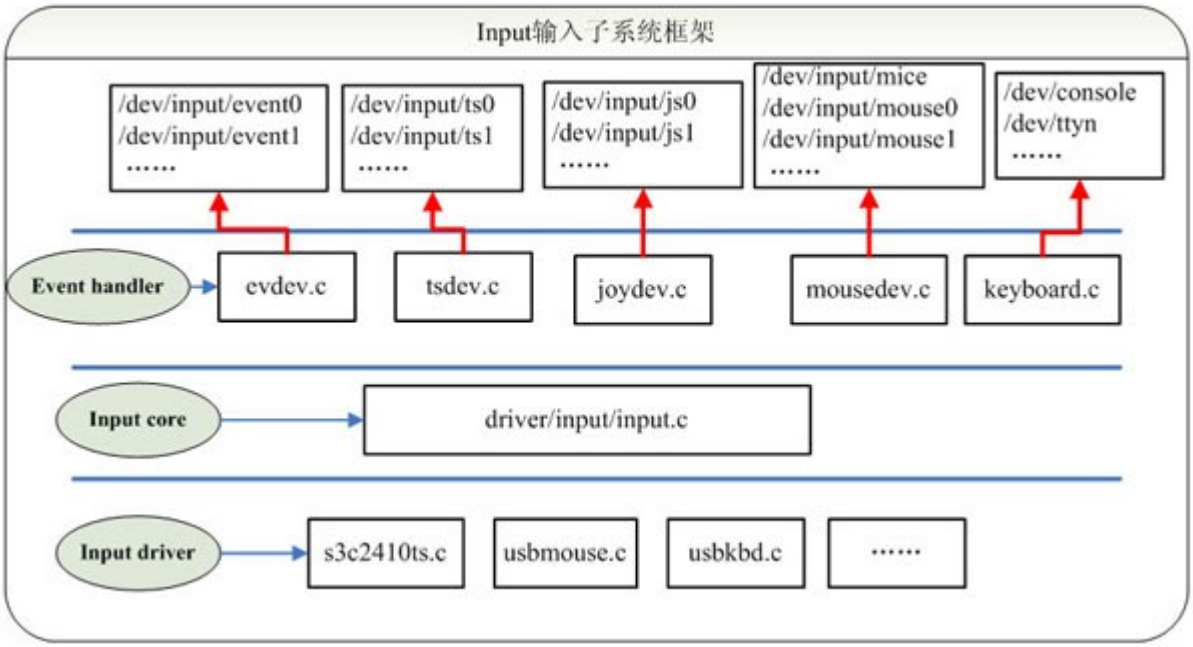
努力成为 linux kernel hacker 的人李万鹏原创作品，为梦而战。转载请标明出处
http://blog.csdn.net/woshixingaaa/archive/2011/05/19/6431094.aspx
内核的输入子系统是对分散的，多种不同类别的输入设备(如键盘，鼠标，跟踪球，操纵杆，触摸屏，加速计和手写板)等字符设备进行统一处理的一层抽象，就是在字符设备驱动上抽象出的一层。输入子系统包括两类驱动程序：事件驱动程序和设备驱动程序。事件驱动程序负责和应用程序的接口，而设备驱动程序负责和底层输入设备的通信。鼠标事件生成文件 mousedev 属于事件驱动程序，而 PS/2 鼠标驱动程序是设备驱动程序。事件驱动程序是标准的，对所有的输入类都是可用的，所以要实现的是设备驱动程序而不是事件驱动程序。设备驱动程序可以利用一个已经存在的，合适的事件驱动程序通过输入核心和用户应用程序接口。
输入子系统带来了如下好处：
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
如下图，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。



现在了解了 input 子系统的基本思想，下面来看一下 input 子系统的 3 个基本的数据结构：

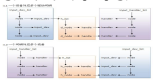
```
1. struct input_dev {
2.   const char *name;
```

```
3.    const char *phys;
4.    const char *uniq;
5.    struct input_id id;                    //与 input_handler 匹配的时会用到
6.
7.    unsigned long evbit[BITS_TO_LONGS(EV_CNT)];     //支持的所有事件类型
8.    unsigned long keybit[BITS_TO_LONGS(KEY_CNT)];   //按键事件支持的子事件
9.    unsigned long relbit[BITS_TO_LONGS(REL_CNT)];   //相对坐标事件支持的子事件
10.   unsigned long absbit[BITS_TO_LONGS(ABS_CNT)];   //绝对坐标事件支持的子事件
11.   unsigned long mscbit[BITS_TO_LONGS(MSC_CNT)];   //其他事件支持的子事件
12.   unsigned long ledbit[BITS_TO_LONGS(LED_CNT)];   //LED 灯事件支持的子事件
13.   unsigned long sndbit[BITS_TO_LONGS(SND_CNT)];   //声音事件支持的子事件
14.   unsigned long ffbit[BITS_TO_LONGS(FF_CNT)];     //受力事件支持的子事件
15.   unsigned long swbit[BITS_TO_LONGS(SW_CNT)];     //开关事件支持的子事件
16.
17.   unsigned int keycodemax;
18.   unsigned int keycodesize;
19.   void *keycode;
20.   int (*setkeycode)(struct input_dev *dev, int scancode, int keycode);
21.   int (*getkeycode)(struct input_dev *dev, int scancode, int *keycode);
22.
23.   struct ff_device *ff;
24.
25.   unsigned int repeat_key;
26.   struct timer_list timer;
27.
28.   int sync;
29.
30.   int abs[ABS_MAX + 1];            //绝对坐标上报的当前值
31.   int rep[REP_MAX + 1];            //这个参数主要是处理重复按键，后面遇到再讲
32.   unsigned long key[BITS_TO_LONGS(KEY_CNT)]; //按键有两种状态，按下和抬起，这个字段就是记录这
    两个状态。
33.   unsigned long led[BITS_TO_LONGS(LED_CNT)];
34.   unsigned long snd[BITS_TO_LONGS(SND_CNT)];
35.   unsigned long sw[BITS_TO_LONGS(SW_CNT)];
36.
37.   int absmax[ABS_MAX + 1];          //绝对坐标的最大值
38.   int absmin[ABS_MAX + 1];       //绝对坐标的最小值
39.   int absfuzz[ABS_MAX + 1];
40.   int absflat[ABS_MAX + 1];
41.
42.   int (*open)(struct input_dev *dev);
43.   void (*close)(struct input_dev *dev);
44.   int (*flush)(struct input_dev *dev, struct file *file);
45.   int (*event)(struct input_dev *dev, unsigned int type, unsigned int code, int value);
46.
```

```
47.    struct input_handle *grab;        //当前使用的 handle
48.
49.    spinlock_t event_lock;
50.    struct mutex mutex;
51.
52.    unsigned int users;
53.    int going_away;
54.
55.    struct device dev;
56.
57.    struct list_head    h_list;    //h_list 是一个链表头，用来把 handle 挂载在这个上
58.    struct list_head    node;      //这个 node 是用来连到 input_dev_list 上的
59.};
60.
61.struct input_handler {
62.
63.    void *private;
64.
65.    void (*event)(struct input_handle *handle, unsigned int type, unsigned int code, int value);
66.    int (*connect)(struct input_handler *handler, struct input_dev *dev, const struct input_devic
    e_id *id);
67.    void (*disconnect)(struct input_handle *handle);
68.    void (*start)(struct input_handle *handle);
69.
70.    const struct file_operations *fops;
71.    int minor;                          //次设备号
72.    const char *name;
73.
74.    const struct input_device_id *id_table;
75.    const struct input_device_id *blacklist;
76.
77.    struct list_head    h_list;    //h_list 是一个链表头，用来把 handle 挂载在这个上
78.    struct list_head    node;      //这个 node 是用来连到 input_handler_list 上的
79.};
80.
81.struct input_handle {
82.
83.    void *private;
84.
85.    int open;
86.    const char *name;
87.
88.    struct input_dev *dev;            //指向 input_dev
89.    struct input_handler *handler;      //指向 input_handler
90.
```

```
    struct list_head    d_node;     //连到 input_dev 的 h_list 上
    struct list_head    h_node;     //连到 input_handler 的 h_list 上
};
```

如下图代表了 input_dev，input_handler，input_handle，3 者之间的关系。一类 handler 可以和多个硬件设备相关联，一个硬件设备可以和多个 handler 相关联。例如：一个触摸屏设备可以作为一个 event 设备，作为一个鼠标设备，也可以作为一个触摸设备，所以一个设备需要与多个平台驱动进行连接。而一个平台驱动也不只为一个设备服务，一个触摸平台驱动可能要为 A,B,C3 个触摸设备提供上层驱动，所以需要这样一对多的连接。



下面来看看 input 子系统的初始化函数：

```
static int __init input_init(void)
{
    int err;

    input_init_abs_bypass();
    /*创建一个类 input_class*/
    err = class_register(&input_class);
    if (err) {
        printk(KERN_ERR "input: unable to register input_dev class/n");
        return err;
    }
    /*在/proc 下创建入口项*/
    err = input_proc_init();
    if (err)
        goto fail1;
    /*注册设备号 INPUT_MAJOR 的设备，记住 input 子系统的设备的主设备号都是 13，即 INPUT_MAJOR 为 13，并与 input_fops 相关联*/
    err = register_chrdev(INPUT_MAJOR, "input", &input_fops);
    if (err) {
        printk(KERN_ERR "input: unable to register char major %d", INPUT_MAJOR);
        goto fail2;
    }

    return 0;

fail2: input_proc_exit();
fail1: class_unregister(&input_class);
    return err;
}
subsys_initcall(input_init);
```

下面来看 input 子系统的 file_operations，这里只有一个打开函数 input_open_file，这个在事件传递部分讲解。

```
1. static const struct file_operations input_fops = {
2.     .owner = THIS_MODULE,
3.     .open = input_open_file,
4. };
```

下边来看 input_dev 设备的注册：

```
1. int input_register_device(struct input_dev *dev)
2. {
3.     static atomic_t input_no = ATOMIC_INIT(0);
4.     struct input_handler *handler;
5.     const char *path;
6.     int error;
7.
8.     __set_bit(EV_SYN, dev->evbit);
9.
10.    /*
11.     * If delay and period are pre-set by the driver, then autorepeating
12.     * is handled by the driver itself and we don't do it in input.c.
13.     */
14.
15.    init_timer(&dev->timer);
16.    /*
17.     *rep 主要是处理重复按键，如果没有定义 dev->rep[REP_DELAY]和 dev->rep[REP_PERIOD],
18.     *则将其赋值为默认值。dev->rep[REP_DELAY]是指第一次按下多久算一次，这里是 250ms，
19.     *dev->rep[REP_PERIOD]指如果按键没有被抬起，每 33ms 算一次。
20.     */
21.    if (!dev->rep[REP_DELAY] && !dev->rep[REP_PERIOD]) {
22.        dev->timer.data = (long) dev;
23.        dev->timer.function = input_repeat_key;
24.        dev->rep[REP_DELAY] = 250;
25.        dev->rep[REP_PERIOD] = 33;
26.    }
27.    /*如果 dev 没有定义 getkeycode 和 setkeycode，则赋默认值。他们的作用一个是获得键的扫描码，一个是
   设置键的扫描码*/
28.    if (!dev->getkeycode)
29.        dev->getkeycode = input_default_getkeycode;
30.
31.    if (!dev->setkeycode)
32.        dev->setkeycode = input_default_setkeycode;
33.
34.    dev_set_name(&dev->dev, "input%ld",
```

```
        (unsigned long) atomic_inc_return(&input_no) - 1);
/*将 input_dev 封装的 dev 注册到 sysfs*/
error = device_add(&dev->dev);
if (error)
    return error;

path = kobject_get_path(&dev->dev.kobj, GFP_KERNEL);
printk(KERN_INFO "input: %s as %s/n",
    dev->name ? dev->name : "Unspecified device", path ? path : "N/A");
kfree(path);

error = mutex_lock_interruptible(&input_mutex);
if (error) {
    device_del(&dev->dev);
    return error;
}
/*将 input_dev 挂在 input_dev_list 上*/
list_add_tail(&dev->node, &input_dev_list);
/*匹配所有的 input_handler，这个就是刚才那幅图里的一个设备对应多个 handler 的由来*/
list_for_each_entry(handler, &input_handler_list, node)
    input_attach_handler(dev, handler);

input_wakeup_procfs_readers();

mutex_unlock(&input_mutex);

return 0;
}
```

跟踪程序，来看看 input_attach_handler 的实现：

```
static int input_attach_handler(struct input_dev *dev, struct input_handler *handler)
{
  const struct input_device_id *id;
  int error;
  /*handler 有一个黑名单，如果存在黑名单，并且这个 id 匹配就退出*/
  if (handler->blacklist && input_match_device(handler->blacklist, dev))
    return -ENODEV;
  /*匹配 id，实现在下边可以看到*/
  id = input_match_device(handler->id_table, dev);
  if (!id)
    return -ENODEV;
  /*如果匹配，则调用具体的 handler 的 connect 函数*/
  error = handler->connect(handler, dev, id);
  if (error && error != -ENODEV)
    printk(KERN_ERR
```

```
16.         "input: failed to attach handler %s to device %s, "
17.         "error: %d/n",
18.         handler->name, kobject_name(&dev->dev.kobj), error);
19.
20.   return error;
21.}
```

下边来看看这个匹配函数：如果 id->flags 存在，并且相应的标志为被设定则进行比较。

```
1.  static const struct input_device_id *input_match_device(const struct input_device_id *id,
2.                      struct input_dev *dev)
3.  {
4.    int i;
5.
6.    for (; id->flags || id->driver_info; id++) {
7.
8.      if (id->flags & INPUT_DEVICE_ID_MATCH_BUS)
9.        if (id->bustype != dev->id.bustype)
10.         continue;
11.
12.     if (id->flags & INPUT_DEVICE_ID_MATCH_VENDOR)
13.       if (id->vendor != dev->id.vendor)
14.         continue;
15.
16.     if (id->flags & INPUT_DEVICE_ID_MATCH_PRODUCT)
17.       if (id->product != dev->id.product)
18.         continue;
19.
20.     if (id->flags & INPUT_DEVICE_ID_MATCH_VERSION)
21.       if (id->version != dev->id.version)
22.         continue;
23.
24.     MATCH_BIT(evbit,  EV_MAX);
25.     MATCH_BIT(keybit, KEY_MAX);
26.     MATCH_BIT(relbit, REL_MAX);
27.     MATCH_BIT(absbit, ABS_MAX);
28.     MATCH_BIT(mscbit, MSC_MAX);
29.     MATCH_BIT(ledbit, LED_MAX);
30.     MATCH_BIT(sndbit, SND_MAX);
31.     MATCH_BIT(ffbit,  FF_MAX);
32.     MATCH_BIT(swbit,  SW_MAX);
33.
34.     return id;
35.   }
36.
```

```
    return NULL;
}
```

```
#define MATCH_BIT(bit, max) /
        for (i = 0; i < BITS_TO_LONGS(max); i++) /
            if ((id->bit[i] & dev->bit[i]) != id->bit[i]) /
                break; /
        if (i != BITS_TO_LONGS(max)) /
            continue;
```

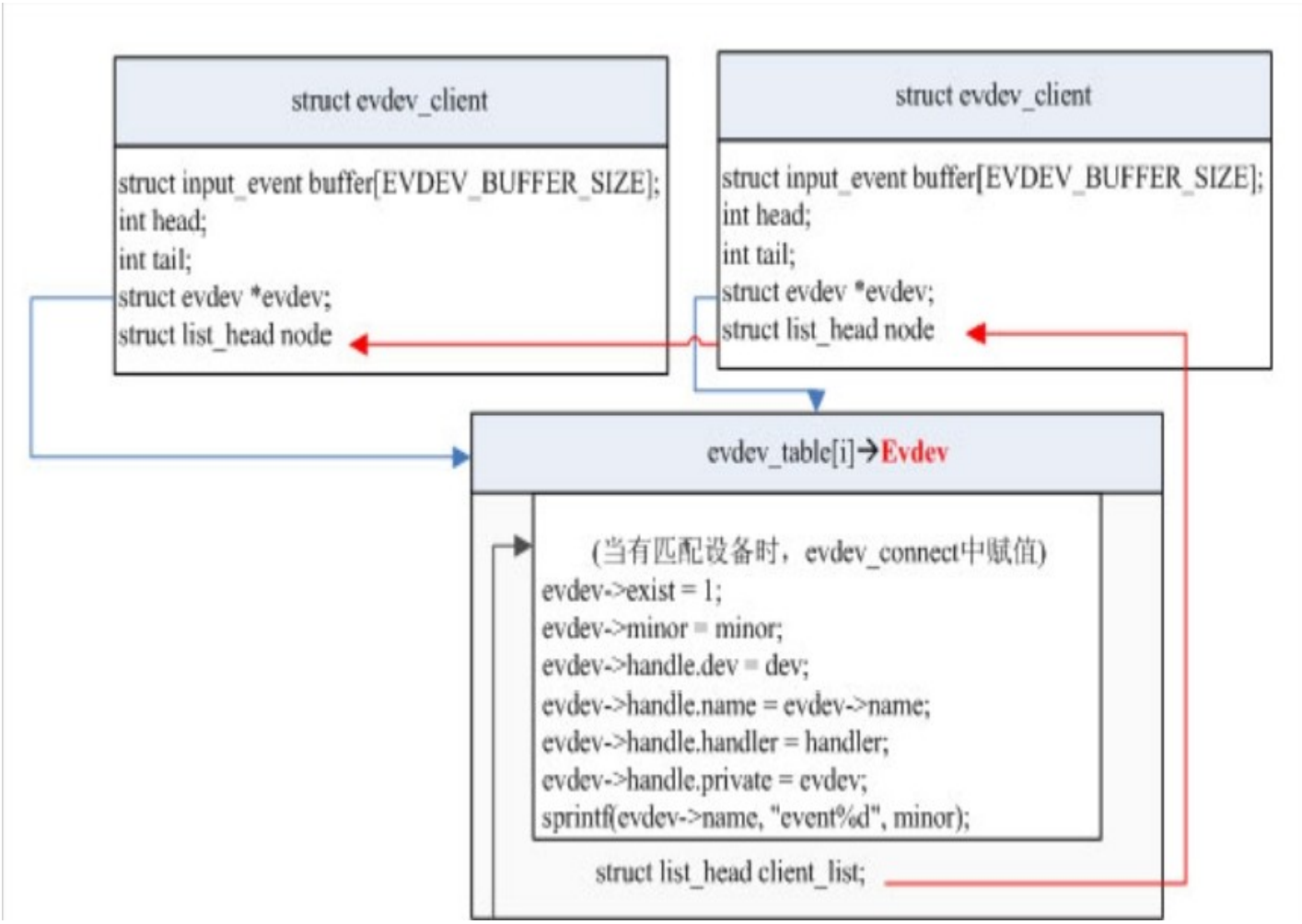
下边是刚刚看到的 connect，这里假设这个 handler 是 evdev_handler。如果匹配上了就会创建一个 evdev，它里边封装了一个 handle，会把 input_dev 和 input_handler 关联到一起。

```
/*
 * Create new evdev device. Note that input core serializes calls
 * to connect and disconnect so we don't need to lock evdev_table here.
 */
static int evdev_connect(struct input_handler *handler, struct input_dev *dev,
             const struct input_device_id *id)
{
    struct evdev *evdev;
    int minor;
    int error;
    /*
     *首先补充几个知识点：
     *static struct input_handler *input_table[8];
     *#define INPUT_DEVICES 256
     *一共有 8 个 input_handler，对应 256 个设备，所以一个 handler 对应 32 个设备。
     *这个问题在我参加的一次 linux 驱动的面试中被问到，当时真是汗啊！！！
     *static struct evdev *evdev_table[EVDEV_MINORS];
     *#define EVDEV_MINORS  32
     *evdev 理论上可对应 32 个设备，其对应的设备节点一般位于/dev/input/event0~/dev/input/event4
     *下边的 for 循环，在 evdev_table 数组中找一个未使用的地方
     */
    for (minor = 0; minor < EVDEV_MINORS; minor++)
        if (!evdev_table[minor])
            break;

    if (minor == EVDEV_MINORS) {
        printk(KERN_ERR "evdev: no more free evdev devices/n");
        return -ENFILE;
    }
    /*下边的代码是分配一个 evdev 结构体，并对成员进行初始化*/
    evdev = kzalloc(sizeof(struct evdev), GFP_KERNEL);
    if (!evdev)
        return -ENOMEM;
```

```
34.
35.  INIT_LIST_HEAD(&evdev->client_list);
36.  spin_lock_init(&evdev->client_lock);
37.  mutex_init(&evdev->mutex);
38.  init_waitqueue_head(&evdev->wait);
39.
40.  snprintf(evdev->name, sizeof(evdev->name), "event%d", minor);
41.  evdev->exist = 1;
42.  evdev->minor = minor;
43.
44.  evdev->handle.dev = input_get_device(dev);
45.  evdev->handle.name = evdev->name;
46.  evdev->handle.handler = handler;
47.  evdev->handle.private = evdev;
48.
49.  dev_set_name(&evdev->dev, evdev->name);
50.  evdev->dev.devt = MKDEV(INPUT_MAJOR, EVDEV_MINOR_BASE + minor);
51.  evdev->dev.class = &input_class;
52.  evdev->dev.parent = &dev->dev;
53.  evdev->dev.release = evdev_free;
54.  /**/
55.  device_initialize(&evdev->dev);
56.  /*
57.      *input_register_handle 完成的主要功能是:
58.      *list_add_tail_rcu(&handle->d_node, &dev->h_list);
59.   *list_add_tail(&handle->h_node, &handler->h_list);
60.   */
61.  error = input_register_handle(&evdev->handle);
62.  if (error)
63.      goto err_free_evdev;
64.  /*evdev_install_chrdev 完成的功能是 evdev_table[evdev->minor]=evdev;*/
65.  error = evdev_install_chrdev(evdev);
66.  if (error)
67.      goto err_unregister_handle;
68.
69.  error = device_add(&evdev->dev);
70.  if (error)
71.      goto err_cleanup_evdev;
72.
73.  return 0;
74.  。。。。。。。。。。
75.}
```

看一下这张图会对上边的结构有清楚的认知了：


struct evdev_client
struct input_event buffer[EVDEV_BUFFER_SIZE];
int head;
int tail;
struct evdev *evdev;
struct list_head node
struct evdev_client
struct input_event buffer[EVDEV_BUFFER_SIZE];
int head;
int tail;
struct evdev *evdev;
struct list_head node
evdev_table[i]→Evdev
(当有匹配设备时，evdev_connect中赋值)
evdev->exist = 1;
evdev->minor = minor;
evdev->handle.dev = dev;
evdev->handle.name = evdev->name;
evdev->handle.handler = handler;
evdev->handle.private = evdev;
sprintf(evdev->name, "event%d", minor);
struct list_head client_list;