Input 子系统详解

http://blog.csdn.net/ylyuanlu/article/details/6704744

一. Input 子系统架构

Linux 系统提供了 input 子系统，按键、触摸屏、键盘、鼠标等输入都可以利用 input 接口函数来实现设备驱动，下面是 Input 子系统架构：

Input 子系统架构

二. Input 系统的组成

输入子系统由驱动层（Drivers），输入子系统核心层（ Input Core ）和事件处理层（Event Handler）三部份组成。一个输入事件，如鼠标移动，键盘按键按下等都是通过 Driver -> InputCore -> Eventhandler -> userspace 的顺序到达用户空间传给应用程序。下面介绍各部分的功能：

（1）驱动层功能：负责和底层的硬件设备打交道，将底层硬件设备对用户输入的响应转换为标准的输入事件以后再向上发送给输入子系统核心层（Input Core）。

（2）Input 系统核心层：Input Core 即 Input Layer，由 driver/input/input.c 及相关头文件实现，它对下提供了设备驱动层的接口，对上提供了事件处理层（Event Handler）的编程接口。

（3）事件处理层将硬件设备上报的事件分发到用户空间和内核。

三. Input 设备驱动编写

在 Linux 内核中，input 设备用 input_dev 结构体描述，使用 input 子系统实现输入设备驱动的时候，驱动的核心工作是向系统报告按键、触摸屏、键盘、鼠标等输入事件（event，通过 input_event 结构体描述），不再需要关心文件操作接口，因为 input 子系统已经完成了文 件操作接口。驱动报告的事件经过 InputCore 和 Eventhandler 最终到达用户空间。下面给出一个使用 input 子系统的例子，通过这个例子来解析 input 子系统的方方面面。

（1）键盘驱动

```
static void button_interrupt(int irq, void *dummy, struct pt_regs *fp)
{

input_report_key(&button_dev, BTN_1, inb(BUTTON_PORT) & 1);

input_sync(&button_dev);
}
```

```
static int __init button_init(void)
{

      if (request_irq(BUTTON_IRQ, button_interrupt, 0, "button", NULL)) {
       printk(KERN_ERR "button.c: Can''t allocate irq %d\n", button_irq);
       return -EBUSY;
       }

button_dev.evbit[0] = BIT(EV_KEY);

button_dev.keybit[LONG(BTN_0)] = BIT(BTN_0);

input_register_device(&button_dev);
}



static void __exit button_exit(void)
{
input_unregister_device(&button_dev);

free_irq(BUTTON_IRQ, button_interrupt);
}

module_init(button_init);
module_exit(button_exit);
```

这是个最简单使用 input 子系统的例子，权且引出这 input 子系统，这个驱动中主要涉及 input 子系统的函数下面一一列出，后面会有详细的介绍：

1)set_bit(EV_KEY, button_dev.evbit);

set_bit(BTN_0, button_dev.keybit);

分别用来设置设备所产生的事件以及上报的按键值。Struct iput_dev 中有两个成员，一个是 evbit.一个是 keybit，分别用表示设备所支持的动作和按键类型。

2)input_register_device(&button_dev);

用来注册一个 input device.

3) input_report_key()

用于给上层上报一个按键动作

4）input_sync()

用来告诉上层，本次的事件已经完成了.

5) input_unregister_device ()

用来注销一个 input_dev 设备

四. Input 子系统探幽

（1）input 设备注册分析

Input 设备注册的接口为：input_register_device()。代码如下：

```
int input_register_device(struct input_dev *dev)
{
static atomic_t input_no = ATOMIC_INIT(0);
struct input_handler *handler;
const char *path;
int error;

__set_bit(EV_SYN, dev->evbit);



init_timer(&dev->timer);
if (!dev->rep[REP_DELAY] && !dev->rep[REP_PERIOD]) {
dev->timer.data = (long) dev;
dev->timer.function = input_repeat_key;
dev->rep[REP_DELAY] = 250;
dev->rep[REP_PERIOD] = 33;
}
```

在 前面的分析中曾分析过。Input_device 的 evbit 表示该设备所支持的事件。在这里将其 EV_SYN 置位，即所有设备都支持这个事件。如果 dev->rep[REP_DELAY]和 dev->rep[REP_PERIOD]没有设值，则将其赋默认值。这主要是处理重复按键的。

```
if (!dev->getkeycode)
dev->getkeycode = input_default_getkeycode;

if (!dev->setkeycode)
dev->setkeycode = input_default_setkeycode;

snprintf(dev->dev.bus_id, sizeof(dev->dev.bus_id),
"input%ld", (unsigned long) atomic_inc_return(&input_no) - 1);
```

```
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;
}
```

如 果 input device 没有定义 getkeycode 和 setkeycode.则将其赋默认值。还记得在键盘驱动中的分析吗?这两个操作函数就可以用来取键的扫描码 和设置键的扫描码。然后调用 device_add()将 input_dev 中封装的 device 注册到 sysfs。

```
list_add_tail(&dev->node, &input_dev_list);

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

input_wakeup_procfs_readers();

mutex_unlock(&input_mutex);

return 0;
}
```

这 里就是重点了，将 input device 挂到 input_dev_list 链表上.然后，对每一个挂在 input_handler_list 的 handler 调用 input_attach_handler().在这里的情况有好比设备模型中的 device 和 driver 的匹配。所有的 input device 都挂在 input_dev_list 链上。所有的 handler 都挂在 input_handler_list 上。

看一下这个匹配的详细过程。匹配是在 input_attach_handler()中完成的。代码如下：

```
static int input_attach_handler(struct input_dev *dev, struct input_handler *handler)
{
const struct input_device_id *id;
int error;

if (handler->blacklist && input_match_device(handler->blacklist, dev))
```

```
return -ENODEV;

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

error = handler->connect(handler, dev, id);
if (error && error != -ENODEV)
printk(KERN_ERR
"input: failed to attach handler %s to device %s, "
"error: %d\n",

handler->name, kobject_name(&dev->dev.kobj), error);

return error;
}
```

如 果handle的blacklist被赋值。要先匹配blacklist中的数据跟dev->id的数据是否匹配。匹配成功过后再来匹配 handle->id 和 dev->id 中的数据。如果匹配成功，则调用 handler->connect()。

来看一下具体的数据匹配过程，这是在 input_match_device()中完成的。代码如下：

```
static const struct input_device_id *input_match_device(const struct input_device_id
*id,
struct input_dev *dev)
{
int i;

for (; id->flags || id->driver_info; id++) {

if (id->flags & INPUT_DEVICE_ID_MATCH_BUS)
if (id->bustype != dev->id.bustype)
continue;

if (id->flags & INPUT_DEVICE_ID_MATCH_VENDOR)
if (id->vendor != dev->id.vendor)
continue;

if (id->flags & INPUT_DEVICE_ID_MATCH_PRODUCT)
if (id->product != dev->id.product)
continue;

if (id->flags & INPUT_DEVICE_ID_MATCH_VERSION)
if (id->version != dev->id.version)
```

```
continue;

MATCH_BIT(evbit, EV_MAX);
MATCH_BIT(,, KEY_MAX);
MATCH_BIT(relbit, REL_MAX);
MATCH_BIT(absbit, ABS_MAX);
MATCH_BIT(mscbit, MSC_MAX);
MATCH_BIT(ledbit, LED_MAX);
MATCH_BIT(sndbit, SND_MAX);
MATCH_BIT(ffbit, FF_MAX);
MATCH_BIT(swbit, SW_MAX);

return id;
}

return NULL;
}
```

MATCH_BIT 宏的定义如下：

```
#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;
```

由 此看到。在 id->flags 中定义了要匹配的项。定义 INPUT_DEVICE_ID_MATCH_BUS。则是要比较 input device 和 input handler 的总线类型。 INPUT_DEVICE_ID_MATCH_VENDOR, INPUT_DEVICE_ID_MATCH_PRODUCT, INPUT_DEVICE_ID_MATCH_VERSION 分别要求设备厂商。设备号和设备版本。如果 id->flags 定义的类型匹配成功。或者是 id->flags 没有定义，就会进入到 MATCH_BIT 的匹配项了。

从 MATCH_BIT 宏的定义可以看出。只有当 iput device 和 input handler 的 id 成员在 evbit, keybit,… swbit 项相同才会匹配成功。而且匹配的顺序是从 evbit, keybit 到 swbit.只要有一项不同，就会循环到 id 中的下一项进行比较。简而言之,注册 input device 的过程就是为 input device 设置默认值，并将其挂以 input_dev_list.与挂载在 input_handler_list 中的 handler 相匹配。如果匹配成 功，就会调用 handler 的 connect 函数.

（2）handler 注册分析

Handler 注册的接口如下所示:

```
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);
return retval;
}
```

handler->minor 表示对应 input 设备节点的次设备号.以 handler->minor 右移五位做为索引值插入到 input_table[ ]中..之后再来分析 input_talbe[ ]的作用。

然后将 handler 挂到 input_handler_list 中.然后将其与挂在 input_dev_list 中的 input device 匹配.这个过程和 input device 的注册有相似的地方.都是注册到各自的链表，然后与另外一条链表的对象相匹配.

（3）handle 的注册

```
int input_register_handle(struct input_handle *handle)
{

struct input_handler *handler = handle->handler;
struct input_dev *dev = handle->dev;
int error;
```

```
/*
* We take dev->mutex here to prevent race with
* input_release_device().
*/
error = mutex_lock_interruptible(&dev->mutex);
if (error)
return error;
list_add_tail_rcu(&handle->d_node, &dev->h_list);
mutex_unlock(&dev->mutex);
synchronize_rcu();

list_add_tail(&handle->h_node, &handler->h_list);

if (handler->start)
handler->start(handle);

return 0;
}
```

在这个函数里所做的处理其实很简单.将 handle 挂到所对应 input device 的 h_list 链表上.还将 handle 挂到对应的 handler 的 hlist 链表上.如果 handler 定义了 start 函数,将调用之。

到这里,我们已经看到了 input device, handler 和 handle 是怎么关联起来的了。

（4）event 事件的处理

我们在开篇的时候曾以 linux kernel 文档中自带的代码作分析.提出了几个事件上报的 API.这些 API 其实都是 input_event()的封装，代码如下:

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

//判断设备是否支持这类事件
if (is_event_supported(type, dev->evbit, EV_MAX)) {

spin_lock_irqsave(&dev->event_lock, flags);
//利用键盘输入来调整随机数产生器
add_input_randomness(type, code, value);
input_handle_event(dev, type, code, value);
spin_unlock_irqrestore(&dev->event_lock, flags);
}
}
```

先判断设备产生的这个事件是否合法.如果合法,流程转入到 input_handle_event()中，代码如下:

```
static void input_handle_event(struct input_dev *dev,
unsigned int type, unsigned int code, int value)
{
int disposition = INPUT_IGNORE_EVENT;

switch (type) {

case EV_SYN:
switch (code) {
case SYN_CONFIG:
disposition = INPUT_PASS_TO_ALL;
break;

case SYN_REPORT:
if (!dev->sync) {
dev->sync = 1;
disposition = INPUT_PASS_TO_HANDLERS;
}
break;
}
break;

case EV_KEY:
//判断按键值是否被支持
if (is_event_supported(code, dev->keybit, KEY_MAX) &&
!!test_bit(code, dev->key) != value) {

if (value != 2) {
__change_bit(code, dev->key);
if (value)
input_start_autorepeat(dev, code);
}

disposition = INPUT_PASS_TO_HANDLERS;
}
break;

case EV_SW:
if (is_event_supported(code, dev->swbit, SW_MAX) &&
!!test_bit(code, dev->sw) != value) {
```

```
__change_bit(code, dev->sw);
disposition = INPUT_PASS_TO_HANDLERS;
}
break;

 

case EV_ABS:

if (is_event_supported(code, dev->absbit, ABS_MAX)) {

value = input_defuzz_abs_event(value,
dev->abs[code], dev->absfuzz[code]);

if (dev->abs[code] != value) {
dev->abs[code] = value;
disposition = INPUT_PASS_TO_HANDLERS;
}
}
break;

case EV_REL:
if (is_event_supported(code, dev->relbit, REL_MAX) && value)
disposition = INPUT_PASS_TO_HANDLERS;

break;

case EV_MSC:
if (is_event_supported(code, dev->mscbit, MSC_MAX))
disposition = INPUT_PASS_TO_ALL;

break;

case EV_LED:
if (is_event_supported(code, dev->ledbit, LED_MAX) &&
!!test_bit(code, dev->led) != value) {

__change_bit(code, dev->led);
disposition = INPUT_PASS_TO_ALL;
}
break;

case EV_SND:
if (is_event_supported(code, dev->sndbit, SND_MAX)) {
```

```
if (!!test_bit(code, dev->snd) != !!value)
__change_bit(code, dev->snd);
disposition = INPUT_PASS_TO_ALL;
}
break;

case EV_REP:
if (code <= REP_MAX && value >= 0 && dev->rep[code] != value) {
dev->rep[code] = value;
disposition = INPUT_PASS_TO_ALL;
}
break;

case EV_FF:
if (value >= 0)
disposition = INPUT_PASS_TO_ALL;
break;

case EV_PWR:
disposition = INPUT_PASS_TO_ALL;
break;
}

if (type != EV_SYN)
dev->sync = 0;

if ((disposition & INPUT_PASS_TO_DEVICE) && dev->event)
dev->event(dev, type, code, value);

if (disposition & INPUT_PASS_TO_HANDLERS)
input_pass_event (dev, type, code, value);
}
```

在 这里，我们忽略掉具体事件的处理.到最后,如果该事件需要 input device 来完成的,就会将 disposition 设置成 INPUT_PASS_TO_DEVICE.如果需要 handler 来完成的,就将 dispostion 设为 INPUT_PASS_TO_HANDLERS.如果需要两者都参与，将 disposition 设置为 INPUT_PASS_TO_ALL。

需要输入设备参与的,回调设备的 event 函数.如果需要 handler 参与的.调用 input_pass_event().代码如下:

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

```
rcu_read_lock();

handle = rcu_dereference(dev->grab);
if (handle)
handle->handler->event(handle, type, code, value);
else
list_for_each_entry_rcu(handle, &dev->h_list, d_node)
if (handle->open)
handle->handler->event(handle,type, code, value);
rcu_read_unlock();
}
```

如果 input device 被强制指定了 handler,则调用该 handler 的 event 函数。结合 handle 注册的分析，我们知道会将 handle 挂到 input device 的 h_list 链表上。如 果没有为 input device 强制指定 handler.就会遍历 input device->h_list 上的 handle 成员.如果该 handle 被打开,则调用与输入设备对应的 handler 的 event()函数.注 意,只有在 handle 被打开的情况下才会接收到事件。

另外,输入设备的 handler 强制设置一般是用带 EVIOCGRAB 标志的 ioctl 来完成的.如下是发图的方示总结 evnet 的处理过程。我们 已经分析了 input device,handler 和 handle 的注册过程以及事件的上报和处理.下面以 evdev 为实例做分析.来贯穿理解一下整个过程.

五. evdev 概述

Evdev 对应的设备节点一般位于/dev/input/event0 ~ /dev/input/event4.理论上可以对应 32 个设备节点.分别代表被 handler 匹配的 32 个 input device。可以用 cat /dev/input/event0.然后移动鼠标或者键盘按键就会有数据输出(两者之间只能选一.因为一个设备文件只能关能一个输入设备).还可以往这 个文件里写数据,使其产生特定的事件.这个过程我们之后再详细分析。为了分析这一过程,必须从 input 子系统的初始化说起。

(1) input 子系统的初始化

Input 子系统的初始化函数为 input_init().代码如下：

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

err = class_register(&input_class);
if (err) {
printk(KERN_ERR "input: unable to register input_dev class\n");
return err;
}
```

```
err = input_proc_init();
if (err)
goto fail1;

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;
}
```

在这个初始化函数里,先注册了一个名为”input”的类.所有 input device 都属于这个类.在 sysfs 中表现就是. 所有 input device 所代表的目录都位于/dev/class/input 下面，然后调用 input_proc_init()在/proc 下面建立相关的交互文 件，再后调用 register_chrdev()注册了主设备号为 INPUT_MAJOR(13).次设备号为 0~255 的字符设备.它的操作指针为 input_fops.

在这里,我们看到.所有主设备号 13 的字符设备的操作最终都会转入到 input_fops 中.在前面分析的/dev/input/event0~ /dev/input/event4 的主设备号为 13.操作也不例外的落在了 input_fops 中。Input_fops 定义如下:

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

打开文件所对应的操作函数为 input_open_file.代码如下示:

```
static int input_open_file(struct inode *inode, struct file *file)
{
struct input_handler *handler = input_table[iminor(inode) >> 5];
const struct file_operations *old_fops, *new_fops = NULL;
int err;


if (!handler || !(new_fops = fops_get(handler->fops)))
return -ENODEV;
```

iminor(inode) 为打开文件所对应的次设备号。input_table 是一个 struct input_handler 全局数组，在这里它先设备结点的次设备号右移 5 位做为索引值到 input_table 中取对应项。

从这里我们也可以看到.一 个 handle 代表 1<<5 个设备节点(因为在 input_table 中取值是以次备号右移 5 位为索引的.即低 5 位相同的次备号对应的是同一 个索引)。在这里,终于看到了 input_talbe 大显身手的地方了， input_talbe[ ]中取值和 input_talbe[ ]的赋值,这两个过程是相对应的.在 input_table 中找到对应的 handler 之后,就会检验这个 handle 是否存,是否带有 fops 文件操作 集.如果没有.则返回一个设备不存在的错误.

```
if (!new_fops->open) {
fops_put(new_fops);
return -ENODEV;
}
old_fops = file->f_op;
file->f_op = new_fops;

err = new_fops->open(inode, file);

if (err) {
fops_put(file->f_op);
file->f_op = fops_get(old_fops);
}
fops_put(old_fops);
return err;
}
```

然后将 handler 中的 fops 替换掉当前的 fops.如果新的 fops 中有 open()函数,则调用它。

（2）evdev 的初始化

Evdev 的模块初始化函数为 evdev_init().代码如下:

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

它调用了 input_register_handler 注册了一个 handler， 注意在这里 evdev_handler 中定义的 minor 为 EVDEV_MINOR_BASE(64)。也就是说 evdev_handler 所表示的设备文件范围为(13,64)到(13,64+32)。

从之前的分析我们知道.匹配成功的关键在于 handler 中的 blacklist 和 id_talbe.
Evdev_handler 的 id_table 定义如下:

```
static const struct input_device_id evdev_ids[] = {
{ .driver_info = 1 },
{ },
};
```

它没有定义 flags.也没有定义匹配属性值.这个 handler 是匹配所有 input device 的.从前面的分析我们知道.匹配成功之后会调用 handler->connect 函数。在 Evdev_handler 中,该成员函数如下所示:

```
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;

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

EVDEV_MINORS 定义为 32.表示 evdev_handler 所表示的 32 个设备文件.evdev_talbe 是一个 struct evdev 类型的数组.struct evdev 是模块使用的封装结构.在接下来的代码中我们可以看到这个结构的使用。这一段代码的在 evdev_talbe 找到为空的那一项.minor 就是 数组中第一项为空的序号。

```
evdev = kzalloc(sizeof(struct evdev), GFP_KERNEL);
if (!evdev)
return -ENOMEM;

INIT_LIST_HEAD(&evdev->client_list);
spin_lock_init(&evdev->client_lock);

mutex_init(&evdev->mutex);
init_waitqueue_head(&evdev->wait);

snprintf(evdev->name, sizeof(evdev->name), "event%d", minor);
evdev->exist = 1;
evdev->minor = minor;

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

接 下来,分配了一个 evdev 结构,并对这个结构进行初始化.在这里我们可以看到,这个结构封装了一个 handle 结构,这结构与我们之前所讨论的 handler 是不相同的.注意有一个字母的差别哦.我们可以把 handle 看成是 handler 和 input device 的信息集合体.在这个结构里集合了匹配成功的 handler 和 input device。

```
strlcpy(evdev->dev.bus_id, evdev->name, sizeof(evdev->dev.bus_id));
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);
```

在这段代码里主要完成 evdev 封装的 device 的初始化.注意在这里,使它所属的类指向 input_class.这样在/sysfs 中创建的设备目录就会在/sys/class/input/下面显示。

```
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)
goto err_cleanup_evdev;

return 0;

err_cleanup_evdev:
evdev_cleanup(evdev);
err_unregister_handle:
input_unregister_handle(&evdev->handle);
err_free_evdev:
put_device(&evdev->dev);
return error;
}
```

注册 handle,如果是成功的,那么调用 evdev_install_chrdev 将 evdev_table 的 minor 项指向 evdev. 然后将 evdev->device 注册到 sysfs.如果失败,将进行相关的错误处理。万事俱备了,但是要接收事件，还是要打开相应的 handle，这个打开过程是在文件的 open()中完成的。

（3）evdev 设备结点的 open()操作

我们知道.对主设备号为 INPUT_MAJOR 的设备节点进行操作,会将操作集转换成 handler 的操作集，在 evdev 中,这个操作集就是 evdev_fops 对应的 open 函数如下示:

```
static int evdev_open(struct inode *inode, struct file *file)
{
struct evdev *evdev;
struct evdev_client *client;
int i = iminor(inode) - EVDEV_MINOR_BASE;
int error;

if (i >= EVDEV_MINORS)
return -ENODEV;

error = mutex_lock_interruptible(&evdev_table_mutex);
if (error)
return error;
evdev = evdev_table[i];
if (evdev)
get_device(&evdev->dev);
mutex_unlock(&evdev_table_mutex);

if (!evdev)
return -ENODEV;

client = kzalloc(sizeof(struct evdev_client), GFP_KERNEL);
if (!client) {
error = -ENOMEM;
goto err_put_evdev;
}
spin_lock_init(&client->buffer_lock);
client->evdev = evdev;
evdev_attach_client(evdev, client);

error = evdev_open_device(evdev);
if (error)
goto err_free_client;

file->private_data = client;
return 0;

err_free_client:
evdev_detach_client(evdev, client);
kfree(client);
err_put_evdev:
put_device(&evdev->dev);
return error;
}
```

iminor(inode) - EVDEV_MINOR_BASE 就得到了在 evdev_table[ ]中的序号.然后将数组中对应的 evdev 取出.递增 devdev 中 device 的引用计数。

分配并初始化一个 client.并将它和 evdev 关联起来: client->evdev 指向它所表示的 evdev.将 client 挂到 evdev->client_list 上. 将 client 赋为 file 的私有区。对应 handle 的打开是在此 evdev_open_device()中完成的，代码如下:

```
static int evdev_open_device(struct evdev *evdev)
{
int retval;

retval = mutex_lock_interruptible(&evdev->mutex);
if (retval)
return retval;

if (!evdev->exist)
retval = -ENODEV;
else if (!evdev->open++) {
retval = input_open_device(&evdev->handle);
if (retval)
evdev->open--;
}

mutex_unlock(&evdev->mutex);
return retval;
}
```

如果 evdev 是第一次打开,就会调用 input_open_device()打开 evdev 对应的 handle，跟踪一下这个函数:

```
int input_open_device(struct input_handle *handle)
{
struct input_dev *dev = handle->dev;
int retval;

retval = mutex_lock_interruptible(&dev->mutex);
if (retval)
return retval;

if (dev->going_away) {
retval = -ENODEV;
goto out;
}

handle->open++;
```

```
if (!dev->users++ && dev->open)
retval = dev->open(dev);

if (retval) {
dev->users--;if (!--handle->open) {

synchronize_rcu();
}
}

out:
mutex_unlock(&dev->mutex);
return retval;
}
```

在这个函数中,我们看到递增 handle 的打开计数，如果是第一次打开.则调用 input device 的 open()函数.

（4）evdev 的事件处理

经过上面的分析.每当 input device 上报一个事件时,会将其交给和它匹配的 handler 的 event 函数处理.在 evdev 中.这个 event 函数对应的代码为:

```
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;

do_gettimeofday(&event.time);
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);
}
```

首先构造一个 struct input_event 结构.并设备它的 type.code,value 为处理事件的相关属性.如果该设备被强制设置了 handle.则调用如之对应的 client。

我们在 open 的时候分析到.会初始化 clinet 并将其链入到 evdev->client_list. 这样,就可以通过 evdev->client_list 找到这个 client 了。对于找到的第一个 client 都会调用 evdev_pass_event( ), 代码如下:

```
static void evdev_pass_event(struct evdev_client *client,
struct input_event *event)
{

spin_lock(&client->buffer_lock);
client->buffer[client->head++] = *event;
client->head &= EVDEV_BUFFER_SIZE - 1;
spin_unlock(&client->buffer_lock);

kill_fasync(&client->fasync, SIGIO, POLL_IN);
}
```

这里的操作很简单.就是将 event 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。

（5）设备节点的 read 处理

对于 evdev 设备节点的 read 操作都会由 evdev_read()完成.它的代码如下:

```
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 < evdev_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 + evdev_event_size() <= count &&
evdev_fetch_next_event(client, &event)) {

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

retval += evdev_event_size();
}

return retval;
}
```

首先,它判断缓存区大小是否足够.在读取数据的情况下,可能当前缓存区内没有数据可读.在这里先睡眠等待缓存区中有数据。如果在睡眠的时候，条件满 足，是不会进行睡眠状态而直接返回的。然后根据 read()提够的缓存区大小.将 client 中的数据写入到用户空间的缓存区中。

（6）设备节点的写操作

同样.对设备节点的写操作是由 evdev_write()完成的.代码如下:

```
static ssize_t evdev_write(struct file *file, const 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;

retval = mutex_lock_interruptible(&evdev->mutex);
if (retval)
return retval;

if (!evdev->exist) {
retval = -ENODEV;
```

```
goto out;
}

while (retval < count) {

if (evdev_event_from_user(buffer + retval, &event)) {
retval = -EFAULT;
bsp; goto out;
}

input_inject_event(&evdev->handle,
event.type, event.code, event.value);
retval += evdev_event_size();
}

out:
mutex_unlock(&evdev->mutex);
return retval;
}
```

首先取得操作设备文件所对应的 evdev，实际上,这里写入设备文件的是一个 event 结构的数组.我们在之前分析过,这个结构里包含了事件的 type.code 和 event，将写入设备的 event 数组取出，然后对每一项调用 event_inject_event()。

这个函数的操作和 input_event()差不多，就是将第一个参数 handle 转换为输入设备结构，然后这个设备再产生一个事件。代码如下:

```
void input_inject_event(struct input_handle *handle,
unsigned int type, unsigned int code, int value)
{
struct input_dev *dev = handle->dev;
struct input_handle *grab;
unsigned long flags;

if (is_event_supported(type, dev->evbit, EV_MAX)) {
spin_lock_irqsave(&dev->event_lock, flags);

rcu_read_lock();
grab = rcu_dereference(dev->grab);
if (!grab || grab == handle)
input_handle_event(dev, type, code, value);
rcu_read_unlock();

spin_unlock_irqrestore(&dev->event_lock, flags);
```

```
}
}
```

我们在这里也可以跟input_event()对比一下,这里设备可以产生任意事件,而不需要和设备所支持的事件类型相匹配，由此可见，对于写操作而言.就是让与设备文件相关的输入设备产生一个特定的事件。

六.小结

在 这一节点,分析了整个 input 子系统的架构,各个环节的流程。最后还以 evdev 为例将各个流程贯穿在一起，以加深对 input 子系统的理解。

由此也可以看出：linux 设备驱动采用了分层的模式，从最下层的设备模型到设备,驱动和总线，再到 input 子系统最后到 input device。这样的分层结构使得最上层的驱动不必关心下层是怎么实现的，而下层驱动又为多种型号同样功能的驱动提供了一个统一的接口。