昵称：yangjunhe
园龄：2年9个月
粉丝：0
关注：1
+加关注

< 2021年9月 >

日	一	二	三	四	五	六
29	30	31	1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	1	2
3	4	5	6	7	8	9

搜索
找找看
谷歌搜索

常用链接
我的随笔
我的评论
我的参与
最新评论
我的标签

随笔分类
android 基础(2)
git (1)
Java primary(4)
linux(19)
mcu(4)
Qt相关(8)

随笔档案
2020年9月(1)
2020年8月(2)
2020年7月(2)
2020年5月(2)
2019年12月(7)
2019年11月(4)
2019年10月(7)
2019年9月(1)
2019年6月(2)
2019年4月(4)
2019年3月(6)
2019年1月(3)

about QT
雪莲
朝闻道

阅读排行榜

yangjunhe

博客园 首页 新随笔 联系 订阅 管理

linux设备驱动程序--sysfs用户接口的使用

https://www.cnblogs.com/downey-blog/p/10501514.html

自2.6版本开始，linux内核开始使用sysfs文件系统，它的作用是将设备和驱动程序的信息导出到用户空间，方便了用户读取设备信息，同时支持修改和调整。

与ext系列和fat等文件系统不同的是，sysfs是一个系统在启动时构建在内存中虚拟文件系统，一般被挂载在/sys目录下，既然是存储在内存中，自然掉电不保存，不能存储用户数据。

事实上，在之前也有同样的虚拟文件系统建立了内核与用户系统信息的交互，它就是procfs，但是procfs并非针对设备和驱动程序，而是针对整个内核信息的抽象接口。

所以，内核开发人员觉得有必要使用一个独立的抽象接口来描述设备和驱动信息，毕竟直到目前，驱动代码在内核代码中占比非常大，内容也是非常庞杂。这样可以避免procfs的混乱，子系统之间的分层和分离总是能带来更清晰地框架。

sysfs的默认目录结构

上文中提到，sysfs一般被挂载在/sys目录下，我们可以通过ls /sys来查看sysfs的内容：

```
block  bus  class  dev  devices  firmware  fs  kernel  module  power
```

首先需要注意的是，sysfs目录下的各个子目录中存放的设备信息并非独立的，我们可以看成不同的目录是从不同的角度来描述某个设备信息。

一个设备可能同时有多个属性，所以对于同一个驱动设备，同时存在于不同的子目录下，例如：在之前的章节中，我们使用create_dev_node.c编译出create_dev_node.ko模块，加载完成之后，我们可以在/sys下面看到当前驱动相关的目录：

- /sys/module/create_device_node/
- /sys/class/basic_class/basic_demo (basic class为驱动程序中创建的class名称,basic_demo为设备名)
- /sys/devices/virtual/basic_class/basic_demo (basic class为驱动程序中创建的class名称,basic_demo为设备名)

理解了这个概念，我们再来简览/sys各目录的功能：

- /sys/block:该子目录包含在系统上发现的每个块设备的一个符号链接。符号链接指向/sys/devices下的相应目录。
- /sys/bus:该目录包含linux下的总线设备，每个子目录下主要包含两个目录：device和driver，后面会讲到linux的总线驱动模型，几乎都是分层为device和driver来实现的。
- /sys/class:每一个在内核中注册了class的驱动设备都会在这里创建一个class设备。
- /sys/dev:这个目录下包含两个子目录：block和char，分别代表块设备和字符设备，特别的是，它的组织形式是以major:minor来描述的，即每一个字符设备或者块设备在这里对应的目录为其相应的设备号major:minor.
- /sys/devices:包含整个目录内核设备树的描述，其他目录下的设备多为此目录的链接符号。
- /sys/firmware:包含查看和操作的接口
- /sys/fs:包含某些文件系统的子目录
- /sys/kernel:包含各种正在运行的内核描述文件。
- /sys/module:包含当前系统中被加载的模块信息。
- /sys/power：官方暂时没有描述，但是根据里面文件内容和命名习惯推测，这里存放的是一些与电源管理相关的模块信息。

如果你手头上有设备的话，博主强烈建议动手操作一遍看看，这样才能加深理解和记忆。

如果在、sys中添加描述文件

既然是承载用户与内核接口的虚拟文件系统，那肯定是要能被用户所使用的，那么我们应该怎样在/sys中添加描述文件呢？

首先，在上文中提到了，sysfs负责向用户展示驱动在内核中的信息，那么，肯定是要从内核出发，在内核中进行创建。

kobject kset

Linux设备模型的核心是使用Bus、Class、Device、Driver四个核心数据结构，将大量的、不同功能的硬件设备（以及驱动该硬件设备的方法），以树状结构的形式，进行归纳、抽象，从而方便Kernel的统一管理。

而硬件设备的数量、种类是非常多的，这就决定了Kernel中将会有大量的有关设备模型的数据结构。
这些数据结构一定有一些共同的功能，需要抽象出来统一实现，否则就会不可避免的产生冗余代码。这就是Kobject诞生的背景。

目前为止，Kobject主要提供如下功能：

- 通过parent指针，可以将所有Kobject以层次结构的形式组合起来。
- 使用一个引用计数（reference count），来记录Kobject被引用的次数，并在引用次数变为0时把它释放（这是Kobject诞生时的唯一功能）。
- 和sysfs虚拟文件系统配合，将每一个Kobject及其特性，以文件的形式，开放到用户空间（有关sysfs，会在其它文章中专门描述，本文不会涉及太多内容）。

  注1：在Linux中，Kobject几乎不会单独存在。它的主要功能，就是内嵌在一个大型的数据结构中，为这个数据结构提供一些底层的功能实现。
  注2：Linux driver开发者，很少会直接使用Kobject以及它提供的接口，而是使用构建在Kobject之上的设备模型接口。

至于kset，其实可以看成是kobject的集合，它也可以当成kobject来使用，下面来看看这两个结构体的内容：

```
struct kset {
    /*链表，记录所有连入这个kset的kobject*/
    struct list_head list;
    /*kset要在文件系统中生成一个目录，同样需要包含一个kobj结构体，以插入内核树中*/
    struct kobject kobj;
    ...
} __randomize_layout;
```

```
struct kobject {
    const char      *name;
    /*当前kobj的父节点，在文件系统中的表现就是父目录*/
    struct kobject      *parent;
    /*kobj属于的kset*/
    struct kset     *kset;
    /*kobj的类型描述，最主要的是其中的属性描述，包含其读写方式*/
    struct kobj_type    *ktype;
    /*当前kobj的引用，只有当引用为0时才能被删除*/
    struct kref     kref;
    ...
};
```

虽然linux基于C语言开发，但是其面向对象的思想无处不在，同时我们可以将kobject结构体看成是一个基类，提供基础的功能，而其他更为复杂的结构继承自这个结构体，延伸出不同的属性。

创建实例

介绍完kobject和kset的概念，当然是给出一个具体的实例来说明kobject和kset的使用：
kobject_create_test.c:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/kobject.h>
#include <linux/sysfs.h>
#include <linux/slab.h>

//指定license版本
MODULE_LICENSE("GPL");
```

```
static struct kobject *kob;
static struct kset *kst;

//设置初始化入口函数
static int __init hello_world_init(void)
{

    int ret = 0;
    kst = kset_create_and_add("test_kset",NULL,kernel_kobj->parent);
    if(!kst)
    {
        printk(KERN_ALERT "Create kset failed\n");
        kset_put(kst);
    }
    kob = kzalloc(sizeof(*kob),GFP_KERNEL);
    if(IS_ERR(kob)){
        printk(KERN_ALERT "alloc failed!!\n");
        return -ENOMEM;
    }

    ret = kobject_init_and_add(kob, NULL, NULL, "%s", "test_obj");
    if(ret)
    {
        kobject_put(kob);
        kset_unregister(kst);
    }

    printk(KERN_DEBUG "kobj test project!!!\n");
    return 0;
}

//设置出口函数
static void __exit hello_world_exit(void)
{
    kobject_put(kob);
    kset_unregister(kst);
    printk(KERN_DEBUG "goodbye !!!\n");
}

//将上述定义的init()和exit()函数定义为模块入口/出口函数
module_init(hello_world_init);
module_exit(hello_world_exit);
```

在上文代码中我们创建了一个kset对象和一个kobject对象：

- kset名为"test_kset",父节点为kernel_kobj->parent，这个kernel_kobj事实上就是/sys/kernel节点，这里相当于在/sys目录下创建一个test_kset目录。
- kobject名为"test_obj"，没有指定父节点，默认父节点为/sys.

编译加载运行

修改Makefile，然后编译kobject_create_test.c:

```
make
```

加载模块到内核：

```
sudo insmod kobject_create_test.ko
```

查看结果

我们可以使用下面的指令查看：

```
ls -l /sys/test*
```

输出：

```
/sys/test_kset
total 0
```

```
/sys/test_obj:
total 0
```

果然，在/sys目录下生成了相应目录。

添加属性

事实上严格来说，上面的示例是有问题的：

- 首先，这两个文件仅仅是存在在那里，任何作用也起不了
- 如果你有同时查看log信息，会发现，上面的示例在加载时内核会报错：

  ```
  Dec 23 08:44:28 beaglebone kernel: [21705.791009] kobject (daa8d880):
  must have a ktype to be initialized properly!
  ```

  报错信息可以看到，对于kobject而言，必须对kobject添加相应的操作属性。

ktype

既然需要添加相应操作属性，那我们就再来详细看看kobject结构体的源码(为避免陷入一些不必要的细节，博主只列出主干部分，有兴趣的朋友可以自行查看源码)：

```
struct kobject {
    ...
    struct kobj_type    *ktype;
    ...
};
```

先从kobject中找到kobj_type，这是描述kobject属性的结构体

```
struct kobj_type {
    void (*release)(struct kobject *kobj);
    const struct sysfs_ops *sysfs_ops;
    struct attribute **default_attrs;
    ...
};
```

在kobj_type结构体中：

- release函数在当前kobject的引用计数为0时，释放当前kobject的资源。
- sysfs_ops：对文件的操作函数
- default_attrs：表示当前object的属性

我们再来看看sysfs_ops，这是对应文件的操作函数：

```
struct sysfs_ops {
    ssize_t (*show)(struct kobject *, struct attribute *, char *);
    ssize_t (*store)(struct kobject *, struct attribute *, const char *,
};
```

default_attrs描述了当前kobject的属性：

```
struct attribute {
    const char      *name;          //作为当前kobject目录下的文件名
    umode_t         mode;           //文件操作权限
}
```

必须来个小结

不知道上面的结构体关系有没有把你绕晕，我们按照主干线再来总结一下：

- kobject和kset将会在相应的/sys目录下创建一个目录，父目录由参数parent指定，本目录名由参数name指定。
- 每个kobject需要填充kobj_type结构体，这个结构体指定本目录的相关操作信息,也可以使用默认值。
- kobj_type结构体主要包含三个部分：
  - release主要负责当前kobject的释放
  - sysfs_ops的内容为两个函数指针，store对应用户对文件写操作的回调函数，show对应用户读文件的回调函数，这两个函数一般有开发者来决定执行什么操作，这个接口实现了用户与内核数据的交互。
  - attribute描述kobject的属性，它有两个元素，name和mode，分别表示kobject目录下的文件名和文件操作权限，定义为二级指针，在使用时传入的是指针数组。

示例

光说不练假把式，我们来看看下面的示例kobject_create_with_attrs：

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/kthread.h>
#include <linux/delay.h>
#include <linux/kobject.h>
#include <linux/sysfs.h>
#include <linux/slab.h>
#include <linux/string.h>
#include <linux/gpio.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Downey");
MODULE_DESCRIPTION("Kobject test!");
MODULE_VERSION("0.1");

static int led_status = 0;
#define LED_PIN   26
/************************kobject***************************/
static struct kobject *kob;

static ssize_t led_show(struct kobject* kobjs,struct kobj_attribute *att
{
    printk(KERN_INFO "Read led\n");
    return sprintf(buf,"The led_status status = %d\n",led_status);
}

static ssize_t led_status_show(struct kobject* kobjs,struct kobj_attribu
{
    printk(KERN_INFO "led status show\n");
    return sprintf(buf,"led status : \n%d\n",led_status);
}


static ssize_t led_status_store(struct kobject *kobj, struct kobj_attrib
{
    printk(KERN_INFO "led status store\n");
    if(0 == memcmp(buf,"on",2))
    {
        gpio_set_value(LED_PIN,1);
        led_status = 1;
    }
    else if(0 == memcmp(buf,"off",3))
    {
        gpio_set_value(LED_PIN,0);
        led_status = 0;
    }
    else
    {
        printk(KERN_INFO "Not support cmd\n");
    }

    return count;
}


static struct kobj_attribute status_attr = __ATTR_RO(led);
static struct kobj_attribute led_attr = __ATTR(led_status,0660,led_statu


static struct attribute *led_attrs[] = {
    &status_attr.attr,
    &led_attr.attr,
    NULL,
};
```

```
static struct attribute_group attr_g = {
    .name = "kobject_test",
    .attrs = led_attrs,
};


int create_kobject(void)
{
    kob = kobject_create_and_add("obj_test",kernel_kobj->parent);
    return 0;
}

static void gpio_config(void)
{
    if(!gpio_is_valid(LED_PIN)){
        printk(KERN_ALERT "Error wrong gpio number\n");
        return ;
    }
    gpio_request(LED_PIN,"led_ctr");
    gpio_direction_output(LED_PIN,1);
    gpio_set_value(LED_PIN,1);
    led_status = 1;
}

static void gpio_deconfig(void)
{
    gpio_free(LED_PIN);
}

static int __init sysfs_ctrl_init(void){
    printk(KERN_INFO "Kobject test!\n");
    gpio_config();
    create_kobject();
    sysfs_create_group(kob, &attr_g);
    return 0;
}


static void __exit sysfs_ctrl_exit(void){

    gpio_deconfig();
    kobject_put(kob);
    printk(KERN_INFO "Goodbye!\n");
}


module_init(sysfs_ctrl_init);
module_exit(sysfs_ctrl_exit);
```

在上述的示例中，我们依旧引入了一个指示灯，值得注意的是，在示例中，博主并没有将led_attrs传入给kobject本身，而是使用sysfs_create_group()接口创建了一个目录，目录下的文件有led和led_status.

编译加载运行

修改Makefile，然后使用make进行编译。

加载相应内核模块：

```
sudo insmod kobject_create_with_attrs.ko
```

加载完成之后如果你有在gpio26连上指示灯，可以看到指示灯现在处于亮的状态，同时我们可以用指令查看是否在/sys目录下生成了相应的目录：

```
ls -l /sys/obj_test/kobject_test/
```

输出结果：

```
-r--r--r-- 1 root root 4096 Dec 25 14:45 led
-rw-rw---- 1 root root 4096 Dec 25 14:52 led_status
```

根据程序中的实现，led显示的内容是led的状态，同时我们可以通过向led_status文件来控制led灯的状态。

我们先查看led文件：

```
cat /sys/obj_test/kobject_test/led
```

输出：

```
The led_status status = 1
```

如我们所料，对led的读调用了led_show()函数，我们再来试试led_status文件，在这之前，我们先要赋予文件操作权限：

```
chmod 666 /sys/obj_test/kobject_test/led_status
```

然后往led_status文件中写off来关闭led：

```
echo "off" > /sys/obj_test/kobject_test/led_status
```

果然，led被关闭，此时我们再查看led文件发现led状态为0。

相信到这里，大家对kobject、kset和sysfs有了一个基本的理解，博主在这里再贴上一些kobject的注意事项：

- 上文说到kobject常常不会单独存在，而是作为一部分嵌入到其他对象中，一个对象struct只能包含一个kobject，不然会导致混乱
- kobject不能在栈上分配，也不推荐将其作为静态存储，最好的是在堆上申请资源，原因可以自己想想。
- 释放kobject时不要使用kfree,要使用kobject_put()函数释放
- 不要在release函数中对kobject改名，会造成内存泄漏。

关于kobject和kset更详细的部分欢迎大家访问官方文档，这里有更详细的资料。
同时建议大家多多尝试，这样才能有更深地理解。

kobject描述部分参考大牛的博客 (博主目前看过最好的讲解linux内核的系列博客，强烈推荐！)

分类: linux

yangjunhe
关注 - 1
粉丝 - 0
+加关注

0 0

« 上一篇： linux 内核符号
» 下一篇： 设备类class理解

posted on 2019-10-18 09:52 yangjunhe 阅读(571) 评论(0) 编辑 收藏 举报

刷新评论 刷新页面 返回顶部

(评论功能已被禁用)