Linux设备驱动之USB hub驱动

一：前言

继UHCI的驱动之后，我们对USB Control的运作有了一定的了解。在接下来的分析中，我们对USB设备的驱动做一个全面的分析，我们先从HUB的驱动说起。关于HUB,usb2.0 spec上有详细的定义，基于这部份的代码位于linux-2.6.25/drivers/usb/core下，也就是说，这部份代码是位于core下，和具体设备是无关的，因为各厂商的hub都是按照spec的要求来设计的。

二：UHCI驱动中的root hub

记得在分析UHCI驱动的时候，曾详细分析过root hub的初始化操作。为了分析方便，将代码片段列出如下：

usb_add_hcd() à usb_alloc_dev():

```
struct usb_device *usb_alloc_dev(struct usb_device *parent,

struct usb_bus *bus, unsigned port1)

{

……

……

//usb_device,内嵌有struct device结构，对这个结构进行初始化

device_initialize(&dev->dev);

dev->dev.bus = &usb_bus_type;

dev->dev.type = &usb_device_type;

……

……

}
```

一看到前面对dev的赋值，根据我们对设备模型的理解，一旦这个device进行注册，就会发生driver和device的匹配过程了。

不过，现在还不是分析这个过程的时候，我们先来看一下，USB子系统中的两种驱动。

三：USB子系统中的两种驱动

linux-2.6.25/drivers/usb/core/driver.c中，我们可以找到两种register driver的方式，分别为usb_register_driver()和usb_register_device_driver()。分别来分析一下这两个接口。

usb_register_device_driver()接口的代码如下：

```
int usb_register_device_driver(struct usb_device_driver *new_udriver,
```

```
struct module *owner)
{
int retval = 0;
if (usb_disabled())
return -ENODEV;
new_udriver->drvwrap.for_devices = 1;
new_udriver->drvwrap.driver.name = (char *) new_udriver->name;
new_udriver->drvwrap.driver.bus = &usb_bus_type;
new_udriver->drvwrap.driver.probe = usb_probe_device;
new_udriver->drvwrap.driver.remove = usb_unbind_device;
new_udriver->drvwrap.driver.owner = owner;
retval = driver_register(&new_udriver->drvwrap.driver);
if (!retval) {
pr_info("%s: registered new device driver %s\n",
usbcore_name, new_udriver->name);
usbfs_update_special();
} else {
printk(KERN_ERR "%s: error %d registering device "
" driver %s\n",
usbcore_name, retval, new_udriver->name);
}
return retval;
}
```

首先，通过usb_disabled()来判断一下usb是否被禁用，如果被禁用，当然就不必执行下面的流程了，直接退出即可。

从上面的代码，很明显可以看到， struct usb_device_driver 对struct device_driver进行了一次封装，我们注意一下这里的赋值操作：new_udriver->drvwrap.for_devices = 1. 等等。这些在后面都是用派上用场的。

usb_register_driver()的代码如下：

```
int usb_register_driver(struct usb_driver *new_driver, struct module *owner,
```

```
const char *mod_name)
{
int retval = 0;
if (usb_disabled())
return -ENODEV;
new_driver->drvwrap.for_devices = 0;
new_driver->drvwrap.driver.name = (char *) new_driver->name;
new_driver->drvwrap.driver.bus = &usb_bus_type;
new_driver->drvwrap.driver.probe = usb_probe_interface;
new_driver->drvwrap.driver.remove = usb_unbind_interface;
new_driver->drvwrap.driver.owner = owner;
new_driver->drvwrap.driver.mod_name = mod_name;
spin_lock_init(&new_driver->dynids.lock);
INIT_LIST_HEAD(&new_driver->dynids.list);
retval = driver_register(&new_driver->drvwrap.driver);
if (!retval) {
pr_info("%s: registered new interface driver %s\n",
usbcore_name, new_driver->name);
usbfs_update_special();
usb_create_newid_file(new_driver);
} else {
printk(KERN_ERR "%s: error %d registering interface "
" driver %s\n",
usbcore_name, retval, new_driver->name);
}
return retval;
}
```

很明显，在这里接口里，将new_driver->drvwrap.for_devices设为了0.而且两个接口的porbe()函数也不一样。

其实，对于usb_register_driver()可以看作是usb设备中的接口驱动，而usb_register_device_driver()是一个单纯的USB设备驱动。

四： hub的驱动分析

4.1: usb_bus_type->match()的匹配过程

usb_bus_type->match()用来判断驱动和设备是否匹配，它的代码如下：

```
static int usb_device_match(struct device *dev, struct device_driver *drv)

{

/* devices and interfaces are handled separately */

//usb device的情况

if (is_usb_device(dev)) {

/* interface drivers never match devices */

if (!is_usb_device_driver(drv))

return 0;

/* TODO: Add real matching code */

return 1;

}

//interface的情况

else {

struct usb_interface *intf;

struct usb_driver *usb_drv;

const struct usb_device_id *id;

/* device drivers never match interfaces */

if (is_usb_device_driver(drv))

return 0;

intf = to_usb_interface(dev);

usb_drv = to_usb_driver(drv);

id = usb_match_id(intf, usb_drv->id_table);

if (id)

return 1;
```

```
id = usb_match_dynamic_id(intf, usb_drv);

if (id)

return 1;

}

return 0;

}
```

这里的match会区分上面所说的两种驱动，即设备的驱动和接口的驱动。

is_usb_device()的代码如下：

```
static inline int is_usb_device(const struct device *dev)

{

return dev->type == &usb_device_type;

}
```

很明显，对于root hub来说，这个判断是肯定会满足的。

```
static inline int is_usb_device_driver(struct device_driver *drv)

{

return container_of(drv, struct usbdrv_wrap, driver)->

for_devices;

}
```

回忆一下，我们在分析usb_register_device_driver()的时候，不是将new_udriver->drvwrap.for_devices置为了1么？所以对于usb_register_device_driver()注册的驱动来说，这里也是会满足的。

因此，对应root hub的情况，从第一个if就会匹配到usb_register_device_driver()注册的驱动。

对于接口的驱动，我们等遇到的时候再来进行分析。

4.2:root hub的驱动入口

既然我们知道，root hub会匹配到usb_bus_type->match()的驱动，那这个驱动到底是什么呢？我们从usb子系统的初始化开始说起。

在linux-2.6.25/drivers/usb/core/usb.c中。有这样的一段代码：

```
subsys_initcall(usb_init);
```

对于subsys_initcall()我们已经不陌生了，在很多地方都会遇到它。在系统初始化的时候，会调用到它对应的函数。在这里，即为usb_init()。

在usb_init()中，有这样的代码片段：

```
static int __init usb_init(void)

{

……

……

1

if (!retval)

goto out;

……

}
```

在这里终于看到usb_register_device_driver()了。 usb_generic_driver会匹配到所有usb 设备。定义如下：

```
struct usb_device_driver usb_generic_driver = {

.name = "usb",

.probe = generic_probe,

.disconnect = generic_disconnect,

#ifdef CONFIG_PM

.suspend = generic_suspend,

.resume = generic_resume,

#endif

.supports_autosuspend = 1,

};
```

现在是到分析probe()的时候了。我们这里说的并不是usb_generic_driver中的probe,而是封装在struct usb_device_driver中的driver对应的probe函数。

在上面的分析， usb_register_device_driver()将封装的driver的probe()函数设置为了usb_probe_device()。代码如下：

```
static int usb_probe_device(struct device *dev)

{

struct usb_device_driver *udriver = to_usb_device_driver(dev->driver);

struct usb_device *udev;
```

```
int error = -ENODEV;

dev_dbg(dev, "%s\n", __FUNCTION__);

//再次判断dev是否是usb device

if (!is_usb_device(dev)) /* Sanity check */

return error;

udev = to_usb_device(dev);

/* TODO: Add real matching code */

/* The device should always appear to be in use

* unless the driver suports autosuspend.

*/

//pm_usage_cnt: autosuspend计数。如果此计数为1,则不允许autosuspend

udev->pm_usage_cnt = !(udriver->supports_autosuspend);

error = udriver->probe(udev);

return error;

}
```

首先，可以通过container_of()将封装的struct device, struct device_driver转换为struct usb_device和struct usb_device_driver.

然后，再执行一次安全检查，判断dev是否是属于一个usb device.

在这里，我们首次接触到了hub suspend.如果不支持suspend(udriver->supports_autosuspend为0)，则udev->pm_usage_cnt被设为1,也就是说，它不允许设备suspend.否则，将其初始化为0.

最后，正如你所看到的，流程转入到了usb_device_driver->probe()。

对应到root hub,流程会转入到generic_probe()。代码如下：

```
static int generic_probe(struct usb_device *udev)

{

int err, c;

/* put device-specific files into sysfs */

usb_create_sysfs_dev_files(udev);

/* Choose and set the configuration. This registers the interfaces

* with the driver core and lets interface drivers bind to them.
```

```
*/
if (udev->authorized == 0)
dev_err(&udev->dev, "Device is not authorized for usage\n");
else {
//选择和设定一个配置
c = usb_choose_configuration(udev);
if (c >= 0) {
err = usb_set_configuration(udev, c);
if (err) {
dev_err(&udev->dev, "can't set config #%d, error %d\n",
c, err);
/* This need not be fatal. The user can try to
* set other configurations. */
}
}
}
/* USB device state == configured … usable */
usb_notify_add_device(udev);
return 0;
}
```

usb_create_sysfs_dev_files()是在sysfs中显示几个属性文件，不进行详细分析，有兴趣的可以结合之前分析的《linux设备模型详解》来看下代码。

usb_notify_add_device()是有关notify链表的操作，这里也不做详细分析。

至于udev->authorized,在root hub的初始化中，是会将其初始化为1的。后面的逻辑就更简单了。为root hub 选择一个配置然后再设定这个配置。

还记得我们在分析root hub的时候，在usb_new_device()中，会将设备的所有配置都取出来，然后将它们放到了usb_device-> config.现在这些信息终于会派上用场了。不太熟悉的，可以看下本站之前有关usb控制器驱动的文档。

Usb2.0 spec上规定，对于hub设备，只能有一个config,一个interface,一个endpoint.实际上，在这里，对hub的选择约束不大，反正就一个配置，不管怎么样，选择和设定都是这个配置。

不过，为了方便以后的分析，我们还是跟进去看下usb_choose_configuration()和

usb_set_configuration()的实现。

实际上，经过这两个函数之后，设备的probe()过程也就会结束了。

4.2.1:usb_choose_configuration()函数分析

usb_choose_configuration()的代码如下：

```
//为usb device选择一个合适的配置

int usb_choose_configuration(struct usb_device *udev)

{

int i;

int num_configs;

int insufficient_power = 0;

struct usb_host_config *c, *best;

best = NULL;

//config数组

c = udev->config;

//config项数

num_configs = udev->descriptor.bNumConfigurations;

//遍历所有配置项

for (i = 0; i < num_configs; (i++, c++)) {

struct usb_interface_descriptor *desc = NULL;

/* It's possible that a config has no interfaces! */

//配置项的接口数目

//取配置项的第一个接口

if (c->desc.bNumInterfaces > 0)

desc = &c->intf_cache[0]->altsetting->desc;

/*

* HP's USB bus-powered keyboard has only one configuration

* and it claims to be self-powered; other devices may have

* similar errors in their descriptors. If the next test

* were allowed to execute, such configurations would always
```

```
* be rejected and the devices would not work as expected.
* In the meantime, we run the risk of selecting a config
* that requires external power at a time when that power
* isn't available. It seems to be the lesser of two evils.
*
* Bugzilla #6448 reports a device that appears to crash
* when it receives a GET_DEVICE_STATUS request! We don't
* have any other way to tell whether a device is self-powered,
* but since we don't use that information anywhere but here,
* the call has been removed.
*
* Maybe the GET_DEVICE_STATUS call and the test below can
* be reinstated when device firmwares become more reliable.
* Don't hold your breath.
*/
#if 0
/* Rule out self-powered configs for a bus-powered device */
if (bus_powered && (c->desc.bmAttributes &
USB_CONFIG_ATT_SELFPOWER))
continue;
#endif
/*
* The next test may not be as effective as it should be.
* Some hubs have errors in their descriptor, claiming
* to be self-powered when they are really bus-powered.
* We will overestimate the amount of current such hubs
* make available for each port.
*
* This is a fairly benign sort of failure. It won't
```

```
* cause us to reject configurations that we should have
* accepted.
*/
/* Rule out configs that draw too much bus current */
//电源不足。配置描述符中的电力是所需电力的1/2
if (c->desc.bMaxPower * 2 > udev->bus_mA) {
insufficient_power++;
continue;
}
/* When the first config's first interface is one of Microsoft's
* pet nonstandard Ethernet-over-USB protocols, ignore it unless
* this kernel has enabled the necessary host side driver.
*/
if (i == 0 && desc && (is_rndis(desc) || is_activesync(desc))) {
#if !defined(CONFIG_USB_NET_RNDIS_HOST) &&
!defined(CONFIG_USB_NET_RNDIS_HOST_MODULE)
continue;
#else
best = c;
#endif
}
/* From the remaining configs, choose the first one whose
* first interface is for a non-vendor-specific class.
* Reason: Linux is more likely to have a class driver
* than a vendor-specific driver. */
//选择一个不是USB_CLASS_VENDOR_SPEC的配置
else if (udev->descriptor.bDeviceClass !=
USB_CLASS_VENDOR_SPEC &&
(!desc || desc->bInterfaceClass !=
```

```
USB_CLASS_VENDOR_SPEC)) {

best = c;

break;

}

/* If all the remaining configs are vendor-specific,

* choose the first one. */

else if (!best)

best = c;

}

if (insufficient_power > 0)

dev_info(&udev->dev, "rejected %d configuration%s "

"due to insufficient available bus power\n",

insufficient_power, plural(insufficient_power));

//如果选择好了配置，返回配置的序号，否则，返回-1

if (best) {

i = best->desc.bConfigurationValue;

dev_info(&udev->dev,

"configuration #%d chosen from %d choice%s\n",

i, num_configs, plural(num_configs));

} else {

i = -1;

dev_warn(&udev->dev,

"no configuration chosen from %d choice%s\n",

num_configs, plural(num_configs));

}

return i;

}
```

Linux按照自己的喜好选择好了配置之后，返回配置的序号。不过对于HUB来说，它有且仅有一个配置。

4.2.2:usb_set_configuration()函数分析

既然已经选好配置了，那就告诉设备选好的配置，这个过程是在usb_set_configuration()中完成的。它的代码如下：

```
int usb_set_configuration(struct usb_device *dev, int configuration)

{

int i, ret;

struct usb_host_config *cp = NULL;

struct usb_interface **new_interfaces = NULL;

int n, nintf;

if (dev->authorized == 0 || configuration == -1)

configuration = 0;

else {

for (i = 0; i < dev->descriptor.bNumConfigurations; i++) {

if (dev->config[i].desc.bConfigurationValue ==

configuration) {

cp = &dev->config[i];

break;

}

}

}

if ((!cp && configuration != 0))

return -EINVAL;

/* The USB spec says configuration 0 means unconfigured.

* But if a device includes a configuration numbered 0,

* we will accept it as a correctly configured state.

* Use -1 if you really want to unconfigure the device.

*/

if (cp && configuration == 0)

dev_warn(&dev->dev, "config 0 descriptor??\n");
```

首先，根据选择好的配置号找到相应的配置，在这里要注意了， dev->config[]数组中的配置并不是按照配置的序号来存放的，而是按照遍历到顺序来排序的。因为有些设备在发送配置描述符的时候，并不是按照配置序号来发送的，例如，配置2可能在第一次GET_CONFIGURATION就被发送了，而配置1可能是在第二次GET_CONFIGURATION才能发送。

取得配置描述信息之后，要对它进行有效性判断，注意一下本段代码的最后几行代码：usb2.0 spec上规定，0号配置是无效配置，但是可能有些厂商的设备并末按照这一约定，所以在linux中，遇到这种情况只是打印出警告信息，然后尝试使用这一配置。

```
/* Allocate memory for new interfaces before doing anything else,

* so that if we run out then nothing will have changed. */

n = nintf = 0;

if (cp) {

//接口总数

nintf = cp->desc.bNumInterfaces;

//interface指针数组，

new_interfaces = kmalloc(nintf * sizeof(*new_interfaces),

GFP_KERNEL);

if (!new_interfaces) {

dev_err(&dev->dev, "Out of memory\n");

return -ENOMEM;

}

for (; n < nintf; ++n) {

new_interfaces[n] = kzalloc(

sizeof(struct usb_interface),

GFP_KERNEL);

if (!new_interfaces[n]) {

dev_err(&dev->dev, "Out of memory\n");

ret = -ENOMEM;

free_interfaces:

while (--n >= 0)

kfree(new_interfaces[n]);

kfree(new_interfaces);
```

```
return ret;

}

}

//如果总电源小于所需电流，打印警告信息

i = dev->bus_mA - cp->desc.bMaxPower * 2;

if (i < 0)

dev_warn(&dev->dev, "new config #%d exceeds power "

"limit by %dmA\n",

configuration, -i);

}
```

在这里，注要是为new_interfaces分配空间，要这意的是， new_interfaces是一个二级指针，它的最终指向是struct usb_interface结构。特别的，如果总电流数要小于配置所需电流，则打印出警告消息。实际上，这种情况在usb_choose_configuration()中已经进行了过滤。

```
/* Wake up the device so we can send it the Set-Config request */

//要对设备进行配置了，先唤醒它

ret = usb_autoresume_device(dev);

if (ret)

goto free_interfaces;

/* if it's already configured, clear out old state first.

* getting rid of old interfaces means unbinding their drivers.

*/

//不是处于ADDRESS状态，先清除设备的状态

if (dev->state != USB_STATE_ADDRESS)

usb_disable_device(dev, 1); /* Skip ep0 */

//发送控制消息，选取配置

ret = usb_control_msg(dev, usb_sndctrlpipe(dev, 0),

USB_REQ_SET_CONFIGURATION, 0, configuration, 0,

NULL, 0, USB_CTRL_SET_TIMEOUT);

if (ret < 0) {
```

/* All the old state is gone, so what else can we do?

* The device is probably useless now anyway.

*/

cp = NULL;

}

//dev->actconfig存放的是当前设备选取的配置

dev->actconfig = cp;

if (!cp) {

usb_set_device_state(dev, USB_STATE_ADDRESS);

usb_autosuspend_device(dev);

goto free_interfaces;

}

//将状态设为CONFIGURED

usb_set_device_state(dev, USB_STATE_CONFIGURED);

接下来，就要对设备进行配置了，首先，将设备唤醒。回忆一下我们在分析UHCI驱动时，列出来的设备状态图。只有在ADDRESS状态才能转入到CONFIG状态。(SUSPEND状态除外)。 所以，如果设备当前不是处于ADDRESS状态，就需要将设备的状态初始化。usb_disable_device()函数是个比较重要的操作，在接下来再对它进行详细分析。

接着，发送SET_CONFIGURATION的Control消息给设备，用来选择配置

最后，将dev->actconfig指向选定的配置，将设备状态设为CONFIG

/* Initialize the new interface structures and the

* hc/hcd/usbcore interface/endpoint state.

*/

//遍历所有的接口

for (i = 0; i < nintf; ++i) {

struct usb_interface_cache *intfc;

struct usb_interface *intf;

struct usb_host_interface *alt;

cp->interface[i] = intf = new_interfaces[i];

intfc = cp->intf_cache[i];

```
intf->altsetting = intfc->altsetting;
intf->num_altsetting = intfc->num_altsetting;
//是否关联的接口描述符，定义在minor usb 2.0 spec中
intf->intf_assoc = find_iad(dev, cp, i);
kref_get(&intfc->ref);
//选择0号设置
alt = usb_altnum_to_altsetting(intf, 0);
/* No altsetting 0? We'll assume the first altsetting.
* We could use a GetInterface call, but if a device is
* so non-compliant that it doesn't have altsetting 0
* then I wouldn't trust its reply anyway.
*/
//如果0号设置不存在，选排在第一个设置
if (!alt)
alt = &intf->altsetting[0];
//当前的配置
intf->cur_altsetting = alt;
usb_enable_interface(dev, intf);
intf->dev.parent = &dev->dev;
intf->dev.driver = NULL;
intf->dev.bus = &usb_bus_type;
intf->dev.type = &usb_if_device_type;
intf->dev.dma_mask = dev->dev.dma_mask;
device_initialize(&intf->dev);
mark_quiesced(intf);
sprintf(&intf->dev.bus_id[0], "%d-%s:%d.%d",
dev->bus->busnum, dev->devpath,
configuration, alt->desc.bInterfaceNumber);
}
```

kfree(new_interfaces);

if (cp->string == NULL)

cp->string = usb_cache_string(dev, cp->desc.iConfiguration);

之前初始化的new_interfaces在这里终于要派上用场了。初始化各接口，从上面的初始化过程中，我们可以看出：

Intf->altsetting,表示接口的各种设置

Intf->num_altsetting:表示接口的设置数目

Intf->intf_assoc:接口的关联接口(定义于minor usb 2.0 spec)

Intf->cur_altsetting:接口的当前设置。

结合之前在UHCI中的分析，我们总结一下：

Usb_dev->config,其实是一个数组，存放设备的配置。usb_dev->config[m]->interface[n]表示第m个配置的第n个接口的intercace结构。(m,n不是配置序号和接口序号*^_^*)。

注意这个地方对intf内嵌的struct devcie结构赋值，它的type被赋值为了usb_if_device_type.bus还是usb_bus_type.可能你已经反应过来了，要和这个device匹配的设备是interface的驱动。

特别的，这里的device的命名：

sprintf(&intf->dev.bus_id[0], "%d-%s:%d.%d",

dev->bus->busnum, dev->devpath,

configuration, alt->desc.bInterfaceNumber);

dev指的是这个接口所属的usb_dev,结合我们之前在UHCI中关于usb设备命名方式的描述。可得出它的命令方式如下：

USB总线号-设备路径：配置号。接口号。

例如，在我的虚拟机上：

[root@localhost devices]# pwd

/sys/bus/usb/devices

[root@localhost devices]# ls

1-0:1.0 usb1

[root@localhost devices]#

可以得知，系统只有一个usb control.

1-0:1.0:表示，第一个usb control下的root hub的1号配置的0号接口。