Tekkaman Ninja

tekkamanninja.blog.chinaunix.net

tekkamanninj

博客访问： 75932
博文数量： 263
博客积分： 15936
博客等级： 上将
技术积分： 13951
用 户 组： 普通用户
注册时间： 2007-03-27 11:22

加关注 短消息
论坛 加好友

个人简介

Fedora-ARM

文章分类

文章存档

Linux设备驱动程序学习（13）-Linux设备模型（总线、设备、驱动程序和类）

2007-12-27 11:04:00

分类： LINUX

Linux设备驱动程序学习（13）
-Linux设备模型（总线、设备、驱动程序和类）

文章的例子和实验使用《LDD3》所配的lddbus模块（稍作修改）。

提示：在学习这部分内容是一定要分析所有介绍的源代码，知道他们与上一部分内容（kobject、kset、attribute等等）的关系，最好要分析一个实际的**"flatform device"**设备，不然会只学到表象，到后面会不知所云的。

总线

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。在 Linux 设备模型中，总线由 bus_type 结构表示，定义在 <linux/device.h> ：

```
struct bus_type {
    const char          * name;/*总线类型名称*/
    struct module         * owner;/*指向模块的指针(如果有)，此模块负责操作这个总线*/

    struct kset         subsys;/*与该总线相关的子系统*/
    struct kset         drivers;/*总线驱动程序的kset*/
    struct kset         devices;/* 挂在该总线的所有设备的kset*/

    struct klist         klist_devices;/*与该总线相关的驱动程序链表*/
    struct klist         klist_drivers;/*挂接在该总线的设备链表*/

    struct blocking_notifier_head bus_notifier;

    struct bus_attribute    * bus_attrs; /*总线属性*/
    struct device_attribute * dev_attrs; /*设备属性,指向为每个加入总线的设备建立的默认属性链表*/
    struct driver_attribute * drv_attrs; /*驱动程序属性*/
    struct bus_attribute drivers_autoprobe_attr;/*驱动自动探测属性*/
    struct bus_attribute drivers_probe_attr;/*驱动探测属性*/

    int         (*match)(struct device * dev, struct device_driver * drv);
    int         (*uevent)(struct device *dev, char **envp,
                  int num_envp, char *buffer, int buffer_size);
    int         (*probe)(struct device * dev);
    int         (*remove)(struct device * dev);
    void         (*shutdown)(struct device * dev);

    int (*suspend)(struct device * dev, pm_message_t state);
    int (*suspend_late)(struct device * dev, pm_message_t state);
    int (*resume_early)(struct device * dev);
    nt (*resume)(struct device * dev);
/*处理热插拔、电源管理、探测和移除等事件的方法*/
```

```
    unsigned int drivers_autoprobe:1;
};
```

在更新的内核里，这个结构体变得更简洁了，隐藏了无需驱动编程人员知道的一些成员：

```
/*in Linux 2.6.26.5*/
struct bus_type {
    const char        *name;
    struct bus_attribute    *bus_attrs;
    struct device_attribute    *dev_attrs;
    struct driver_attribute    *drv_attrs;

    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 (*suspend)(struct device *dev, pm_message_t state);
    int (*suspend_late)(struct device *dev, pm_message_t state);
    int (*resume_early)(struct device *dev);
    int (*resume)(struct device *dev);

    struct bus_type_private *p;
};

struct bus_type_private {
    struct kset subsys;
    struct kset *drivers_kset;
    struct kset *devices_kset;
    struct klist klist_devices;
    struct klist klist_drivers;
    struct blocking_notifier_head bus_notifier;
    unsigned int drivers_autoprobe:1;
    struct bus_type *bus;
};
```

总线的注册和删除

总线的主要注册步骤：

（1）申明和初始化 bus_type 结构体。只有很少的 bus_type 成员需要初始化，大部分都由设备模型核心控制。但必须为总线指定名字及一些必要的方法。例如：

```
struct bus_type ldd_bus_type = {
    .name = "ldd",
    .match = ldd_match,
    .uevent = ldd_uevent,
};
```

（2）调用bus_register函数注册总线。

```
int bus_register(struct bus_type * bus)
```

调用可能失败，所以必须始终检查返回值。若成功，新的总线子系统将被添加进系统，并可在 sysfs 的 /sys/bus 下看到。之后可以向总线添加设备。

例如：

```
ret = bus_register(&ldd_bus_type);
if (ret)
 return ret;
```

当必须从系统中删除一个总线时，调用：

```
void bus_unregister(struct bus_type *bus);
```

总线方法

在 bus_type 结构中定义了许多方法，它们允许总线核心作为设备核心和单独的驱动程序之间提供服务的中介，主要介绍以下两个方法：

```
int (*match)(struct device * dev, struct device_driver * drv);
/*当一个新设备或者驱动被添加到这个总线时，这个方法会被调用一次或多次，若指定的驱动程序能够处理指定的设备，则返回非零值。必须在总线层使用这个函数，因为那里存在正确的逻辑，核心内核不知道如何为每个总线类型匹配设备和驱动程序*/

int (*uevent)(struct device *dev, char **envp, int num_envp, char *buffer, int buffer_size);
/*在为用户空间产生热插拔事件之前，这个方法允许总线添加环境变量（参数和 kset 的uevent方法相同）*/
```

lddbus的match和uevent方法：

```
static int ldd_match(struct device *dev, struct device_driver *driver)
{
 return !strncmp(dev->bus_id, driver->name, strlen(driver->name));
}/*仅简单比较驱动和设备的名字*/
/*当涉及实际硬件时，match 函数常常对设备提供的硬件 ID 和驱动所支持的 ID 做比较*/

static int ldd_uevent(struct device *dev, char **envp, int num_envp, char *buffer, int buffer_size)
{
 envp[0] = buffer;
 if (snprintf(buffer, buffer_size, "LDDBUS_VERSION=%s",
 Version) >= buffer_size)
 return -ENOMEM;
 envp[1] = NULL;
 return 0;
}/*在环境变量中加入 lddbus 源码的当前版本号*/
```

对设备和驱动的迭代

若要编写总线层代码，可能不得不对所有已经注册到总线的设备或驱动进行一些操作，这可能需要仔细研究嵌入到 bus_type 结构中的其他数据结构，但最好使用内核提供的辅助函数：

```
int bus_for_each_dev(struct bus_type *bus, struct device *start, void *data, int (*fn)(struct device *, void *));
int bus_for_each_drv(struct bus_type *bus, struct device_driver *start, void *data,
int (*fn)(struct device_driver *, void *));

/*这两个函数迭代总线上的每个设备或驱动程序，将关联的 device 或 device_driver 传递给 fn，同时传递 data 值。若 start 为 NULL，则从第一个设备开始；否则从 start 之后的第一个设备开始。若 fn 返回非零值，迭代停止并且那个值从 bus_for_each_dev 或bus_for_each_drv 返回。*/
```

总线属性

几乎 Linux 设备模型中的每一层都提供添加属性的函数，总线层也不例外。bus_attribute 类型定义在 <linux/device.h> 如下：

```
struct bus_attribute {
    struct attribute    attr;
    ssize_t (*show)(struct bus_type *, char * buf);
    ssize_t (*store)(struct bus_type *, const char * buf, size_t count);
};
```

可以看出struct bus_attribute 和struct attribute 很相似，其实大部分在 kobject 级上的设备模型层都是以这种方式工作。

内核提供了一个宏在编译时创建和初始化 bus_attribute 结构：

```
BUS_ATTR(_name,_mode,_show,_store)/*这个宏声明一个结构，将 bus_attr_ 作为给定 _name 的前缀来创建总线的真正名称*/

/*总线的属性必须显式调用 bus_create_file 来创建:*/
```

```
int bus_create_file(struct bus_type *bus, struct bus_attribute *attr);

/*删除总线的属性调用:*/
void bus_remove_file(struct bus_type *bus, struct bus_attribute *attr);
```

例如创建一个包含源码版本号简单属性文件方法如下:

```
static ssize_t show_bus_version(struct bus_type *bus, char *buf)
{
 return snprintf(buf, PAGE_SIZE, "%s\n", Version);
}

static BUS_ATTR(version, S_IRUGO, show_bus_version, NULL);

/*在模块加载时创建属性文件:*/
if (bus_create_file(&ldd_bus_type, &bus_attr_version))
 printk(KERN_NOTICE "Unable to create version attribute\n");

/*这个调用创建一个包含 lddbus 代码的版本号的属性文件(/sys/bus/ldd/version)*/
```

设备

在最底层，Linux 系统中的每个设备由一个 struct device 代表:

```
struct device {
    struct klist        klist_children;
    struct klist_node    knode_parent;   /* node in sibling list */
    struct klist_node    knode_driver;
    struct klist_node    knode_bus;
    struct device        *parent;/* 设备的 "父" 设备，该设备所属的设备，通常一个父设备是某种总线或者主控制器. 如果 parent 是 NULL，则该设备是顶层设备，较少见 */

    struct kobject kobj;/*代表该设备并将其连接到结构体系中的 kobject； 注意：作为通用的规则，device->kobj->parent 应等于 device->parent->kobj*/
    char    bus_id[BUS_ID_SIZE];/*在总线上唯一标识该设备的字符串;例如: PCI 设备使用标准的 PCI ID 格式，包含:域，总线，设备，和功能号.*/
    struct device_type    *type;
    unsigned        is_registered:1;
    unsigned        uevent_suppress:1;
    struct device_attribute uevent_attr;
    struct device_attribute *devt_attr;

    struct semaphore    sem;  /* semaphore to synchronize calls to its driver. */
    struct bus_type    * bus;     /*标识该设备连接在何种类型的总线上*/
    struct device_driver *driver;    /*管理该设备的驱动程序*/
    void        *driver_data;    /*该设备驱动使用的私有数据成员*/
    void        *platform_data;    /* Platform specific data, device core doesn't touch it */
    struct dev_pm_info    power;

#ifdef CONFIG_NUMA
    int        numa_node;   /* NUMA node this device is close to */
#endif
    u64        *dma_mask;    /* dma mask (if dma'able device) */
    u64        coherent_dma_mask;/* Like dma_mask, but for
                     alloc_coherent mappings as
                     not all hardware supports
                     64 bit addresses for consistent
                     allocations such descriptors. */

    struct list_head    dma_pools;    /* dma pools (if dma'ble) */

    struct dma_coherent_mem    *dma_mem; /* internal for coherent mem override */
```

```
    /* arch specific additions */
    struct dev_archdata    archdata;

    spinlock_t        devres_lock;
    struct list_head    devres_head;

    /* class_device migration path */
    struct list_head    node;
    struct class        *class;
    dev_t          devt;       /* dev_t, creates the sysfs "dev" */
    struct attribute_group    **groups;    /* optional groups */

    void    (*release)(struct device * dev);/*当这个设备的最后引用被删除时，内核调用该方法；它从被嵌入的 kobject 的 release 方法中调用。所有注册到核心的设备结构必须有一个 release 方法，否则内核将打印错误信息*/
};
/*在注册 struct device 前，最少要设置parent，bus_id，bus，和 release 成员*/
```

设备注册

设备的注册和注销函数为:

```
int device_register(struct device *dev);
void device_unregister(struct device *dev);
```

一个实际的总线也是一个设备，所以必须单独注册，以下为 lddbus 在编译时注册它的虚拟总线设备源码:

```
static void ldd_bus_release(struct device *dev)
{
 printk(KERN_DEBUG "lddbus release\n");
}

struct device ldd_bus = {
 .bus_id = "ldd0",
 .release = ldd_bus_release

}; /*这是顶层总线，parent 和 bus 成员为 NULL*/

/*作为第一个(并且唯一)总线，它的名字为 ldd0，这个总线设备的注册代码如下:*/
ret = device_register(&ldd_bus);
if (ret)
 printk(KERN_NOTICE "Unable to register ldd0\n");
/*一旦调用完成，新总线会在 sysfs 中 /sys/devices 下显示，任何挂到这个总线的设备会在 /sys/devices/ldd0 下显示*/
```

设备属性

sysfs 中的设备入口可有属性,相关的结构是:

```
/* interface for exporting device attributes 这个结构体和《LDD3》中的不同，已经被更新过了，请特别注意！*/
struct device_attribute {
    struct attribute attr;
    ssize_t (*show)(struct device *dev, struct device_attribute *attr,char *buf);
    ssize_t (*store)(struct device *dev, struct device_attribute *attr, const char *buf, size_t count);
};

/*设备属性结构可在编译时建立，使用以下宏:*/
DEVICE_ATTR(_name,_mode,_show,_store);
/*这个宏声明一个结构，将 dev_attr_ 作为给定 _name 的前缀来命名设备属性

/*属性文件的实际处理使用以下函数:*/
```

```
 int device_create_file(struct device *device,     struct device_attribute * entry);
 void device_remove_file(struct device * dev, struct device_attribute * attr);
```

设备结构的嵌入

device 结构包含设备模型核心用来模拟系统的信息。但大部分子系统记录了关于它们又拥有的设备的额外信息，所以很少单纯用 device 结构代表设备，而是，通常将其嵌入一个设备的高层表示中。底层驱动几乎不知道 struct device。

lddbus 驱动创建了它自己的 device 类型，并期望每个设备驱动使用这个类型来注册它们的设备:

```
struct ldd_device {
 char *name;
 struct ldd_driver *driver;
 struct device dev;
};
#define to_ldd_device(dev) container_of(dev, struct ldd_device, dev);
```

lddbus 导出的注册和注销接口如下:

```
/*
 * LDD devices.
 */

/*
 * For now, no references to LDDbus devices go out which are not
 * tracked via the module reference count, so we use a no-op
 * release function.
 */
static void ldd_dev_release(struct device *dev)
{ }

int register_ldd_device(struct ldd_device *lddev)
{
    lddev->dev.bus = &ldd_bus_type;
    lddev->dev.parent = &ldd_bus;
    lddev->dev.release = ldd_dev_release;
    strncpy(lddev->dev.bus_id, lddev->name, BUS_ID_SIZE);
    return device_register(&lddev->dev);
}
EXPORT_SYMBOL(register_ldd_device);

void unregister_ldd_device(struct ldd_device *lddev)
{
    device_unregister(&lddev->dev);
}
EXPORT_SYMBOL(unregister_ldd_device);
```

sculld 驱动添加一个自己的属性到它的设备入口，称为 dev，仅包含关联的设备号，源码如下：

```
static ssize_t sculld_show_dev(struct device *ddev,struct device_attribute *attr, char
*buf)
{
 struct sculld_dev *dev = ddev->driver_data;
 return print_dev_t(buf, dev->cdev.dev);
}

static DEVICE_ATTR(dev, S_IRUGO, sculld_show_dev, NULL);

/*接着，在初始化时间，设备被注册，并且 dev 属性通过下面的函数被创建:*/
static void sculld_register_dev(struct sculld_dev *dev, int index)
{
 sprintf(dev->devname, "sculld%d", index);
```

```
 dev->ldev.name = dev->devname;
 dev->ldev.driver = &sculld_driver;
 dev->ldev.dev.driver_data = dev;
 register_ldd_device(&dev->ldev);
 if (device_create_file(&dev->ldev.dev, &dev_attr_dev))
    printk( "Unable to create dev attribute ! \n");
} /*注意：程序使用 driver_data 成员来存储指向我们自己的内部的设备结构的指针。请检查device_create_file的返回值，否则编译时会有警告。*/
```

设备驱动程序

设备模型跟踪所有系统已知的驱动，主要目的是使驱动程序核心能协调驱动和新设备之间的关系。一旦驱动在系统中是已知的对象就可能完成大量的工作。驱动程序的结构体 device_driver 定义如下：

```
/*定义在<linux/device.h>*/
struct device_driver {
    const char        * name;/*驱动程序的名字( 在 sysfs 中出现 )*/
    struct bus_type         * bus;/*驱动程序所操作的总线类型*/

    struct kobject         kobj;/*内嵌的kobject对象*/
    struct klist         klist_devices;/*当前驱动程序能操作的设备链表*/
    struct klist_node     knode_bus;

    struct module         * owner;
    const char         * mod_name;     /* used for built-in modules */
    struct module_kobject     * mkobj;

    int     (*probe)     (struct device * dev);/*查询一个特定设备是否存在及驱动是否可以使用它的函数*/
    int     (*remove)     (struct device * dev);/*将设备从系统中删除*/
    void     (*shutdown)     (struct device * dev);/*关闭设备*/
    int     (*suspend)     (struct device * dev, pm_message_t state);
    int     (*resume)     (struct device * dev);
};

/*注册device_driver 结构的函数是:*/
int driver_register(struct device_driver *drv);
void driver_unregister(struct device_driver *drv);

/*driver的属性结构在:*/
struct driver_attribute {
 struct attribute attr;
 ssize_t (*show)(struct device_driver *drv, char *buf);
 ssize_t (*store)(struct device_driver *drv, const char *buf, size_t count);
};
DRIVER_ATTR(_name,_mode,_show,_store)

/*属性文件创建的方法:*/
int driver_create_file(struct device_driver * drv, struct driver_attribute * attr);
void driver_remove_file(struct device_driver * drv, struct driver_attribute * attr);

/*bus_type 结构含有一个成员( drv_attrs ) 指向一组为属于该总线的所有设备创建的默认属性*/
```

在更新的内核里，这个结构体变得更简洁了，隐藏了无需驱动编程人员知道的一些成员：

```
/*in Linux 2.6.26.5*/
struct device_driver {
    const char        *name;
    struct bus_type         *bus;

    struct module         *owner;
```

```
    const char          *mod_name;    /* used for built-in modules */

    int (*probe) (struct device *dev);
    int (*remove) (struct device *dev);
    void (*shutdown) (struct device *dev);
    int (*suspend) (struct device *dev, pm_message_t state);
    int (*resume) (struct device *dev);
    struct attribute_group **groups;

    struct driver_private *p;
};


struct driver_private {
    struct kobject kobj;
    struct klist klist_devices;
    struct klist_node knode_bus;
    struct module_kobject *mkobj;
    struct device_driver *driver;
};
#define to_driver(obj) container_of(obj, struct driver_private, kobj)
```

驱动程序结构的嵌入

对大多数驱动程序核心结构，device_driver 结构通常被嵌入到一个更高层的、总线相关的结构中。以lddbus 子系统为例，它定义了ldd_driver 结构：

```
struct ldd_driver {
 char *version;
 struct module *module;
 struct device_driver driver;
 struct driver_attribute version_attr;
};
#define to_ldd_driver(drv) container_of(drv, struct ldd_driver, driver);
```

lddbus总线中相关的驱动注册和注销函数是：

```
/*
 * Crude driver interface.
 */
static ssize_t show_version(struct device_driver *driver, char *buf)
{
    struct ldd_driver *ldriver = to_ldd_driver(driver);
    sprintf(buf, "%s\n", ldriver->version);
    return strlen(buf);
}

int register_ldd_driver(struct ldd_driver *driver)
{
 int ret;
 driver->driver.bus = &ldd_bus_type;
 ret = driver_register(&driver->driver);/*注册底层的 device_driver 结构到核心*/
 if (ret)
 return ret;
 driver->version_attr.attr.name = "version";/* driver_attribute 结构必须手工填充*/
 driver->version_attr.attr.owner = driver->module;/*注意:设定 version 属性的拥有者为驱动模块，不是 lddbus 模块！因为 show_version 函数是使用驱动模块所创建的 ldd_driver 结构，若 ldd_driver 结构在一个用户空间进程试图读取版本号时已经注销，就会出错*/
 driver->version_attr.attr.mode = S_IRUGO;
 driver->version_attr.show = show_version;
 driver->version_attr.store = NULL;
 return driver_create_file(&driver->driver, &driver->version_attr);/*建立版本属性，因为这个属性在运行时被创建，所以不能使用 DRIVER_ATTR 宏*/
}

void unregister_ldd_driver(struct ldd_driver *driver)
```

```
{
    driver_unregister(&driver->driver);
}
EXPORT_SYMBOL(register_ldd_driver);
EXPORT_SYMBOL(unregister_ldd_driver);
```

在sculld 中创建的 ldd_driver 结构如下:

```
/* Device model stuff */
static struct ldd_driver sculld_driver = {
    .version = "$Revision: 1.21 $",
    .module = THIS_MODULE,
    .driver = {
        .name = "sculld",
    },
};/*只要一个简单的 register_ldd_driver 调用就可添加它到系统中。一旦完成初始化，驱动信息可在 sysfs 中显示*/
```

类 子系统

类是一个设备的高层视图，它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。但有些情况下驱动也需要直接处理类。

几乎所有的类都显示在 /sys/class 目录中。出于历史的原因，有一个例外：块设备显示在 /sys/block 目录中。在许多情况，类子系统是向用户空间导出信息的最好方法。当类子系统创建一个类时，它将完全拥有这个类,根本不用担心哪个模块拥有那些属性，而且信息的表示也比较友好。

为了管理类，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，所以udev的使用离不开类。 类函数和结构与设备模型的其他部分遵循相同的模式，所以真正崭新的概念是很少的。

注意：class_simple 是老接口，在2.6.13中已被删除，这里不再研究。

管理类的接口

类由 struct class 的结构体来定义:

```
/*
 * device classes
 */
struct class {
    const char        * name;/*每个类需要一个唯一的名字，它将显示在 /sys/class 中*/
    struct module        * owner;

    struct kset        subsys;
    struct list_head    children;
    struct list_head    devices;
    struct list_head    interfaces;
    struct kset        class_dirs;
    struct semaphore    sem;    /* locks both the children and interfaces lists */

    struct class_attribute        * class_attrs;/* 指向类属性的指针（以NULL结尾） */
    struct class_device_attribute    * class_dev_attrs;/* 指向类中每个设备的一组默认属性的指针 */
    struct device_attribute        * dev_attrs;

    int    (*uevent)(struct class_device *dev, char **envp,
             int num_envp, char *buffer, int buffer_size);/* 类热插拔产生时添加环境变量的函数 */
    int    (*dev_uevent)(struct device *dev, char **envp, int num_envp,
                char *buffer, int buffer_size);/* 类中的设备热插拔时添加环境变量的函数 */

    void    (*release)(struct class_device *dev);/* 把设备从类中删除的函数 */
```

```
    void    (*class_release)(struct class *class);/* 删除类本身的函数 */
    void    (*dev_release)(struct device *dev);

    int    (*suspend)(struct device *, pm_message_t state);
    int    (*resume)(struct device *);
};


/*类注册函数:*/
int class_register(struct class *cls);
void class_unregister(struct class *cls);

/*类属性的接口:*/
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);
```

在更新的内核里，这个结构体变得简洁了，删除了一些成员：

```
/*in Linux 2.6.26.5*/

/*
 * device classes
 */
struct class {
    const char        *name;
    struct module        *owner;

    struct kset        subsys;
    struct list_head    devices;
    struct list_head    interfaces;
    struct kset        class_dirs;
    struct semaphore    sem; /* locks children, devices, interfaces */
    struct class_attribute        *class_attrs;
    struct device_attribute        *dev_attrs;

    int (*dev_uevent)(struct device *dev, struct kobj_uevent_env *env);

    void (*class_release)(struct class *class);
    void (*dev_release)(struct device *dev);

    int (*suspend)(struct device *dev, pm_message_t state);
    int (*resume)(struct device *dev);
};
```

类设备（在新内核中已被删除）

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：

```
struct class_device {
    struct list_head    node;/*for internal use by the driver core only*/
    struct kobject        kobj;/*for internal use by the driver core only*/
    struct class        * class;    /* 指向该设备所属的类，必须*/
    dev_t            devt;        /* dev_t, creates the sysfs "dev" , for internal use by the driver core only*/
    struct class_device_attribute *devt_attr;/*for internal use by the driver core only*/
    struct class_device_attribute uevent_attr;
    struct device        * dev;        /* 指向此设备相关的 device 结构体，可选。若不为NULL，应是一个从类入口
到/sys/devices 下相应入口的符号连接，以便用户空间查找设备入口*/
    void            * class_data;    /* 私有数据指针 */
    struct class_device    *parent;    /* parent of this child device, if there is one */
    struct attribute_group ** groups;    /* optional groups */


    void    (*release)(struct class_device *dev);
```

```
    int     (*uevent)(struct class_device *dev, char **envp,
             int num_envp, char *buffer, int buffer_size);
    char    class_id[BUS_ID_SIZE];    /* 此类中的唯一的名字 */
};

/*类设备注册函数:*/
int class_device_register(struct class_device *cd);
void class_device_unregister(struct class_device *cd);

/*重命名一个已经注册的类设备入口:*/
int class_device_rename(struct class_device *cd, char *new_name);

/*类设备入口属性:*/
struct class_device_attribute {
 struct attribute attr;
 ssize_t (*show)(struct class_device *cls, char *buf);
 ssize_t (*store)(struct class_device *cls, const char *buf,
 size_t count);
};

CLASS_DEVICE_ATTR(_name, _mode, _show, _store);

/*创建和删除除struct class中设备默认属性外的属性*/
int class_device_create_file(struct class_device *cls, const struct class_device_attribute *attr);
void class_device_remove_file(struct class_device *cls, const struct class_device_attribute *attr);
```

类接口

类子系统有一个 Linux 设备模型的其他部分找不到的附加概念，称为“接口”，可将它理解为一种设备加入或离开类时获得信息的触发机制，结构体如下：

```
struct class_interface {
    struct list_head    node;
    struct class        *class;/* 指向该接口所属的类*/

    int (*add) (struct class_device *, struct class_interface *);
/*当一个类设备被加入到在 class_interface 结构中指定的类时，将调用接口的 add 函数，进行一些设备需要的额外设置，通常是添加更多属性或其他的一些工作*/
    void (*remove)    (struct class_device *, struct class_interface *);/*一个接口的功能是简单明了的. 当设备从类中删除，将调用remove 方法来进行必要的清理*/
    int (*add_dev)     (struct device *, struct class_interface *);
    void (*remove_dev) (struct device *, struct class_interface *);
};

/*注册或注销接口的函数:*/
int class_interface_register(struct class_interface *class_intf);
void class_interface_unregister(struct class_interface *class_intf);
/*一个类可注册多个接口*/
```


上一篇：Linux设备驱动程序学习（12）-Linux设备模型（底层原理简介）
下一篇：Linux设备驱动程序学习（14）-Linux设备模型（各环节的整合）

Linux设备驱动程序学习（1）-...	【ROOTFS搭建】busybox的httpd...	系统提供的库函数存在内存泄漏...
Linux设备驱动程序学习（2）-...	xmanager 2.0 for linux配置	linux虚拟机 求教
Linux设备驱动程序学习（3）-...	什么是shell	初学UNIX环境高级编程的，关于...
Linux设备驱动程序学习（4）-...	linux socket的bug??	chinaunix博客什么时候可以设...

给主人留下些什么吧！~~

评论热议

请登录后评论。

登录 注册