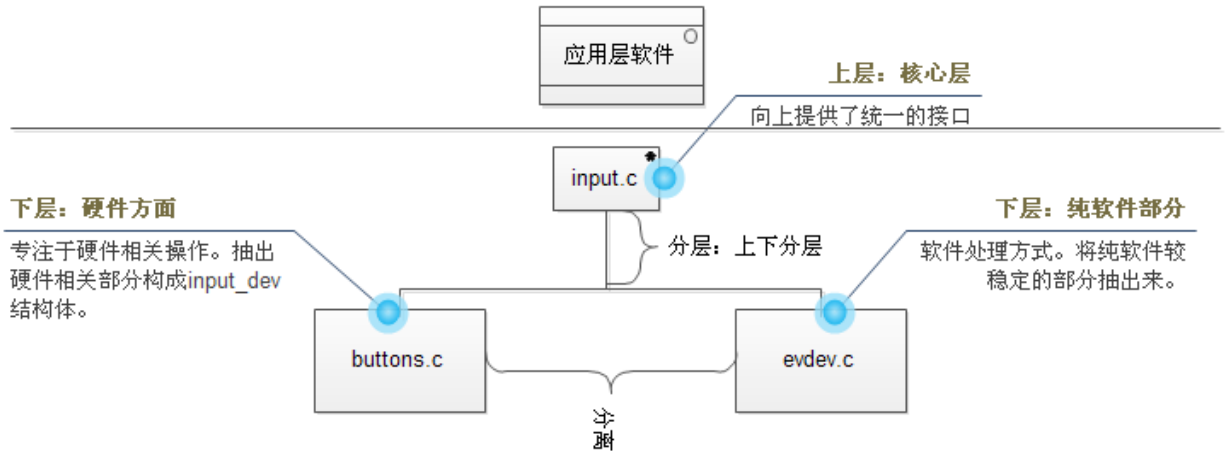
网名“鱼树”的学员聂龙浩，

学习“韦东山 Linux 视频第 2 期”时所写的笔记很详细，供大家参考。

也许有错漏，请自行分辨。

目录

驱动程序的分离分层：



分离：

把硬件相关的东西抽出来；把相对稳定的软件部分抽出来。

分层：

input.c 向上提供统一给 APP 操作的统一接口。每一层专注于自己的事件。

在“输入子系统”中有一个实例：gpio_keys.c.
它的入口函数“gpio_keys_init(void)”：

```
static int __init gpio_keys_init(void)
{
    return platform_driver_register(&gpio_keys_device_driver);
}
```

注册一个平台 driver 结构体。

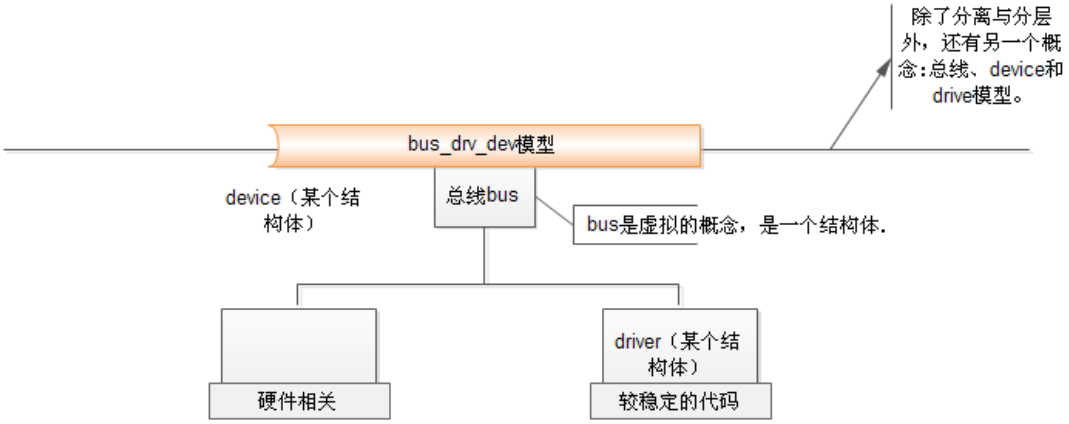
这个注册的“platform_driver”我们关心其中的“.probe”函数：

```
struct platform_driver gpio_keys_device_driver = {
.probe                  = gpio_keys_probe,
.remove                 = __devexit_p(gpio_keys_remove),
.driver                 = {
.name           = "gpio-keys",
}
};
int __devinit gpio_keys_probe(struct platform_device *pdev)这里有一个平台设备形参。
```

-->input = input_allocate_device();分配一个“input_dev”结构体（就是之前的输入子系统框

架部分）。
其中的“平台设备”驱动就涉及分离分层的概念。

之所以看到注册的一个平台结构体，就看这个结构体中的“.probe”函数？



这个模型的使用方法，可以看“gpio_keys.c”中的代码：

平台总线是一条虚拟总线。

驱动部分：

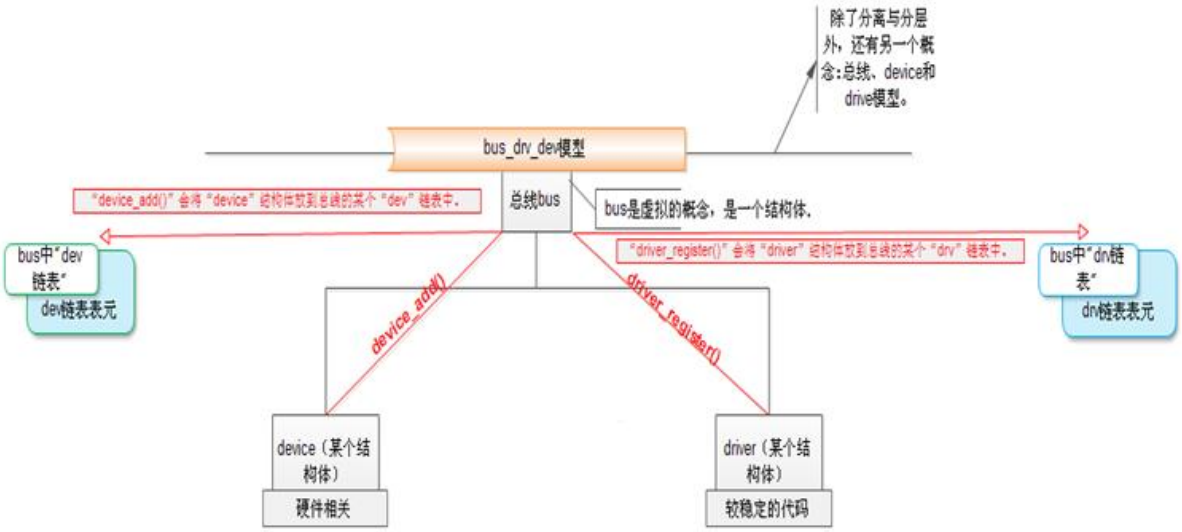
```
int __init gpio_keys_init(void)
-->platform_driver_register(&gpio_keys_device_driver);向上面注册一个driver。
-->drv->driver.bus = &platform_bus_type;
-->return driver_register(&drv->driver);
```

"driver_register()"会将“bus_drv_dev”模型中的较稳定代码“driver”结构体放到虚拟总线的某个链表（drv 链表）中。

硬件部分：

```
int platform_device_add(struct platform_device *pdev)
-->ret = device_add(&pdev->dev);
```

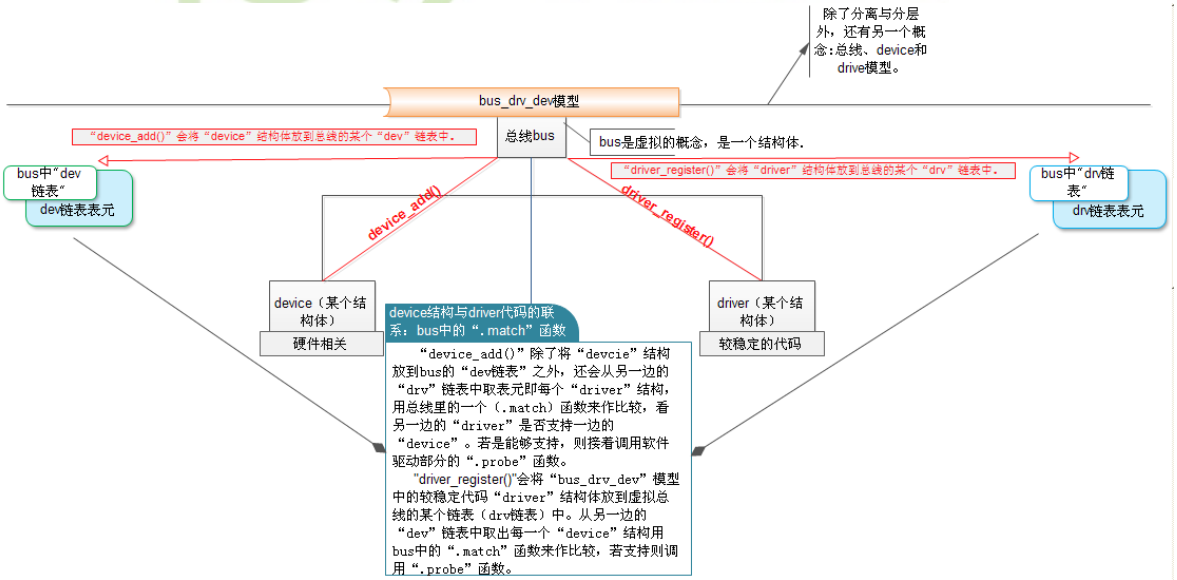
"device_add()"会将“bus_drv_dev”模型中的硬件部分“device”结构体放到虚拟总线的某个链表（dev 链表）中。



一边的“device”结构体和另一边的“较稳定的 drivice 代码”的联系：

“device_add()”除了将“devcie”结构放到 bus 的“dev 链表”之外，还会从另一边的“drv”链表中取表元即某个“driver”结构，用总线里的一个（.match）函数来作比较，看另一边的“driver”是否支持一边的“device”。若是能够支持，则接着调用软件驱动部分的“.probe”函数。

"driver_register()"会将“bus_drv_dev”模型中的较稳定代码“driver”结构体放到虚拟总线的某个链表（drv 链表）中。从另一边的“dev”链表中取出每一个“device”结构用 bus 中的“.match”函数来作比较，若支持则调用“.probe”函数。



“只不过”左右两个注册建立起来的一种机制。在“.probe”函数中做的事件由自己决定，打印一句话，或注册一个字符设备，再或注册一个“input_dev”结构体等等都是由自己决定。

强制的把一个驱动程序分为左右两边这种机制而已，可以把这套东西放在任何地方，这里的“driver”只是个结构体不要被这个名字迷惑，“device”也只是个结构体，里面放什么内容都是由自己决定的。

int __init gpio_keys_init(void)

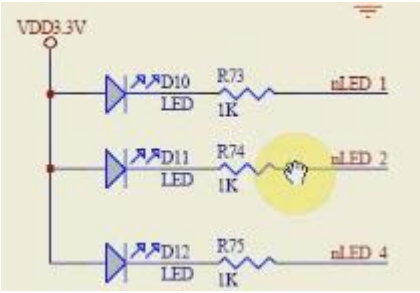
-->platform_driver_register(&gpio_keys_device_driver);注册一个平台 driver。
-->drv->driver.bus = &platform_bus_type; 总线是平台总线（虚拟出的总线，有一个函数）。
-->.match = platform_match, (看左右 device 与 driver 结构两边是否匹配)
-->struct platform_device *pdev = container_of(dev, struct platform_device, dev);
//平台 dev 的名字和 drv 的名字相同了就认为它们能匹配。
-->return (strncmp(pdev->name, drv->name, BUS_ID_SIZE) == 0);
匹配后就会调用 platform_driver gpio_keys_device_driver 结构中的 .probe = gpio_keys_probe 函数,若想“.probe”函数能够调用，左边“device”硬件结构部分应该有一个同名的“.name = "gpio-keys"”平台 device.

```
struct platform_driver gpio_keys_device_driver = {
    .probe      = gpio_keys_probe,
    .remove     = __devexit_p(gpio_keys_remove),
    .driver     = {
        .name   = "gpio-keys",
    }
};
```

可以搜索这个平台“device”名字“gpio-keys”，但 Linux-2.6.22 源码中并没有，所以这个“gpio_keys.c”只不过是个示例程序而已。

实验：用平台分层分离思想点 LED。

1，有三个 LED 灯：




```
/* 配置GPF4,5,6为输出 */
*gpfcon &= ~((0x3<<(4*2)) | (0x3<<(5*2)) | (0x3<<(6*2)));
*gpfcon |= ((0x1<<(4*2)) | (0x1<<(5*2)) | (0x1<<(6*2)));
```

现在只想点一个 LED 灯，这里强制的把代码分成左右两边：
想写一个驱动，想达到一个目的：
左边“device”表示某一个 LED 灯。要想修改是哪个 LED 灯，就只需要修改左边这个 led_platform_dev.c 即可。而
右边那个“led_platform_drv.c”保持稳定不变。

Led_platform_dev.c
模仿：这是打过补丁后的“linux-2.6.22.6\arch\arm\mach-s3c2440\Mach-smdk2440.c"中的代码。

```
static struct platform_device s3c2440_device_sdi = {
```

```
    .name           = "s3c2440-sdi",  名字。
    .id             = -1,
    .num_resources      = ARRAY_SIZE(s3c2440_sdi_resource),
    .resource           = s3c2440_sdi_resource,
};
```

“s3c2440_sdi_resource”：平台设备结构“platform_device”有所谓的资源。

下面是“s3c2440_sdi_resource”平台资源具体的内容：

平台资源结构体：

```
struct resource {
resource_size_t start; //资源开始地址。
resource_size_t end;    //资源结束地址。
const char *name;       //资源名字。
unsigned long flags;    //flags表示是哪一类资源。
struct resource *parent, *sibling, *child;
};
```

“flags”表示是指哪一类资源，也只是定义而已，也是一些结构体：

```
#define IORESOURCE_MEM          0x00000200
#define IORESOURCE_IRQ          0x00000400
```

查看 LED1 是哪个引脚：*gpfdat &= ~((1<<4) | (1<<5) | (1<<6));

```
#define S3C2410_PA_SDI    (0x5A000000)
#define S3C24XX_SZ_SDI    SZ_1M
```

```
static struct resource s3c2440_sdi_resource[] = {
    [0] = {  寄存器的物理地址
        .start = S3C2410_PA_SDI,
        .end   = S3C2410_PA_SDI + S3C24XX_SZ_SDI - 1,
        .flags = IORESOURCE_MEM,
    },
    [1] = {
        .start = IRQ_SDI,
        .end   = IRQ_SDI,
        .flags = IORESOURCE_IRQ,
    }

};
```

仿照上面的示例写：

```
gpfcon = (volatile unsigned long *)ioremap(0x56000050, 16);
gpfdat = gpfcon + 1;
```

还有硬件地址，看 LED 灯的寄存器：

初始物理地址：0x56000050

结束地址：0x56000050 + 8 - 1;

驱动程序里定义了一个平台设备：

struct platform_device led_dev，这个平台设备里还有一些所谓的平台资源。

资源里有它的寄存器地址：

```
[0] = {
        .start = 0x56000050, //LED 的寄存器起始地址:gpfcon = (volatile unsigned long *)ioremap(0x56000050, 16);
        .end   = 0x56000050 + 8 - 1, //LED 寄存器结束地址:gpfdat = gpfcon + 1;
        .flags = IORESOURCE_MEM,
  },
```

要修改寄存器，就直接修改上面的寄存器起始地址就可以。

资源里还有哪一个引脚的信息：

```
[1] = {
        .start = 4, //LED1 引脚 4 是"4,5,6"中的第一个 LED 灯。
        .end   = 4,
        .flags = IORESOURCE_IRQ,
    }
```

以后想换另一个 LED 灯，就只需要换这里的引脚就好，如第 2 个 LED 灯引脚是 5：

```
1] = {
        .start = 5, //LED2 引脚 5 是"4,5,6"中的第二个 LED 灯。
        .end   = 5,
        .flags = IORESOURCE_IRQ,
    }
```


写“平台设备”：

Led_dev.c 设备部分：

1，定义平台设备：

1，分配、设置、注册一个 platform_device 结构体。

```
static struct resource led_resource[] = {
    [0] = {//哪个寄存器：LED的寄存器起始地址:gpfcon = (volatile unsigned long *)ioremap(0x56000050, 16);
        .start = 0x56000050, //ioremap
        .end   = 0x56000050 + 8 - 1, //LED寄存器结束地址:gpfdat = gpfcon + 1;
        .flags = IORESOURCE_MEM,      //内存资源
    },
    [1] = { //哪个引脚
        .start = 4, //LED1引脚4是"4,5,6"中的第1个LED灯
        .end   = 4,
        .flags = IORESOURCE_IRQ,     //中断资源
    }
```

```
};

static struct platform_device led_dev = {
	.name		= "myled",
	.id		= -1,
	.num_resources	= ARRAY_SIZE(led_resource),
	.resource	= led_resource,
};
```

2，入口函数：

```
static int led_dev_init(void)
{
	//2.1，注册一个平台设备.
	platform_device_register (&led_dev);
	return 0;
}
```

2.1，看注册平台设备的过程：

```
platform_device_register (&led_dev);
-->platform_device_add(pdev);
-->device_add(&pdev->dev); 将device放到平台总线的“dev”链表中去。
```

3，出口函数：

```
static void led_dev_exit(void)
{//3.1,卸载平台设备
	platform_device_unregister(&led_dev);
}
```

4，修饰入口函数：

```
module_exit(led_dev_init);
MODULE_LICENSE("GPL");
module_init(led_dev_init);
以上便写好了平台设备部分代码。
```

接着写“平台驱动”：

1，定义平台驱动：

```
//1,分配、设置、注册一个 platform_driver 结构体。
//1,1，定义一个平台驱动.因为平台总线的.match 函数比较的是"平台设备"和"平台驱动"的名字.所以两边名字要相同.
//这样才会认为这个 drv 能支持这个 dev。才会调用平台驱动里面的".probe"函数。
struct platform_driver led_drv = {
.probe		= led_probe, //自己写一个 probe 函数.
.remove		= led_remove; //自己写 led_remove 函数.与 led_remov 倒过来写.
.driver		= {
.name	= "myled", //名字要与平台设备结构体中的名字一致.
}
};
```

2，构造平台驱动结构中的“.probe”函数：

```
//2,平台驱动结构中的".probe"函数.这个函数是自己按照自己的要求写的。
static int led_probe(struct platform_device *pdev)
{
	//2.1,根据 platform_device 的资源进行 ioremap .

	//2.2,注册字符设备驱动程序.
	printk("led_probe, found led\n");
	return 0;
}
```

3，构造平台驱动结构中的“.remove”函数：做与“.probe”相反的事件。

```
//3,led_remove,与".probe"函数相反.
static int led_remove(struct platform_device *pdev)
{
	//3.1,根据 platform_device 的资源进行 iounmap .

	//3.2,卸载字符设备驱动程序.
	printk("led_remove, remove led\n");
	return 0;
}
```

4，入口函数：

```
//4,入口函数.
static int led_drv_init(void)
{
	//4.1,注册一个台平台驱动。
	platform_driver_register (&led_drv);
	return 0;
}
```

5，出口函数：

```
//5,出口函数
static void led_drv_exit(void)
{
	//5.1,卸载平台驱动。
	platform_driver_unregister (&led_drv);
}
```

简单的框架代码如下：

编译加载：

```
root@ian:/work/nfs_root/romfs/9th_led_bus_drv_dev# make
make -C /work/system/linux-2.6.22.6 M=`pwd` modules
make[1]: Entering directory `/work/system/linux-2.6.22.6'
  CC [M]  /work/nfs_root/romfs/9th_led_bus_drv_dev/led_platform_dev.o
  CC [M]  /work/nfs_root/romfs/9th_led_bus_drv_dev/led_platform_drv.o
  Building modules, stage 2.
  MODPOST 2 modules
  CC      /work/nfs_root/romfs/9th_led_bus_drv_dev/led_platform_dev.mod.o
  LD [M]  /work/nfs_root/romfs/9th_led_bus_drv_dev/led_platform_dev.ko
  CC      /work/nfs_root/romfs/9th_led_bus_drv_dev/led_platform_drv.mod.o
  LD [M]  /work/nfs_root/romfs/9th_led_bus_drv_dev/led_platform_drv.ko
make[1]: Leaving directory `/work/system/linux-2.6.22.6'
root@ian:/work/nfs_root/romfs/9th_led_bus_drv_dev#
```

单板挂接 NFS 文件系统：

注册"led_platform_drv.ko":

```
# cd my_test_ko/
# ls
led_platform_dev.ko  led_platform_drv.ko
# insmod led_platform_drv.ko
```

这时候 insmod led_platform_dev.ko 后没有任何输入，是因为只是把平台驱动结构放到了 bus 平台总线的“drv”链表中。而此时另一边的“设备链表”中还没有同名的“device”。接着再注册“led_platform_dev.ko”

```
# insmod led_platform_drv.ko
# insmod led_platform_dev.ko
led_probe, found led
#
```

两边有同名设备时注册。

卸载时，就会打印“platform_driver led_drv”结构的“.remove = led_remove”函数：

platform_driver led_drv->(.remove = led_remove)

-->int led_remove(struct platform_device *pdev)

-->printk("led_remove, remove led\n");

卸载是“void led_drv_exit(void)”

-->platform_driver_unregister (&led_drv);

-->driver_unregister(&drv->driver);

-->bus_remove_driver(drv);

卸载是从平台总线的“derive”链表上取下注册上的平台设备结构，根据设备的“同名名字”找到“平台驱动”层之前相对应的“平台驱动结构”，调用里面的“.remove”函数。

卸载时出错：提示为没有“release()”函数。

```
# rmmod led_dev
led_remove, remove led        没有"release()"函数。
Device 'myled' does not have a release() function, it is broken and must be fixed.
WARNING: at drivers/base/core.c:107 device_release()
[<c002fde8>] (dump_stack+0x0/0x14) from [<c01ba614>] (device_release+0x84/0x98)
[<c01ba590>] (device_release+0x0/0x98) from [<c0178714>] (kobject_cleanup+0x68/0x80)
[<c01786ac>] (kobject_cleanup+0x0/0x80) from [<c0178740>] (kobject_release+0x14/0x18)
 r7:00000000 r6:c3faa000 r5:c017872c r4:bf002488
[<c017872c>] (kobject_release+0x0/0x18) from [<c01794d0>] (kref_put+0x8c/0xa4)
[<c0179444>] (kref_put+0x0/0xa4) from [<c01786a4>] (kobject_put+0x20/0x28)
 r5:bf0025a0 r4:bf002400
[<c0178684>] (kobject_put+0x0/0x28) from [<c01bad1c>] (put_device+0x1c/0x20)
[<c01bad00>] (put_device+0x0/0x20) from [<c01bf434>] (platform_device_put+0x1c/0x20)
[<c01bf418>] (platform_device_put+0x0/0x20) from [<c01bf75c>] (platform_device_unregister+0x1c/0x20)
[<c01bf740>] (platform_device_unregister+0x0/0x20) from [<bf002034>] (led_dev_exit+0x14/0x1c [led_dev])
 r4:c036717c
[<bf002020>] (led_dev_exit+0x0/0x1c [led_dev]) from [<c00620e4>] (sys_delete_module+0x214/0x29c)
[<c0061ed0>] (sys_delete_module+0x0/0x29c) from [<c002bea0>] (ret_fast_syscall+0x0/0x2c)
 r8:c002c044 r7:00000081 r6:0009fbac r5:be943ec4 r4:000a104c
#
```

找一个类似的 release()函数的用法（平台设备里的 release()函数用法）：

```
struct platform_device {
    const char  * name;
    u32     id;
    struct device   dev;
    u32     num_resources;
    struct resource * resource;
};
```

平台设备 结构体定义中有一个“device”结构，这个“device”结构在“device.h”中定义：其中有定义“void (*release)(struct device * dev);”。我们要提供这个 relase 函数，这里什么都不用做。以后可以放一些硬件相关的到里面。这个例子里暂时不做任何事情。

修改代码如下：

```
void led_release(struct device * dev)
{//release函数留空什么也不做.虽然什么也不作但卸载"led_dev.ko"时需要它，不然提示没有release函数.

}

static struct platform_device led_dev = {
    .name           = "myled", //平台设备的名字为"myled".
    .id         = -1,
    .num_resources      = ARRAY_SIZE(led_resource),
    .resource       = led_resource,
    .dev = {
        .release = led_release, //需要这个release函数.在"device"结构体中定义.
    },
};
```

```
# insmod led_dev.ko
led_probe, found led
# rmmod led_dev
led_remove, remove led
#
```

重新注册“dev.ko”后，是将一个平台设备结构注册到平台总线下的“平台设备”链表，这时另一方“平台驱动”通过“设备名”匹配到一个“平台驱动”结构后，调用此结构下的“.probe”函数，为这个“平台设备”作相关的处理；当“insmod xx_dev.ko”后，是从“平台设备”结构链表中删除了 这个“xx_dev.ko”，这时另一端相关同名的“平台驱动”就调用自己结构中的“.remove”函数作相关清理工作。

第二部分:下面在“.probe”函数中做有意义的事情：注册字符设备。

6.1,定义主设备号：

```
static int major;
-->int led_probe(struct platform_device *pdev)
-->//6.2.注册字符设备：major = register_chrdev(0, "myled", led_fops);
```

7.1，构造一个 file_operations 结构体。

```
static struct file_operations led_fops = {
    .owner = THIS_MODULE,    //这是一个宏推向编译模块时自动创建__this_module变量.
    .open    = led_open,
    .write = led_write,
}
```

8.1，创建‘类’

（可以不需要，只是让系统自动创建‘设备节点’）,先定义 class 结构。

```
static struct class *cls;
-->int led_probe(struct platform_device *pdev)
-->//8.2,创建设备类: cls = class_create(THIS_MODULE, "myled");
-->//8.3,在cls类中建设备:class_device_create(cls, NULL, MKDEV(major, 0), NULL, "led");
   //则设备节点为/dev/led。
```


第三部分：要操作硬件（哪个寄存器等等操作），

则要：根据 platform_device 的资源进行 ioremap .
int led_probe(struct platform_device *pdev)

2.1,根据 platform_device 的资源进行 ioremap .

```
    //2.1.1，定义资源.
 struct resource *res;
    //2.1.2,获得资源：led_dev中有IORESOURC_MEM 内存资源，IERSOURCE_IRQ中断资源。
    res = platform_get_resource(pdev, IORESOURCE_MEM, 0); //pdev平台设备，IORESOURCE_MEM内存类资源.

//2.1.3,定义GPIO_CONF控制寄存器 和 GPIO_DAT寄存器。
static volatile unsigned long *gpio_con;
static volatile unsigned long *gpio_dat;

int led_probe(struct platform_device *pdev)
-->//2.1.4，ioremap：res资源里有start，ioremap需要大小，资源res中有大小'资源起始地址减去资源结束地址'加1.
    gpio_con = ioremap(res->start, res->end - res->start +1);
```

```
	gpio_dat = gpio_con + 1; //指针加1就相当于加4字节，就指向了gpio_dat寄存器了。

//2.1.5，哪个pin引脚。
static int pin;

int led_probe(struct platform_device *pdev)
-->//2.1.6,获得IORESOURCE_IRQ中断资源。
	res = platform_get_resource(pdev, IORESOURCE_IRQ, 0); //0表示这类IORESOURCE_IRQ资源里的第0个.
	pin = res->start;
	//以上寄存器有了'gpio_con,gpio_dat'。pin引脚也有了。led_probe()就操作完成。
```

第四部分：file_operations led_fops 结构中的 open 函数的处理。

open 后要把这些引脚配置成“输出”引脚。

9.1,open 后，要把引脚配置成"输出"引脚。

```
static    int led_open(struct inode *inode, struct file *file)
{
	//9.1.1,配置为输出引脚。
	*gpio_con &= ~(0x3<<(pin*2));	//pin清为0.
	*gpio_con |=	(0x1<<(pin*2));	//pin清为0后再'或'上1.
	return 0;
}
//9.2,'.write'函数。
static ssize_t led_write(struct file *file, const char __user *buf, size_t count, loff_t * ppos)
{
	int val;
	copy_from_user(&val, buf, count); //	copy_to_user();

	if (val == 1)
	{
		// 点灯
		*gpio_dat &= ~(1<<pin); //led_drv.c可以保持稳定，到底是哪个引脚点LED只需要修改led_dev.c中
的资源定义。
	}
	else
	{
		// 灭灯
		*gpio_dat |= (1<<pin);
	}
	return 0;
}
```

第五部分：出口函数

```
static void led_drv_exit(void)
{
    //5.1,卸载平台驱动。
    platform_driver_unregister (&led_drv);
    //8.3_1,在cls类下面创建设备，则要卸载掉类设备。
    class_device_destroy(cls, MKDEV(major, 0));
    //8.4_1,去掉类。
    class_destroy(cls);
    //6.2_1.卸载字符设备：
    unregister_chrdev(major, "myled");
    //2.1.4_1: ioremap()，则这里"iounmap()":
    iounmap(gpio_con);
}
```

最后作测试：

```
# insmod led_dev.ko
# insmod led_drv.ko
led_drv: Unknown symbol iounmap
led_drv: Unknown symbol copy_from_user
led_drv: Unknown symbol ioremap
insmod: cannot insert `led_drv.ko': Unknown symbol in module (-1): No such file or directory
#
```

先装载led_dev.ko一开始找不平始驱动，但当led_drv.ko装载后还会找一遍平台驱动的"drv链表"。

有些头文件不有包含。


```
#include <asm/uaccess.h>
#include <asm/io.h>
```

这两个头文件要包含。

接着再编译再装载：

```
# insmod led_drv.ko
led_probe, found led
# ls /dev/led -l
crw-rw----    1 0        0         252,   0 Jan  1 00:57 /dev/led
#
```

最后就可以“led_test_on”"led_test_off"来测试 LED 的亮灭。

要是想测试其他 LED 的情况，则修改“led_dev.c”中的引脚资源：

```
static struct resource led_resource[] = {
    [0] = {
        .start = 0x56000050,
        .end   = 0x56000050 + 8 - 1,
        .flags = IORESOURCE_MEM,
    },
    [1] = {
        .start = 5,
        .end   = 5,
        .flags = IORESOURCE_IRQ,
    }
```

换成另一盏LED

而“led_drv.c”就可以保持不动。

分层是每一层专注做自己的事情。分离经常使用的就是“bus_drv_dev 模型”中，这个模型只是提供一种机制，在这个机制中做什么事件是由自己定义的，在“.probe”函数中的操作完全 是由自己决定的（.probe 中的操作是核心内容）。