



下载APP



31 | 瞧一瞧Linux：如何获取所有设备信息？

2021-07-19 LMOS

《操作系统实战45讲》

课程介绍 >



讲述：陈晨

时长 16:39 大小 15.26M



你好，我是 LMOS。

前面我们已经完成了 Cosmos 的驱动设备的建立，还写好了一个真实的设备驱动。

今天，我们就来看看 Linux 是如何管理设备的。我们将从 Linux 如何组织设备开始，然后研究设备驱动相关的数据结构，最后我们还是要一起写一个 Linux 设备驱动实例，这样才能真正理解它。

感受一下 Linux 下的设备信息

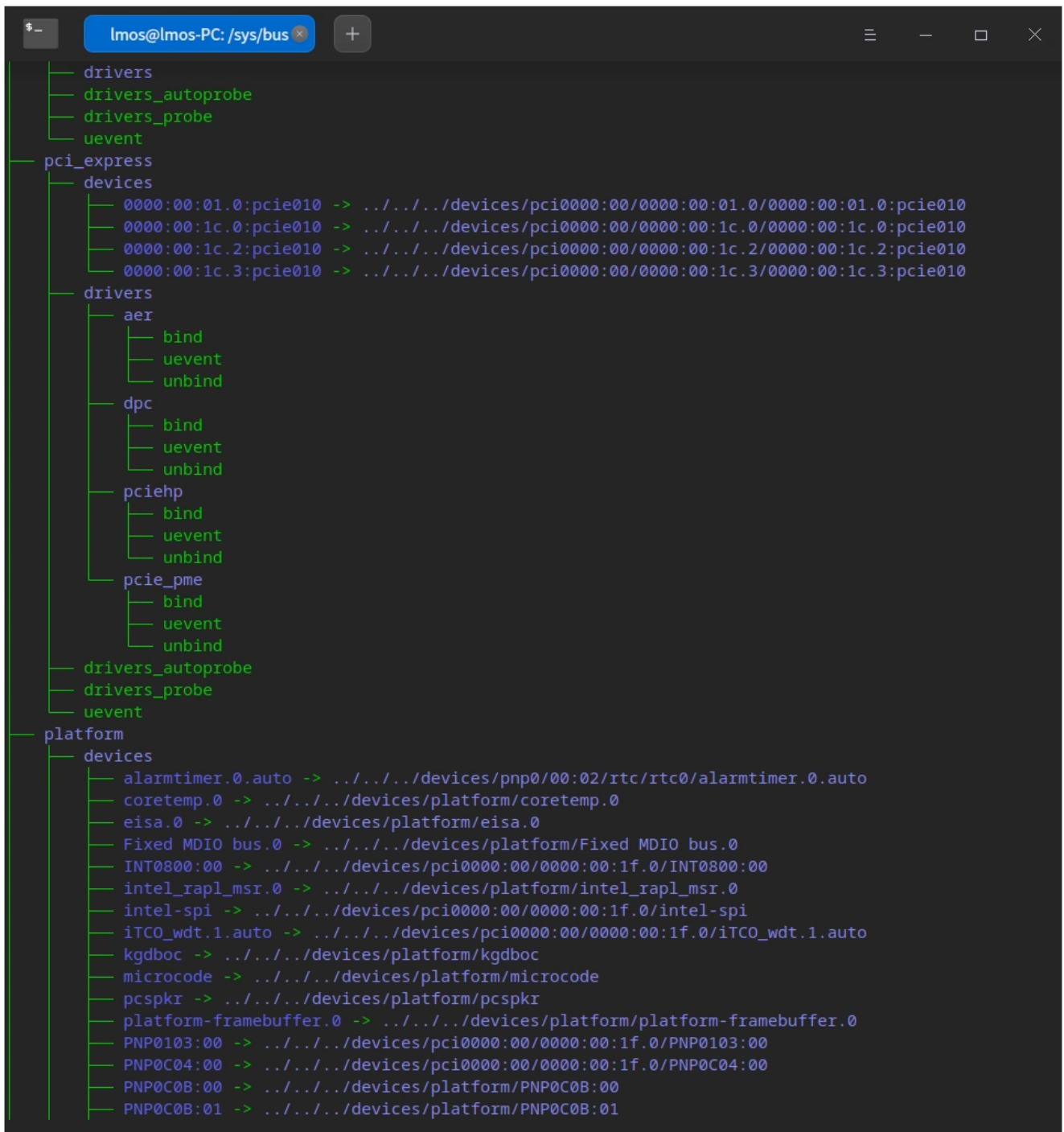


Linux 的设计哲学就是一切皆文件，各种设备在 Linux 系统下自然也是一个个文件。不过这个文件并不对应磁盘上的数据文件，而是对应着存在内存当中的设备文件。实际上，我

们对设备文件进行操作，就等同于操作具体的设备。

既然我们了解万事万物，都是从最直观的感受开始的，想要理解 Linux 对设备的管理，自然也是同样的道理。那么 Linux 设备文件在哪个目录下呢？其实现在我们在 /sys/bus 目录下，就可以查看所有的设备了。

Linux 用 BUS (总线) 组织设备和驱动，我们在 /sys/bus 目录下输入 tree 命令，就可以看到所有总线下的所有设备了，如下图所示。



```
lmos@lmos-PC: /sys/bus
├── drivers
│   ├── drivers_autoprobe
│   ├── drivers_probe
│   └── uevent
├── pci_express
│   ├── devices
│   │   ├── 0000:00:01.0:pcie010 -> ../../../../devices/pci0000:00/0000:00:01.0/0000:00:01.0:pcie010
│   │   ├── 0000:00:1c.0:pcie010 -> ../../../../devices/pci0000:00/0000:00:1c.0/0000:00:1c.0:pcie010
│   │   ├── 0000:00:1c.2:pcie010 -> ../../../../devices/pci0000:00/0000:00:1c.2/0000:00:1c.2:pcie010
│   │   └── 0000:00:1c.3:pcie010 -> ../../../../devices/pci0000:00/0000:00:1c.3/0000:00:1c.3:pcie010
│   └── drivers
│       ├── aer
│       │   ├── bind
│       │   ├── uevent
│       │   └── unbind
│       ├── dpc
│       │   ├── bind
│       │   ├── uevent
│       │   └── unbind
│       ├── pciehp
│       │   ├── bind
│       │   ├── uevent
│       │   └── unbind
│       └── pcie_pme
│           ├── bind
│           ├── uevent
│           └── unbind
├── drivers_autoprobe
├── drivers_probe
├── uevent
└── platform
    ├── devices
    │   ├── alarmtimer.0.auto -> ../../../../devices/pnp0/00:02/rtc/rtc0/alarmtimer.0.auto
    │   ├── coretemp.0 -> ../../../../devices/platform/coretemp.0
    │   ├── eisa.0 -> ../../../../devices/platform/eisa.0
    │   ├── Fixed MDIO bus.0 -> ../../../../devices/platform/Fixed MDIO bus.0
    │   ├── INT0800:00 -> ../../../../devices/pci0000:00/0000:00:1f.0/INT0800:00
    │   ├── intel_rapl_msr.0 -> ../../../../devices/platform/intel_rapl_msr.0
    │   ├── intel-spi -> ../../../../devices/pci0000:00/0000:00:1f.0/intel-spi
    │   ├── iTCO_wdt.1.auto -> ../../../../devices/pci0000:00/0000:00:1f.0/iTCO_wdt.1.auto
    │   ├── kgdboc -> ../../../../devices/platform/kgdboc
    │   ├── microcode -> ../../../../devices/platform/microcode
    │   ├── pcspkr -> ../../../../devices/platform/pcspkr
    │   ├── platform-framebuffer.0 -> ../../../../devices/platform/platform-framebuffer.0
    │   ├── PNP0103:00 -> ../../../../devices/pci0000:00/0000:00:1f.0/PNP0103:00
    │   ├── PNP0C04:00 -> ../../../../devices/pci0000:00/0000:00:1f.0/PNP0C04:00
    │   ├── PNP0C0B:00 -> ../../../../devices/platform/PNP0C0B:00
    │   └── PNP0C0B:01 -> ../../../../devices/platform/PNP0C0B:01
```

Linux设备文件

上图中，显示了部分 Linux 设备文件，有些设备文件是链接到其它目录下文件，这不是重点，重点是你要在心中有这个目录层次结构，即**总线目录下有设备目录，设备目录下是设备文件**。

数据结构

我们接着刚才的图往下说，我们能感觉到 Linux 的驱动模型至少有三个核心数据结构，分别是总线、设备和驱动，但是要像上图那样有层次化地组织它们，只有总线、设备、驱动这三个数据结构是不够的，还得有两个数据结构来组织它们，那就是 kobject 和 kset，下面我们就去研究它们。

kobject 与 kset

kobject 和 kset 是构成 /sys 目录下的目录节点和文件节点的核心，也是层次化组织总线、设备、驱动的核心数据结构，kobject、kset 数据结构都能表示一个目录或者文件节点。下面我们先来研究一下 kobject 数据结构，代码如下所示。

[复制代码](#)

```
1 struct kobject {
2     const char      *name;           //名称，反映在sysfs中
3     struct list_head entry;          //挂入kset结构的链表
4     struct kobject  *parent;         //指向父结构
5     struct kset      *kset;          //指向所属的kset
6     struct kobj_type *ktype;
7     struct kernfs_node *sd;          //指向sysfs文件系统目录项
8     struct kref       kref;          //引用计数器结构
9     unsigned int state_initialized:1; //初始化状态
10    unsigned int state_in_sysfs:1;    //是否在sysfs中
11    unsigned int state_add_uevent_sent:1;
12    unsigned int state_remove_uevent_sent:1;
13    unsigned int uevent_suppress:1;
14 };
```

每一个 kobject，都对应着 /sys 目录下（其实是 sysfs 文件系统挂载在 /sys 目录下）的一个目录或者文件，目录或者文件的名称就是 kobject 结构中的 name。

我们从 kobject 结构中可以看出，它挂载在 kset 下，并且指向了 kset，那 kset 是什么呢？我们来分析分析，它是 kobject 结构的容器吗？

其实是也不是，因为 kset 结构中本身又包含一个 kobject 结构，所以它既是 kobject 的容器，同时本身还是一个 kobject。kset 结构代码如下所示。

[复制代码](#)

```
1 struct kset {
2     struct list_head list; //挂载kobject结构的链表
3     spinlock_t list_lock; //自旋锁
4     struct kobject kobj; //自身包含一个kobject结构
5     const struct kset_uevent_ops *uevent_ops; //暂时不关注
6 } __randomize_layout;
```

看到这里你应该知道了，kset 不仅仅自己是个 kobject，还能挂载多个 kobject，这说明 kset 是 kobject 的集合容器。在 Linux 内核中，至少有两个顶层 kset，代码如下所示。

[复制代码](#)

```
1 struct kset *devices_kset; //管理所有设备
2 static struct kset *bus_kset; //管理所有总线
3 static struct kset *system_kset;
4 int __init devices_init(void)
5 {
6     devices_kset = kset_create_and_add("devices", &device_uevent_ops, NULL); //
7     return 0;
8 }
9 int __init buses_init(void)
10 {
11     bus_kset = kset_create_and_add("bus", &bus_uevent_ops, NULL); //建立总线kset
12     if (!bus_kset)
13         return -ENOMEM;
14     system_kset = kset_create_and_add("system", NULL, &devices_kset->kobj); //在
15     if (!system_kset)
16         return -ENOMEM;
17     return 0;
18 }
```

我知道，你可能很难想象许多个 kset 和 kobject 在逻辑上形成的层次结构，所以我为你画了一幅图，你可以结合这张示意图理解这个结构。




kset与kobject

上图中展示了一个类似文件目录的结构，这正是 kset 与 kobject 设计的目标之一。kset 与 kobject 结构只是基础数据结构，但是仅仅只有它的话，也就只能实现这个层次结构，其它的什么也不能干，根据我们以往的经验可以猜出，kset 与 kobject 结构肯定是嵌入到更高级的数据结构之中使用，下面我们继续探索。

总线

kset、kobject 结构只是开胃菜，这个基础了解了，我们还要回到研究 Linux 设备与驱动的正题上。我们之前说过了，Linux 用总线组织设备和驱动，由此可见总线是 Linux 设备的基础，它可以表示 CPU 与设备的连接，那么总线的数据结构是什么样呢？我们一起来看看。

Linux 把总线抽象成 bus_type 结构，代码如下所示。

 复制代码

```

1 struct bus_type {
2     const char      *name; //总线名称
3     const char      *dev_name; //用于列举设备，如 ("foo%u", dev->id)
4     struct device    *dev_root; //父设备
5     const struct attribute_group **bus_groups; //总线的默认属性
6     const struct attribute_group **dev_groups; //总线上设备的默认属性
7     const struct attribute_group **drv_groups; //总线上驱动的默认属性
8     //每当有新的设备或驱动程序被添加到这个总线上时调用
9     int (*match)(struct device *dev, struct device_driver *drv);
10    //当一个设备被添加、移除或其他一些事情时被调用产生uevent来添加环境变量。
11    int (*uevent)(struct device *dev, struct kobj_uevent_env *env);
12    //当一个新的设备或驱动程序添加到这个总线时被调用，并回调特定驱动程序探查函数，以初始化匹
13    int (*probe)(struct device *dev);
14    //将设备状态同步到软件状态时调用
15    void (*sync_state)(struct device *dev);
16    //当一个设备从这个总线上删除时被调用
17    int (*remove)(struct device *dev);
18    //当系统关闭时被调用
19    void (*shutdown)(struct device *dev);
20    //调用以使设备重新上线（在下线后）
21    int (*online)(struct device *dev);
22    //调用以使设备离线，以便热移除。可能会失败。
23    int (*offline)(struct device *dev);
24    //当这个总线上的设备想进入睡眠模式时调用
25    int (*suspend)(struct device *dev, pm_message_t state);
26    //调用以使该总线上的一个设备脱离睡眠模式
27    int (*resume)(struct device *dev);
28    //调用以找出该总线上的一个设备支持多少个虚拟设备功能
29    int (*num_vf)(struct device *dev);
30    //调用以在该总线上的设备配置DMA
31    int (*dma_configure)(struct device *dev);
32    //该总线的电源管理操作，回调特定的设备驱动的pm-ops
33    const struct dev_pm_ops *pm;
34    //此总线的IOMMU具体操作，用于将IOMMU驱动程序实现到总线上
35    const struct iommu_ops *iommu_ops;
36    //驱动核心的私有数据，只有驱动核心能够接触这个
37    struct subsys_private *p;
38    struct lock_class_key lock_key;
39    //当探测或移除该总线上的一个设备时，设备驱动核心应该锁定该设备
40    bool need_parent_lock;
41 };

```

可以看出，上面代码的 bus_type 结构中，包括总线名字、总线属性，还有操作该总线下所有设备通用操作函数的指针，其各个函数的功能我在代码注释中已经写清楚了。

从这一点可以发现，**总线不仅仅是组织设备和驱动的容器，还是同类设备的共有功能的抽象层**。下面我们来看看 `subsys_private`，它是总线的驱动核心的私有数据，其中有我们想知道的秘密，代码如下所示。

[复制代码](#)

```

1 //通过kobject找到对应的subsys_private
2 #define to_subsys_private(obj) container_of(obj, struct subsys_private, subsys
3 struct subsys_private {
4     struct kset subsys; //定义这个子系统结构的kset
5     struct kset *devices_kset; //该总线的"设备"目录，包含所有的设备
6     struct list_head interfaces; //总线相关接口的列表
7     struct mutex mutex; //保护设备，和接口列表
8     struct kset *drivers_kset; //该总线的"驱动"目录，包含所有的驱动
9     struct klist klist_devices; //挂载总线上所有设备的可迭代链表
10    struct klist klist_drivers; //挂载总线上所有驱动的可迭代链表
11    struct blocking_notifier_head bus_notifier;
12    unsigned int drivers_autoprobe:1;
13    struct bus_type *bus; //指向所属总线
14    struct kset glue_dirs;
15    struct class *class; //指向这个结构所关联类结构的指针
16 };

```

看到这里，你应该明白 `kset` 的作用了，我们通过 `bus_kset` 可以找到所有的 `kset`，通过 `kset` 又能找到 `subsys_private`，再通过 `subsys_private` 就可以找到总线了，也可以找到该总线上所有的设备与驱动。

设备

虽然 Linux 抽象出了总线结构，但是 Linux 还需要表示一个设备，下面我们来探索 Linux 是如何表示一个设备的。

其实，在 Linux 系统中设备也是一个数据结构，里面包含了一个设备的所有信息。代码如下所示。

[复制代码](#)

```

1 struct device {
2     struct kobject kobj;
3     struct device *parent; //指向父设备
4     struct device_private *p; //设备的私有数据
5     const char *init_name; //设备初始化名字
6     const struct device_type *type; //设备类型
7     struct bus_type *bus; //指向设备所属总线

```

```

8     struct device_driver *driver; //指向设备的驱动
9     void          *platform_data; //设备平台数据
10    void          *driver_data; //设备驱动的私有数据
11    struct dev_links_info links; //设备供应商链接
12    struct dev_pm_info power; //用于设备的电源管理
13    struct dev_pm_domain *pm_domain; //提供在系统暂停时执行调用
14 #ifdef CONFIG_GENERIC_MSI_IRQ
15     struct list_head msi_list; //主机的MSI描述符链表
16 #endif
17     struct dev_archdata archdata;
18     struct device_node *of_node; //用访问设备树节点
19     struct fwnode_handle *fwnode; //设备固件节点
20     dev_t          devt; //用于创建sysfs "dev"
21     u32            id; //设备实例id
22     spinlock_t      devres_lock; //设备资源链表锁
23     struct list_head devres_head; //设备资源链表
24     struct class      *class; //设备的类
25     const struct attribute_group **groups; //可选的属性组
26     void      (*release)(struct device *dev); //在所有引用结束后释放设备
27     struct iommu_group *iommu_group; //该设备属于的IOMMU组
28     struct dev_iommu   *iommu; //每个设备的通用IOMMU运行时数据
29 };

```

device 结构很大，这里删除了我们不需要关心的内容。另外，我们看到 device 结构中同样包含了 kobject 结构，这使得设备可以加入 kset 和 kobject 组建的层次结构中。device 结构中有总线和驱动指针，这能帮助设备找到自己的驱动程序和总线。

驱动

有了设备结构，还需要有设备对应的驱动，Linux 是如何表示一个驱动的呢？同样也是一个数据结构，其中包含了驱动程序的相关信息。其实在 device 结构中我们就看到了，就是 device_driver 结构，代码如下。

```

1 struct device_driver {
2     const char      *name; //驱动名称
3     struct bus_type  *bus; //指向总线
4     struct module    *owner; //模块持有者
5     const char      *mod_name; //用于内置模块
6     bool suppress_bind_attrs; //禁用通过sysfs的绑定/解绑
7     enum probe_type probe_type; //要使用的探查类型（同步或异步）
8     const struct of_device_id *of_match_table; //开放固件表
9     const struct acpi_device_id *acpi_match_table; //ACPI匹配表
10    //被调用来查询一个特定设备的存在
11    int (*probe) (struct device *dev);

```

 复制代码


```
12 //将设备状态同步到软件状态时调用
13 void (*sync_state)(struct device *dev);
14 //当设备被从系统中移除时被调用，以便解除设备与该驱动的绑定
15 int (*remove) (struct device *dev);
16 //关机时调用，使设备停止
17 void (*shutdown) (struct device *dev);
18 //调用以使设备进入睡眠模式，通常是进入一个低功率状态
19 int (*suspend) (struct device *dev, pm_message_t state);
20 //调用以使设备从睡眠模式中恢复
21 int (*resume) (struct device *dev);
22 //默认属性
23 const struct attribute_group **groups;
24 //绑定设备的属性
25 const struct attribute_group **dev_groups;
26 //设备电源操作
27 const struct dev_pm_ops *pm;
28 //当sysfs目录被写入时被调用
29 void (*coredump) (struct device *dev);
30 //驱动程序私有数据
31 struct driver_private *p;
32 };
33 struct driver_private {
34     struct kobject kobj;
35     struct klist klist_devices; //驱动管理的所有设备的链表
36     struct klist_node knode_bus; //加入bus链表的节点
37     struct module_kobject *mkobj; //指向用kobject管理模块节点
38     struct device_driver *driver; //指向驱动本身
39 };
```

在 `device_driver` 结构中，包含了驱动程序的名字、驱动程序所在模块、设备探查和电源相关的回调函数的指针。在 `driver_private` 结构中同样包含了 `kobject` 结构，用于组织所有的驱动，还指向了驱动本身，你发现没有，`bus_type` 中的 `subsys_private` 结构的机制如出一辙。

文件操作函数

前面我们学习的都是 Linux 驱动程序的核心数据结构，我们很少用到，只是为了让你了解最基础的原理。

其实，在 Linux 系统中提供了更为高级的封装，Linux 将设备分成几类分别是：字符设备、块设备、网络设备以及杂项设备。具体情况你可以参考我后面梳理的图表。

设备类型	描述&举例
字符设备	以字节流形式被访问的设备，比如字符终端和串口设备
块设备	以数据块形式被访问的设备，比如硬盘、光盘等
网络设备	主机与主机之间进行数据交换的设备
杂项设备	一些不符合 Linux 预先确定的字符设备，划分为杂项设备



设备类型一览表

这些类型的设备的数据结构，都会直接或者间接包含基础的 device 结构，我们以杂项设备为例子研究一下，Linux 用 miscdevice 结构表示一个杂项设备，代码如下。

复制代码

```
1 struct miscdevice {
2     int minor;//设备号
3     const char *name;//设备名称
4     const struct file_operations *fops;//文件操作函数结构
5     struct list_head list;//链表
6     struct device *parent;//指向父设备的device结构
7     struct device *this_device;//指向本设备的device结构
8     const struct attribute_group **groups;
9     const char *nodename;//节点名字
10    umode_t mode;//访问权限
11 };
```

miscdevice 结构就是一个杂项设备，它一般在驱动程序代码文件中静态定义。我们清楚地看见有个 this_device 指针，它指向下层的、属于这个杂项设备的 device 结构。

但是这里重点是 **file_operations 结构**，设备一经注册，就会在 sys 相关的目录下建立设备对应的文件结点，对这个文件结点打开、读写等操作，最终会调用到驱动程序对应的函数，而对应的函数指针就保存在 file_operations 结构中，我们现在来看看这个结构。

复制代码

```
1 struct file_operations {
2     struct module *owner;//所在的模块
```

```

3     loff_t (*llseek) (struct file *, loff_t, int); //调整读写偏移
4     ssize_t (*read) (struct file *, char __user *, size_t, loff_t *); //读
5     ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *); //
6     int (*mmap) (struct file *, struct vm_area_struct *); //映射
7     int (*open) (struct inode *, struct file *); //打开
8     int (*flush) (struct file *, fl_owner_t id); //刷新
9     int (*release) (struct inode *, struct file *); //关闭
10 }     randomize layout

```

file_operations 结构中的函数指针有 31 个，我删除了我们不熟悉的函数指针，我们了解原理，不需要搞清楚所有函数指针的功能。

那么，Linux 如何调用到这个 file_operations 结构中的函数呢？我以打开操作为例给你讲讲，Linux 的打开系统调用接口会调用 filp_open 函数，filp_open 函数的调用路径如下所示。

[复制代码](#)

```

1 //filp_open
2 //file_open_name
3 //do_filp_open
4 //path_openat
5 static int do_o_path(struct nameidata *nd, unsigned flags, struct file *file)
6 {
7     struct path path;
8     int error = path_lookupat(nd, flags, &path); //解析文件路径得到文件inode节点
9     if (!error) {
10         audit_inode(nd->name, path.dentry, 0);
11         error = vfs_open(&path, file); //vfs层打开文件接口
12         path_put(&path);
13     }
14     return error;
15 }
16 int vfs_open(const struct path *path, struct file *file)
17 {
18     file->f_path = *path;
19     return do_dentry_open(file, d_backing_inode(path->dentry), NULL);
20 }
21 static int do_dentry_open(struct file *f, struct inode *inode, int (*open)(stru
22 {
23     //略过我们不想看的代码
24     f->f_op = fops_get(inode->i_fop); //获取文件节点的file_operations
25     if (!open) //如果open为空则调用file_operations结构中的open函数
26         open = f->f_op->open;
27     if (open) {
28         error = open(inode, f);
29     }
30     //略过我们不想看的代码

```

```
31     return 0;
32 }
```

看到这里，我们就知道了，file_operations 结构的地址存在一个文件的 inode 结构中。在 Linux 系统中，都是用 inode 结构表示一个文件，不管它是数据文件还是设备文件。

到这里，我们已经清楚了文件操作函数以及它的调用流程。


驱动程序实例

我们想要真正理解 Linux 设备驱动，最好的方案就是写一个真实的驱动程序实例。下面我们一起应用前面的基础，结合 Linux 提供的驱动程序开发接口，一起实现一个真实驱动程序。

这个驱动程序的主要工作，就是获取所有总线和其下所有设备的名字。为此我们需要先了解驱动程序的整体框架，接着建立我们总线和设备，然后实现驱动程序的打开、关闭，读写操作函数，最后我们写个应用程序，来测试我们的驱动程序。

驱动程序框架

Linux 内核的驱动程序是在一个可加载的内核模块中实现，可加载的内核模块只需要两个函数和模块信息就行，但是我们要在模块中实现总线和设备驱动，所以需要更多的函数和数据结构，它们的代码如下。

 复制代码

```
1 #define DEV_NAME  "devicesinfo"
2 #define BUS_DEV_NAME  "devicesinfobus"
3
4 static int misc_find_match(struct device *dev, void *data)
5 {
6     printk(KERN_EMERG "device name is:%s\n", dev->kobj.name);
7     return 0;
8 }
9 //对应于设备文件的读操作函数
10 static ssize_t misc_read (struct file *pfile, char __user *buff, size_t size,
11 {
12     printk(KERN_EMERG "line:%d,%s is call\n",__LINE__,__FUNCTION__);
13     return 0;
14 }
15 //对应于设备文件的写操作函数
16 static ssize_t misc_write(struct file *pfile, const char __user *buff, size_t
```

```
17 {
18     printk(KERN_EMERG "line:%d,%s is call\n",__LINE__,__FUNCTION__);
19     return 0;
20 }
21 //对应于设备文件的打开操作函数
22 static int misc_open(struct inode *pinode, struct file *pfile)
23 {
24     printk(KERN_EMERG "line:%d,%s is call\n",__LINE__,__FUNCTION__);
25     return 0;
26 }
27 //对应于设备文件的关闭操作函数
28 static int misc_release(struct inode *pinode, struct file *pfile)
29 {
30     printk(KERN_EMERG "line:%d,%s is call\n",__LINE__,__FUNCTION__);
31     return 0;
32 }
33
34 static int devicesinfo_bus_match(struct device *dev, struct device_driver *dri
35 {
36     return !strcmp(dev->kobj.name, driver->name, strlen(driver->name));
37 }
38 //对应于设备文件的操作函数结构
39 static const struct file_operations misc_fops = {
40     .read      = misc_read,
41     .write     = misc_write,
42     .release   = misc_release,
43     .open      = misc_open,
44 };
45 //misc设备的结构
46 static struct miscdevice misc_dev = {
47     .fops      = &misc_fops,           //设备文件操作方法
48     .minor     = 255,                  //次设备号
49     .name      = DEV_NAME,             //设备名/dev/下的设备节点名
50 };
51 //总线结构
52 struct bus_type devicesinfo_bus = {
53     .name      = BUS_DEV_NAME, //总线名字
54     .match     = devicesinfo_bus_match, //总线match函数指针
55 };
56 //内核模块入口函数
57 static int __init miscdrv_init(void)
58 {
59     printk(KERN_EMERG "INIT misc\n");
60     return 0;
61 }
62 //内核模块退出函数
63 static void __exit miscdrv_exit(void)
64 {
65     printk(KERN_EMERG "EXIT,misc\n");
66 }
67 module_init(miscdrv_init); //申明内核模块入口函数
68 module_exit(miscdrv_exit); //申明内核模块退出函数
```

```
69 MODULE_LICENSE("GPL");//模块许可
70 MODULE_AUTHOR("LMOS");//模块开发者
```

一个最简单的驱动程序框架的内核模块就写好了，该有的函数和数据结构都有了，那些数据结构都是静态定义的，它们的内部字段我们在前面也已经了解了。这个模块一旦加载就会执行 `miscdrv_init` 函数，卸载时就会执行 `miscdrv_exit` 函数。

建立设备

Linux 系统也提供了很多专用接口函数，用来建立总线和设备。下面我们先来建立一个总线，然后在总线下建立一个设备。

首先来说说建立一个总线，Linux 系统提供了一个 `bus_register` 函数向内核注册一个总线，相当于建立了一个总线，我们需要在 `miscdrv_init` 函数中调用它，代码如下所示。

[复制代码](#)

```
1 static int __init miscdrv_init(void)
2 {
3     printk(KERN_EMERG "INIT misc\n");
4     busok = bus_register(&devicesinfo_bus);//注册总线
5     return 0;
6 }
```

`bus_register` 函数会在系统中注册一个总线，所需参数就是总线结构的地址 (`&devicesinfo_bus`)，返回非 0 表示注册失败。现在来看看，在 `bus_register` 函数中都做了些什么事情，代码如下所示。

[复制代码](#)

```
1 int bus_register(struct bus_type *bus)
2 {
3     int retval;
4     struct subsys_private *priv;
5     //分配一个subsys_private结构
6     priv = kzalloc(sizeof(struct subsys_private), GFP_KERNEL);
7     //bus_type和subsys_private结构互相指向
8     priv->bus = bus;
9     bus->p = priv;
10    //把总线的名称加入subsys_private的kobject中
11    retval = kobject_set_name(&priv->subsys.kobj, "%s", bus->name);
12    priv->subsys.kobj.kset = bus_kset;//指向bus_kset
```



```

13 //把subsys_private中的kset注册到系统中
14 retval = kset_register(&priv->subsys);
15 //建立总线的文件结构在sysfs中
16 retval = bus_create_file(bus, &bus_attr_uevent);
17 //建立subsys_private中的devices和drivers的kset
18 priv->devices_kset = kset_create_and_add("devices", NULL,
19                                         &priv->subsys.kobj);
20 priv->drivers_kset = kset_create_and_add("drivers", NULL,
21                                         &priv->subsys.kobj);
22 //建立subsys_private中的devices和drivers链表,用于属于总线的设备和驱动
23 klist_init(&priv->klist_devices, klist_devices_get, klist_devices_put);
24 klist_init(&priv->klist_drivers, NULL, NULL);
25 return 0;
26 }

```

我删除了很多你不用关注的代码,看到这里,你应该知道总线是怎么通过 `subsys_private` 把设备和驱动关联起来的(通过 `bus_type` 和 `subsys_private` 结构互相指向),下面我们看看怎么建立设备。我们这里建立一个 `misc` 杂项设备。`misc` 杂项设备需要定一个数据结构,然后调用 `misc` 杂项设备注册接口函数,代码如下。

[复制代码](#)


```

1 #define DEV_NAME "devicesinfo"
2 static const struct file_operations misc_fops = {
3     .read      = misc_read,
4     .write     = misc_write,
5     .release   = misc_release,
6     .open     = misc_open,
7 };
8 static struct miscdevice misc_dev = {
9     .fops      = &misc_fops,           //设备文件操作方法
10    .minor     = 255,                   //次设备号
11    .name      = DEV_NAME,              //设备名/dev/下的设备节点名
12 };
13 static int __init miscdrv_init(void)
14 {
15     misc_register(&misc_dev); //注册misc杂项设备
16     printk(KERN_EMERG "INIT misc busok\n");
17     busok = bus_register(&devicesinfo_bus); //注册总线
18     return 0;
19 }

```


上面的代码中,静态定义了 `miscdevice` 结构的变量 `misc_dev`, `miscdevice` 结构我们在前面已经了解过了,最后调用 `misc_register` 函数注册了 `misc` 杂项设备。

`misc_register` 函数到底做了什么，我们一起来看看，代码如下所示。

 复制代码

```
1 int misc_register(struct miscdevice *misc)
2 {
3     dev_t dev;
4     int err = 0;
5     bool is_dynamic = (misc->minor == MISC_DYNAMIC_MINOR);
6     INIT_LIST_HEAD(&misc->list);
7     mutex_lock(&misc_mtx);
8     if (is_dynamic) { //minor次设备号如果等于255就自动分配次设备
9         int i = find_first_zero_bit(misc_minors, DYNAMIC_MINORS);
10        if (i >= DYNAMIC_MINORS) {
11            err = -EBUSY;
12            goto out;
13        }
14        misc->minor = DYNAMIC_MINORS - i - 1;
15        set_bit(i, misc_minors);
16    } else { //否则检查次设备号是否已经被占有
17        struct miscdevice *c;
18        list_for_each_entry(c, &misc_list, list) {
19            if (c->minor == misc->minor) {
20                err = -EBUSY;
21                goto out;
22            }
23        }
24    }
25    dev = MKDEV(MISC_MAJOR, misc->minor); //合并主、次设备号
26    //建立设备
27    misc->this_device =
28        device_create_with_groups(misc_class, misc->parent, dev,
29                                misc, misc->groups, "%s", misc->name);
30    //把这个misc加入到全局misc_list链表
31    list_add(&misc->list, &misc_list);
32    out:
33    mutex_unlock(&misc_mtx);
34    return err;
35 }
```

可以看出，`misc_register` 函数只是负责分配设备号，以及把 `miscdev` 加入链表，真正的核心工作由 `device_create_with_groups` 函数来完成，代码如下所示。

 复制代码

```
1 struct device *device_create_with_groups(struct class *class,
2                                         struct device *parent, dev_t devt, void *drvdata, const str
3 {
```

```

4     va_list vargs;
5     struct device *dev;
6     va_start(vargs, fmt);
7     dev = device_create_groups_vargs(class, parent, devt, drvdata, groups,fmt,
8     va_end(vargs);
9     return dev;
10 }
11 struct device *device_create_groups_vargs(struct class *class, struct device *
12 {
13     struct device *dev = NULL;
14     int retval = -ENODEV;
15     dev = kzalloc(sizeof(*dev), GFP_KERNEL); //分配设备结构的内存空间
16     device_initialize(dev); //初始化设备结构
17     dev->devt = devt; //设置设备号
18     dev->class = class; //设置设备类
19     dev->parent = parent; //设置设备的父设备
20     dev->groups = groups; //设置设备属性
21     dev->release = device_create_release;
22     dev_set_drvdata(dev, drvdata); //设置miscdev的地址到设备结构中
23     retval = kobject_set_name_vargs(&dev->kobj, fmt, args); //把名称设置到设备的ko
24     retval = device_add(dev); //把设备加入到系统中
25     if (retval)
26         goto error;
27     return dev; //返回设备
28 error:
29     put_device(dev);
30     return ERR_PTR(retval);
31 }

```

到这里，misc 设备的注册就搞清楚了，下面我们来测试一下看看结果，看看 Linux 系统是不是多了一个总线和设备。

你可以在本课程的代码目录中，执行 make 指令，就会产生一个 miscdrv.ko 内核模块文件，我们把这个模块文件加载到 Linux 系统中就行了。

为了看到效果，我们还必须要做另一件事情。在终端中用 `sudo cat /proc/kmsg` 指令读取 `/proc/kmsg` 文件，该文件是内核 `printk` 函数输出信息的文件。指令如下所示。

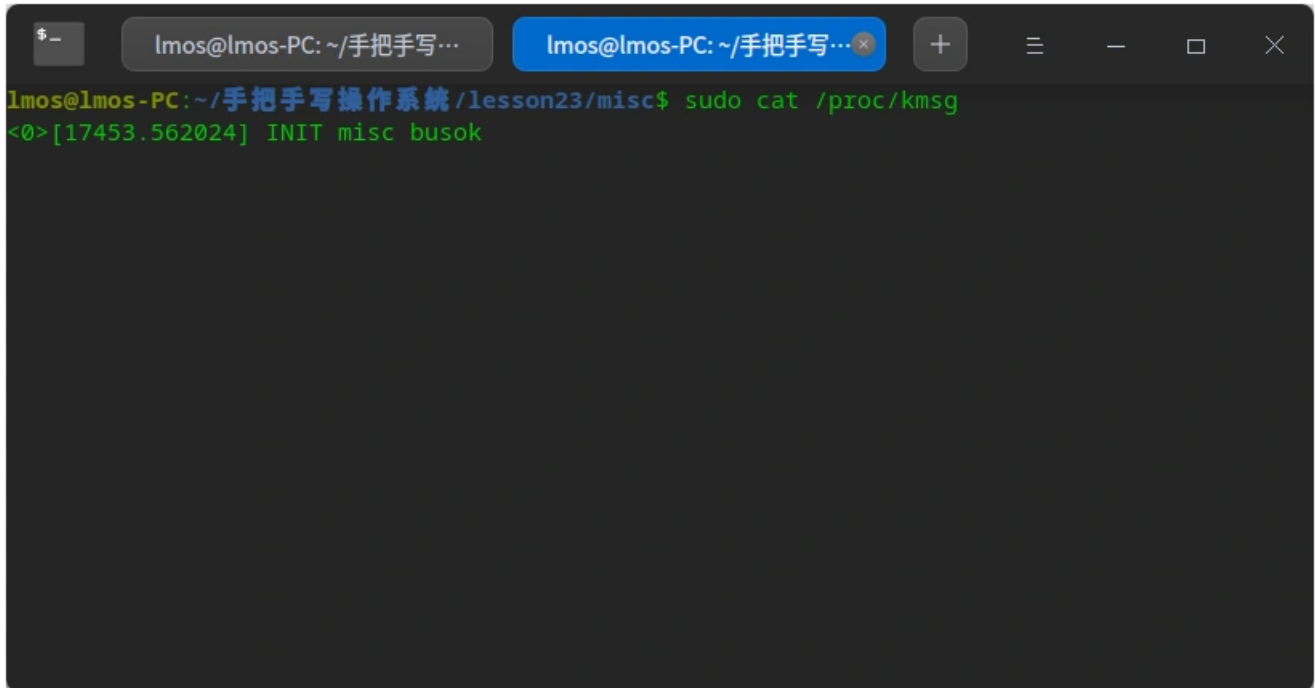
 复制代码

```

1 #第一步在终端中执行如下指令
2 sudo cat /proc/kmsg
3 #第二步在另一个终端中执行如下指令
4 make
5 sudo insmod miscdrv.ko
6 #不用这个模块了可以用以下指令卸载
7 sudo rmmod miscdrv.ko

```

insmod 指令是加载一个内核模块，一旦加载成功就会执行 miscdrv_init 函数。如果不出意外，你在终端中会看到如下图所示的情况。



```
lmos@lmos-PC: ~/手把手写... lmos@lmos-PC: ~/手把手写...  
lmos@lmos-PC:~/手把手写操作系统/lesson23/misc$ sudo cat /proc/kmsg  
<0>[17453.562024] INIT misc busok
```

驱动测试

这说明我们设备已经建立了，你应该可以在 /dev 目录看到一个 devicesinfo 文件，同时你在 /sys/bus/ 目录下也可以看到一个 devicesinfobus 文件。这就是我们建立的设备和总线的文件节点的名称。

打开、关闭、读写函数

建立了设备和总线，有了设备文件节点，应用程序就可以打开、关闭以及读写这个设备文件了。

虽然现在确实可以操作设备文件了，只不过还不能完成任何实际功能，因为我们只是写好了框架函数，所以我们下面就去写好并填充这些框架函数，代码如下所示。

复制代码

```
1 //打开  
2 static int misc_open(struct inode *pinode, struct file *pfile)  
3 {  
4     printk(KERN_EMERG "line:%d,%s is call\n",__LINE__,__FUNCTION__); //打印这个函  
5     return 0;
```

```

6 }
7 //关闭
8 static int misc_release(struct inode *pinode, struct file *pfile)
9 {
10     printk(KERN_EMERG "line:%d,%s is call\n",__LINE__,__FUNCTION__);//打印这个函数
11     return 0;
12 }
13 //写
14 static ssize_t misc_write(struct file *pfile, const char __user *buff, size_t
15 {
16     printk(KERN_EMERG "line:%d,%s is call\n",__LINE__,__FUNCTION__);//打印这个函数
17     return 0;
18 }

```

以上三个函数，仍然没干什么实际工作，就是打印该函数所在文件的行号和名称，然后返回 0 就完事了。回到前面，我们的目的是要获取 Linux 中所有总线上的所有设备，所以在读函数中来实现是合理的。

具体实现的代码如下所示。

 复制代码

```

1 #define to_subsys_private(obj) container_of(obj, struct subsys_private, subsys
2 struct kset *ret_buskset(void)
3 {
4     struct subsys_private *p;
5     if(!busok)
6         return NULL;
7     if(!devicesinfo_bus.p)
8         return NULL;
9     p = devicesinfo_bus.p;
10    if(!p->subsys.kobj.kset)
11        return NULL;
12    //返回devicesinfo_bus总线上的kset，正是bus_kset
13    return p->subsys.kobj.kset;
14 }
15 static int misc_find_match(struct device *dev, void *data)
16 {
17     struct bus_type* b = (struct bus_type*)data;
18     printk(KERN_EMERG "%s---->device name is:%s\n", b->name, dev->kobj.name);/
19     return 0;
20 }
21 static ssize_t misc_read (struct file *pfile, char __user *buff, size_t size,
22 {
23     struct kobject* kobj;
24     struct kset* kset;
25     struct subsys_private* p;
26     kset = ret_buskset();//获取bus_kset的地址

```

```
27     if(!kset)
28         return 0;
29     printk(KERN_EMERG "line:%d,%s is call\n",__LINE__,__FUNCTION__); //打印这个函数
30     //扫描所有总线的kobject
31     list_for_each_entry(kobj, &kset->list, entry)
32     {
33         p = to_subsys_private(kobj);
34         printk(KERN_EMERG "Bus name is:%s\n",p->bus->name);
35         //遍历具体总线上的所有设备
36         bus_for_each_dev(p->bus, NULL, p->bus, misc_find_match);
37     }
38     return 0;
39 }
```

正常情况下，我们是不能获取 bus_kset 地址的，它是所有总线的根，包含了所有总线的 kobject，Linux 为了保护 bus_kset，并没有在 bus_type 结构中直接包含 kobject，而是让总线指向一个 subsys_private 结构，在其中包含了 kobject 结构。

所以，我们要注册一个总线，这样就能拔出萝卜带出泥，得到 bus_kset，根据它又能找到所有 subsys_private 结构中的 kobject，接着找到 subsys_private 结构，反向查询到 bus_type 结构的地址。

然后调用 Linux 提供的 bus_for_each_dev 函数，就可以遍历一个总线上的所有设备，它每遍历到一个设备，就调用一个函数，这个函数是用参数的方式传给它的，在我们代码中就是 misc_find_match 函数。


在调用 misc_find_match 函数时，会把一个设备结构的地址和另一个指针作为参数传递进来。最后就能打印每个设备的名称了。

测试驱动

驱动程序已经写好，加载之后会自动建立设备文件，但是驱动程序不会主动工作，我们还需要写一个应用程序，对设备文件进行读写，才能测试驱动。我们这里这个驱动对打开、关闭、写操作没有什么实际的响应，但是只要一读就会打印所有设备的信息了。

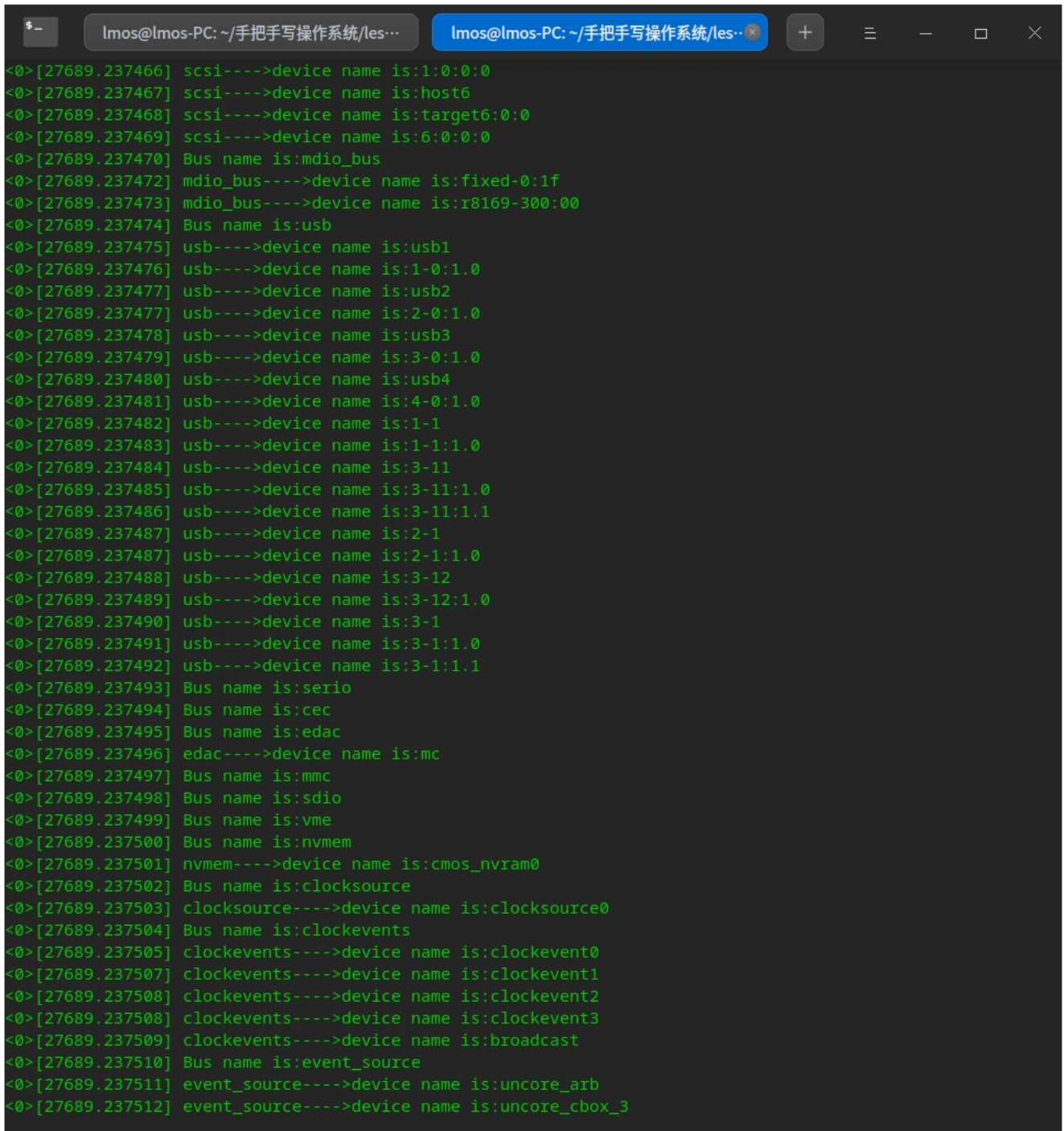
下面我们来写好这个应用，代码如下所示。

```
1 #include <stdio.h>
2 #include <stdlib.h>
```

 复制代码


```
3 #include <string.h>
4 #include <unistd.h>
5 #include <sys/types.h>
6 #include <sys/stat.h>
7 #include <fcntl.h>
8 #define DEV_NAME "/dev/devicesinfo"
9 int main(void)
10 {
11     char buf[] = {0, 0, 0, 0};
12     int fd;
13     //打开设备文件
14     fd = open(DEV_NAME, O_RDWR);
15     if (fd < 0) {
16         printf("打开 :%s 失败!\n", DEV_NAME);
17     }
18     //写数据到内核空间
19     write(fd, buf, 4);
20     //从内核空间中读取数据
21     read(fd, buf, 4);
22     //关闭设备,也可以不调用,程序关闭时系统自动调用
23     close(fd);
24     return 0;
25 }
```

你可以这样操作：切换到本课程的代码目录 make 一下，然后加载 miscdrv.ko 模块，最后在终端中执行 `sudo ./app`，就能在另一个已经执行了 `sudo cat /proc/kmsg` 的终端中，看到后面图片这样形式的数据。



```
lmos@lmos-PC: ~/手把手写操作系统/les...
lmos@lmos-PC: ~/手把手写操作系统/les...
<0>[27689.237466] scsi---->device name is:1:0:0:0
<0>[27689.237467] scsi---->device name is:host6
<0>[27689.237468] scsi---->device name is:target6:0:0
<0>[27689.237469] scsi---->device name is:6:0:0:0
<0>[27689.237470] Bus name is:mdio_bus
<0>[27689.237472] mdio_bus---->device name is:fixed-0:1f
<0>[27689.237473] mdio_bus---->device name is:r8169-300:00
<0>[27689.237474] Bus name is:usb
<0>[27689.237475] usb---->device name is:usb1
<0>[27689.237476] usb---->device name is:1-0:1.0
<0>[27689.237477] usb---->device name is:usb2
<0>[27689.237477] usb---->device name is:2-0:1.0
<0>[27689.237478] usb---->device name is:usb3
<0>[27689.237479] usb---->device name is:3-0:1.0
<0>[27689.237480] usb---->device name is:usb4
<0>[27689.237481] usb---->device name is:4-0:1.0
<0>[27689.237482] usb---->device name is:1-1
<0>[27689.237483] usb---->device name is:1-1:1.0
<0>[27689.237484] usb---->device name is:3-11
<0>[27689.237485] usb---->device name is:3-11:1.0
<0>[27689.237486] usb---->device name is:3-11:1.1
<0>[27689.237487] usb---->device name is:2-1
<0>[27689.237487] usb---->device name is:2-1:1.0
<0>[27689.237488] usb---->device name is:3-12
<0>[27689.237489] usb---->device name is:3-12:1.0
<0>[27689.237490] usb---->device name is:3-1
<0>[27689.237491] usb---->device name is:3-1:1.0
<0>[27689.237492] usb---->device name is:3-1:1.1
<0>[27689.237493] Bus name is:serio
<0>[27689.237494] Bus name is:cec
<0>[27689.237495] Bus name is:edac
<0>[27689.237496] edac---->device name is:mc
<0>[27689.237497] Bus name is:mmc
<0>[27689.237498] Bus name is:sdio
<0>[27689.237499] Bus name is:vme
<0>[27689.237500] Bus name is:nvmm
<0>[27689.237501] nvmm---->device name is:cmos_nvram0
<0>[27689.237502] Bus name is:clocksource
<0>[27689.237503] clocksource---->device name is:clocksource0
<0>[27689.237504] Bus name is:clockevents
<0>[27689.237505] clockevents---->device name is:clockevent0
<0>[27689.237507] clockevents---->device name is:clockevent1
<0>[27689.237508] clockevents---->device name is:clockevent2
<0>[27689.237508] clockevents---->device name is:clockevent3
<0>[27689.237509] clockevents---->device name is:broadcast
<0>[27689.237510] Bus name is:event_source
<0>[27689.237511] event_source---->device name is:uncore_arb
<0>[27689.237512] event_source---->device name is:uncore_cbox_3
```

获取设备名称

上图是我系统中总线名和设备名，你的计算机上可能略有差异，因为我们的计算机硬件可能不同，所以有差异是正常的，不必奇怪。

重点回顾

尽管 Linux 驱动模型异常复杂，我们还是以最小的成本，领会了 Linux 驱动模型设计的要点，还动手写了个小小的驱动程序。现在我来为你梳理一下这节课的重点。

首先，我们通过查看 `sys` 目录下的文件层次结构，直观感受了一下 Linux 系统的总线、设备、驱动是什么情况。

然后，我们了解一些重要的数据结构，它们分别是总线、驱动、设备、文件操作函数结构，还有非常关键的 **kset** 和 **kobject**，这两个结构一起组织了总线、设备、驱动，最终形成了类目录文件这样的层次结构。

最后，我们建立一个驱动程序实例，从驱动程序框架开始，我们了解如何建立一个总线和设备，编写了对应的文件操作函数，在读操作函数中实现扫描了所有总线上的所有设备，并打印总线名称和设备名称，还写了个应用程序进行了测试，检查有没有达到预期的功能。

如果你对 Linux 是怎么在总线上注册设备和驱动，又对驱动和设备怎么进行匹配感兴趣的话，也可以自己阅读 Linux 内核代码，其中有很多驱动实例，你可以研究和实验，动手和动脑相结合，我相信你一定可以搞清楚的。

思考题

为什么无论是我们加载 `miscdrv.ko` 内核模块，还是运行 app 测试，都要在前面加上 `sudo` 呢？

欢迎你在留言区记录你的学习收获，也欢迎你把这节课分享给你身边的小伙伴，一起拿下 Linux 设备驱动的内容。

我是 LMOS，我们下节课见！

分享给需要的人，Ta订阅后你可得 **20** 元现金奖励

 赞 0  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 [30 | 部门响应：设备如何处理内核I/O包？](#)

下一篇 [用户故事：成为面向“知识库”的工程师](#)

更多学习推荐

极客时间 | 训练营

大厂面试必考 100 题

2021 最新版 | 算法篇

限量免费领取

仅限前 99 名



精选留言 (2)

写留言



青玉白露

2021-07-19

在Linux系统中，sudo可以获取超级用户的权利，它之后的命令可以在内核态下进行工作。

而加载miscdrv.ko模块和app测试都需要在内核态下进行。

展开

作者回复: 是的



3



pedro

2021-07-19

加载内核模块，使用内核驱动，得 sudo 权限

展开

作者回复: 哈哈 这问题又太简单了



