dev->mutex);                      //  互斥锁初始化
11        spin_lock_init(&dev->event_lock);           //  自旋锁初始化
12        INIT_LIST_HEAD(&dev->h_list);               //  input_dev -> h_list 链表初始化
13        INIT_LIST_HEAD(&dev->node);                //  input_dev -> node 链表初始化
14
15        __module_get(THIS_MODULE);
16    }
17
18    return dev;
19 }
```

(2)input_set_capability函数：

函数原型：input_set_capability(struct input_dev *dev, unsigned int type, unsigned int code)

参数：dev就是设备的input_dev结构体变量

type表示设备可以上报的事件类型

code表示上报这类事件中的那个事件

注意：input_set_capability函数一次只能设置一个具体事件，如果设备可以上报多个事件，则需要重复调用这个函数来进行设置，例如：

input_set_capability(dev, EV_KEY, KEY_Q);　　// 至于函数内部是怎么设置的，将会在后面进行分析。

input_set_capability(dev, EV_KEY, KEY_W);

input_set_capability(dev, EV_KEY, KEY_E);

```
#define EV_SYN          0x00    /* 同步 */
#define EV_KEY          0x01    /* 按键 */
#define EV_REL          0x02    /* 相对运动 */
#define EV_ABS          0x03    /* 绝对运动 */
#define EV_MSC          0x04    /* 其他杂类 */
#define EV_SW           0x05    /* 转换 */
#define EV_LED          0x11    /* 设备上的LED */
#define EV_SND          0x12    /* 声音效果 */
#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)
```

具体的这些类下面有哪些具体的输入事件，请看 drivers\input\input.h 这个文件。

(3)input_register_device函数：

```
1 int input_register_device(struct input_dev *dev)      //  注册input输入设备
2 {
3     static atomic_t input_no = ATOMIC_INIT(0);
4     struct input_handler *handler;                          //  定义一个  input_handler 结构体指针
5     const char *path;
6     int error;
7
8     /* Every input device generates EV_SYN/SYN_REPORT events. */
9     __set_bit(EV_SYN, dev->evbit);                  //   每一个input输入设备都会发生这个事件
10
11    /* KEY_RESERVED is not supposed to be transmitted to userspace. */
12    __clear_bit(KEY_RESERVED, dev->keybit);  //  清除KEY_RESERVED 事件对应的bit位，也就是不传输这种类型的事件
13
14    /* Make sure that bitmasks not mentioned in dev->evbit are clean. */
15    input_cleanse_bitmasks(dev);          //   确保input_dev中的用来记录事件的变量中没有提到的位掩码是干净的。
16
17    /*
```

```
18      * If delay and period are pre-set by the driver, then autorepeating
19      * is handled by the driver itself and we don't do it in input.c.
20      */
21     init_timer(&dev->timer);
22     if (!dev->rep[REP_DELAY] && !dev->rep[REP_PERIOD]) {
23         dev->timer.data = (long) dev;
24         dev->timer.function = input_repeat_key;
25         dev->rep[REP_DELAY] = 250;
26         dev->rep[REP_PERIOD] = 33;
27     }
28
29     if (!dev->getkeycode)
30         dev->getkeycode = input_default_getkeycode;
31
32     if (!dev->setkeycode)
33         dev->setkeycode = input_default_setkeycode;
34
35     dev_set_name(&dev->dev, "input%ld",                                  //   设置input设备对象的名字     input+数字
36              (unsigned long) atomic_inc_return(&input_no) - 1);
37
38     error = device_add(&dev->dev);         //   添加设备        例如:          /sys/devices/virtual/input/input0
39     if (error)
40         return error;
41
42             path   =   kobject_get_path(&dev->dev.kobj,   GFP_KERNEL);     //     获 取 input 设 备 对 象 所 在 的 路 径
/sys/devices/virtual/input/input_xxx
43     printk(KERN_INFO "input: %s as %s\n",
44         dev->name ? dev->name : "Unspecified device", path ? path : "N/A");
45     kfree(path);
46
47     error = mutex_lock_interruptible(&input_mutex);
48     if (error) {
49         device_del(&dev->dev);
50         return error;
51     }
52
53       list_add_tail(&dev->node, &input_dev_list);                 //    链表挂接:      将 input_dev->node 作为节点挂接到
input_dev_list  链表上
54
55     list_for_each_entry(handler, &input_handler_list, node)  //  遍历input_handler_list 链表上的所有handler
56         input_attach_handler(dev, handler);                          //  将handler与input设备进行匹配
57
58     input_wakeup_procfs_readers();                //  更新proc 文件系统
59
60     mutex_unlock(&input_mutex);
61
62     return 0;
63 }
```

(4)input_attach_handler函数：

input_attach_handler就是input_register_device函数中用来对下层的设备驱动和上层的handler进行匹配的一个函数，只有匹配成功之后就会调用上层handler中的connect函数

进行连接绑定。

```
1 static int input_attach_handler(struct input_dev *dev, struct input_handler *handler)
2 {
3     const struct input_device_id *id;               //   定义一个input_device_id 的指针
4     int error;
5
6     id = input_match_device(handler, dev);   //  通过这个函数进行handler与input设备的匹配工作
7     if (!id)
8         return -ENODEV;
9
10    error = handler->connect(handler, dev, id);  //  匹配成功则调用 handler 中的 connect 函数进行连接
11    if (error && error != -ENODEV)
12        printk(KERN_ERR
13            "input: failed to attach handler %s to device %s, "
14            "error: %d\n",
15            handler->name, kobject_name(&dev->dev.kobj), error);
16
17    return error;
18 }
19
20
21
22 static const struct input_device_id *input_match_device(struct input_handler *handler,
23                             struct input_dev *dev)
24 {
```

```
25      const struct input_device_id *id;            //   定义一个 input_device_id  指针
26      int i;
27
28        for (id = handler->id_table; id->flags || id->driver_info; id++) {   //   依次遍历handler->id_table 所指向的input_device_id 数组中的各个元素
29                                                                                                                                                  //依次进行下面的匹配过程
30          if (id->flags & INPUT_DEVICE_ID_MATCH_BUS)          //     匹配总线
31              if (id->bustype != dev->id.bustype)
32                  continue;
33
34          if (id->flags & INPUT_DEVICE_ID_MATCH_VENDOR)  //  匹配供应商
35              if (id->vendor != dev->id.vendor)
36                  continue;
37
38          if (id->flags & INPUT_DEVICE_ID_MATCH_PRODUCT)  //  匹配产品
39              if (id->product != dev->id.product)
40                  continue;
41
42          if (id->flags & INPUT_DEVICE_ID_MATCH_VERSION)  //  匹配版本
43              if (id->version != dev->id.version)
44                  continue;
45
46      //     下面的这些是匹配我们上传的事件是否属实
47          MATCH_BIT(evbit,  EV_MAX);
48          MATCH_BIT(keybit, KEY_MAX);
49          MATCH_BIT(relbit, REL_MAX);
50          MATCH_BIT(absbit, ABS_MAX);
51          MATCH_BIT(mscbit, MSC_MAX);
52          MATCH_BIT(ledbit, LED_MAX);
53          MATCH_BIT(sndbit, SND_MAX);
54          MATCH_BIT(ffbit,  FF_MAX);
55          MATCH_BIT(swbit,  SW_MAX);
56
57          if (!handler->match || handler->match(handler, dev))
58              return id;         //     如果数组中的某个匹配成功了就返回他的地址
59      }
60
61      return NULL;
62  }
```

input_attach_handler函数做的事情有两件：调用input_match_device函数进行设备与handler的匹配、匹配成功调用handler的连接函数进行连接（至于如何连接将会在后面说到）。

3、核心层提供给事件驱动层的接口函数

在input输入核心层向事件驱动层提供的接口主要有两个：

input_register_handler。事件驱动层向核心层注册handler

input_register_handle。事件驱动层向核心层注册handle。 注意上面的是handler，这里是handle，不一样，后面会说到。

(1)input_register_handler函数：

```
1  int input_register_handler(struct input_handler *handler)    //  向核心层注册handler
2  {
3      struct input_dev *dev;          //  定义一个input_dev 指针
4      int retval;
5
6      retval = mutex_lock_interruptible(&input_mutex);
7      if (retval)
8          return retval;
9
10     INIT_LIST_HEAD(&handler->h_list);      //  初始化 handler->h_list 链表
11
12     if (handler->fops != NULL) {          //  如果 handler -> fops 存在
13         if (input_table[handler->minor >> 5]) {  //  如果input_table 数组中没有该handler  的位置了 则返回
14             retval = -EBUSY;
15             goto out;
16         }
17         input_table[handler->minor >> 5] = handler;  //  将 handler 指针存放在input_table 数组中去
18     }
19
20         list_add_tail(&handler->node,  &input_handler_list);     //   将  handler  通过  handler  ->  node  节点  挂接到input_handler_list 链表上
21
22     list_for_each_entry(dev, &input_dev_list, node)     //  遍历 input_dev_list 链表下挂接的所有的 input_dev 设备
23         input_attach_handler(dev, handler);          //  然后进行匹配
```

```
24
25      input_wakeup_procfs_readers();              //  更新proc 文件系统
26
27  out:
28      mutex_unlock(&input_mutex);
29      return retval;
30  }
```

通过分析了上面的input_register_device和这里的input_register_handler函数可以知道：注册设备的时候，不一定是先注册了handler才能够注册设备。当注册设备时，会先将

设备挂接到设备管理链表(input_dev_list)上，然后再去遍历input_handler_list链表匹配hander。同样对于handler注册的时候，也会先将handler挂接到handler管理链表

(input_handler_list)上，然后再去遍历input_dev_list链表匹配设备。所以从这里可以看出来，这种机制好像之前说过的platform总线下设备和驱动的匹配过程。

而且一个input_dev可以与多个handler匹配成功，从而可以在sysfs中创建多个设备文件，也可以在/dev/目录下创建多个设备节点，并且他们的次设备号是不一样的，这个很好理解。

所以就是导致一个设备对应多个次设备号，那这样有没有错呢？当然是没有错的。例如在我们的Ubuntu中，/dev/input/event3 和

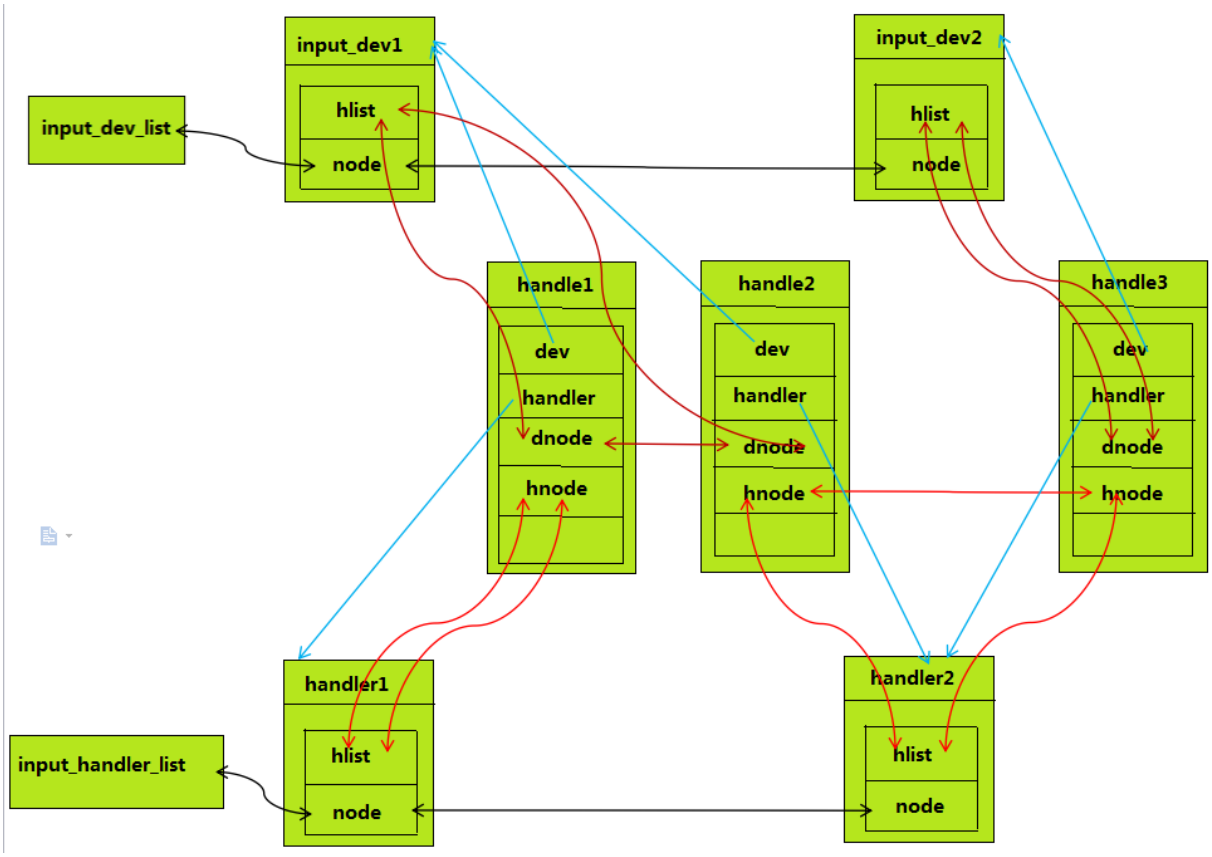
/dev/input/mouse1 都是对应鼠标这个设备。

(2)input_register_handle函数

这个函数的作用就是注册一个handle，也就是实现上图中的将各个handle连接起来构成一个环形的结构，再调用这个函数之前已经将handle中的dev和handler已经是填充好了的，

具体的这个函数代码就不去分析了。

其实handler、input_dev、handle3这之间的关系，在之前就已经接触过了，讲Linux设备驱动模型底层架构的时候遇到过，下面用一副关系图来描述他们之间的一个关系：



从本质上讲，input_dev与handler是多对多的关系，从上图可以看出来，一个input_dev可以对应多个handler，一个handler也可以对应多个input_dev。因为在匹配的时候，

一个input_dev会与所有的handler都进行匹配的，并不是匹配成功一次就退出。

从图中可以看出来，一个handle就是用来记录系统中一对匹配成功的handler和device，我们可以从这个handle出发得到handler的信息，还可以得到device的信息。所以正因为有这样的

功能，所以可以由handler经过handle最终获取到device的信息，同理也可以从device从发经过handle最终获取到handler的信息。这种运用方法将会在后面的分析中看到。

4、总结：

核心层（其实就是驱动框架）提供的服务有哪些：

(1)创建设备类、注册字符设备

(2)向设备驱动层提供注册接口

(3)提供上层handler和下层device之间的匹配函数

(4)向上层提供注册handler的接口

二、输入事件驱动层源码分析

input输入子系统的输入事件驱动层（上层）其实是由各个handler构成的，各个handler之间是属于平行关系，不存在相互调用的现象。目前用的最多是event，今天就以这个handler

为例分析他的源代码，以便对handler的实现有一定的了解，前面说到过，input输入子系统的源代码都在drivers\input\这个目录下，其中 drivers\input\evdev.c就是event

的源代码文件。

从evdev.c文件的末尾可以看到使用了module_init、module_exit这些宏，说明内核中将这部分实现为模块的方式，这其实很好理解，因为input核心层都是实现为模块的方式，而

上层是要依赖于核心层才能够注册、才能够工作的，而核心层都已经实现为模块了，那么上层不更得需要这样做吗。好了，废话不多说开始分析代码。

1、模块注册函数：

```
static int __init evdev_init(void)
{
    return input_register_handler(&evdev_handler);
}

static void __exit evdev_exit(void)
{
    input_unregister_handler(&evdev_handler);
}

static struct input_handler evdev_handler = {
    .event      = evdev_event,
    .connect    = evdev_connect,
    .disconnect = evdev_disconnect,
    .fops       = &evdev_fops,
    .minor      = EVDEV_MINOR_BASE,   /* 次设备号起始编号 64 */
    .name       = "evdev",
    .id_table   = evdev_ids,
};
```

evdev_handler变量就是本次分析的handler对应的结构体变量，变量中填充最重要的有3个：

evdev_event函数：

evdev_connect函数：

evdev_fops变量：

2、相关的数据结构

```
1 struct evdev {
2     int exist;
```

```
    int open;                               //  这个是用来作为设备被打开的计数
    int minor;                              //   handler 与 input设备匹配成功之后创建的设备对应的device的次设备号相对于基准次设备号的偏移量
    struct input_handle handle;   //   内置的一个  handle ，里面记录了匹配成功的input_dev 和 handler
    wait_queue_head_t wait;
    struct evdev_client *grab;
    struct list_head client_list;       //   用来挂接与 evdev 匹配成功的evdev_client 的一个链表头
    spinlock_t client_lock; /* protects client_list */
    struct mutex mutex;             //  互斥锁
    struct device dev;                //  这个是handler 与 input设备匹配成功之后创建的设备对应的device
};
```

```
struct evdev_client {
    struct input_event buffer[EVDEV_BUFFER_SIZE];    //  用来存放input_dev 事件的缓冲区
    int head;
    int tail;
    spinlock_t buffer_lock; /* protects access to buffer, head and tail */
    struct fasync_struct *fasync;
    struct evdev *evdev;              //   evdev 指针
    struct list_head node;            //  作为一个链表节点挂接到相应的 evdev->client_list 链表上
    struct wake_lock wake_lock;
    char name[28];            //  名字
};
```

```
struct input_event {
    struct timeval time;        //  事件发生的事件
    __u16 type;                   //  事件的类型
    __u16 code;                  //   事件的码值
    __s32 value;                  //  事件的状态
};
```

3、函数详解

(1)evdev_connect函数分析：

```
static int evdev_connect(struct input_handler *handler, struct input_dev *dev,
             const struct input_device_id *id)
{
    struct evdev *evdev;                //  定义一个 evdev 指针
    int minor;
    int error;

    for (minor = 0; minor < EVDEV_MINORS; minor++)  //  从evdev_table 数组中找到一个没有被使用的最小的数组项  最大值32
        if (!evdev_table[minor])
            break;

    if (minor == EVDEV_MINORS) {
        printk(KERN_ERR "evdev: no more free evdev devices\n");
        return -ENFILE;
    }

    evdev = kzalloc(sizeof(struct evdev), GFP_KERNEL);   //  给evdev 申请分配内存
    if (!evdev)
        return -ENOMEM;

    INIT_LIST_HEAD(&evdev->client_list);            //  初始化 evdev->client_list 链表
    spin_lock_init(&evdev->client_lock);              //  初始化自旋锁 evdev->client_lock
    mutex_init(&evdev->mutex);                          //  初始化互斥锁 evdev->mutex
    init_waitqueue_head(&evdev->wait);

    dev_set_name(&evdev->dev, "event%d", minor);  // 设置input设备的名字
    evdev->exist = 1;
    evdev->minor = minor;                                       //  input设备的次设备号的偏移量

    evdev->handle.dev = input_get_device(dev);              //  将我们传进来的 input_dev 指针存放在 evdev->handle.dev 中
    evdev->handle.name = dev_name(&evdev->dev);      //  设置 evdev -> dev 对象的名字，并且把名字赋值给 evdev->handle.name
    evdev->handle.handler = handler;          //  将我们传进来的 handler 指针存放在 handle.handler 中
    evdev->handle.private = evdev;             //  把evdev 作为handle 的私有数据

    evdev->dev.devt = MKDEV(INPUT_MAJOR, EVDEV_MINOR_BASE + minor);        //  设置 evdev->device 设备的设备号
    evdev->dev.class = &input_class;                                                    //  将 input_class 作为 evdev->device 的设备类
    evdev->dev.parent = &dev->dev;                                                  // 将input_dev  -> device 作为evdev-
```

```
>device 的父设备
38      evdev->dev.release = evdev_free;                    //  evdev -> device 设备的卸载函数
39      device_initialize(&evdev->dev);                       //  设备初始化
40
41      error = input_register_handle(&evdev->handle);        //  注册handle
42      if (error)
43          goto err_free_evdev;
44
45       error = evdev_install_chrdev(evdev);         // 安装evdev    其实就是将evdev 结构体指针存放在evdev_table数组当中   下标就是
evdev->minor
46      if (error)
47          goto err_unregister_handle;
48
49        error = device_add(&evdev->dev);        //   添加设备到系统                /sys/devices/virtual/input/input0/event0
event0就是表示建立的设备文件
50      if (error)
51          goto err_cleanup_evdev;
52
53      return 0;
54
55   err_cleanup_evdev:
56      evdev_cleanup(evdev);
57   err_unregister_handle:
58      input_unregister_handle(&evdev->handle);
59   err_free_evdev:
60      put_device(&evdev->dev);
61      return error;
62  }
```

这里搞清楚： /sys/devices/virtual/input/input0 这个设备是在注册input_dev时创建的，而input0/event0就是在handler和input_dev匹配成功之后创建的，也会在/dev/目录

下创建设备节点。

(2)evdev_open分析

```
1  static int evdev_open(struct inode *inode, struct file *file)
2  {
3      struct evdev *evdev;                          //  定义一个 evdev 结构体指针
4      struct evdev_client *client;                //   定义一个evdev_client 指针
5      int i = iminor(inode) - EVDEV_MINOR_BASE;   //  通过inode 获取 需要打开的设备对应的evdev_table 数组中的下标变量
6      int error;
7
8      if (i >= EVDEV_MINORS)
9          return -ENODEV;
10
11     error = mutex_lock_interruptible(&evdev_table_mutex);
12     if (error)
13         return error;
14     evdev = evdev_table[i];              //  从evdev_table  数组中找到evdev
15     if (evdev)
16         get_device(&evdev->dev);
17     mutex_unlock(&evdev_table_mutex);
18
19     if (!evdev)
20         return -ENODEV;
21
22     client = kzalloc(sizeof(struct evdev_client), GFP_KERNEL);      //  给 client 申请分配内存
23     if (!client) {
24         error = -ENOMEM;
25         goto err_put_evdev;
26     }
27
28     spin_lock_init(&client->buffer_lock);
29     snprintf(client->name, sizeof(client->name), "%s-%d",
30              dev_name(&evdev->dev), task_tgid_vnr(current));
31     wake_lock_init(&client->wake_lock, WAKE_LOCK_SUSPEND, client->name);
32     client->evdev = evdev;                            //  通过client->evdev 指针指向 evdev
33     evdev_attach_client(evdev, client);   //  其实这个函数就是做了一个链表挂接：  client->node  挂接到 evdev->client_list
34
35     error = evdev_open_device(evdev); //  打开 evdev 设备    最终就会打开 input_dev -> open 函数
36     if (error)
37         goto err_free_client;
38
39     file->private_data = client;              //   将evdev_client 作为file 的私有数据存在
40     nonseekable_open(inode, file);
41
42     return 0;
43
44  err_free_client:
```

```
45     evdev_detach_client(evdev, client);
46     kfree(client);
47  err_put_evdev:
48     put_device(&evdev->dev);
49     return error;
50 }
```

4、总结：

(1)其实下层可以上报的事件都在我们的内核中是定义好的，我们都可以上报这些事，但是input子系统的上层输入事件驱动层的各个handler只能够处理某一些事件（event除外），

例如joy handler只能处理摇杆类型的事件，key handler只能处理键盘，内部实现的原理就是会在核心层做handler和device匹配的过程。如果我们的上报的事件与多个handler都

能够匹配成功，那么绑定之后核心层会向这多个handler都上报事件，再由handler上报给应用层。

(2)input设备注册的流程:

下层通过调用核心层的函数来向子系统注册input输入设备

/***/

input_register_device

device_add: /sys/devices/virtual/input/input0

链表挂接: input_dev->node -------> input_dev_list

input_attach_handler // 进行input_dev和handler之间的匹配

调用handler->connect进行连接

构建evdev结构体，加入evdev_table数组

input_register_handle

device_add: /sys/devices/virtual/input/input0/event0

/***/

(3)handler注册流程

/***/

input_register_handler

input_table[handler->minor >> 5] = handler

链表挂接: handler->node -----> input_handler_list

input_attach_handler

handler->connect // 调用handler的connect函数进行连接

/***/

(4)事件如何传递到应用层

input子系统下层通过调用input_event函数项核心层上报数据

input_event

input_handle_event

input_pass_event

handler->event() // 最终会调用到handler 中的event函数

evdev_pass_event

client->buffer[client->head++] = *event; // 会将input输入事件数据存放在evdev_client结构体中的缓冲去中

当我们的应用层通过open打开event0这个设备节点时最终会调用到input_init函数中注册的字符设备input时注册的file_operations->open() 函数

```
input_open_file
    handler = input_table[iminor(inode) >> 5]
    handler->fops->open()
        evdev = evdev_table[i];
        evdev_open_device
            input_open_device
                input_dev->open()        //  最终就是执行input设备中的open函数
        file->private_data = evdev_client;
```

所以当我们在应用层调用read函数时，最终会调用到handler->fops->read函数

```
evdev_read
    evdev_fetch_next_event
        *event = client->buffer[client->tail++]      //  将evdev_client->buffer中的数据取走
    input_event_to_user
        copy_to_user                //  拷贝到用户空间
/*****************************************************************************
*****************/
```

到此为止，input输入子系统中还有设备驱动层没有说到，将会在下一篇博文中补充。。。。。。。。。。

分类: Linux驱动

涛少&
关注 - 0
粉丝 - 12
+加关注

« 上一篇：内核链表操作函数/宏
» 下一篇：触摸屏基本原理介绍

发表于 2016-11-26 14:31 涛少& 阅读(986) 评论(5) 编辑 收藏

评论

#1楼
赞
支持(0) 反对(0)
竹林海宝 评论于 2017-07-30 15:13

#2楼
请教下："input输入子系统中还有设备驱动层"这个博主分析了吗？没有找到啊！支持博主更新，写的太好了。学习！
支持(0) 反对(0)
竹林海宝 评论于 2017-07-30 15:15

#3楼[楼主]
@ 竹林海宝
谢谢啊
支持(0) 反对(0)
涛少& 评论于 2017-07-30 15:27

#4楼[楼主]
@ 竹林海宝
现在工作有点忙，暂时不会更新了
支持(0) 反对(0)

涛少& 评论于 2017-07-30 15:29

#5楼

那好可惜，找了好多，你的输入子系统写的最棒！

支持(0) 反对(0)

竹林海宝 评论于 2017-07-30 15:30

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。

最新IT新闻:
· CDR来了！回A股标准确定，符合条件的独角兽原来就这几家
· 腾讯微保上线"微出行"保险产品
· 微软Windows主管梅尔森致全体员工告别信
· 深交所向乐视发问询函：孙宏斌变卖核心资产不够还债言论请予说明
· 360电话手表X1 PRO发布：1499元
» 更多新闻...

最新知识库文章:
· 写给自学者的入门指南
· 和程序员谈恋爱
· 学会学习
· 优秀技术人的管理陷阱
· 作为一个程序员，数学对你到底有多重要
» 更多知识库文章...