Linux内核驱动自动创建设备节点文件 *2012-06-16 00:04:23*

Linux下生成驱动设备节点文件的方法有3个：1、手动mknod；2、利用devfs；3、利用udev

在刚开始写Linux设备驱动程序的时候，很多时候都是利用mknod命令手动创建设备节点，实际上Linux内核为我们提供了一组函数，可以用来在模块加载的时候自动在/dev目录下创建相应设备节点，并在卸载模块时删除该节点。

在2.6.17以前，在/dev目录下生成设备文件很容易，

devfs_mk_bdev
devfs_mk_cdev
devfs_mk_symlink
devfs_mk_dir
devfs_remove

这几个是纯devfs的api，2.6.17以前可用，但后来devfs被sysfs+udev的形式取代，同时期sysfs文件系统可以用的api：

class_device_create_file，在2.6.26以后也不行了，现在，使用的是device_create ，从2.6.18开始可用

struct device *device_create(struct class *class, struct device *parent,
dev_t devt, const char *fmt, ...)

从2.6.26起又多了一个参数drvdata： the data to be added to the device for callbacks

不会用可以给此参数赋NULL

struct device *device_create(struct class *class, struct device *parent,
dev_t devt, void *drvdata, const char *fmt, ...)

下面着重讲解第三种方法udev

在驱动用加入对udev的支持主要做的就是：在驱动初始化的代码里调用class_create(...)为该设备创建一个class，再为每个设备调用device_create(...)(在2.6较早的内核中用class_device_create)创建对应的设备。

内核中定义的struct class结构体，顾名思义，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。这样，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。

struct class和class_create(…) 以及device_create(…)都包含在在/include/linux/device.h中，使用的时候一定要包含这个头文件，否则编译器会报错。

struct class定义在头文件include/linux/device.h中

class_create(…)在/drivers/base/class.c中实现

device_create(…)函数在/drivers/base/core.c中实现

class_destroy(...),device_destroy(...)也在/drivers/base/core.c中实现调用过程类似如下：

```
static struct class *spidev_class;

/*-------------------------------------------------------------------------*/

static int __devinit spidev_probe(struct spi_device *spi)
{
	....

	dev = device_create(spidev_class, &spi->dev, spidev->devt,
		spidev, "spidev%d.%d",
		spi->master->bus_num, spi->chip_select);
	...
}

static int __devexit spidev_remove(struct spi_device *spi)
{
   ......
   device_destroy(spidev_class, spidev->devt);
```

```
    .....

    return 0;
}

static struct spi_driver spidev_spi = {
    .driver = {
        .name =        "spidev",
        .owner =    THIS_MODULE,
    },
    .probe =    spidev_probe,
    .remove =    __devexit_p(spidev_remove),

};

/*-------------------------------------------------------------------------*/

static int __init spidev_init(void)
{
    ....

    spidev_class = class_create(THIS_MODULE, "spidev");
    if (IS_ERR(spidev_class)) {
        unregister_chrdev(SPIDEV_MAJOR, spidev_spi.driver.name);
        return PTR_ERR(spidev_class);
    }
    ....
}
module_init(spidev_init);

static void __exit spidev_exit(void)
{
    ......
    class_destroy(spidev_class);
    ......
}
module_exit(spidev_exit);

MODULE_DESCRIPTION("User mode SPI device interface");
MODULE_LICENSE("GPL");
```

下面以一个简单字符设备驱动来展示如何使用这几个函数

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/device.h>

int HELLO_MAJOR = 0;
int HELLO_MINOR = 0;
int NUMBER_OF_DEVICES = 2;

struct class *my_class;
```

```
//struct cdev cdev;
//dev_t devno;

struct hello_dev {
   struct device *dev;
   dev_t chrdev;
   struct cdev cdev;
};

static struct hello_dev *my_hello_dev = NULL;

struct file_operations hello_fops = {
 .owner = THIS_MODULE
};

static int __init hello_init (void)
{
int err = 0;
struct device *dev;

my_hello_dev = kzalloc(sizeof(struct hello_dev), GFP_KERNEL);
if (NULL == my_hello_dev) {
   printk("%s kzalloc failed!\n",__func__);
   return -ENOMEM;
}

devno = MKDEV(HELLO_MAJOR, HELLO_MINOR);
if (HELLO_MAJOR)
   err= register_chrdev_region(my_hello_dev->chrdev, 2, "memdev");
else
{
   err = alloc_chrdev_region(&my_hello_dev->chrdev, 0, 2, "memdev");
   HELLO_MAJOR = MAJOR(devno);
}
if (err) {
   printk("%s alloc_chrdev_region failed!\n",__func__);
   goto alloc_chrdev_err;
}
printk("MAJOR IS %d\n",HELLO_MAJOR);

cdev_init(&(my_hello_dev->cdev), &hello_fops);
my_hello_dev->cdev.owner = THIS_MODULE;
err = cdev_add(&(my_hello_dev->cdev), my_hello_dev->chrdev, 1);
if (err) {
   printk("%s cdev_add failed!\n",__func__);
   goto cdev_add_err;
}
printk (KERN_INFO "Character driver Registered\n");

my_class = class_create(THIS_MODULE,"hello_char_class");  //类名为hello_char_class
if(IS_ERR(my_class))
{
   err = PTR_ERR(my_class);
   printk("%s class_create failed!\n",__func__);
```

```
    goto class_err;
}

dev = device_create(my_class,NULL,my_hello_dev->chrdev,NULL,"memdev%d",0);     //设备名为memdev
if (IS_ERR(dev)) {
    err = PTR_ERR(dev);
    gyro_err("%s device_create failed!\n",__func__);
    goto device_err;
}

printk("hello module initialization\n");
return 0;

device_err:
    device_destroy(my_class, my_hello_dev->chrdev);
class_err:
    cdev_del(my_hello_dev->chrdev);
cdev_add_err:
    unregister_chrdev_region(my_hello_dev->chrdev, 1);
alloc_chrdev_err:
    kfree(my_hello_dev);
return err;
}

static void __exit hello_exit (void)
{
cdev_del (&(my_hello_dev->cdev));
unregister_chrdev_region (my_hello_dev->chrdev,1);
device_destroy(my_class, devno);        //delete device node under /dev//必须先删除设备，再删除class类
class_destroy(my_class);                //delete class created by us
printk (KERN_INFO "char driver cleaned up\n");
}

module_init (hello_init);
module_exit (hello_exit);

MODULE_LICENSE ("GPL");
```

这样，模块加载后，就能在/dev目录下找到memdev这个设备节点了。

例2:内核中的drivers/i2c/i2c-dev.c

在i2cdev_attach_adapter中调用device_create(i2c_dev_class, &adap->dev,

```
        MKDEV(I2C_MAJOR, adap->nr), NULL,
        "i2c-%d", adap->nr);
```

这样在dev目录就产生i2c-0 或i2c-1节点

接下来就是udev应用，udev是应用层的东西，udev需要内核sysfs和tmpfs的支持，sysfs为udev提供设备入口和uevent通道，tmpfs为udev设备文件提供存放空间

udev的源码可以在去相关网站下载，然后就是对其在运行环境下的移植，指定交叉编译环境，修改Makefile下的CROSS_COMPILE，如为mipsel-linux-，DESTDIR=xxx，或直接make CROSS_COMPILE=mipsel-linux- ,DESTDIR=xxx 并install

把主要生成的udevd、udevstart拷贝rootfs下的/sbin/目录内，udev的配置文件udev.conf和rules.d下的rules文件拷贝到rootfs下的/etc/目录内

并在rootfs/etc/init.d/rcS中添加以下几行:

echo “Starting udevd...”

/sbin/udevd --daemon
/sbin/udevstart
（原rcS内容如下:
mount filesystems
/bin/mount -t proc /proc /proc
/bin/mount -t sysfs sysfs /sys
/bin/mount -t tmpfs tmpfs /dev
create necessary devices
/bin/mknod /dev/null c 1 3
/bin/mkdir /dev/pts
/bin/mount -t devpts devpts /dev/pts
/bin/mknod /dev/audio c 14 4
/bin/mknod /dev/ts c 10 16
)
这样当系统启动后，udevd和udevstart就会解析配置文件，并自动在/dev下创建设备节点文件

Udev简介(from wikipedia)

udev 是Linux kernel 2.6系列的设备管理器。它主要的功能是管理/dev目录底下的设备节点。它同时也是用来接替devfs及hotplug的功能，这意味着它要在添加／删除硬件时处理/dev目录以及所有用户空间的行为，包括加载firmware时。

udev的最新版本依赖于升级后的Linux kernel 2.6.13的uevent接口的最新版本。使用新版本udev的系统不能在2.6.13以下版本启动，除非使用*noudev*参数来禁用udev并使用传统的／dev来进行设备读取。

概要

在传统的Linux系统中，/dev目录下的设备节点为一系列静态存在的文件，而udev则动态提供了在系统中实际存在的设备节点。虽然devfs提供了类似功能，udev的支持者也给出了很多udev实现得比devfs好的理由[1]:

- udev支持设备的固定命名，而并不依赖于设备插入系统的顺序。默认的udev设置提供了存储设备的固定命名。任何硬盘都根据其唯一的文件系统id、磁盘名称及硬件连接的物理位置来进行识别。
- udev完全在用户空间执行，而不是像devfs在内核空间一样执行。结果就是udev将命名策略从内核中移走，并可以在节点创建前用任意程序在设备属性中为设备命名。

运行方式

udev是一个通用的内核设备管理器。它以守护进程的方式运行于Linux系统，并监听在新设备初始化或设备从系统中移除时，内核（通过netlink socket）所发出的uevent。

系统提供了一套规则用于匹配可发现的设备事件和属性的导出值。匹配规则可能命名并创建设备节点，并运行配置程序来对设备进行设置。udev规则可以匹配像内核子系统、内核设备名称、设备的物理等属性，或设备串行号的属性。规则也可以请求外部程序提供信息来命名设备，或指定一个永远一样的自定义名称来命名设备，而不管设备什么时候被系统发现。

udev系统可以分为三个部分：

- libudev函数库，可以用来获取设备的信息。
- udevd守护进程，处于用户空间，用于管理虚拟／dev
- 管理命令udevadm，用来诊断出错情况。

系统获取内核通过netlink socket发出的信息。早期的版本使用hotplug，并在`/etc/hotplug.d/default`添加一个链接到自身来达到目的。

udev的命令格式

BUS 总线 KERNEL 内核名如sd* ID 设备id 如总线id PLACE

SYSFS{filename}

PROGRAM 调用外部程序 RESULT 匹配program返回的结果 NAME

SYMLINK 连接规则