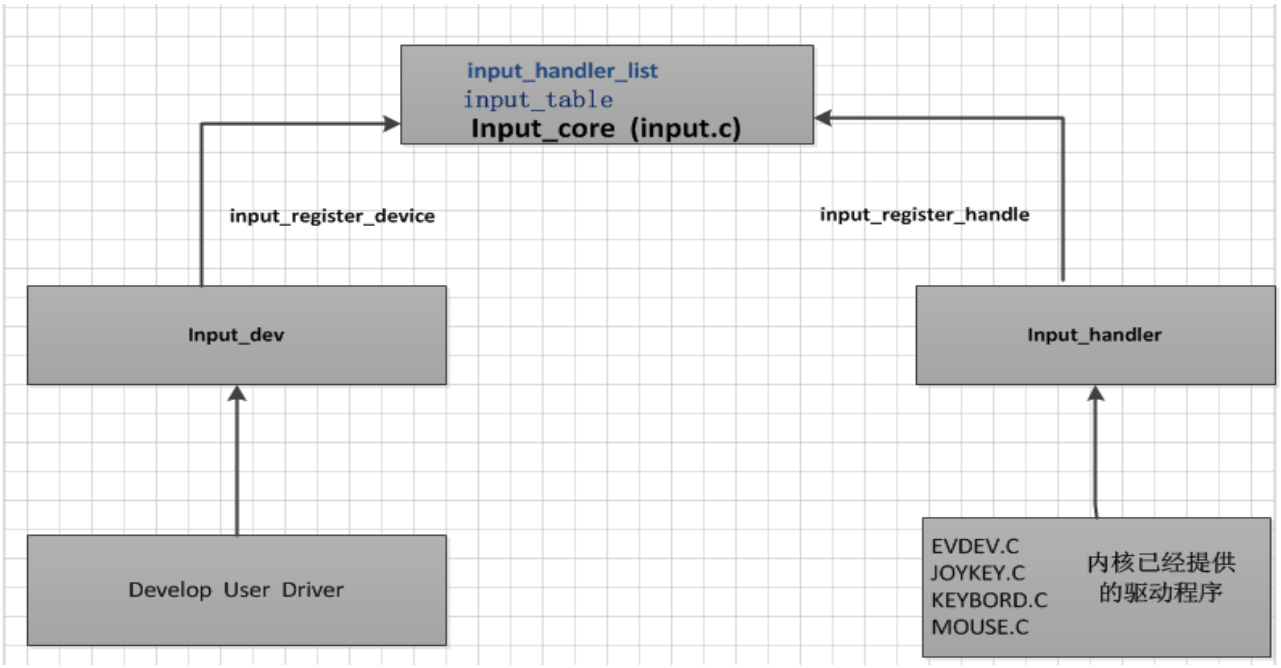
Linux 输入子系统

1. 系统框架图结构



2.驱动机制分析：

输入子系统将输入设备的驱动程序按类型按照固定的主设备号在内核中进行实现。此时的 driver 的编写将按照固定的格式进行输入设备的编写，这样做的目的，采用统一的设备节点对应相同类型的输入设备，在应用层进行设备的统一。不同的地方只需要进行设备 IDtabe 的设置进行相应的配置，之后进行输入设备的注册。自动按照设备类型进行输入设备的匹配，自动的创建设备节点进行操作。下边是设备端程序的编写，分配，构造，注册之后，会生成相对应类型的设备节点

```
/* 1. 分配一个input_dev结构体 */
buttons_dev = input_allocate_device();;

/* 2. 设置 */
/* 2.1 能产生哪类事件 */
set_bit(EV_KEY, buttons_dev->evbit);
set_bit(EV_REP, buttons_dev->evbit);

/* 2.2 能产生这类操作里的哪些事件: L,S,ENTER,LEFTSHIT */
set_bit(KEY_L, buttons_dev->keybit);
set_bit(KEY_S, buttons_dev->keybit);
set_bit(KEY_ENTER, buttons_dev->keybit);
set_bit(KEY_LEFTSHIFT, buttons_dev->keybit);

/* 3. 注册 */
input_register_device(buttons_dev);
```

其中的 ev->evbit ，设置的是设备产生的输入类型，主要用于设备产生的事件类型，用于与 driver 的事件进行匹配，若匹配才能产生设备节点。

内核驱动的层次结构和驱动实现流程：见 evdev.c

```
static struct input_handler evdev_handler = {
    .event      = evdev_event,
    .connect    = evdev_connect,
    .disconnect = evdev_disconnect,
    .fops       = &evdev_fops,
    .minor      = EVDEV_MINOR_BASE,
    .name       = "evdev",
    .id_table   = evdev_ids,
};

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

内核设备注册一个 handler 对应产生的设备节点在/dev/input/event0~/dev/input/event..,这些设备节点，如何在系统中产生这些节点并读取这些节点产生的设备输入。

（1）打开设备分析：

不同类型的输入设备的次设备号是系统写死分配的，这样在每次进行

匹配的时候，按照基础子设备号进行依次的累加，进行创建不同的设备信息。那么打开的时候所打开的设备的 open 入口在哪里呢？系统所有的输入设备字符设备的主设备号都是 13 那么在最顶层的 input.c 中进行打开的，跳到 input.c 中的 open 函数

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

```
    err = register_chrdev(INPUT_MAJOR, "input", &input_fops);
    if (err) {
        printk(KERN_ERR "input: unable to register char major %d", INPUT_MAJOR);
        goto fail2;
    }
```

```
  /* No load-on-demand here? */
  handler = input_table[iminor(inode) >> 5];
  if (handler)
      new_fops = fops_get(handler->fops);
```

在主设备好为 13 的主设备中进行设备节点的打开，输入节点的打开函数并没有直接打开设备的节点，而是在 input_table 中按照设备节点的次设备号进行查找 handler,在 handler 的 fileoperation 进行数据 fileoperation 的替换，那么 input_table 是在什么时候进行设置和注册的呢？是在 evdev 调用 inpit_handler 进行注册的，注意此时并没有进行设备的创建。系统初始化的时候将不同类型的输入设备按照不同区域的此设备号进行划分，但是其方法还是按照 32 区域》》5 放置在全局的 input_table 中。而设备节点是在双方都注册匹配之后进行创建的。

open--->handler_open ，open 其实是调用的 handlerde open 函数。

在 evdev.c 中其实是这样的一个过程：

```
static ssize_t evdev_read(struct file *file, char __user *buffer,
              size_t count, loff_t *ppos)
{
    struct evdev_client *client = file->private_data;
    struct evdev *evdev = client->evdev;
    struct input_event event;
    int retval;

    if (count < input_event_size())
        return -EINVAL;

    if (client->head == client->tail && evdev->exist &&
        (file->f_flags & O_NONBLOCK))
        return -EAGAIN;

    retval = wait_event_interruptible(evdev->wait,
        client->head != client->tail || !evdev->exist);
    if (retval)
        return retval;

    if (!evdev->exist)
        return -ENODEV;

    while (retval + input_event_size() <= count &&
           evdev_fetch_next_event(client, &event)) {

        if (input_event_to_user(buffer + retval, &event))
            return -EFAULT;

        retval += input_event_size();
    }
```

读取自身的环形缓冲区，有数据的时候就读取返回，没有数据的时候按照打开设备的方式进行休眠，那么在哪进行的数据放入，和读取设备节点的唤醒呢？

当然是在设备端有数据的时候进行的上报了，且看上报函数：

```
void input_event(struct input_dev *dev,
         unsigned int type, unsigned int code, int value)
{
    unsigned long flags;

    if (is_event_supported(type, dev->evbit, EV_MAX)) {

        spin_lock_irqsave(&dev->event_lock, flags);
        add_input_randomness(type, code, value);
        input_handle_event(dev, type, code, value);
        spin_unlock_irqrestore(&dev->event_lock, flags);
    }
```

```
static void input_pass_event(struct input_dev *dev,
                 unsigned int type, unsigned int code, int value)
{
    struct input_handler *handler;
    struct input_handle *handle;

    rcu_read_lock();

    handle = rcu_dereference(dev->grab);
    if (handle)
        handle->handler->event(handle, type, code, value);
    else {
        bool filtered = false;

        list_for_each_entry_rcu(handle, &dev->h_list, d_node) {
            if (!handle->open)
                continue;

            handler = handle->handler;
            if (!handler->filter) {
                if (filtered)
                    break;

                handler->event(handle, type, code, value);

            } else if (handler->filter(handle, type, code, value))
                filtered = true;
        }
    }
```

通过建立两端的中间件 handle 找到对应的 handler，调用对应的 handler 的 event 函数，且分析 evdev.c 的 handiler event 函数，

```
static void evdev_event(struct input_handle *handle,
            unsigned int type, unsigned int code, int value)
{
    struct evdev *evdev = handle->private;
    struct evdev_client *client;
    struct input_event event;
    struct timespec ts;

    ktime_get_ts(&ts);
    event.time.tv_sec = ts.tv_sec;
    event.time.tv_usec = ts.tv_nsec / NSEC_PER_USEC;

    event.type = type;
    event.code = code;
    event.value = value;

    rcu_read_lock();

    client = rcu_dereference(evdev->grab);
    if (client)
        evdev_pass_event(client, &event);
    else
        list_for_each_entry_rcu(client, &evdev->client_list, node)
            evdev_pass_event(client, &event);

    rcu_read_unlock();

    wake_up_interruptible(&evdev->wait);
}
```

且看到 wait_up_intrrupttible 函数。这个函数环形读取的设备读取函数。另在 evdev_pass_event()中按照 event 的数据格式进行放置数据。

设备节点是怎样建立的和其中的关系：

```
int input_register_handler(struct input_handler *handler)
{
    struct input_dev *dev;
    int retval;

    retval = mutex_lock_interruptible(&input_mutex);
    if (retval)
        return retval;

    INIT_LIST_HEAD(&handler->h_list);

    if (handler->fops != NULL) {
        if (input_table[handler->minor >> 5]) {
            retval = -EBUSY;
            goto out;
        }
        input_table[handler->minor >> 5] = handler;
    }

    list_add_tail(&handler->node, &input_handler_list);

    list_for_each_entry(dev, &input_dev_list, node)
        input_attach_handler(dev, handler);

    input_wakeup_procfs_readers();

 out:
    mutex_unlock(&input_mutex);
```

每次进行注册的时候，就会进行 handler 和 dev 进行匹配，这样在进行 match idtable 之后就会调用 handler connect 函数。

```
    id = input_match_device(handler, dev);
    if (!id)
        return -ENODEV;

    error = handler->connect(handler, dev, id);
    if (error && error != -ENODEV)
```

其进行分析其中的 evdev handler 的 connect 函数：

```
 */
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;

    for (minor = 0; minor < EVDEV_MINORS; minor++)
        if (!evdev_table[minor])
            break;
```

按照 evdev_table 的不同次设备号进行放置，另设置名字信息，进行创建设备节点（设备节点的创建见 plateform 驱动）

```
dev_set_name(&evdev->dev, "event%d", minor);
evdev->exist = 1;
evdev->minor = minor;

evdev->handle.dev = input_get_device(dev);
evdev->handle.name = dev_name(&evdev->dev);
evdev->handle.handler = handler;
evdev->handle.private = evdev;

evdev->dev.devt = MKDEV(INPUT_MAJOR, EVDEV_MINOR_BASE + minor);
evdev->dev.class = &input_class;
evdev->dev.parent = &dev->dev;
evdev->dev.release = evdev_free;
device_initialize(&evdev->dev);

error = input_register_handle(&evdev->handle);
if (error)
    goto err_free_evdev;

error = evdev_install_chrdev(evdev);
if (error)
    goto err_unregister_handle;

error = device_add(&evdev->dev);
if (error)
```

到此整个流程已经分析完毕。