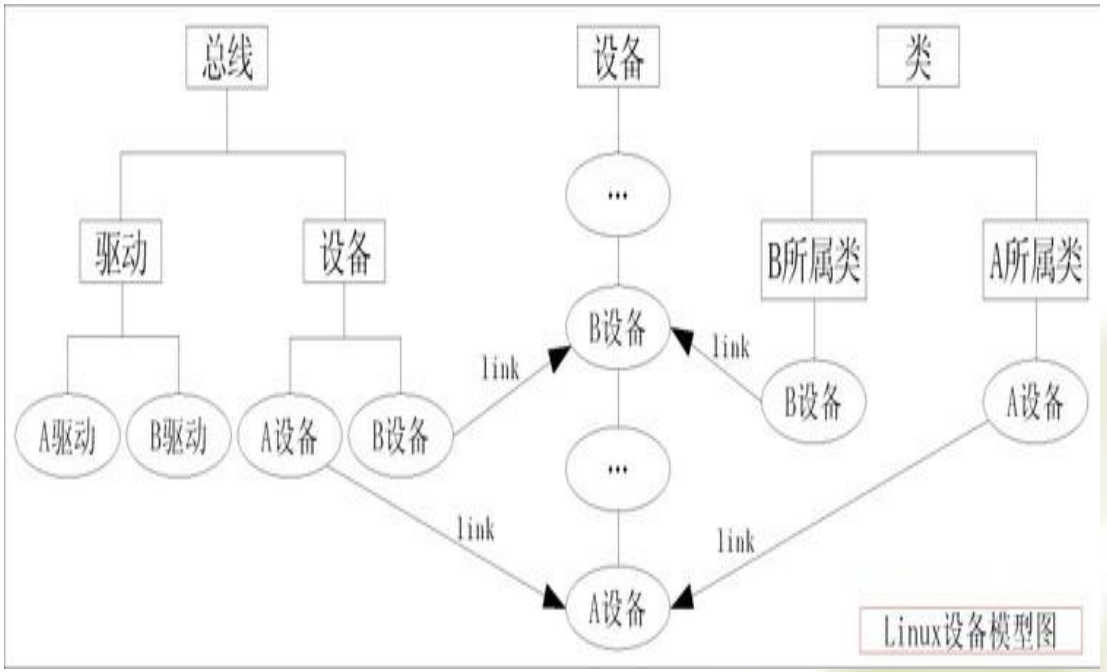
# 9.3 设备驱动模型


西安邮电大学

# 设备驱动模型的引入


总线
设备
类
驱动
设备
...
B所属类
A所属类
B设备
link
link
A驱动
B驱动
A设备
B设备
B设备
A设备
...
link
link
A设备
Linux设备模型图

# 设备驱动模型的引入

由于Linux支持世界上几乎所有的、不同功能的硬件设备，导致Linux内核中有一半的代码是设备驱动，而且随着硬件的快速升级换代，设备驱动的代码量也在快速增长。为了降低设备多样性带来的Linux驱动开发的复杂度，以及设备热拔插处理、电源管理等，Linux内核提出了设备模型（也称作Driver Model）的概念。设备模型将硬件设备归纳、分类，然后抽象出一套标准的数据结构和接口。驱动的开发，就简化为对内核所规定的数据结构的填充和实现。

因此，Linux设备驱动模型是一种抽象，为内核建立起统一的设备模型。其目的是：

提供一个对系统结构的一般性抽象描述。

Linux设备模型跟踪所有系统所知道的设备，以便让设备驱动模型的核心程序协调驱动与新设备之间的关系。

# Linux设备驱动模型引入的目的

| 功能 | 描述 |
| --- | --- |
| 电源管理和系统关机 | 设备之间大多情况下有依赖、耦合，因此要实现电源管理就必须对系统的设备结构有清楚的理解，应知道先关哪个然后才能再关哪个。 |
| 与用户空间通信 | sys虚拟文件系统的实现与设备模型密切相关，并且向外界展示了它所表述的结构。向用户空间所提供的系统信息，以及改变操作参数的接口，都要通过/sys文件系统实现，即通过设备模型实现。 |
| 热插拔设备 | 处理与用户空间进行热插拔设备的通信是通过设备模型管理的。 |
| 设备分类机制 | 设备模型包括了对设备分类的机制，它会在更高的功能层上描述这些设备，并使得这些设备对用户空间可见。尤其是将命名设备的功能从内核层转移到用户层，大大提高了设备管理的灵活性。 |
| 对象生命周期管理 | 设备模型实现一系列机制以处理对象的生命周期、对象之间的关系，以及这些对象在用户空间中的表示。简化编程人员创建和管理对象的工作。 |

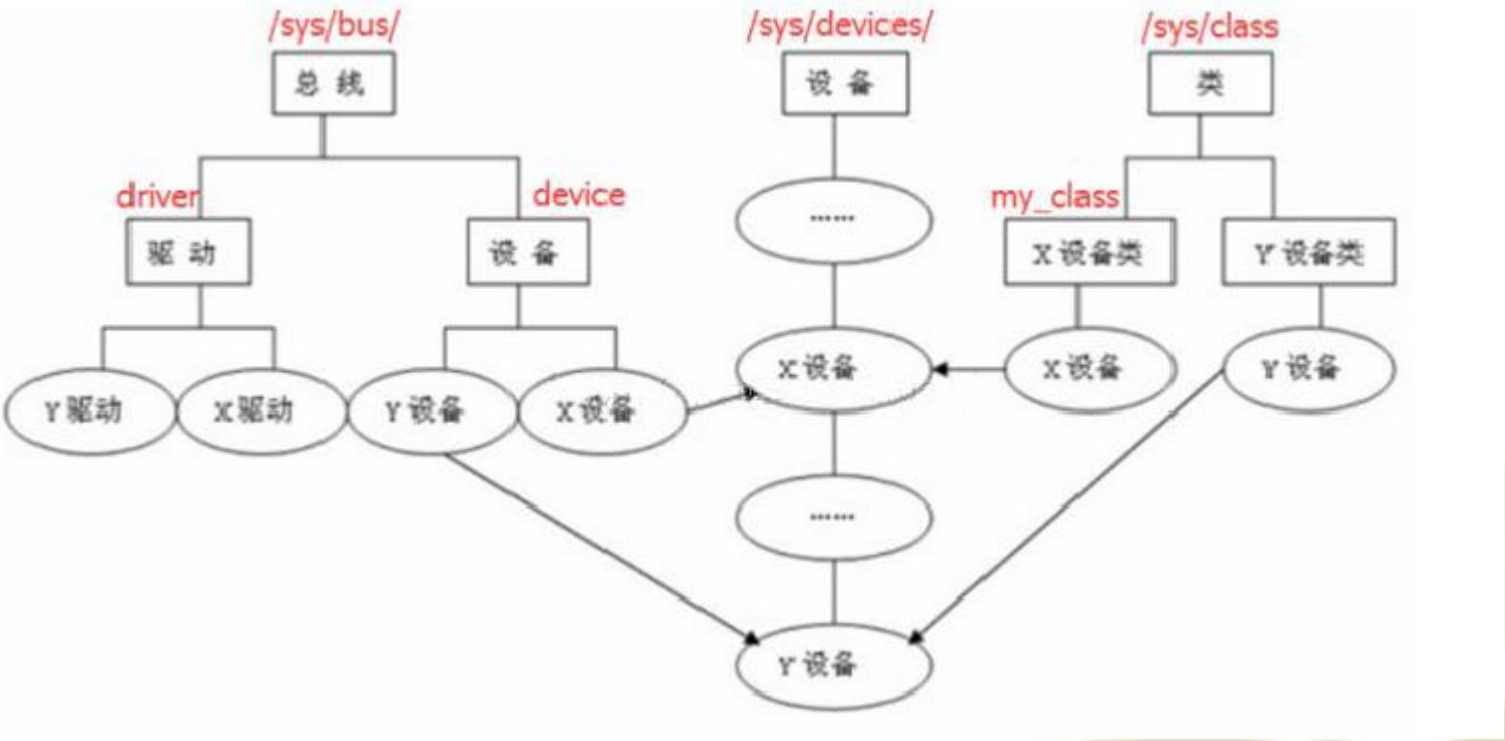
# sys文件系统

Sysfs文件系统是一个类似于proc文件系统的特殊文件系统，用于将系统中的设备组织成层次结构，并向用户程序提供详细的内核数据信息。

也就是说，在用户态可以通过对sys文件系统的访问，来看内核态的一些驱动或者设备等信息，如图为sys文件系统目录。

```
[clj@localhost ~]$ cd /sys
[clj@localhost sys]$ ls
block  class  devices   fs          kernel  power
bus    dev    firmware  hypervisor  module
[clj@localhost sys]$ cd bus
[clj@localhost bus]$ ls
acpi         edac          ishtp         nvmem        scsi         usb-serial
clockevents  event_source  machinecheck  pci          serio        virtio
clocksource  hid           mdio_bus      pci_express  spi          workqueue
container    i2c           memory        platform     thunderbolt  xen
cpu          iio           node          pnp          usb
[clj@localhost bus]$ cd ../devices/
[clj@localhost devices]$ ls
breakpoint   msr          platform  software  tracepoint
LNXSYSTM:00  pci0000:00   pnp0      system    virtual
```

# Linux设备模型




Linux设备驱动模型使用一系列抽象（面向对象设计里的类），提供统一的设备管理视图，这些抽象包括：总线、类、设备和设备驱动。

Bus（总线）：总线是CPU和一个或多个设备之间信息交互的通道。为了方便设备模型的抽象，所有的设备都应连接到总线上。

# Linux设备模型

Class（分类）：在Linux设备模型中，Class的概念非常类似面向对象程序设计中的Class（类），它主要是集合具有相似功能或属性的设备，这样就可以抽象出一套可以在多个设备之间共用的数据结构和接口函数。因而从属于相同Class的设备的驱动程序，就不再需要重复定义这些公共资源，直接从Class中继承即可。

Device（设备）：抽象系统中所有的硬件设备，描述它的名字、属性、从属的Bus、从属的Class等信息。

Device Driver（设备驱动）：Linux设备模型用Driver抽象硬件设备的驱动程序，它包含设备初始化、电源管理相关的接口实现。而Linux内核中的驱动开发，基本都围绕该抽象进行（实现所规定的接口函数）。

# 核心对象之kobject

```
struct kobject {
    const char          *name;      //   对象的名字
    struct list_head     entry;
    struct kobject       *parent;
    struct kset          *kset;     //   用来指向父类对象的kset
    struct kobj_type     *ktype;    //   指向一个kobj_type对象
    struct sysfs_dirent     *sd;
    struct kref          kref;          //   kobject的引用计数
    unsigned int state_initialized:1; //   该对象是否被初始化了
                                          的状态标志位
    ... ...
};
```

# 核心对象之kobject

kobject结构体是设备驱动模型底层的一个结构体，这个结构体是设备驱动模型下的所有对象的一个基本单元，他是对设备驱动模型下所有对象抽象出来的共有的部分；

kobject结构体提供了一些公共型的服务：对象引用计数、维护对象链表、对象上锁、对用户空间的表示。

设备驱动模型中的各种对象其内部都会包含一个kobject，地位相当于面向对象思想中的总基类。

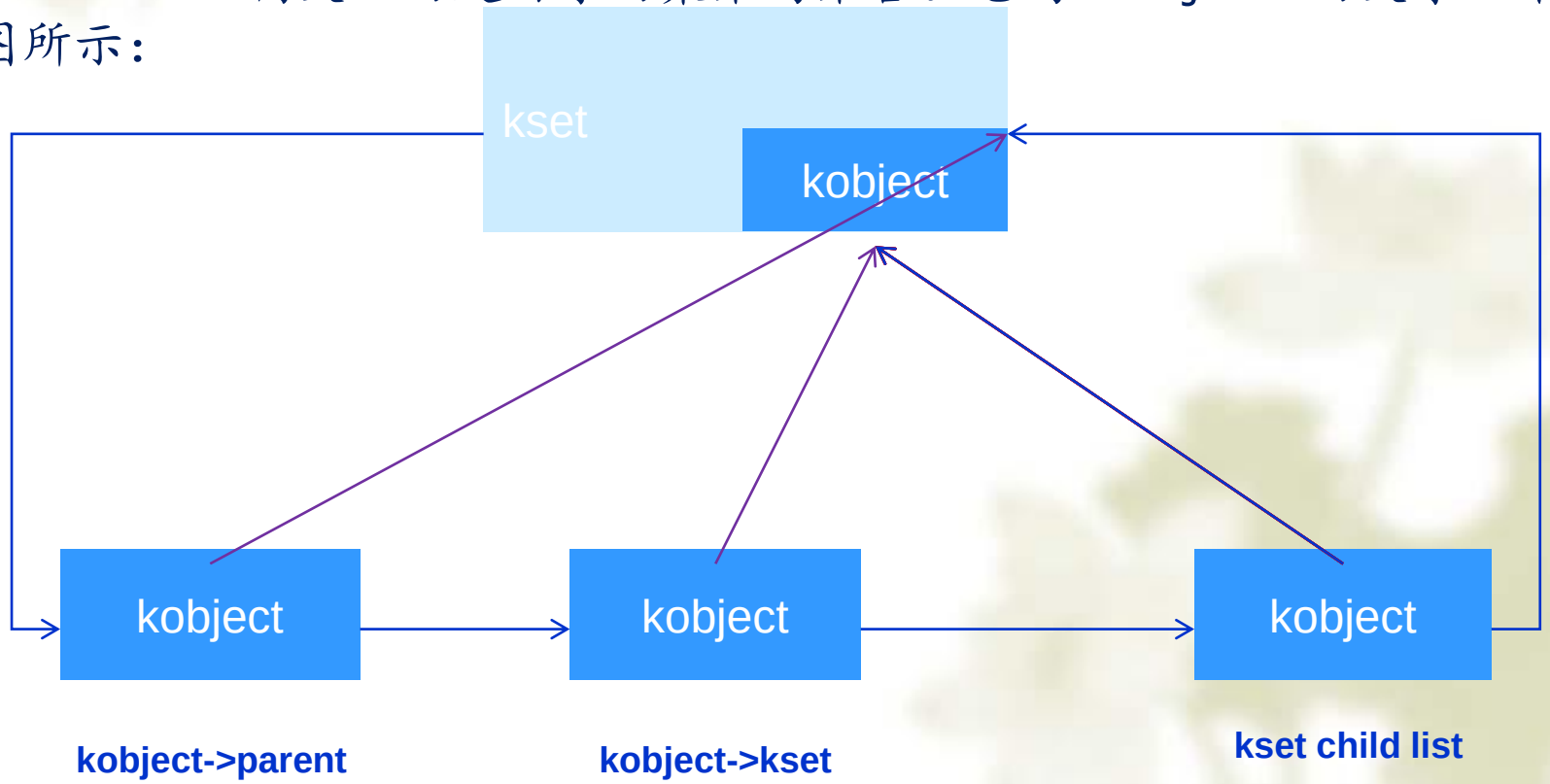
# 对象集合体kset

```
struct kset {
    struct list_head list;    //用来链接该目录下的所有kobject对象
    spinlock_t list_lock;
    struct kobject kobj;    //这个kobject就是本目录对应的对象结构体
    const struct kset_uevent_ops *uevent_ops; //指向一个用于处理集合中kobject对象的热插拔操作的结构体
};
```

# 对象集合体kset

kset 是嵌入相同类型结构的 kobject 的集合，可以把它看成是一个容器，可将所有相关的kobject对象聚集起来，比如“全部的块设备”就是一个kset

kset 结构关心的是对象的聚集与集合。它与 kobject 的关系如下图所示：

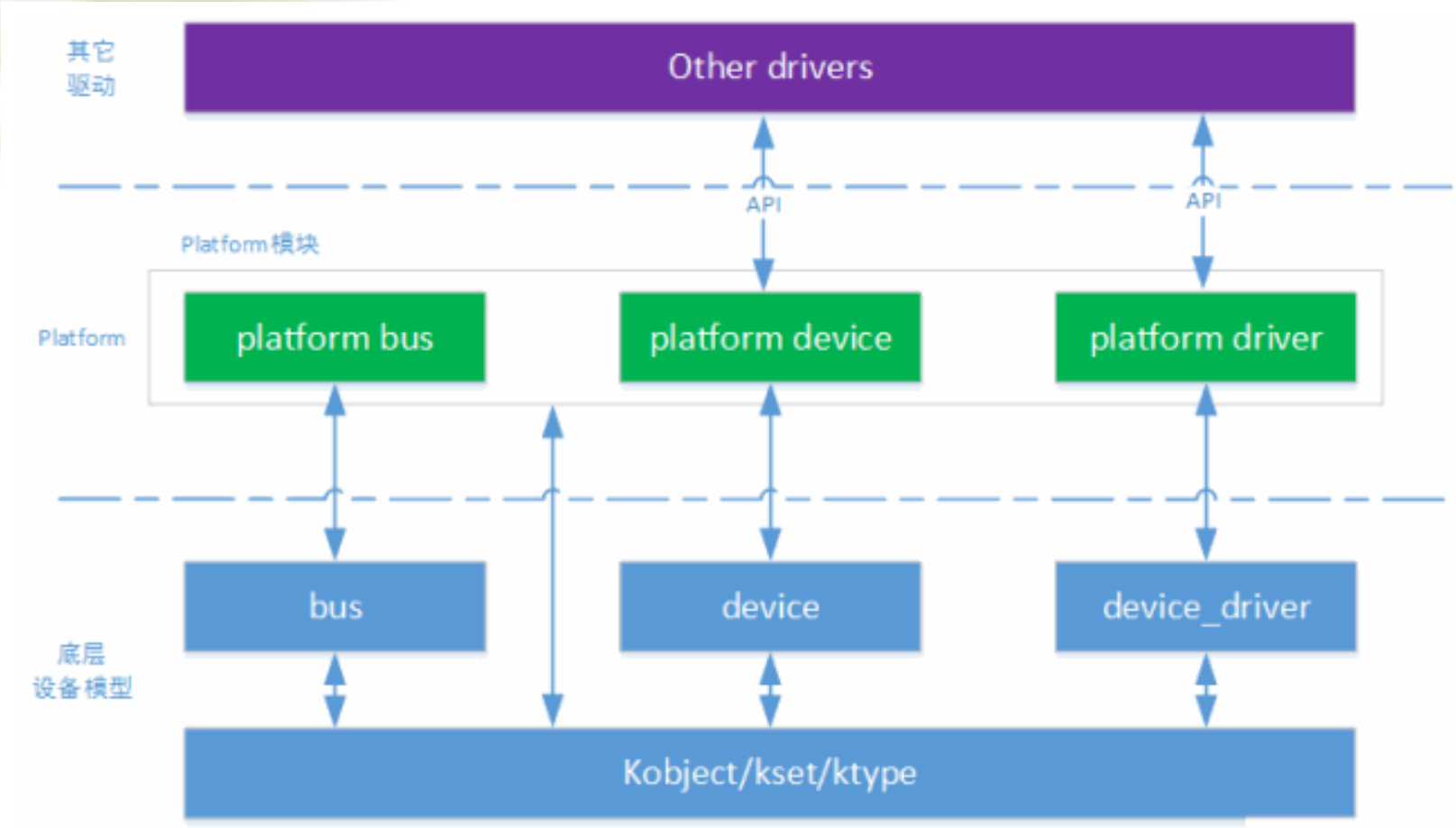


# 共同特性的ktype

kobject是一个抽象而基本的对象。对于一族具有共同特性的kobject，就是用定义在头文件<linux/kobject.h>中的ktype来描述：

```
struct kobj_type {
    void (*release)(structkobject *); //析构函数
    struct sysfs_ops  *sysfs_ops; //对属性进行操作的读写
                                  函数show()和store()。
    struct attribute  **default_attrs;//描述了给定对象的特征
};
```

# platform平台总线驱动模型

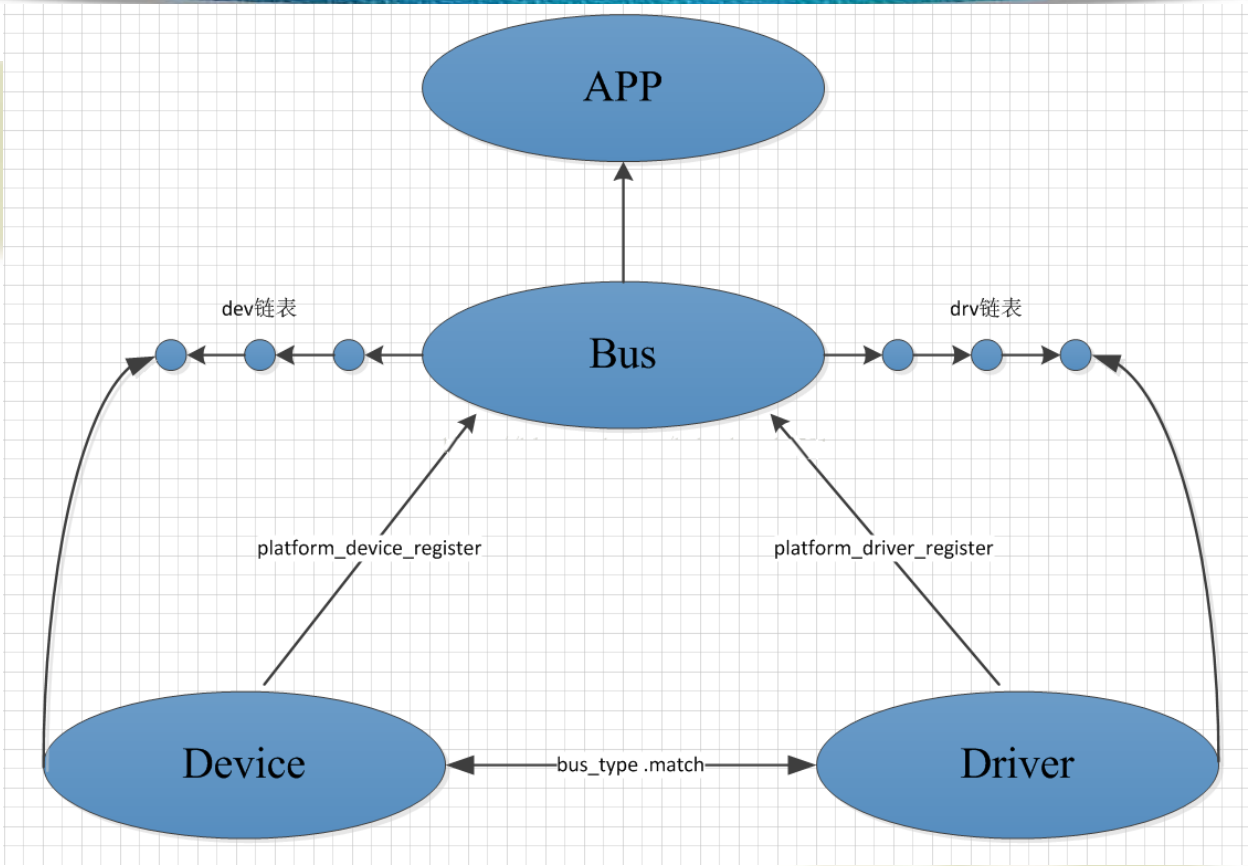


# platform平台总线驱动模型

为了解决驱动代码和设备信息耦合,Linux提出了platform bus(平台总线)的概念，即使用虚拟总线将设备信息和驱动程序进行分离。平台总线会维护两条链表，分别管理设备和驱动，当一个设备被注册到总线上的时候，总线会根据其名字搜索对应的驱动，如果找到就将设备信息导入驱动程序并执行驱动；当一个驱动被注册到平台总线的时候，总线也会搜索设备。总之，平台总线负责将设备信息和驱动代码匹配，这样就可以做到驱动和设备信息的分离。

与传统的bus/device/driver机制相比，platform由内核进行统一管理，在驱动中使用资源，提高了代码的安全性和可移植性。当硬件部分的时序变了或者芯片替换了，我们只需要修改硬件部分的代码，还有一部分代码是属于内核的稳定部分是不用修改的，这就是一种通用的接口。因此，本讲重点讲解platform模型。

# platform平台总线驱动模型




platform平台总线是一条虚拟总线。其中platform_device为相应的设备，platform_driver为相应的驱动。

# 描述平台的结构platform_driver

```
1  struct platform_driver {
2          int (*probe)(struct platform_device *);
3          int (*remove)(struct platform_device *);
4          void (*shutdown)(struct platform_device *);
5          int (*suspend)(struct platform_device *, pm_message_t state);
6          int (*resume)(struct platform_device *);
7          struct device_driver driver;
8          const struct platform_device_id *id_table;
9          bool prevent_deferred_probe;
10 };
```

可以看到platform_driver结构体中包含了probe和remove等相关操作，同时还内嵌了device_driver结构体。

# 描述设备驱动的结构device_driver

```
1  struct device_driver {
2          const char              *name;          // 驱动的名字
3          struct bus_type         *bus;           // 所属总线
4
5          struct module           *owner;
6          const char              *mod_name;      /* used for built-in modules */
7
8          bool suppress_bind_attrs;       /* disables bind/unbind via sysfs */
9
10         const struct of_device_id       *of_match_table;
11         const struct acpi_device_id     *acpi_match_table;
12
13         int (*probe) (struct device *dev);      // 驱动挂载的时候调用
14         int (*remove) (struct device *dev);     // 驱动卸载的时候调用
15         void (*shutdown) (struct device *dev);
16         int (*suspend) (struct device *dev, pm_message_t state);
17         int (*resume) (struct device *dev);
18         const struct attribute_group **groups;
19
20         const struct dev_pm_ops *pm;
21
22         struct driver_private *p;
23 };
```

从中看到最后一个域指针p指向driver_private，这是一个描述驱动私有数据的结构

# 描述私有数据的driver_private

```
1  struct driver_private {
2          struct kobject kobj;              // 在sysfs中代表目录本身
3          struct klist klist_devices;       // 驱动链表即我们上面所说的drv链表
4          struct klist_node knode_bus;      // 挂载在总线上的驱动链表的节点
5          struct module_kobject *mkobj;     // driver与相关的module之间的联系
6          struct device_driver *driver;
7  };
8  #define to_driver(obj) container_of(obj, struct driver_private, kobj)
```

无论是从名字上，还是上面的注释中 我们都可以看到 void *driver_data;就是为设计用来放置不同驱动数据的

# 描述设备的结构platform_device

```
struct platform_device {
        const char      *name;          // 设备的名字这将代替device->dev_id，用作sys/device下显示的目录
        int             id;             // 设备id，用于给插入给该总线并且具有相同name的设备编号，如果只有一
        bool            id_auto;
        struct device   dev;            // 内嵌device结构
        u32             num_resources;  // 资源的数目
        struct resource *resource;      // 资源

        const struct platform_device_id *id_entry;

        /* MFD cell pointer */
        struct mfd_cell *mfd_cell;

        /* arch specific additions */
        struct pdev_archdata    archdata;
};
```

其中sturct resource在前面的I/O空间管理一讲已经讲过。

# 设备与驱动匹配的过程

设备与驱动如何匹配，打个比喻：

a -- 红娘（总线）负责男方（设备）和女方（驱动）的撮合；

b -- 男方（女方）找到红娘，说我来登记一下，看有没有合适的姑娘（小伙子）—— 设备或驱动的注册；

c -- 红娘这时候就需要看看有没有匹配的姑娘（小伙子）——match 函数进行匹配，看name是否相同；

d -- 如果没有找到匹配的，就告诉男方（女方）没有合适的对象，先等着，别急 —— 设备和驱动会等待，直到匹配成功；

e -- 终于遇到匹配的，那就结婚呗！结完婚，男方就向女方交代，他有那些资源（比如，存款，房子等）（struct resource *resource），女方说好啊，于是去拿钱买东西（买书，买衣服等）（int (*probe)(struct platform_device *) 匹配成功后驱动执行的第一个函数）。

# 设备与驱动匹配的代码

首先是总线出场，系统为platform总线定义了一个bus_type 的实例platform_bus_type，其定义如下：

```
1  struct bus_type platform_bus_type = {
2          .name          = "platform",
3          .dev_groups     = platform_dev_groups,
4          .match          = platform_match,
5          .uevent          = platform_uevent,
6          .pm          = &platform_dev_pm_ops,
7  };
```

# 设备与驱动匹配的代码

其后，又是怎样工作的呢？在platform.c 可以看到注册函数：

```
/**
 * __platform_driver_register - register a driver for platform-level devices
 * @drv: platform driver structure
 * @owner: owning module/driver
 */
int __platform_driver_register(struct platform_driver *drv,
                               struct module *owner)
{
        drv->driver.owner = owner;
        drv->driver.bus = &platform_bus_type;
        drv->driver.probe = platform_drv_probe;
        drv->driver.remove = platform_drv_remove;
        drv->driver.shutdown = platform_drv_shutdown;

        return driver_register(&drv->driver);
}
EXPORT_SYMBOL_GPL(__platform_driver_register);
```

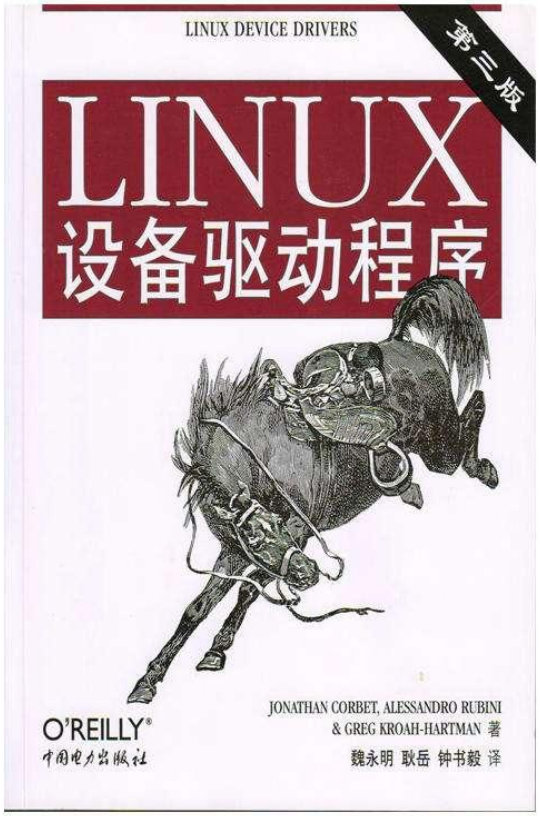
# 设备与驱动匹配的代码

```
1  static int platform_match(struct device *dev, struct device_driver *drv)
2  {
3          struct platform_device *pdev = to_platform_device(dev);
4          struct platform_driver *pdrv = to_platform_driver(drv);
5
6          匹配设备树信息，如果有设备树，就调用 of_driver_match_device() 函数进行匹配
7          if (of_driver_match_device(dev, drv))
8                  return 1;
9
10
11         匹配id_table
12         if (pdrv->id_table)
13                 return platform_match_id(pdrv->id_table, pdev) != NULL;
14
15         最基本匹配规则
16         return (strcmp(pdev->name, drv->name) == 0);
17 }
```

# 设备与驱动匹配的代码

在 platform_bus_type 中调用 了platform_match，基本的匹配规则是名字匹配。设备驱动中引入platform概念，隔离BSP和驱动。在BSP中定义platform设备和设备使用的资源、设备的具体匹配信息，而在驱动中，只需要通过API去获取资源和数据，做到了板相关代码和驱动代码的分离，使得驱动具有更好的可扩展性和跨平台性。

# 参考文献




- 1.《Linux 驱动开发》
- 2.网上有大量详尽的驱动开发资料，读者可自行查阅，推荐一篇：
- https://blog.csdn.net/zqixiao_09/article/details/50888795

# 带着疑问上路

platform平台模型的优势是什么？

# 谢谢大家！

THANK YOU