Linux设备驱动分层

Linux下的驱动往往是分层的。用以在不同的层处理不同的内容。因此linux引入了platform驱动。

简介

Platform驱动框架分为总线、设备和驱动。这里主要以设备树下的platform驱动。其中，platform总线部分是由内核来管理的，不需要我们来写，设备呢我们这里就直接用设备树了，所以设备的一些描述是直接写进了设备树。因此我们主要是来写驱动。也就是platform driver.

platform总线

linux内核使用bus_type结构体表示总线，该结构体定义在device.h头文件里。其结构体如下：

```
struct bus_type{
    const char *name; /* 总线名字 */
    const char *dev_name;
    struct device *dev_root;
    struct device_attribute *dev_attrs;
    const struct attribute_group **bus_groups; /* 总线属性 */
    const struct attribute_group **dev_groups; /* 设备属性 */
    const struct attribute_group **drv_groups; /* 驱动属性 */

    int (*match)(struct device *dev, struct device_driver *drv);
    int (*uevent)(struct device *dev, struct kobj_uevent_env *env);
    int (*probe)(struct device *dev);
    int (*remove)(struct device *dev);
    void (*shutdown)(struct device *dev);
    int (*online)(struct device *dev);
    int (*offline)(struct device *dev);
    int (*suspend)(struct device *dev, pm_message_t state);
    int (*resume)(struct device *dev);
    const struct dev_pm_ops *pm;
    const struct iommu_ops *iommu_ops;
    struct subsys_private *p;
    struct lock_class_key lock_key;
};
```

其中要重点关注match函数，这个函数主要是用来完成设备和驱动之间的匹配，总线会根据match函数，来根据注册的设备来查找相应的驱动，或者根据注册的驱动查找设备，所以不管是platform总线或者什么spi、IIC之类的总线，都必须实现这个match函数，函数的两个参数分别是dev和drv，就是设备和驱动。

而platform只是bus_type的一个实例，定义在platform.c,platform总线的定义：

```
struct bus_type platform_bus_type = {
    .name = "platform",
    .dev_groups = platform_dev_groups,
    .match = platform_match,
    .uevent = platform_uevent,
```

```
    .pm = &platform_dev_ops,
};
```

重点还是看match函数，定义：

```
static int platform_match(struct device *dev, struct device_driver *drv)
{
    struct platform_device *pdev = to_platform_device(dev);
    struct platform_driver *pdrv = to_platform_driver(drv);

    if (pdev->driver_override)
        return !strcmp(pdev->driver_override, drv->name);

    if (of_driver_match_device(dev, drv))
        return 1;

    if (acpi_driver_match_device(dev, drv))
        return 1;

    if (pdrv->id_table)
        return platform_match_id(pdrv->id_table, pdev) != NULL;

    return (strcmp(pdev->name, drv->name) == 0);
}
```

可以看到有四种匹配方法：

- 第一种是of类型的匹配

设备树下创建节点

总线是由内核管理，而设备就是我们在设备树里描述设备信息。

这里有一个比较重要的点，是设备树下的compatible属性的内容。因为platform总线是靠设备节点的compatible属性的内容来匹配驱动的。所以一定要注意这个。

比如来写一个LED灯的设备。

设备树里描述为：

```
gpioled{
    #address-cells=<1>;
    #size-cells=<1>;
    compatible = "keys-gpioled";
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_led>;
    led-gpio = <&gpio1 3 GPIO_ACTIVE_LOW>;
    status = "okay";
};
```

驱动里

使用设备树时platform会使用of_match_table来保存兼容性值。所以要在驱动里写：

```
statuc const struct of_device_id leds_of_match[] = {
    {.compatible = "keys-gpioled"},
    {}
};

MODULE_DEVICE_TABLE(of, leds_of_match);

static struct platform_driver leds__platform_driver = {
    .driver = {
        .name = "imx6ul-led",
        .of_match_table = leds_of_match,
    },
    .probe = leds_probe,
    .remove = leds_remove,
};
```

当驱动和设备匹配成功后，就会执行probe函数，然后在probe函数里面执行设备驱动哪些东西了。