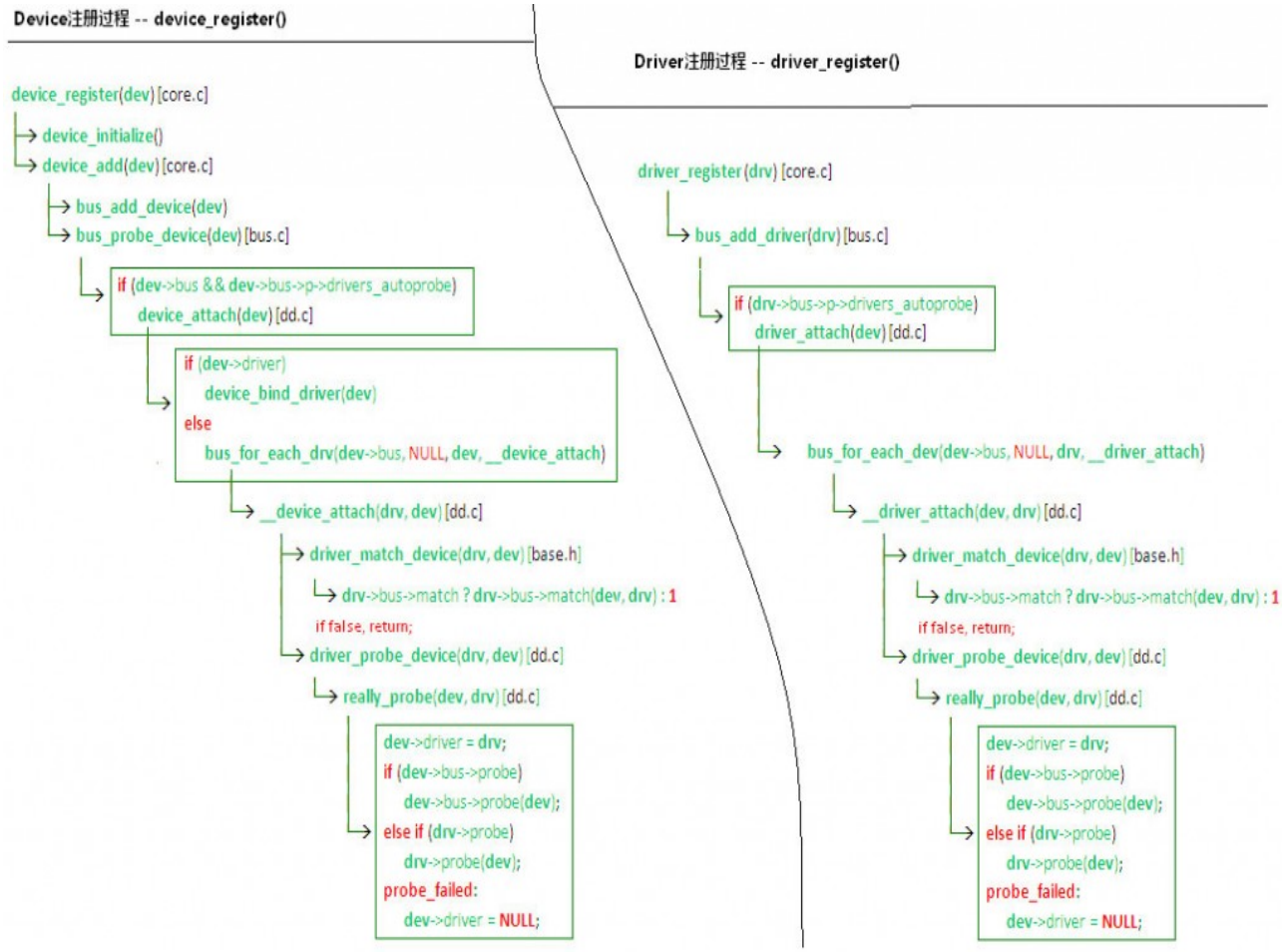
轉載本文解釋linux下設備和驅動的不同註冊順序時設備probe的時機；增加兩個case以解決PCI/USB等可熱插拔設備不同插入過程的probe時機的疑問。

Linux 2.6的設備驅動模型中，所有的device都是通過Bus相連。device_register() / driver_register()執行時通過枚舉BUS上的Driver/Device來實現綁定，本文詳解這一過程。這是整個LINUX設備驅動的基礎，PLATFORM設備，I2C上的設備等諸設備的註冊最終也是調用本文講述的註冊函數來實現的。

Linux Device的註冊最終都是通過device_register()實現，Driver的註冊最終都是通過driver_register()實現。下圖對照說明了Device和Driver的註冊過程。



上面的圖解一目瞭然，詳細過程不再贅述。注意以下幾點說明：

- BUS的p->drivers_autoprobe;1默認是true。
- bus_for_each_drv()是對BUS上所有的Driver都進行__device_attach()操作；同樣的，bus_for_each_dev()是對BUS上所有的Device都進行__driver_attach()操作。

- BUS上實現的.match()函數，定義了Device和Driver綁定時的規則。比如Platform實現的就是先比較id_table，然後比較name的規則。如果BUS的match()函數沒實現，認為BUS上的所有的Device和Driver都是match的，具體後續過程要看probe()的實現了。
- Probe的規則是：如果BUS上實現了probe就用BUS的probe；否則才會用driver的probe。

Device一般是先於Driver註冊，但也不全是這樣的順序。

Linux的Device和Driver的註冊過程分別枚舉掛在該BUS上所有的Driver和Device實現了這種時序無關性。

[增加兩個例子以解惑]

1 一個設備A已經attach驅動，不管兩個註冊的順序如何，完成這一步，說明driver已經加載；同類設備B再次hot pluggin加入，則device_attach僅為設備B與驅動attach上，不會重做設備A的attach；

2 一個設備A註冊，但是沒有找到驅動，用戶也不加載驅動；同類設備B hot pluggin，這時用戶加載驅動，驅動註冊，driver_attach，針對總線上的每個設備掃瞄，此時匹配的設備肯定沒有加載驅動（如果沒有一個設備對應同層次兩個驅動的情況），則對設備A和B都attach該driver；

3 如果一個設備可以對應同層次兩個驅動，是否允許，什麼策略？需要時研究代碼。

[附really_probe的代碼]

```
250static int really_probe(struct device *dev, struct device_driver *drv)
251{
252 int ret = 0;
253
254 atomic_inc(&probe_count);
255 pr_debug("bus: '%s': %s: probing driver %s with device %s\n",
256      drv->bus->name, __func__, drv->name, dev_name(dev));
257 WARN_ON(!list_empty(&dev->devres_head));
258
259 dev->driver = drv; // 已經bus->match上，所以可以將該device和driver關聯起來
260 if (driver_sysfs_add(dev)) {
261     printk(KERN_ERR "%s: driver_sysfs_add(%s) failed\n",
262         __func__, dev_name(dev));
263     goto probe_failed;
264 }
265
266 if (dev->bus->probe) {  // 調用bus->probe，由bus->probe調用'具體'dev_drv->probe
267     ret = dev->bus->probe(dev);
268     if (ret)
269         goto probe_failed;
```

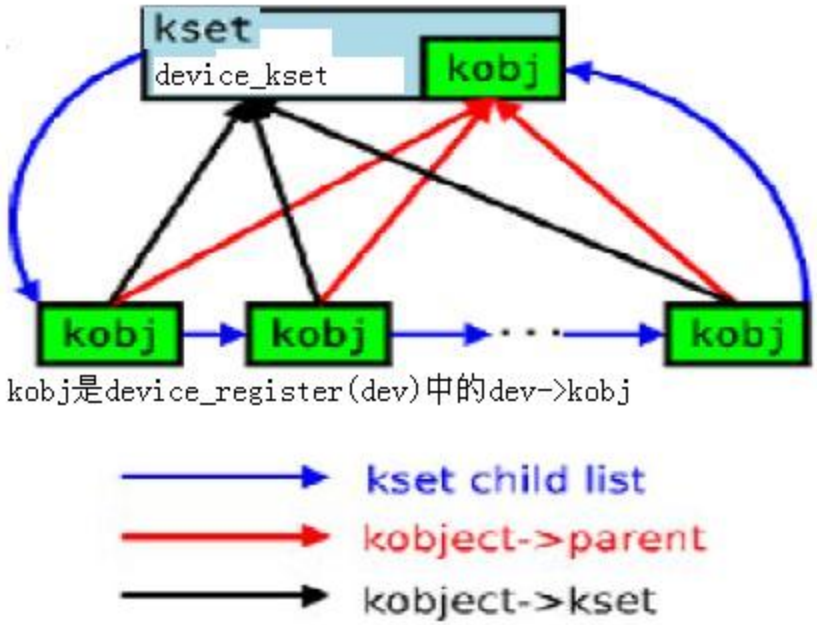
```
270 } else if (drv->probe) { // 使用'頂層'驅動device_driver的probe
271     ret = drv->probe(dev);
272     if (ret)
273         goto probe_failed;
274 }
275
276 driver_bound(dev);      // 設備與驅動已經關聯好了
277 ret = 1;
278 pr_debug("bus: '%s': %s: bound device %s to driver %s\n",
279      drv->bus->name, __func__, dev_name(dev), drv->name);
280 goto done;
281
282probe_failed:
283 devres_release_all(dev);
284 driver_sysfs_remove(dev);
285 dev->driver = NULL;
286
287 if (ret == -EPROBE_DEFER) {
288     /* Driver requested deferred probing */
289     dev_info(dev, "Driver %s requests probe deferral\n", drv->name);
290     driver_deferred_probe_add(dev);
291 } else if (ret != -ENODEV && ret != -ENXIO) {
292     /* driver matched but the probe failed */
293     printk(KERN_WARNING
294            "%s: probe of %s failed with error %d\n",
295            drv->name, dev_name(dev), ret);
296 } else {
297     pr_debug("%s: probe of %s rejects match %d\n",
298            drv->name, dev_name(dev), ret);
299 }
300 /*
301  * Ignore errors returned by ->probe so that the next driver can try
302  * its luck.
303  */
304 ret = 0;
305done:
306 atomic_dec(&probe_count);
307 wake_up(&probe_waitqueue);
308 return ret;
```

```
60. 309}
61.
```

device_register 分析

這篇文章也是從別的地方轉載的，我的目的是搞清楚：當調用device_register()函數向系統註冊一個設備的時候，我註冊進去的設備是如何和他父設備關聯起來的，以及如何加入到他所在的總線設備中的，但針對這個問題，好像通過這篇文章瞭解的並不透徹。但具體到代碼分析的最後關於設備和驅動是如何綁定的，這並不是我這篇文章的重點，但大概看了一下，有點類型i2c總線上設備和驅動的匹配過程。

看下圖：



在分析程序的過程中看到了把kobj->kset賦值為(kset)device_kset(即圖中黑線實現的部分),但沒有看到什麼時候把dev->kobj->parent賦值為device_kset->kobj(圖中的紅線實現的部分)，在調用函數setup_parent()中是對dev->kobject->parent賦值了，但不明白在setup_parent()函數中是怎麼找到device_kset的。說實在話，對setup_parent()函數不明白，也沒分析清楚。

（在此補充一下，分析了一下setup_parent()函數，實現了紅線的部分）

這篇文章將那個3個註冊函數說說，把整個設備模型框架搭建起來，當然，是重點部分了。在這之前希望你已經懂得總線、設備、驅動的數據結構及其裡面的有關數據結構。關於調用的函數，如果顯示為粗體，那麼在下面我有分析。

轉載於：http://student.csdn.net/space.php?uid=111596&do=blog&id=56043

來自：drivers/base/core.c

```
int device_register(struct device *dev)
{
    device_initialize(dev);                 //初始化設備
        return device_add(dev);                 //添加設備
}

void device_initialize(struct device *dev)
{
    //圖中的黑線實現部分的代碼
    dev->kobj.kset = devices_kset;              //設置設備的kobject所屬集合，devices_kset其實在第一層，sys/devices/
        kobject_init(&dev->kobj, &device_ktype);        //初始化設備的kobject
        INIT_LIST_HEAD(&dev->dma_pools);            //初始化設備的DMA池，用於傳遞大數據
        mutex_init(&dev->mutex);                    //初始化互斥鎖
        lockdep_set_novalidate_class(&dev->mutex);
        spin_lock_init(&dev->devres_lock);              //初始化自旋鎖，用於同步子設備鏈表
        INIT_LIST_HEAD(&dev->devres_head);                  //初始化子設備鏈表頭
        device_pm_init(dev);
        set_dev_node(dev, -1);
```

```
}

int device_add(struct device *dev)
{
	struct device *parent = NULL;
	struct class_interface *class_intf;
	int error = -EINVAL;

	dev = get_device(dev);		//增加設備的kobject的引用計數
	if (!dev)
		goto done;

	if (!dev->p) {
		error = device_private_init(dev);		//初始化設備的私有成員
		if (error)
			goto done;
	}

	/*
	 * for statically allocated devices, which should all be converted
	 * some day, we need to initialize the name. We prevent reading back
	 * the name, and force the use of dev_name()
	 */
	if (dev->init_name) {
		dev_set_name(dev, "%s", dev->init_name);		//設置設備kobject的名稱
		dev->init_name = NULL;
	}
```

```
    if (!dev_name(dev)) {
        error = -EINVAL;
        goto name_error;
    }

    pr_debug("device: '%s': %s/n", dev_name(dev), __func__);

    parent = get_device(dev->parent);          //增加父設備kobject的引用
    setup_parent(dev, parent);                 //設置該設備kobject父對象（父對象是誰呢）

    /* use parent numa_node */
    if (parent)
        set_dev_node(dev, dev_to_node(parent));

    /* first, register with generic layer. */
    /* we require the name to be set before, and pass NULL */
    error = kobject_add(&dev->kobj, dev->kobj.parent, NULL);   //將設備kobject添加
進父對象設備模型
    if (error)
        goto Error;

    /* notify platform of device entry */
    if (platform_notify)
        platform_notify(dev);

    error = device_create_file(dev, &uevent_attr);
```

```
if (error)
        goto attrError;

if (MAJOR(dev->devt)) {
        error = device_create_file(dev, &devt_attr);
        if (error)
                goto ueventattrError;

        error = device_create_sys_dev_entry(dev);
        if (error)
                goto devtattrError;

        devtmpfs_create_node(dev);
}

error = device_add_class_symlinks(dev);
if (error)
        goto SymlinkError;
error = device_add_attrs(dev);
if (error)
        goto AttrsError;
```

調用bus_add_device在sysfs中添加兩個鏈接：一個在總線目錄下指向設備，另一個在設備的目錄下指向總線子系統。

```
error = bus_add_device(dev);        //將設備添加進總線中
if (error)
        goto BusError;
error = dpm_sysfs_add(dev);
```

```
if (error)
        goto DPMError;
device_pm_add(dev);

/* Notify clients of device addition.  This call must come
 * after dpm_sysf_add() and before kobject_uevent().
 */
if (dev->bus)
        blocking_notifier_call_chain(&dev->bus->p->bus_notifier,
                                     BUS_NOTIFY_ADD_DEVICE, dev);

kobject_uevent(&dev->kobj, KOBJ_ADD);
```

bus_probe_device試圖自動探測設備。如果能夠找到合適的驅動程序，則將設備添加到bus->klist_devices.設備還需要添加到父結點的子結點鏈表中，圖中藍色線的實現部分（此前，設備知道其父結點，但父結點不知道子結點的存在）

```
bus_probe_device(dev);                          //現在該為設備在總線上尋找合適的驅動了
if (parent)
        klist_add_tail(&dev->p->knode_parent,
                       &parent->p->klist_children);          //將設備添加到父設備的子設備鏈表中

if (dev->class) {
        mutex_lock(&dev->class->p->class_mutex);
        /* tie the class to the device */
        klist_add_tail(&dev->knode_class,
                       &dev->class->p->class_devices);
```

```
                /* notify any interfaces that the device is here */
                list_for_each_entry(class_intf,
                                    &dev->class->p->class_interfaces, node)
                        if (class_intf->add_dev)
                                class_intf->add_dev(dev, class_intf);
                mutex_unlock(&dev->class->p->class_mutex);
        }
done:
        put_device(dev);
        return error;
 DPMError:
        bus_remove_device(dev);
 BusError:
        device_remove_attrs(dev);
 AttrsError:
        device_remove_class_symlinks(dev);
 SymlinkError:
        if (MAJOR(dev->devt))
                devtmpfs_delete_node(dev);
        if (MAJOR(dev->devt))
                device_remove_sys_dev_entry(dev);
 devtattrError:
        if (MAJOR(dev->devt))
                device_remove_file(dev, &devt_attr);
 ueventattrError:
        device_remove_file(dev, &uevent_attr);
```

```
 attrError:
        kobject_uevent(&dev->kobj, KOBJ_REMOVE);
        kobject_del(&dev->kobj);
 Error:
        cleanup_device_parent(dev);
        if (parent)
                put_device(parent);
name_error:
        kfree(dev->p);
        dev->p = NULL;
        goto done;
}

int device_private_init(struct device *dev)
{
        dev->p = kzalloc(sizeof(*dev->p), GFP_KERNEL);
        if (!dev->p)
                return -ENOMEM;
        dev->p->device = dev;                                   //指向設備自己
        klist_init(&dev->p->klist_children, klist_children_get,
                   klist_children_put);             //初始化設備私有成員的子設備鏈表，還有兩個函
數，關於增加和減少子設備引用計數的
        return 0;
}
static void setup_parent(struct device *dev, struct device *parent)
{
        struct kobject *kobj;
```

```
        kobj = get_device_parent(dev, parent);          //得到設備kobject的父對象
        if (kobj)
            dev->kobj.parent = kobj;
}
int bus_add_device(struct device *dev)
{
        struct bus_type *bus = bus_get(dev->bus);
        int error = 0;

        if (bus) {
                pr_debug("bus: '%s': add device %s/n", bus->name, dev_name(dev));
                error = device_add_attrs(bus, dev);
                if (error)
                        goto out_put;
                error = sysfs_create_link(&bus->p->devices_kset->kobj,
                                                &dev->kobj, dev_name(dev));
                if (error)
                        goto out_id;
                error = sysfs_create_link(&dev->kobj,
                                &dev->bus->p->subsys.kobj, "subsystem");
                if (error)
                        goto out_subsys;
                error = make_deprecated_bus_links(dev);
                if (error)
                        goto out_deprecated;
                klist_add_tail(&dev->p->knode_bus, &bus->p->klist_devices);     //關鍵點了，
將設備添加進總線的設備鏈表
```

```
        }
        return 0;

out_deprecated:
        sysfs_remove_link(&dev->kobj, "subsystem");
out_subsys:
        sysfs_remove_link(&bus->p->devices_kset->kobj, dev_name(dev));
out_id:
        device_remove_attrs(bus, dev);
out_put:
        bus_put(dev->bus);
        return error;
}
void bus_probe_device(struct device *dev)
{
        struct bus_type *bus = dev->bus;
        int ret;

        if (bus && bus->p->drivers_autoprobe) {       //如果需要自動匹配驅動
                ret = device_attach(dev);                       //為設備尋找驅動
                WARN_ON(ret < 0);
        }
}
int device_attach(struct device *dev)
{
        int ret = 0;
```

```
	device_lock(dev);	//鎖住設備
	if (dev->driver) {				//如果設備有驅動
		ret = device_bind_driver(dev);			//那麼將設備和驅動綁定
		if (ret == 0)
			ret = 1;
		else {
			dev->driver = NULL;
			ret = 0;
		}
	} else {
		pm_runtime_get_noresume(dev);
		ret = bus_for_each_drv(dev->bus, NULL, dev, __device_attach); //否則，
在總線上尋找驅動與該設備進行匹配
		pm_runtime_put_sync(dev);
	}
	device_unlock(dev);
	return ret;
}
int device_bind_driver(struct device *dev)
{
	int ret;

	ret = driver_sysfs_add(dev);
	if (!ret)
		driver_bound(dev);  //驅動綁定設備
	return ret;
}
```

```
int bus_for_each_drv(struct bus_type *bus, struct device_driver *start,
                     void *data, int (*fn)(struct device_driver *, void *))
{
        struct klist_iter i;
        struct device_driver *drv;
        int error = 0;

        if (!bus)
                return -EINVAL;

        klist_iter_init_node(&bus->p->klist_drivers, &i,
                             start ? &start->p->knode_bus : NULL);   //初始化i結構體
        while ((drv = next_driver(&i)) && !error)   //遍歷總線上的驅動
                error = fn(drv, data);                    //將驅動和設備進行匹配，這裡的
fn=__device_attach
        klist_iter_exit(&i);
        return error;
}

static int __device_attach(struct device_driver *drv, void *data)
{
        struct device *dev = data;

        if (!driver_match_device(drv, dev))          //現用總線上的match匹配函數進行低級匹
配
                return 0;
```

```
        return driver_probe_device(drv, dev);   //在來高級匹配
}

static inline int driver_match_device(struct device_driver *drv,  struct device *dev)
{
        return drv->bus->match ? drv->bus->match(dev, drv) : 1;    //看到沒，這裡要調用總線
上定義的match函數
}

int driver_probe_device(struct device_driver *drv, struct device *dev)
{
        int ret = 0;

        if (!device_is_registered(dev))  //設備是否註冊
                return -ENODEV;

        pr_debug("bus: '%s': %s: matched device %s with driver %s/n",
                 drv->bus->name, __func__, dev_name(dev), drv->name);

        pm_runtime_get_noresume(dev);
        pm_runtime_barrier(dev);
        ret = really_probe(dev, drv);  //調用真正的匹配
        pm_runtime_put_sync(dev);

        return ret;
}
```

```
static int really_probe(struct device *dev, struct device_driver *drv)
{
        int ret = 0;

        atomic_inc(&probe_count);
        pr_debug("bus: '%s': %s: probing driver %s with device %s/n",
                 drv->bus->name, __func__, drv->name, dev_name(dev));
        WARN_ON(!list_empty(&dev->devres_head));

        dev->driver = drv;
        if (driver_sysfs_add(dev)) {
                printk(KERN_ERR "%s: driver_sysfs_add(%s) failed/n",
                        __func__, dev_name(dev));
                goto probe_failed;
        }

        if (dev->bus->probe) {          //現用總線上定義的probe函數嘗試一下
                ret = dev->bus->probe(dev);
                if (ret)
                        goto probe_failed;
        } else if (drv->probe) {        //如果不行，在用驅動上的probe嘗試
                ret = drv->probe(dev);
                if (ret)
                        goto probe_failed;
        }

        driver_bound(dev);      //驅動綁定設備
```

```
        ret = 1;
        pr_debug("bus: '%s': %s: bound device %s to driver %s/n",
                 drv->bus->name, __func__, dev_name(dev), drv->name);
        goto done;

probe_failed:
        devres_release_all(dev);
        driver_sysfs_remove(dev);
        dev->driver = NULL;

        if (ret != -ENODEV && ret != -ENXIO) {
                /* driver matched but the probe failed */
                printk(KERN_WARNING
                        "%s: probe of %s failed with error %d/n",
                        drv->name, dev_name(dev), ret);
        }
        /*
         * Ignore errors returned by ->probe so that the next driver can try
         * its luck.
         */
        ret = 0;
done:
        atomic_dec(&probe_count);
        wake_up(&probe_waitqueue);
        return ret;
}
```