粤嵌教育

Linux设备模型

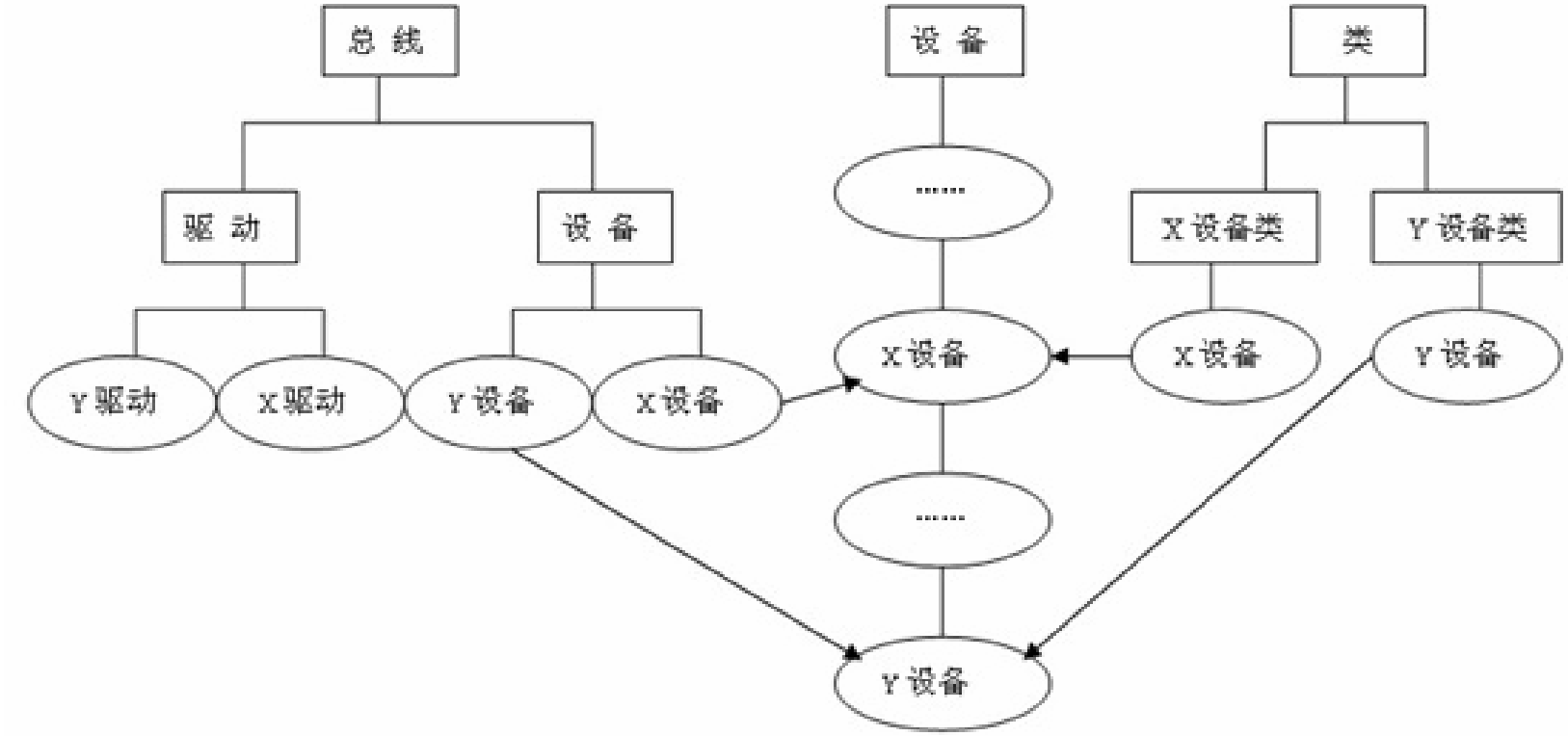
设备模型

- linux在2.6中新引进统一的设备管理模型，主要目的就是对linux的2.6系统所有的设备进行统一的管理，在以前的内核中并没有独立的数据结构让内核对整体的系统做配置和管理。尽管缺乏此类的信息，但是很多时候系统还是能正常工作，然后随着设备越来越多，系统越来越复杂，以及需要支持更多诸如电源管理等新的特征需要，新的内核版本明确提出了需要统一管理设备的要求：需要有一个对系统结构整体统一抽象的描述

- 有很多总线（如SPI、IIC、IIS等等）在Linux下已经被编写成了子系统，无需自己写驱动。

- 但需要自己研究它的子系统构架，甚至要自己添加一个新的总线类型。

设备模型视图



Linux设备模型的目的

- 内核使用设备模型支持多种不同的任务：
  - 电源管理和系统关机：
    这些需要对系统结构的理解，设备模型使OS能以正确顺序遍历系统硬件。
  - 与用户空间的通讯：
    sysfs 虚拟文件系统的实现与设备模型的紧密相关, 并向外界展示它所表述的结构。向用户空间提供系统信息、改变操作参数的接口正越来越多地通过 sysfs， 也就是设备模型来完成。

Linux设备模型的目的

粤嵌教育

- 内核使用设备模型支持多种不同的任务：
  - 热插拔设备
  - 设备类型：设备模型包括了将设备分类的机制，在一个更高的功能层上描述这些设备，并使设备对用户空间可见。
  - 对象生命周期：设备模型的实现需要创建一系列机制来处理对象的生命周期、对象间的关系和对象在用户空间的表示。

Sysfs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，并向用户模式程序提供详细的内核数据结构信息。

- **目录主要有：**
  - Block目录：包含所有的块设备
  - Devices目录：包含系统所有的设备，并根据设备挂接的总线类型组织成层次结构
  - Bus目录：包含系统中所有的总线类型
  - Drivers目录：包括内核中所有已注册的设备驱动程序
  - Class目录：系统中的设备类型（如网卡设备，声卡设备等）

Sysfs文件系统

- sys下面的目录和文件反映了系统状况。比如bus里面就包含了系统用到的一系列总线，比如pci, ide, scsi, usb等等。在usb文件夹中发现使用的U盘，USB鼠标的信息。
- sysfs是一个特殊文件系统，并没有一个实际存放文件的介质。
- sysfs的信息来源是kobject层次结构，读一个sysfs文件，就是动态的从kobject结构提取信息，生成文件。

- kobject层次结构就是linux的设备模型。
- Kobject 是Linux 2.6引入的新的设备管理机制，在内核中由struct kobject表示。

- Kobject使所有设备在底层都具有统一的接口，kobject提供基本的对象管理，是构成Linux2.6设备模型的核心结构。它与sysfs文件系统紧密关联，每个在内核中注册的kobject对象都对应于sysfs文件系统中的一个目录。

Kobject

- Kobject是组成设备模型的基本结构。
- 类似于C++中的基类， Kobject嵌入到更大的对象中用来描述设备模型的组件。如bus,devices, drivers 都是典型的容器。
- 这些对象通过kobject连接起来，形成一个树状结构，与/sys相对应。
- kobject 结构为上层数据结构和子系统提供了基本的对象管理，避免了类似功能的重复实现。这些功能包括:
  - 对象引用计数.
  - 维护对象链表.
  - 对象上锁.
  - 在用户空间的表示.

Kobject

粤嵌教育

```
struct kobject {
        const char              *name;
        struct list_head        entry;
        struct kobject          *parent;
        struct kset             *kset;
        struct kobj_type        *ktype;
        struct sysfs_dirent     *sd;
        struct kref             kref;
        unsigned int state_initialized:1;
        unsigned int state_in_sysfs:1;
        unsigned int state_add_uevent_sent:1;
        unsigned int state_remove_uevent_sent:1;
        unsigned int uevent_suppress:1;
};
```

Kobject

粤嵌教育

- kobject 所处理的任务和支持代码包括:
  - 对象的引用计数：跟踪对象生命周期的一种方法是使用引用计数。当没有内核代码持有该对象的引用时, 该对象将结束自己的有效生命期并可被删除。
  - sysfs 表述：在 sysfs 中出现的每个对象都对应一个 kobject, 它和内核交互来创建它的可见表述。
  - 数据结构关联：整体来看, 设备模型是一个极端复杂的数据结构，通过其间的大量链接而构成一个多层次的体系结构。kobject 实现了该结构并将其聚合在一起。
  - 热插拔事件处理：kobject 子系统将产生的热插拔事件通知用户空间。

Kobj_type

- Kobj_type表示该对象的类型。包含三个域：
  - 一个release方法用于释放kobject占用的资源；
  - 一个sysfs ops指针指向sysfs操作表
  - 一个sysfs文件系统缺省属性列表。

- sysfs操作表包括两个函数store()和show()。
  - 当用户态读取属性时，show()函数被调用，该函数编码指定属性值存入buffer中返回给用户态；
  - 而store()函数用于存储用户态传入的属性值。

Kobj_type的释放

- 释放方法没有在 kobject 自身里面，它被关联到包含 kobject 的结构类型中。这个类型被跟踪，用一个 struct kobj_type 结构类型，常常简单地称为一个 "ktype"。
- 这个结构看来如下：

- struct kobj_type {
- void (*release)(struct kobject *);
- struct sysfs_ops *sysfs_ops;
- struct attribute **default_attrs;
- };

- 在 struct kobj_type 中的 release 成员是一个指向这个 kobject 类型的 release 方法的指针。

- 每一个 kobject 需要有一个关联的 kobj_type 结构，如果这个 kobject 是一个 kset 的成员，kobj_type 指针由 kset 提供。

- 其间, 这个宏定义:
- struct kobj_type *get_ktype(struct kobject *kobj);

Kobj_type

粤嵌教育

- **attribute属性。以文件的形式输出到sysfs的目录当中，在kobject对应的目录下面。文件名就是name。文件读写的方法对应于kobj_type中的sysfs ops。**

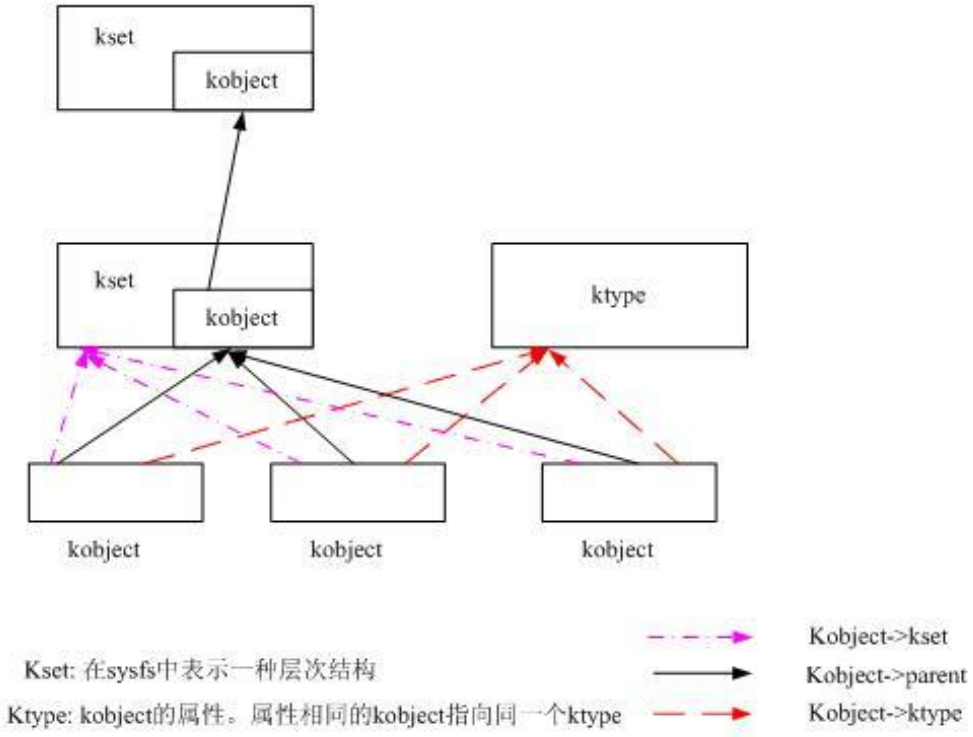
- **struct attribute {**
  char * name;
  struct module * owner;
  mode_t mode;
  };

kset

- kset 象 kobj_type 结构的扩展;
- struct kset {
- struct list_head list;
- spinlock_t list_lock;
- struct kobject kobj;
- struct kset_uevent_ops *uevent_ops;
- };
- kset 是嵌入到相同类型结构的 kobject 的集合
- struct kobj_type 关注的是对象的类型，而struct kset 关心的是对象的聚合和集合。
- 每个 kset 在内部包含自己的 kobject, 并可以用多种处理kobject 的方法处理kset。
- kset 总是在 sysfs 中出现; 一旦设置了 kset 并把它添加到系统中, 将在 sysfs 中创建一个目录；kobjects 不必在 sysfs 中表示, 但kset中的每一个 kobject 成员都在sysfs 中得到表述。

kset

- **kobject、kset、kypte**这三个结构是设备模型中的下层架构。
- 模型中的每一个元素都对应一个**kobject.**
- **kset**和**ktype**可以看成是**kobject**在层次结构与属性结构方面的扩充。将三者之间的关系用图的方示描述如下



kset

粤嵌教育

- 在**sysfs**中每一个目录都对应一个**kobject.**
- 每个**kobject**都有自己的**parent**，在没有指定**parent**的情况下，都会指向它所属的**kset->object**。其次，**kset**也内嵌了**kobject**。这个**kobject**又可以指它上一级的**parent**。就这样构成了一个空间上面的层次关系。
- 每个对象都有属性。例如，电源管理，执插拔事性管理等等。因为大部份的同类设备都有相同的属性，因此将这个属性隔离开来，存放在**ktype**中。这样就可以灵活的管理了。对于**sysfs**中的普通文件读写操作都是由**kobject->ktype->sysfs_ops**来完成的。

kset

粤嵌教育

- 一个 **kset** 的主要功能是当作顶层的**kobjects** 的容器类， 实际上, 每个 **kset** 在内部容纳它自己的 **kobject**，并且在许多情况下， 如同一个 **kobject** 相同的方式被对待。
- **kset** 一直在 **sysfs** 中出现，一旦一个 **kset** 已被建立并且加入到系统，会有一个 **sysfs** 目录给它。每个是 **kset** 成员的 **kobject** 都会出现在那里。

- 增加一个 kobject 到一个 kset：
- **int kobject_add(struct kobject *kobj);**
- **这个函数可能失败(在这个情况下它返回一个负错误码)**

- **内核提供的方便函数:**
- **extern int kobject_register(struct kobject *kobj);**
- **这个函数仅仅是一个 kobject_init 和 kobject_add 的结合.**

- 将一个kobject 从 kset 中移除：
- **void kobject_del(struct kobject *kobj);**
- **kobject_unregister 函数是 kobject_del 和 kobject_put 的结合.**

总线

粤嵌教育

- 总线是处理器和一个或多个设备之间的通道。
- 在设备模型中, 所有的设备都通过总线相连, 甚至是内部的虚拟“platform”总线。
- 总线可以相互插入。设备模型展示了总线和它们所控制的设备之间的实际连接。
- Linux 设备模型中, 总线由 bus_type 结构表示
- 每个bus_type对象都对应/sys/bus目录下的一个子目录，如PCI总线类型对应于/sys/bus/pci。
- 在每个这样的目录下都存在两个子目录：devices和drivers（分别对应于bus type结构中的devices和drivers域）。
- devices子目录描述连接在该总线上的所有设备
- Drivers子目录描述与该总线关联的所有驱动程序。

总线结构体


```
struct bus_type {
        const char              *name;  //总线名称
        struct bus_attribute    *bus_attrs;  //总线属性。
        struct device_attribute *dev_attrs; //该总线上所有设备的默认属性。
        struct driver_attribute *drv_attrs;//该总线上所有驱动的默认属性。

        int (*match)(struct device *dev, struct device_driver *drv);   //总线匹配函数
        int (*uevent)(struct device *dev, struct kobj_uevent_env *env);  //添加环境变量
        int (*probe)(struct device *dev);   //驱动探测
        int (*remove)(struct device *dev);  //驱动移除
        void (*shutdown)(struct device *dev);  //设备关机处理

        int (*suspend)(struct device *dev, pm_message_t state);  //挂起睡眠处理
        int (*suspend_late)(struct device *dev, pm_message_t state);//延迟挂起睡眠操作
        int (*resume_early)(struct device *dev);  //设备提前恢复处理
        int (*resume)(struct device *dev; //设备恢复处理

        struct dev_pm_ops *pm;  //电源管理处理方法集合

        struct bus_type_private *p;  //私有数据
};
```

总线使用

- bus 驱动有一个匹配函数, 一般通过比较驱动和设备的名子进行匹配:

```
static int xxx_match(struct device *dev, struct device_driver *driver)
{
  return !strncmp(dev->bus_id, driver->name, strlen(driver->name));
}
```

- 当涉及到真实硬件, match 函数常常在有设备自身提供的硬件 ID 和驱动提供的 ID 之间, 做一些比较
.match匹配成功则返回1，失配返回0。

- 用户空间的热插拔通知辅助函数:

```
static int xxx_uevent(struct device *dev, struct kobj_uevent_env *env)
{
        struct xxx_device *pdev = to_xxx_device(dev);

        add_uevent_var(env, "MODALIAS=%s:%s", XXX_MODULE, pdev->name);
        add_uevent_var(env, "XXX_BUS=%s:%s", XXX_MODULE, pdev->name);
        return 0;
}
```

- 在设备注册、移除，或者状态更改时，内核负责发送通知事件到用户空间。uevent在事件发送到用户空间之前调用，用来给事件添加总线特定的环境变量。

device

粤嵌教育

- 内核提供了相应的函数用于操作device对象。
  - device_register()函数将一个新的device对象插入设备模型，并自动在/sys/devices下创建一个对应的目录。
  - device_unregister()完成相反的操作，注销设备对象。
- 通常device结构不单独使用，而是包含在更大的结构中作为一个子结构使用，比如描述PCI设备的struct pci_dev

device 结构体

粤嵌教育

```
00551: struct device {
00552:     struct device        *parent;
00553:
00554:     struct device_private   *p;
00555:
00556:     struct kobject kobj;
00557:     const char      *init_name; /* initial name of the device */
00558:     const struct device_type *type;
00559:
00560:     struct mutex         mutex;  /* mutex to synchronize calls to
00561:                          * its driver.
00562:                          */
00563:
00564:     struct bus_type *bus;       /* type of bus device is on */
00565:     struct device_driver *driver;   /* which driver has allocated this
00566:                            device */
00567:     void         *platform_data; /* Platform specific data, device
00568:                            core doesn't touch it */
00569:     struct dev_pm_info  power;
00570:     struct dev_power_domain *pwr_domain;
00571:
00572: #ifdef CONFIG_NUMA
00573:     int     numa_node;  /* NUMA node this device is close to */
00574: #endif
00575:     u64     *dma_mask;  /* dma mask (if dma'able device) */
```

device 结构体


```
00576:     u64     coherent_dma_mask;/* Like dma_mask, but for
00577:                          alloc_coherent mappings as
00578:                          not all hardware supports
00579:                          64 bit addresses for consistent
00580:                          allocations such descriptors. */
00581:
00582:     struct device_dma_parameters *dma_parms;
00583:
00584:     struct list_head    dma_pools;  /* dma pools (if dma'ble) */
00585:
00586:     struct dma_coherent_mem *dma_mem; /* internal for coherent mem
00587:                          override */
00588:     /* arch specific additions */
00589:     struct dev_archdata archdata;
00590:
00591:     struct device_node  *of_node; /* associated device tree node */
00592:
00593:     dev_t           devt;   /* dev_t, creates the sysfs "dev" */
00594:
00595:     spinlock_t      devres_lock;
00596:     struct list_head    devres_head;
00597:
00598:     struct klist_node   knode_class;
00599:     struct class        *class;
00600:     const struct attribute_group **groups;  /* optional groups */
00601:
00602:     void    (*release)(struct device *dev);
00603: };
```

device接口函数


```
struct device *device_create(struct class *cls, struct device *parent,
                             dev_t devt, void *drvdata,
                             const char *fmt, ...)
                             __attribute__((format(printf, 5, 6)));

void device_destroy(struct class *cls, dev_t devt);
```

device_driver结构体

粤嵌教育

```
00185: struct device_driver {
00186:     const char      *name;
00187:     struct bus_type     *bus;
00188:
00189:     struct module       *owner;
00190:     const char      *mod_name;  /* used for built-in modules */
00191:
00192:     bool suppress_bind_attrs;   /* disables bind/unbind via sysfs */
00193:
00194:     const struct of_device_id   *of_match_table;
00195:
00196:     int (*probe) (struct device *dev);
00197:     int (*remove) (struct device *dev);
00198:     void (*shutdown) (struct device *dev);
00199:     int (*suspend) (struct device *dev, pm_message_t state);
00200:     int (*resume) (struct device *dev);
00201:     const struct attribute_group **groups;
00202:
00203:     const struct dev_pm_ops *pm;
00204:
00205:     struct driver_private *p;
00206: };
```

driver

粤嵌教育

- device_driver 结构常常被发现嵌到一个更高级的, 总线特定的结构.

```
struct ldd_driver
{
        char *version;
        struct module *module;
        struct device_driver driver;
        struct driver_attribute version_attr;
};
```

总线特定的驱动注册函数是:

```
int register_ldd_driver(struct ldd_driver *driver) {
    int ret;
     driver->driver.bus = &ldd_bus_type;
      ret = driver_register(&driver->driver);
    if (ret)
          return ret;
        driver->version_attr.attr.name = "version";
        ...
        return driver_create_file(&driver->driver, &driver->version_attr);
}
```

class

粤嵌教育

- 类是一个设备的高层视图, 抽象出了底层的实现细节，从而允许用户空间使用设备所提供的功能, 而不用关心设备是如何连接和工作的。
- 类成员通常由上层代码所控制, 而无需驱动的明确支持。
- 驱动程序核心导出了一些接口，其目的之一是提供包含设备号的属性以便自动创建设备节点，udev的使用离不开类。
- 类存在的真正目的是给作为类成员的各个设备提供一个容器，成员由 struct class_device 来表示

class

- 一个类由一个 **struct class** 的实例来定义:

- **struct class {**
- **char *name;**
- **struct class_attribute *class_attrs;**
- **struct class_device_attribute *class_dev_attrs;**
- **int (*hotplug)(struct class_device *dev, char **envp,**
- **int num_envp, char *buffer, int buffer_size);**
- **void (*release)(struct class_device *dev);**
- **void (*class_release)(struct class *class);**
- **/* Some fields omitted */**
- **};**

- 每个类需要一个唯一的名字, 它在 **/sys/class** 中出现. 当这个类被注册, 由 **class_attrs** 所指向的数组中列出的所有属性被创建. 还有一套缺省属性给每个添加到类中的设备; **class_dev_attrs** 指向它们. 有通常的热插拔函数来添加变量到环境中, 当事件产生时. 还有 **2** 个释放方法: **release** 在无论何时从类中去除一个设备时被调用, 而 **class_release** 在类自己被释放时调用.

class

粤嵌教育

- 注册函数是:

- **int class_register(struct class *cls);**
- **void class_unregister(struct class *cls);**

- **类属性描述结构体**

- **struct class_attribute {**
- **struct attribute attr;**
- **ssize_t (*show)(struct class *cls, char *buf);**
- **ssize_t (*store)(struct class *cls, const char *buf, size_t count);**
- **};**
- **CLASS_ATTR(name, mode, show, store);**

- **int class_create_file(struct class *cls, const struct class_attribute *attr);**
- **void class_remove_file(struct class *cls, const struct class_attribute *attr);**

class

- 类设备
- 一个类的真正目的是作为一个是该类成员的设备的容器. 一个成员由 **struct class_device** 来表示:

- **struct class_device {**
- **struct kobject kobj;**
- **struct class *class;**
- **struct device *dev;**
- **void *class_data;**
- **char class_id[BUS_ID_SIZE];**

- **};**

- **class_id** 成员持有设备名字, 如同它在 **sysfs** 中的一样. **class** 指针应当指向持有这个设备的类, 并且 **dev** 应当指向关联的设备结构. 设置 **dev** 是可选的; 如果它是非 **NULL**, 它用来创建一个符号连接从类入口到对应的在 **/sys/devices** 下的入口, 使得易于在用户空间找到设备入口. 类可以使用 **class_data** 来持有一个私有指针.

- 通常的注册函数已经被提供:

- **int class_device_register(struct class_device *cd);**
- **void class_device_unregister(struct class_device *cd);**

- 类设备接口也允许重命名一个已经注册的入口:
- **int class_device_rename(struct class_device *cd, char *new_name);**

class

粤嵌教育

- 内核中一个**struct class**结构体类型变量对应一个类，内核同时提供了**class_create(…)**函数，可以用它来创建一个类，这个类存放于**sysfs**下面，一旦创建好了这个类，再调用**device_create(…)**函数来在**/dev**目录下创建相应的设备节点。加载模块的时候，用户空间中的**udev**会自动响应**device_create(…)**函数，去**/sysfs**下寻找对应的类从而创建设备节点。

- 在**2.6**较早的内核版本中，**device_create(…)**函数名称不同，是**class_device_create(…)**，所以在新的内核中编译以前的模块程序有时会报错，就是因为函数名称不同，而且里面的参数设置也有一些变化。

- **struct class**和**device_create(…)** 以及**device_create(…)**都定义在**/include/linux/device.h**中，要包含这个头文件，否则编译器会报错。

class接口函数

- #include <linux/device.h>

```
/**
 * class_create - create a struct class structure
 * @owner: pointer to the module that is to "own" this struct class
 * @name: pointer to a string for the name of this class.
 *
 * This is used to create a struct class pointer that can then be used
 * in calls to device_create().
 *
 * Returns &struct class pointer on success, or ERR_PTR() on error.
 *
 * Note, the pointer created here is to be destroyed when finished by
 * making a call to class_destroy().
 */
struct class *class_create(struct module *owner, const char *name)

/**
 * class_destroy - destroys a struct class structure
 * @cls: pointer to the struct class that is to be destroyed
 *
 * Note, the pointer to be destroyed must have been created with a call
 * to class_create().
 */
void class_destroy(struct class *cls)
{
    if ((cls == NULL) || (IS_ERR(cls)))
        return;

    class_unregister(cls);
}
```

platform bus

粤嵌教育

- Linux2.6内核起，引入一套新的驱动管理和注册机制：platform_device 和 platform_driver 。Linux 中大部分的设备驱动，都可以使用这套机制，设备用 platform_device 表示；驱动用 platform_driver 进行注册。

- platform是一个虚拟的地址总线，相比pci，usb，它主要用于描述SOC上的片上资源，比如s5pv210上集成的控制器（lcd，watchdog，rtc等），platform所描述的资源有一个共同点，就是在cpu的总线上直接取址。

```
00029: struct device platform_bus = {
00030:         .init_name   = "platform",
00031: };
00032: EXPORT_SYMBOL_GPL(platform_bus);
```

platform_device

- platform_device会分到一个名称(用在驱动绑定中)以及一系列诸如地址和中断请求号(IRQ)之类的资源.

- 平台设备定义：

```
136 static struct platform_device gec210_led_dev = {
137     .name    = "gec210_led",
138     .num_resources   = ARRAY_SIZE(gec210_led_resources),
139     .resource = gec210_led_resources,
140     .id= -1,
141     .dev   = {
142       .release= gec210_plat_led_release,
143     },
144 };
```

platform_device结构体

粤嵌教育

```
00019: struct platform_device {
00020:     const char   * name;
00021:     int      id;
00022:     struct device    dev;
00023:     u32      num_resources;
00024:     struct resource * resource;
00025:
00026:     const struct platform_device_id *id_entry;
00027:
00028:     /* MFD cell pointer */
00029:     struct mfd_cell *mfd_cell;
00030:
00031:     /* arch specific additions */
00032:     struct pdev_archdata     archdata;
00033: };
```

resource结构体

粤嵌教育

```
43 static struct resource gec210_led_resources[] = {
44     [0] = {
45         .start  = 0xE0200280,
46         .end    = 0xE0200280 + 11,
47         .name   = "GHJ2",
48         .flags  = IORESOURCE_MEM,
49     },
50 };
```

```
00018: struct resource {
00019:         resource_size_t start;
00020:         resource_size_t end;
00021:         const char *name;
00022:         unsigned long flags;
00023:         struct resource *parent, *sibling, *child;
00024: };
```

```
00038: #define IORESOURCE_IO       0x00000100
00039: #define IORESOURCE_MEM      0x00000200
00040: #define IORESOURCE_IRQ      0x00000400
00041: #define IORESOURCE_DMA      0x00000800
00042: #define IORESOURCE_BUS      0x00001000
```

platform获取资源


```
00034: /**
00035:  * platform_get_resource - get a resource for a device
00036:  * @dev: platform device
00037:  * @type: resource type
00038:  * @num: resource index
00039:  */
00040: struct resource *platform_get_resource(struct platform_device *dev,
00041:                                        unsigned int type, unsigned int num)
00042: {
00043:     int i;
00044:
00045:     for (i = 0; i < dev->num_resources; i++) {
00046:         struct resource *r = &dev->resource[i];
00047:
00048:         if (type == resource_type(r) && num-- == 0)
00049:             return r;
00050:     }
00051:     return NULL;
00052: }
00053: EXPORT_SYMBOL_GPL(platform_get_resource);
```

platform_device添加与删除


```
00330: /**
00331:  * platform_device_register - add a platform-level device
00332:  * @pdev: platform device we're adding
00333:  */
00334: int platform_device_register(struct platform_device *pdev)
00335: {
00336:     device_initialize(&pdev->dev);
00337:     return platform_device_add(pdev);
00338: }
00339: EXPORT_SYMBOL_GPL(platform_device_register);
```

```
00341: /**
00342:  * platform_device_unregister - unregister a platform-level device
00343:  * @pdev: platform device we're unregistering
00344:  *
00345:  * Unregistration is done in 2 steps. First we release all resources
00346:  * and remove it from the subsystem, then we drop reference count by
00347:  * calling platform_device_put().
00348:  */
00349: void platform_device_unregister(struct platform_device *pdev)
00350: {
00351:     platform_device_del(pdev);
00352:     platform_device_put(pdev);
00353: }
00354: EXPORT_SYMBOL_GPL(platform_device_unregister);
```

platform_driver结构体

粤嵌教育

```
struct platform_driver {
    int (*probe)(struct platform_device *);
    int (*remove)(struct platform_device *);
    void (*shutdown)(struct platform_device *);
    int (*suspend)(struct platform_device *, pm_message_t state);
    int (*resume)(struct platform_device *);
    struct device_driver driver;
    const struct platform_device_id *id_table;
};
```

```
225 static struct platform_driver gec210_led_drv = {
226     .probe   = gec210_led_probe,
227     .remove  = __devexit_p(gec210_led_remove),
228     .driver  = {
229         .name ="gec210_led",
230         .owner = THIS_MODULE,
231     },
232 };
```

platform平台

粤嵌教育

- 系统为platform总线定义一个bus_type的实例platform_bus_type，通过其成员函数match()，确定device和driver如何匹配。
- 匹配platform_device和platform_driver主要看二者的name字段是否相同。（name必须要相同才能匹配）
- 用platform_device_register(platform_device*)函数注册单个的平台设备。
- 一般是在平台的BSP文件中定义platform_device，通过platform_add_devices(platform_device*)函数将平台设备注册到系统中
- platform_driver 的注册与注销：
- platform_driver_register(platform_driver*)
- platform_driver_unregister(platform_driver*)

platform驱动设计实验

- 第一步：添加头文件
- #include <linux/device.h>
- #include <linux/platform_device.h>

- 第二步：
- 将原有的init模块入口函数修改为platform平台的驱动探测函数

- 第三步：
- 将原有的exit模块退出函数修改为platform平台的驱动移除函数

platform驱动设计实验


```
static int   __devinit my_probe(struct platform_device *)   //设备探测接口
{

        int result;
      /*分配设备编号*/
       printk(" go to my_dev probe \n");

        if(TestMajor)
        {
                dev=MKDEV(TestMajor,TestMinor);//创建设备编号
                result=register_chrdev_region(dev,1,DEVICE_NAME);
        } else {
                result=alloc_chrdev_region(&dev,TestMinor,1,DEVICE_NAME);
                TestMajor=MAJOR(dev);
        }

        if(result<0)
        {
          printk(KERN_WARNING"LED: cannot get major %d \n",TestMajor);
          return result;
        }
```

platform驱动设计实验


```
/* 注册字符设备 */
        test_cdev=cdev_alloc();
        cdev_init(test_cdev,&chardev_fops);
        test_cdev->owner=THIS_MODULE;
        result=cdev_add(test_cdev,dev,1);
        if(result)
                printk("<1>Error %d while register led device!\n",result);

    /* create your own class under /sysfs */
     my_class = class_create(THIS_MODULE, "my_class");
    /* register your own device in sysfs, and this will cause udev to create corresponding
device node */
     device_create( my_class, NULL, dev,NULL,DEVICE_NAME);
     return 0;
}

static int  __devexit  my_remove(struct platform_device *)   //设备移除接口
{
        unregister_chrdev_region(MKDEV(TestMajor,TestMinor),1);
     device_destroy(my_class, MKDEV(TestMajor, 0));        //delete device node under /dev
     class_destroy(my_class);                              //delete class created by us
        cdev_del(test_cdev);
     printk(" goodbye  my_dev probe \n");
}
```

platform驱动设计实验

粤嵌教育

- 第四步：定义platform平台的设备跟驱动

```
struct platform_device my_dev = {
        .name= "my_dev",
        .id= -1,
};
EXPORT_SYMBOL(my_dev);  //声明一个内核的全局符号my_dev；

static struct platform_driver my_driver = {
   .probe  = my_probe,              //驱动探测
   .remove  = __devexit_p(my_remove),  //驱动移除

   .driver  = {
        .name ="my_dev",
        .owner = THIS_MODULE,
    },
};
EXPORT_SYMBOL(my_driver); //声明一个内核的全局符号my_driver；
```

platform驱动设计实验

- 第五步：重新实现该驱动模块的入口及退出接口

```
static int __init my_init(void)
{
    platform_device_register(&my_dev);
     printk(" go to my_dev probe init \n");
     return platform_driver_register(&my_driver );    //平台设备注册
}

static void __exit my_cleanup(void)
{
     platform_device_unregister(&my_dev);
     printk(" goodbye my_dev probe  cleanup \n");
     platform_driver_unregister(&my_driver );     //平台设备注销
}
```


谢谢!
THANKS