# Linux 相关知识

## 平台设备：

### 平台：SC7731C、SC9832 、SC9850

文件系统中的路径：/sys/bus/platform/devices

（1）、module\_init(assistant\_cam\_init);

（2）、

static int \_\_init assistant\_cam\_init(void)

{

    int ret = 0;

    /\* assistant\_cam user space access interface \*/

    ret = platform\_device\_register(&assistant\_cam\_user\_space\_device);

    if (ret) {

        GE\_DBG("\*\*\*\*[assistant\_cam\_init] Unable to device register(%d)\n",

                ret);

        return ret;

    }

    ret = platform\_driver\_register(&assistant\_cam\_user\_space\_driver); **//如果使用dts传入不需要这步**

    if (ret) {

        GE\_DBG("\*\*\*\*[assistant\_cam\_init] Unable to register driver (%d)\n",

                ret);

        return ret;

    }

    return 0;

}

struct platform\_device **assistant\_cam\_user\_space\_device** = {    **//如果使用dts传入不需要这步，**

   .name = "gc0310-user",//这个必须和下面的一样，而且会在目录 /sys/bus/platform/devices/下面生成//gc0310-user文件夹

    .id = -1,

};

static struct platform\_driver **assistant\_cam\_user\_space\_driver** = {

    .probe = assistant\_cam\_user\_space\_probe,

    .driver = {

           .name = "gc0310-user", //这个必须和上面的一样 ,会在devices目录下面创建一个gc0310-user文件夹。

    },

};

static int **assistant\_cam\_user\_space\_probe**(struct platform\_device \*dev)

{

    int ret\_device\_file = 0;

    GE\_DBG("\*\*\*\*\*\*\*\* assistant\_cam\_user\_space\_probe!! \*\*\*\*\*\*\*\*\n");

    ret\_device\_file = device\_create\_file(&(dev->dev), &dev\_attr\_gc0310\_init); //其中&dev\_attr\_gc0310\_init)是由下面的gc0310\_init 前面加上 dev\_attr得来的。

    return 0;

}

（3）、

static DEVICE\_ATTR(gc0310\_init, 0664, show\_assistant\_cam\_init, store\_assistant\_cam\_init);    /\* 664 \*/

**//会在 /sys/bus/platform/devices/gc0310-user/目录下生成一个gc0310\_init的节点。**

static ssize\_t **store\_assistant\_cam\_init**(struct device \*dev, struct device\_attribute \*attr,

                    const char \*buf, size\_t size)

{

    GE\_DBG("[store\_assistant\_cam\_init]\n");

    sensor\_init();

    return size;

}

static ssize\_t **show\_assistant\_cam\_init**(struct device \*dev, struct device\_attribute \*attr, char \*buf)

{

    GE\_DBG("[show\_assistant\_cam\_init]\n");

    sensor\_init();

    return 0;

}

（4）、所得节点的路径：

 /sys/bus/platform/devices/gc0310-user/gc0310\_init

（5）、

读操作：cat   /sys/bus/platform/devices/gc0310-user/gc0310\_init

写操作：echo  /sys/bus/platform/devices/gc0310-user/gc0310\_init

## I2C设备驱动

### 平台：SC7731C、SC9832 、SC9850

文件系统中的路径：/sys/bus/i2c/

好文推荐：[**http://blog.csdn.net/snowwupl/article/details/9134145**](http://blog.csdn.net/snowwupl/article/details/9134145)

**重要函数：**

(1)、I2c\_add\_driver()是一个宏，其原型是：

#define i2c\_add\_driver(driver) \

i2c\_register\_driver(THIS\_MODULE, driver)

**重要的结构体：**

（1）、

struct i2c\_driver {

int id;

unsigned int class;

/\* Notifies the driver that a new bus has appeared. This routine

\* can be used by the driver to test if the bus meets its conditions

\* & seek for the presence of the chip(s) it supports. If found, it

\* registers the client(s) that are on the bus to the i2c admin. via

\* i2c\_attach\_client. (LEGACY I2C DRIVERS ONLY)

\*/

int (\*attach\_adapter)(struct i2c\_adapter \*);/\*依附i2c\_adapter函数指针 \*/

int (\*detach\_adapter)(struct i2c\_adapter \*);/\*脱离i2c\_adapter函数指针\*/

/\* tells the driver that a client is about to be deleted & gives it

\* the chance to remove its private data. Also, if the client struct

\* has been dynamically allocated by the driver in the function above,

\* it must be freed here. (LEGACY I2C DRIVERS ONLY)

\*/

int (\*detach\_client)(struct i2c\_client \*); /\*i2c client脱离函数指针\*/

/\* Standard driver model interfaces, for "new style" i2c drivers.

\* With the driver model, device enumeration is NEVER done by drivers;

\* it's done by infrastructure. (NEW STYLE DRIVERS ONLY)

\*/

int (\*probe)(struct i2c\_client \*, const struct i2c\_device\_id \*);

int (\*remove)(struct i2c\_client \*);

/\* driver model interfaces that don't relate to enumeration \*/

void (\*shutdown)(struct i2c\_client \*);

int (\*suspend)(struct i2c\_client \*, pm\_message\_t mesg);

int (\*resume)(struct i2c\_client \*);

/\* a ioctl like command that can be used to perform specific functions

\* with the device.

\*/

int (\*command)(struct i2c\_client \*client, unsigned int cmd, void \*arg);//类似ioctl\*/

struct device\_driver driver;/\*设备驱动结构体\*/

const struct i2c\_device\_id \*id\_table;

/\* Device detection callback for automatic device creation \*/

int (\*detect)(struct i2c\_client \*, int kind, struct i2c\_board\_info \*);

const struct i2c\_client\_address\_data \*address\_data;

struct list\_head clients;

};

（2）、

/\*\*

\* struct i2c\_client - represent an I2C slave device

\* @flags: I2C\_CLIENT\_TEN indicates the device uses a ten bit chip address;

\* I2C\_CLIENT\_PEC indicates it uses SMBus Packet Error Checking

\* @addr: Address used on the I2C bus connected to the parent adapter.

\* @name: Indicates the type of the device, usually a chip name that's

\* generic enough to hide second-sourcing and compatible revisions.

\* @adapter: manages the bus segment hosting this I2C device

\* @driver: device's driver, hence pointer to access routines

\* @dev: Driver model device node for the slave.

\* @irq: indicates the IRQ generated by this device (if any)

\* @list: list of active/busy clients (DEPRECATED)

\* @detected: member of an i2c\_driver.clients list

\* @released: used to synchronize client releases & detaches and references

\*

\* An i2c\_client identifies a single device (i.e. chip) connected to an

\* i2c bus. The behaviour exposed to Linux is defined by the driver

\* managing the device.

\*/

struct i2c\_client {

unsigned short flags; /\* div., see below \*//标志

unsigned short addr; /\* chip address - NOTE: 7bit \*///低7位为芯片地址

/\* addresses are stored in the \*/

/\* \_LOWER\_ 7 bits \*/

char name[I2C\_NAME\_SIZE];//设备名称

struct i2c\_adapter \*adapter; /\* the adapter we sit on \*/依附的i2c\_adapter

struct i2c\_driver \*driver; /\* and our access routines \*/依附的i2c\_driver

struct device dev; /\* the device structure \*/

int irq; /\* irq issued by device \*/分配给设备的中断号

struct list\_head list; /\* DEPRECATED \*/

struct list\_head detected;

struct completion released;//用于同步

};

（3）、

struct i2c\_adapter {

struct module \*owner; //所属模块

unsigned int id; //algorithm 的类型 ,定义于i2c\_id.h,I2C\_ALGO\_\*\*\*\*\*

unsigned int class; /\* classes to allow probing for \*/

const struct i2c\_algorithm \*algo; /\* the algorithm to access the bus \*/总线的通信方法结构体指针

void \*algo\_data; //algorithm 私有数据

/\* --- administration stuff. \*/

int (\*client\_register)(struct i2c\_client \*); //client 注册时调用

int (\*client\_unregister)(struct i2c\_client \*);//client 注销时调用

/\* data fields that are valid for all devices \*/

u8 level; /\* nesting level for lockdep \*/

struct mutex bus\_lock;/\*控制并发访问的自旋锁\*/

struct mutex clist\_lock;

int timeout;

int retries; //重试次数

struct device dev; /\* the adapter device \*/ //适配器设备

int nr;

struct list\_head clients; /\* DEPRECATED \* /\* client链表头\*/

/

char name[48]; //适配器名称

struct completion dev\_released; //用于同步

}

（4）、

 struct i2c\_msg {  
2 \_\_u16 addr; /\* 设备地址\*/  
3 \_\_u16 flags; /\* 标志 \*/   
4 \_\_u16 len; /\* 消息长度\*/  
5 \_\_u8 \*buf; /\* 消息数据\*/  
6 };

（5）、

struct i2c\_algorithm {

/\* If an adapter algorithm can't do I2C-level access, set master\_xfer

to NULL. If an adapter algorithm can do SMBus access, set

smbus\_xfer. If set to NULL, the SMBus protocol is simulated

using common I2C messages \*/

/\* master\_xfer should return the number of messages successfully

processed, or a negative value on error \*/

int (\*master\_xfer)(struct i2c\_adapter \*adap, struct i2c\_msg \*msgs,

int num); /\*i2c传输函数指针\*/

int (\*smbus\_xfer) (struct i2c\_adapter \*adap, u16 addr,/\*smbus传输函数指针\*/

unsigned short flags, char read\_write,

u8 command, int size, union i2c\_smbus\_data \*data);

/\* To determine what the adapter supports \*/

u32 (\*functionality) (struct i2c\_adapter \*);/\*返回适配器支持的功能\*/

};

实例：

（1）、

        module\_init(assistant\_cam\_init);

        module\_exit(assistant\_cam\_exit);

（2）、

static int \_\_init assistant\_cam\_init(void)

{

    int ret = 0;

    if (i2c\_add\_driver(&assistant\_cam\_driver) != 0) {

        GE\_DBG(

                "[assistant\_cam\_init] failed to register assistant\_cam i2c driver.\n");

    } else {

        GE\_DBG(

                "[assistant\_cam\_init] Success to register assistant\_cam i2c driver.\n");

    }

}

 //如果没有这个DTS，也是要初始化device

#ifdef CONFIG\_OF

static const struct of\_device\_id assistant\_cam\_of\_match[] = {

{.compatible = "sprd,main2",}, //这个值应该和DTS中的值一样。

{},

};

MODULE\_DEVICE\_TABLE(of, assistant\_cam\_of\_match);

#endif

static struct i2c\_driver assistant\_cam\_driver = {

    .driver = {

            .name = "assistant\_cam",

#ifdef CONFIG\_OF

            .of\_match\_table = assistant\_cam\_of\_match,

#endif

    },

    .probe = assistant\_cam\_driver\_probe,

    .id\_table = assistant\_cam\_i2c\_id,

};

static void \_\_exit assistant\_cam\_exit(void)

{

    i2c\_del\_driver(&assistant\_cam\_driver);

}

static int **assistant\_cam\_driver\_probe**(struct i2c\_client \*client, const struct i2c\_device\_id \*id)

{

new\_client = client;

//这里改i2c地址是为了dts中不去动原先sub摄像头的I2C地址

new\_client->addr=0x21;

assistant\_cam\_hw\_component\_detect();

return 0;

}

(3)、I2C读取

**方法一**

调用：i2c\_read\_data(new\_client,addr,1, (u8\*)&get\_byte);

static int i2c\_read\_data(struct i2c\_client \*client, unsigned char command, int length, unsigned char \*values)

{

int retry;

int err;

struct i2c\_msg msgs[] =

{

{

.addr = client->addr,

.flags = 0,

.len = 1,

.buf = &command,

},

{

.addr = client->addr,

.flags = I2C\_M\_RD,

.len = length,

.buf = values,

},

};

for (retry = 0; retry < 5; retry++)

{

err = i2c\_transfer(client->adapter, msgs, 2);

if (err == 2)

break;

else

mdelay(5);

}

if (retry >= 5)

{

GE\_DBG(KERN\_ERR "%s: i2c read fail, err=%d\n", \_\_func\_\_, err);

return -EIO;

}

return 0;

}

**方法二**

调用：static int xxxx\_i2c\_read( struct i2c\_client\* client,uint8\_t reg,uint8\_t \*data)

static int xxxx\_i2c\_read( struct i2c\_client\* client,uint8\_t reg,uint8\_t \*data)

{

// write reg addr

if( 1!= i2c\_master\_send(client,&reg,1) ) {

printk( KERN\_ERR " xxxx\_i2c\_read fail! \n" );

return -1;

}

// wait

msleep(10);

// read

if( 1!= i2c\_master\_recv(client,data,1) ) {

printk( KERN\_ERR " xxxx\_i2c\_read fail! \n" );

return -1;

}

return 0;

}

其中：i2c\_master\_recv 接口的三个参数：client 为此次与主机通信的从机，buf 为接收的数据指针，count 为接收数据的字节数。

(4)、I2C写入

**方法一**

调用：i2c\_write\_data(new\_client,addr,1,&para);

static int i2c\_write\_data(struct i2c\_client \*client, u16 command, u16 length, u16 \*values)

{

int retry;

int err;

unsigned char data[11];

struct i2c\_msg msg;

int index;

if (!client)

return -EINVAL;

else if (length >= 10)

{

GE\_DBG(KERN\_ERR "%s:length %d exceeds 10\n", \_\_func\_\_, length);

return -EINVAL;

}

data[0] = command;

for (index=1;index<=length;index++)

data[index] = values[index-1];

msg.addr = client->addr;

msg.flags = 0;

msg.len = length+1;

msg.buf = data;

for (retry = 0; retry < 5; retry++)

{

err = i2c\_transfer(client->adapter, &msg, 1);

if (err == 1)

break;

else

mdelay(5);

}

if (retry >= 5)

{

GE\_DBG(KERN\_ERR "%s: i2c write fail, err=%d\n", \_\_func\_\_, err);

return -EIO;

}

return 0;

}

**方法二**

调用：static int xxxx\_i2c\_write( struct i2c\_client\* client,uint8\_t reg,uint8\_t data)

static int xxxx\_i2c\_write( struct i2c\_client\* client,uint8\_t reg,uint8\_t data)

{

unsigned char buffer[2];

buffer[0] = reg;

buffer[1] = data;

if( 2!= i2c\_master\_send(client,buffer,2) ) {

printk( KERN\_ERR " xxxx\_i2c\_write fail! \n" );

return -1;

}

return 0;

}

其中：i2c\_master\_send 接口的三个参数：client 为此次与主机通信的从机，buf 为发送的数据指针，count 为发送数据的字节数。

(5)、文件系统的路径：

/sys/bus/i2c/drivers

和 /sys/bus/i2c/devices  //在这里可以看到匹配到的i2c设备的地址。

**I2C小知识：**



**ADB查看I2C挂载情况**

使用ADB，当不确定I2C设备是否挂载成功可以使用以下命令：

adb shell

cd /sys/bus/i2c

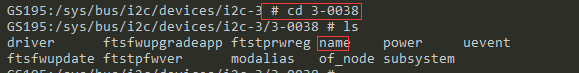
cd devices

cd i2c – x //这边看你的设备是挂载到哪个IC总线下面就进入哪一个总线比如我进入总线i2c-3





这个时候就可以看到对应的I2C总线上面有挂载着几个设备了根据设备地址找到相应的文件夹进入（我这边进入的TP（设备地址0x38））：



就可以看到I2C相关的东西，通过cat就可以得到你想要的东西比如名字还有of\_name的dts配置



## 3、各种节点的创建方法

### 一、proc\_create的使用方法

相关函数讲解：

头文件：#include <linux/proc\_fs.h>

（1）、proc文件夹的创建

struct proc\_dir\_entry \*proc\_mkdir(const char \*name, struct proc\_dir\_entry \*parent);

name就是要创建的文件夹名称。

parent是要创建节点的父节点。也就是要在哪个文件夹之下创建新文件夹，需要将哪个文件夹的proc\_dir\_entry传入。

如果是在/proc目录下创建文件夹，parent为NULL。

例如：

struct proc\_dir\_entry \*mytest\_dir = proc\_mkdir("mytest", NULL);

（2）、proc文件的创建

static inline struct proc\_dir\_entry \*proc\_create(const char \*name, mode\_t mode,

struct proc\_dir\_entry \*parent, const struct file\_operations \*proc\_fops);

name就是要创建的文件名。

mode是文件的访问权限，以UGO的模式表示。

parent与proc\_mkdir中的parent类似。也是父文件夹的proc\_dir\_entry对象。

proc\_fops就是该文件的操作函数了。

例如：

struct proc\_dir\_entry \*mytest\_file = proc\_create("mytest", 0x0644, mytest\_dir, mytest\_proc\_fops);

还有一种方式：

struct proc\_dir\_entry \*mytest\_file = proc\_create("mytest/mytest", 0x0644, NULL, mytest\_proc\_fops);

（3）、mytest\_proc\_fops结构体

Struct file\_operations {

　　struct module \*owner;//拥有该结构的模块的指针，一般为THIS\_MODULES

loff\_t (\*llseek) (struct file \*, loff\_t, int);//用来修改文件当前的读写位置

ssize\_t (\*read) (struct file \*, char \_\_user \*, size\_t, loff\_t \*);//从设备中同步读取数据

ssize\_t (\*write) (struct file \*, const char \_\_user \*, size\_t, loff\_t \*);//向设备发送数据

ssize\_t (\*aio\_read) (struct kiocb \*, const struct iovec \*, unsigned long, loff\_t);//初始化一个异步的读取操作

ssize\_t (\*aio\_write) (struct kiocb \*, const struct iovec \*, unsigned long, loff\_t);//初始化一个异步的写入操作

　　int (\*readdir) (struct file \*, void \*, filldir\_t);//仅用于读取目录，对于设备文件，该字段为NULL

unsigned int (\*poll) (struct file \*, struct poll\_table\_struct \*); //轮询函数，判断目前是否可以进行非阻塞的读写或写入

　　int (\*ioctl) (struct inode \*, struct file \*, unsigned int, unsigned long); //执行设备I/O控制命令

　　long (\*unlocked\_ioctl) (struct file \*, unsigned int, unsigned long); //不使用BLK文件系统，将使用此种函数指针代替ioctl

　　long (\*compat\_ioctl) (struct file \*, unsigned int, unsigned long); //在64位系统上，32位的ioctl调用将使用此函数指针代替

　　int (\*mmap) (struct file \*, struct vm\_area\_struct \*); //用于请求将设备内存映射到进程地址空间

　　int (\*open) (struct inode \*, struct file \*); //打开

　　int (\*flush) (struct file \*, fl\_owner\_t id);

　　int (\*release) (struct inode \*, struct file \*); //关闭

　　int (\*fsync) (struct file \*, struct dentry \*, int datasync); //刷新待处理的数据

　　int (\*aio\_fsync) (struct kiocb \*, int datasync); //异步刷新待处理的数据

　　int (\*fasync) (int, struct file \*, int); //通知设备FASYNC标志发生变化

　　int (\*lock) (struct file \*, int, struct file\_lock \*);

　　ssize\_t (\*sendpage) (struct file \*, struct page \*, int, size\_t, loff\_t \*, int);

　　unsigned long (\*get\_unmapped\_area)(struct file \*, unsigned long, unsigned long, unsigned long, unsigned long);

　　int (\*check\_flags)(int);

　　int (\*flock) (struct file \*, int, struct file\_lock \*);

　　ssize\_t (\*splice\_write)(struct pipe\_inode\_info \*, struct file \*, loff\_t \*, size\_t, unsigned int);

　　ssize\_t (\*splice\_read)(struct file \*, loff\_t \*, struct pipe\_inode\_info \*, size\_t, unsigned int);

　　int (\*setlease)(struct file \*, long, struct file\_lock \*\*);

};

例如：

static const struct file\_operations mytest\_proc\_fops = {

.open = mytest\_proc\_open,

.read = seq\_read,

.write = mytest\_proc\_write,

.llseek = seq\_lseek,

.release = single\_release,

};

解析：

1)、以seq\_和single\_为前缀的函数都是kernel中现成的。可以参考文档：Documentation\filesystems\seq\_file.txt。

在mytest\_proc\_open函数中，**只需要调用single\_open函数**，并传入一个show函数即可。

例如：

static int mytest\_proc\_open(struct inode \*inode, struct file \*file)

{

return single\_open(file, mytest\_proc\_show, inode->i\_private);

}

其中第三个参数，也就是single\_open的data参数，内核中有些地方传入的NULL，有些地方传入的inode->i\_private，也有传入其他值的。来看看data在single\_open函数中如何被使用的：

if (!res)

((struct seq\_file \*)file->private\_data)->private = data;

所以data是seq\_file结构体的private成员。

2)、mytest\_proc\_show函数

传递给single\_open的show函数指针，将在proc file输出时被调用。

例如，执行cat /proc/mytest/mytest时，mytest\_proc\_show函数将会被调用。

static int mytest\_proc\_show (struct seq\_file \*seq, void \*v)

{

seq\_puts(seq, mytest\_flag ? "true\n" : "false\n");

return 0;

}

3）、mytest\_proc\_write函数 //mytest\_proc\_write函数会在写mytest文件时被调用, 功能就是记录写入数据到mytest文件。

static ssize\_t mytest\_proc\_write (struct file \*file, const char \_\_user \*buffer,

size\_t count, loff\_t \*pos)

{

char mode;

if (count > 0) {

if (get\_user(mode, buffer))

return -EFAULT;

mytest\_flag = (mode != '0');

}

return count;

}

应用实例：

（1）、

static int \_\_init zyt\_info\_proc\_init(void) {

proc\_create("zyt\_info", 0, NULL, &zyt\_info\_proc\_fops);

return 0;

}

static void \_\_exit zyt\_info\_proc\_exit(void) {

remove\_proc\_entry("zyt\_info", NULL);

}

MODULE\_LICENSE("GPL");

module\_init(zyt\_info\_proc\_init);

module\_exit(zyt\_info\_proc\_exit);

（2）、

static const struct file\_operations zyt\_info\_proc\_fops = {

.owner = THIS\_MODULE,

.open = zyt\_info\_proc\_open,

.read = seq\_read,

.llseek = seq\_lseek,

.release = single\_release,

};

（3）、

static int zyt\_info\_proc\_open(struct inode \*inode, struct file \*file) {

return single\_open(file, zyt\_info\_proc\_show, NULL);

}

（4）、

static int zyt\_info\_proc\_show(struct seq\_file \*m, void \*v) {

seq\_printf(m, "[LCD] : %s\n[MainSensor] : %s\n[SubSensor] : %s\n%s%s\n",

zyt\_get\_lcm\_info(),

get\_sensor\_info(0),

get\_sensor\_info(1),

zyt\_info\_buffer,

zyt\_battery\_info()

);

return 0;

}

调用方法：cat /proc/zyt\_info

### 二、sys fs下添加文件

作用：在调试驱动，可能需要对驱动里的某些变量进行读写，或函数调用。可通过sysfs接口创建驱动对应的属性，使得可以在用户空间通过sysfs接口的show和store函数与硬件交互；

（1）、创建单个文件

https://blog.csdn.net/njuitjf/article/details/16849333

作用：DEVICE\_ATTR、DRIVER\_ATTR，BUS\_ATTR，CLASS\_ATTR都可以在sys 下面创建文件，区别在于他们创建节点的目录不同，例如 使用DEVICE\_ATTR 创建的文件会在 /sys/devices/目录中 其他的三个也是对号入座。定义在头文件：kernel4.4/include/linux/device.h 中

这里主要介绍DEVICE\_ATTR，其他几个类似。在documentation/driver-model/Device.txt中有对DEVICE\_ATTR的详细介绍，这儿主要说明使用方法。

**DEVICE\_ATTR的原型：**

DEVICE\_ATTR(\_name, \_mode, \_show, \_store)

**DEVICE\_ATTR的定义：**

#define DEVICE\_ATTR(\_name, \_mode, \_show, \_store) \

struct device\_attribute dev\_attr\_##\_name = \_\_ATTR(\_name, \_mode, \_show, \_store)

备注：dev\_attr\_##\_name 就是这边定义的哈

#define \_\_ATTR(\_name,\_mode,\_show,\_store) { \

 .attr = {.name = \_\_stringify(\_name), .mode = \_mode }, \

 .show = \_show,     \

 .store = \_store,     \

}

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);

};

**例如：**

static DEVICE\_ATTR(xxx, 0666, xxx\_show, xxx\_store);

参数分析：

**\_name：**名称，也就是将在sys fs中生成的文件名称（名称可以随便起一个，便于记忆，并能体现其功能即可。）。

**\_mode：**上述文件的访问权限，与普通文件相同，UGO的格式。

可以使用数值设置例如：只读0444，只写0222，或者读写都行的0666

当然也可以使用使用系统定义好的宏更直观：

#define S\_IRWXU 00700 //用户可读写和执行

#define S\_IRUSR 00400//用户可读

#define S\_IWUSR 00200//用户可写

#define S\_IXUSR 00100//用户可执行

#define S\_IRWXG 00070//用户组可读写和执行

#define S\_IRGRP 00040//用户组可读

#define S\_IWGRP 00020//用户组可写

#define S\_IXGRP 00010//用户组可执行

#define S\_IRWXO 00007//其他可读写和执行

#define S\_IROTH 00004//其他可读

#define S\_IWOTH 00002//其他可写

#define S\_IXOTH 00001//其他可执行

**\_show：**显示函数，cat该文件时，此函数被调用。

例如：

static ssize\_t xxx\_show(struct device \*dev,

 struct device\_attribute \*attr, char \*buf)

{

 return scnprintf(buf, PAGE\_SIZE, "%d\n", dma\_dump\_flag);

}

**\_store：**写函数，echo内容到该文件时，此函数被调用。

例如：

static ssize\_t xxx\_store(struct device \*dev,

 struct device\_attribute \*attr, const char \*buf, size\_t count)

{

 unsigned long num;

 if (strict\_strtoul(buf, 0, &num))

  return -EINVAL;

 if (num < 0)

  return -EINVAL;

 mutex\_lock(&xxx\_lock);

 dma\_dump\_flag = num;

 mutex\_unlock(&xxx\_lock);

 return count;

}

接下来就是通过接口 device\_create\_file 来导入DEVICE\_ATTR这个结构体在sys fs下面创建文件了。

例如：

int ret = 0 ;

ret = device\_create\_file(&pdev->dev, &dev\_attr\_xxx); // dev\_attr\_##\_name定义的地方上面有提到

//&pdev->dev 就是你文件夹想要放的设备下面

   if (ret != 0) {

    dev\_err(&pdev->dev,

     "Failed to create xxx sysfs files: %d\n", ret);

    return ret;

   }

这个代码最好放在device的probe函数中。原因么，在documentation/driver-model/Device.txt中有说明。

总结：DEVICE\_ATTR的功能就是定义一个device\_attribute结构体对象。device\_create\_file利用该对象在device下创建文件。

（2）、批量创建文件（大体上和单个创建时类似的这里就举一个实例）

static ssize\_t sprdfgu\_show\_attribute(struct device \*dev,struct device\_attribute \*attr, char \*buf)

{

int i = 0;

const ptrdiff\_t off = attr - sprdfgu\_attribute;

switch (off) {

case FGU\_VOL\_ADC:

i += scnprintf(buf + i, PAGE\_SIZE - i, "%d\n",

fgu\_adi\_read(REG\_FGU\_VOLT\_VAL));

break;

case FGU\_CURRENT\_ADC:

i += scnprintf(buf + i, PAGE\_SIZE - i, "%d\n",

fgu\_adi\_read(REG\_FGU\_CURT\_VAL));

break;

case FGU\_VOL:

i += scnprintf(buf + i, PAGE\_SIZE - i, "%d\n",

sprdfgu\_read\_vbat\_vol());

break;

case FGU\_CURRENT:

i += scnprintf(buf + i, PAGE\_SIZE - i, "%d\n",

sprdfgu\_read\_batcurrent());

break;

case FGU\_CAL\_FROM\_TYPE:

i += scnprintf(buf + i, PAGE\_SIZE - i, "%d\n",

fgu\_cal.cal\_type);

break;

case CC\_TEST\_CMD:

i += scnprintf(buf + i, PAGE\_SIZE - i, "%d\n",

cc\_test.ongoing);

break;

case CC\_TEST\_RESULT:

i += scnprintf(buf + i, PAGE\_SIZE - i,

"GAUGE:%dmAh,TIME:%ds,AVG\_CUR:%dmA\n",

cc\_test.cc\_keep, cc\_test.time\_keep,

cc\_test.avg\_cur);

break;

case QMAX\_FORCE\_S\_SOC:

i += scnprintf(buf + i, PAGE\_SIZE - i, "%d\n",

sprdfgu\_data.qmax.force\_s\_soc);

break;

case QMAX\_STATE:

i += scnprintf(buf + i, PAGE\_SIZE - i, "%d\n",

sprdfgu\_data.qmax.state);

break;

case QMAX\_S\_VOL:

i += scnprintf(buf + i, PAGE\_SIZE - i, "%d\n",

sprdfgu\_data.qmax.s\_vol);

break;

case QMAX\_S\_CUR:

i += scnprintf(buf + i, PAGE\_SIZE - i, "%d\n",

sprdfgu\_data.qmax.s\_cur);

break;

case QMAX\_E\_VOL:

i += scnprintf(buf + i, PAGE\_SIZE - i, "%d\n",

sprdfgu\_data.qmax.e\_vol);

break;

case QMAX\_E\_CUR:

i += scnprintf(buf + i, PAGE\_SIZE - i, "%d\n",

sprdfgu\_data.qmax.e\_cur);

break;

case CNOM:

i += scnprintf(buf + i, PAGE\_SIZE - i, "%d\n",

sprdfgu\_data.cur\_cnom);

break;

case SAVED\_CNOM:

i += scnprintf(buf + i, PAGE\_SIZE - i, "load done\n");

SPRD\_FGU\_DEBUG("saved cnom.read..\n");

sprdfgu\_data.qmax.state = QMAX\_INIT;

queue\_delayed\_work(sprdfgu\_data.fgu\_wqueue,

&sprdfgu\_data.fgu\_qmax\_work, 0);

break;

case FULL\_VOL:

i += scnprintf(buf + i, PAGE\_SIZE - i, "%d\n",

sprdfgu\_data.bat\_full\_vol);

break;

case ALARM\_VOL:

i += scnprintf(buf + i, PAGE\_SIZE - i, "%d\n",

sprdfgu\_data.pdata->alm\_vol);

break;

case SHUTDOWN\_VOL:

i += scnprintf(buf + i, PAGE\_SIZE - i, "%d\n",

sprdfgu\_data.shutdown\_vol);

break;

case SOFT\_UVLO:

i += scnprintf(buf + i, PAGE\_SIZE - i, "%d\n",

sprdfgu\_data.pdata->soft\_vbat\_uvlo);

break;

case CUR\_RINT:

i += scnprintf(buf + i, PAGE\_SIZE - i, "%d\n",

sprdfgu\_data.cur\_rint);

break;

case RSENSE:

i += scnprintf(buf + i, PAGE\_SIZE - i, "%d\n",

sprdfgu\_data.pdata->rsense\_real);

break;

case FGU\_REPORT:

{

int cc, tm, cur;

sprdfgu\_report\_show(&cc, &tm, &cur);

i += scnprintf(buf + i, PAGE\_SIZE - i,

"GAUGE:%dmAh,TIME:%ds,AVG\_CUR:%dmA\n",

cc, tm,

cur);

}

break;

default:

i = -EINVAL;

break;

}

return i;

}

#define SPRDFGU\_ATTR(\_name) \

{ \

.attr = { .name = #\_name, .mode = S\_IRUGO | S\_IWUSR | S\_IWGRP, }, \

.show = sprdfgu\_show\_attribute, \

.store = sprdfgu\_store\_attribute, \

}

#define SPRDFGU\_ATTR\_RO(\_name) \

{ \

.attr = { .name = #\_name, .mode = S\_IRUGO, }, \

.show = sprdfgu\_show\_attribute, \

}

#define SPRDFGU\_ATTR\_WO(\_name) \

{ \

.attr = { .name = #\_name, .mode = S\_IWUSR | S\_IWGRP, }, \

.store = sprdfgu\_store\_attribute, \

}

static struct device\_attribute sprdfgu\_attribute[] = {

SPRDFGU\_ATTR\_RO(fgu\_vol\_adc),

SPRDFGU\_ATTR\_RO(fgu\_current\_adc),

SPRDFGU\_ATTR\_RO(fgu\_vol),

SPRDFGU\_ATTR\_RO(fgu\_current),

SPRDFGU\_ATTR\_WO(fgu\_log\_time),

SPRDFGU\_ATTR\_RO(fgu\_cal\_from\_type),

SPRDFGU\_ATTR(cc\_test\_cmd),

SPRDFGU\_ATTR\_RO(cc\_test\_result),

SPRDFGU\_ATTR\_WO(qmax\_force\_start),

SPRDFGU\_ATTR(qmax\_force\_s\_soc),

SPRDFGU\_ATTR\_RO(qmax\_state),

SPRDFGU\_ATTR(qmax\_s\_vol),

SPRDFGU\_ATTR(qmax\_s\_cur),

SPRDFGU\_ATTR(qmax\_e\_vol),

SPRDFGU\_ATTR(qmax\_e\_cur),

SPRDFGU\_ATTR\_RO(cnom),

SPRDFGU\_ATTR(saved\_cnom),

SPRDFGU\_ATTR(full\_vol),

SPRDFGU\_ATTR(alarm\_vol),

SPRDFGU\_ATTR(shutdown\_vol),

SPRDFGU\_ATTR(soft\_uvlo),

SPRDFGU\_ATTR(cur\_rint),

SPRDFGU\_ATTR(rsense),

SPRDFGU\_ATTR\_RO(fgu\_report),

};

static struct attribute \*sprd\_fgu\_attrs[] = {

&sprdfgu\_attribute[0].attr,

&sprdfgu\_attribute[1].attr,

&sprdfgu\_attribute[2].attr,

&sprdfgu\_attribute[3].attr,

&sprdfgu\_attribute[4].attr,

&sprdfgu\_attribute[5].attr,

&sprdfgu\_attribute[6].attr,

&sprdfgu\_attribute[7].attr,

&sprdfgu\_attribute[8].attr,

&sprdfgu\_attribute[9].attr,

&sprdfgu\_attribute[10].attr,

&sprdfgu\_attribute[11].attr,

&sprdfgu\_attribute[12].attr,

&sprdfgu\_attribute[13].attr,

&sprdfgu\_attribute[14].attr,

&sprdfgu\_attribute[15].attr,

&sprdfgu\_attribute[16].attr,

&sprdfgu\_attribute[17].attr,

&sprdfgu\_attribute[18].attr,

&sprdfgu\_attribute[19].attr,

&sprdfgu\_attribute[20].attr,

&sprdfgu\_attribute[21].attr,

&sprdfgu\_attribute[22].attr,

&sprdfgu\_attribute[23].attr,

NULL

};

static struct attribute\_group sprd\_fgu\_group = {

.name = NULL,

.attrs = sprd\_fgu\_attrs,

};

static int xxxx\_probe(struct platform\_device \*pdev)

{

int ret=0;

。

。

。

ret = sysfs\_create\_group(&pdev ->dev.kobj,&sprd\_fgu\_group);

if (ret)

pr\_err("failed to create sprd\_fgu sysfs device attributes\n");

。

。

。

}

## 4、内核中如何创建一个线程

### 平台：SC7731C、SC9832 、SC9850

函数：kthread\_run

解析：

kthread\_run是一个宏,用来创建一个进程,并且将其唤醒,其定义在头文件include/linux/kthread.h中.

#define kthread\_run(threadfn, data, namefmt, ...) \

({ \

struct task\_struct \*\_\_k \

= kthread\_create(threadfn, data, namefmt, ## \_\_VA\_ARGS\_\_); \

if (!IS\_ERR(\_\_k)) \

wake\_up\_process(\_\_k); \

\_\_k; \

})

先来看看这个宏的参数,threadfn是该线程的执行函数,data是执行函数的传入参数,namefmt是该线程的printf风格的线程名,最后的...类似printf的可变参数列表.从这个宏的实现可以看出kthread\_run是通过kthread\_create来创建一个进程,并返回一个task\_struct,然后使用wake\_up\_process函数将新创建的进程唤醒.下面需要关注一下kthread\_create的实现,其实它也是一个宏,同样也定义在头文件include/linux/kthread.h中.

更详细的解释的链接：<http://blog.csdn.net/cq062364/article/details/39646623>

## 5、队列

### (1)、工作队列

注册工作队列：

INIT\_DELAYED\_WORK(&dev->dispc\_esd\_work, dispc\_esd\_handle);  //参数：触发条件、执行函数

也可以使用宏 INIT\_WORK()

唤醒工作队列：

1. 使用两对函数INIT\_DELAYED\_WORK()对schedule\_delayed\_work(),INIT\_WORK()对schedule\_work()。使这两个函数的时候工作者线程是通过直接使用内核中每个CPU对应的一个 缺省工作者线程envents/n（n代表CPU的序号，从0开始）【扩展：如果是shudule\_work的话，系统中有个events内核线程，这个线程会处理你用shudule\_work接口激活 的所有work。如果是queue\_work的话，一般这种情况都是自己创建了一个单独的处理线程，这样将 你激活的work和这个线程联系起来。至于什么时候运行，那就是events或是你定义的特定线程运行的时】

2、如果缺省的队列不能满足要求，自己创建一个工作者线程也很简单，只需要调用：

struct workqueue\_struct \*create\_workqueue(const char \*name);

调度时使用如下函数：

int queue\_work(struct workqueue\_struct \*wq, struct work\_struct \*work);

或

int queue\_delayed\_work(struct workqueue\_struct \*wq, struct work\_struct

\*work, unsigned long delay);

queue\_delayed\_work(gsl\_monitor\_workqueue, &gsl\_monitor\_work, 1000);

取消工作队列：

cancel\_delayed\_work()

在这下面一般会使用：

flush\_scheduled\_work() // 为了防止有竞争条件的出现, 对于内核模块,如果一个模块使用了工作队列机制,并且利用了events这个缺省队列,那么在卸载这个模块之前,您必须得调用这个函数,这叫做刷新一个工作队列,也就是说,函数会一直等待,直到队列中所有对象都被执行以后才返回.当然,在等待的过程中,这个函数可以进入睡眠.反正刷新完了之后,这个函数会被唤醒,然后它就返回了.

**例子：**

1、头文件路径：include/linux/workqueue.h

2、变量定义：

static struct delayed\_work gsl\_monitor\_work;

static struct workqueue\_struct \*gsl\_monitor\_workqueue = NULL;

3、初始化：

INIT\_DELAYED\_WORK(&gsl\_monitor\_work, gsl\_monitor\_worker);

4、创建一个专用的工作者线程

gsl\_monitor\_workqueue = create\_singlethread\_workqueue("gsl\_monitor\_workqueue");

5、启动队列：

queue\_delayed\_work(gsl\_monitor\_workqueue, &gsl\_monitor\_work, 1000);

6、取消工作队列

cancel\_delayed\_work(&gsl\_monitor\_work)；

7、执行函数：

static void gsl\_monitor\_worker(void)

{

。

。

。

}

### (2)、等待队列

## 6、字符设备驱动

挂载驱动：insmod globalmem.ko

移除驱动：rmmod globalmem.ko

**（1）、查看主次设备号：**字符设备文件由使用 ls -l 的输出的第一列的“c”标识。如果使用 ls -l 命令, 会看到在设备文件项中有 2 个数(由一个逗号分隔) 这些数字就是设备文件的主次设备编号。

**（2）设备号的作用：**主设备号用来标识与设备文件相连的驱动程序。次编号被驱动用来决定操作的是哪个设备。

**（3）、关于内核设备号：**

1）、struct dev\_t 实质为unsigned int 32位整数，其中高12位为主设备号，低20位为次设备号

2）、如何从kdev\_t结构中分解出主设备号： MAJOR(dev\_t dev)

3）、如何从kdev\_t结构中分解出次设备号： MINOR(dev\_t dev)

4）、如何构造设备号： dev\_t MKDEV(unsigned int major, unsigned int minor);

**（4）、内核中如何分配设备号**

**静态分配：**

1）、根据documentxtation/devices.txt,找到一个没有使用的设备号,并手工指定.

2）、使用register\_chrdev\_region函数注册

dev = MKDEV(devmajor, devminor); // 1 获得设备号

result = register\_chrdev\_region(dev, 1, "led"); // 2 分配设备编号

缺点：设备号容易冲突。

其中：**int register\_chrdev\_region(dev\_t from, unsigned count, const char \*name)**

功能：

申请使用从 from 开始的 count 个设备号(主设备号不变，次设备号增加）

参数：

from：希望申请使用的设备号

count：希望申请使用设备号数目

name：设备名(体现在/proc/devices)

**动态分配**

方法：使用动态分配函数 int alloc\_chrdev\_region(dev\_t \*dev, unsigned baseminor, unsigned count,const char \*name) 分配设备号

功能：

请求内核动态分配 count 个设备号，且次设备号从baseminor开始。

参数：

dev：分配到的设备号

baseminor：起始次设备号

count：需要分配的设备号数目

name：设备名(体现在/proc/devices)

例如：

result = alloc\_chrdev\_region(&dev, devminor, 1, “led”);//2 动态分配设备编号

devmajor = MAJOR(dev);

优点：

简单，易于驱动推广

缺点：

无法在安装驱动前创建设备文件,因为还没有分配到主设备号。

解决办法：

一旦驱动程序安装、主设备号分配了, 可从 /proc/devices 中查询到

**（5）、注销设备号：**

1）、定义：不论使用何种方法分配设备号，都应该在不再使用它们时释放这些设备号。

2）、函数：void unregister\_chrdev\_region(dev\_t from,unsigned count)//释放从from开始的count个设备号

**（6）、设备文件**

1）、创建有两种方法

手动：

mknod filename type major minor

Filename:设备文件名

Type: 设备文件类型

Major: 主设备号

Minor: 次设备号

例： mknod /dev/serial0 c 100 0

## 7、关于request\_threaded\_irq----中断线程化

### 平台：SC7731C、SC9832 、SC9850

好文：http://www.wowotech.net/linux\_kenrel/request\_threaded\_irq.html

申请中断函数原型：

int request\_threaded\_irq(unsigned int irq, irq\_handler\_t handler, irq\_handler\_t thread\_fn,unsigned long irqflags,const char \*devname, void \*dev\_id);

函数参数解析：

1>：irq:表示申请的中断号。

2>：handler:表示中断服务例程（相当于中断处理的上半部分）

3.> thread\_fn：中断线程化，此处传递的是NULL。NULL表示没有中断线程化。

在 Linux 中，中断具有最高的优先级。不论在任何时刻，只要产生中断事件，内核将立即执行相应的中断处理程序，等到所有挂起的中断和软中断处理完毕后才能执行正常的任务，因此有可能造成实时任务得不到及时的处理。中断线程化之后，中断将作为内核线程运行而且被赋予不同的实时优先级，实时任务可以有比中断线程更高的优先级。这样，具有最高优先级的实时任务就能得到优先处理，即使在严重负载下仍有实时性保证。but,并不是所有的中断都可以被线程化，比如时钟中断，主要用来维护系统时间以及定时器等，其中定时器是操作系统的脉搏，一旦被线程化，就有可能被挂起，这样后果将不堪设想，所以不应当被线程化。 （相当于中断处理的下半部分）

4>.irqflags:表示中断标志位。

标志位：

IRQF\_TRIGGER\_RISING ：上升沿触发中断

IRQF\_TRIGGER\_FALLING ：下降沿触发中断

IRQF\_TRIGGER\_HIGH ： 高电平触发中断

IRQF\_TRIGGER\_LOW ：低电平触发中断

IRQF\_ONESHOT ：为保证中断在底半部thread\_fn执行完之后才会继续接受中断

IRQF\_NO\_SUSPEND ：这个flag比较好理解，就是说在系统suspend的时候，不用disable这个中断，如果disable，可能会导致系统不能正常的resume。

5>.devname:表示请求中断的设备的名称。

6>.dev\_id: 对应于request\_irq()函数中所传递的第五个参数，可取任意值，但必须唯一能够代表发出中断请求的设备，通常取描述该设备的结构体。 共享中断时所用。

## 8、Linux的相关命令

### 一、重启命令：

1、sudo reboot

2、sudo shutdown -r now 立刻重启(root用户使用)

3、sudo shutdown -r 10 过10分钟自动重启(root用户使用)

4、sudo shutdown -r 20:35 在时间为20:35时候重启(root用户使用)

如果是通过shutdown命令设置重启的话，可以用shutdown -c命令取消重启

### 二、关机命令：

1、halt 立刻关机

2、poweroff 立刻关机

3、shutdown -h now 立刻关机(root用户使用)

4、shutdown -h 10 10分钟后自动关机

如果是通过shutdown命令设置关机的话，可以用shutdown -c命令取消重启

### 三、查看CPU详细信息和ubuntu详情：

CPU:

cat /proc/cpuinfo 查看CPU的所有信息

cat /proc/cpuinfo| grep "physical id"| sort| uniq| wc -l 查看物理CPU个数

cat /proc/cpuinfo| grep "cpu cores"| uniq 查看每个物理CPU中core的个数

cat /proc/cpuinfo| grep "processor"| wc -l 查看逻辑CPU的个数

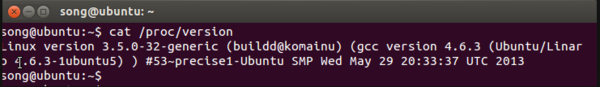
grep 'processor' /proc/cpuinfo | sort -u | wc -l 查看总线程数量

cat /proc/cpuinfo | grep name | cut -f2 -d: | uniq -c 查看CPU信息（型号）

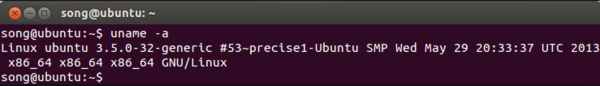
getconf LONG\_BIT 返回值可以知道当前是32位服务器还是64位服务器

Ubuntu：

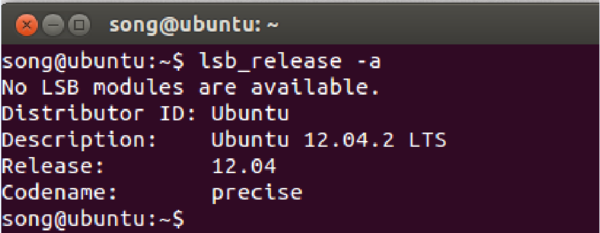
第一种：使用命令：cat /proc/version 查看



第二种：使用命令：uname -a 查看



第三种：使用命令：lsb\_release -a 查看



### 四、查看内存大小

free

### 五、查看磁盘大小

df -h

sudo uname --s 显示内核名字s

sudo uname --r 显示内核版本

sudo uname --n 显示网络主机名

sudo uname --p 显示cpu （比如32位和64位）

### 六、Linux sed 命令

Linux sed 命令是利用脚本来处理文本文件。

sed 可依照脚本的指令来处理、编辑文本文件。

Sed 主要用来自动编辑一个或多个文件、简化对文件的反复操作、编写转换程序等。

**语法**

sed [-hnVl][-e<script>][-f<script文件>][文本文件]

参数说明：

-e<script>或--expression=<script> 以选项中指定的script来处理输入的文本文件。

-f<script文件>或--file=<script文件> 以选项中指定的script文件来处理输入的文本文件。

-h或--help 显示帮助。

-n或--quiet或--silent 仅显示script处理后的结果。

-V或--version 显示版本信息。

动作说明：

a ：新增， a 的后面可以接字串，而这些字串会在新的一行出现(目前的下一行)～

c ：取代， c 的后面可以接字串，这些字串可以取代 n1,n2 之间的行！

d ：删除，因为是删除啊，所以 d 后面通常不接任何咚咚；

i ：插入， i 的后面可以接字串，而这些字串会在新的一行出现(目前的上一行)；

p ：打印，亦即将某个选择的数据印出。通常 p 会与参数 sed -n 一起运行～

s ：取代，可以直接进行取代的工作哩！通常这个 s 的动作可以搭配正规表示法！例如 1,20s/old/new/g 就是啦！

**实例**

在testfile文件的第四行后添加一行，并将结果输出到标准输出，在命令行提示符下输入如下命令：

sed -e 4a\newLine testfile

首先查看testfile中的内容如下：

$ cat testfile #查看testfile 中的内容

HELLO LINUX!

Linux is a free unix-type opterating system.

This is a linux testfile!

Linux test

使用sed命令后，输出结果如下：

$ sed -e 4a\newline testfile #使用sed 在第四行后添加新字符串

HELLO LINUX! #testfile文件原有的内容

Linux is a free unix-type opterating system.

This is a linux testfile!

Linux test

newline

以行为单位的新增/删除

将 /etc/passwd 的内容列出并且列印行号，同时，请将第 2~5 行删除！

[root@www ~]# nl /etc/passwd | sed '2,5d'

1 root:x:0:0:root:/root:/bin/bash

6 sync:x:5:0:sync:/sbin:/bin/sync

7 shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown

.....(后面省略).....

sed 的动作为 '2,5d' ，那个 d 就是删除！因为 2-5 行给他删除了，所以显示的数据就没有 2-5 行罗～ 另外，注意一下，原本应该是要下达 sed -e 才对，没有 -e 也行啦！同时也要注意的是， sed 后面接的动作，请务必以 '' 两个单引号括住喔！

只要删除第 2 行

nl /etc/passwd | sed '2d'

要删除第 3 到最后一行

nl /etc/passwd | sed '3,$d'

在第二行后(亦即是加在第三行)加上『drink tea?』字样！

[root@www ~]# nl /etc/passwd | sed '2a drink tea'

1 root:x:0:0:root:/root:/bin/bash

2 bin:x:1:1:bin:/bin:/sbin/nologin

drink tea

3 daemon:x:2:2:daemon:/sbin:/sbin/nologin

.....(后面省略).....

那如果是要在第二行前

nl /etc/passwd | sed '2i drink tea'

如果是要增加两行以上，在第二行后面加入两行字，例如 Drink tea or ..... 与 drink beer?

[root@www ~]# nl /etc/passwd | sed '2a Drink tea or ......\

> drink beer ?'

1 root:x:0:0:root:/root:/bin/bash

2 bin:x:1:1:bin:/bin:/sbin/nologin

Drink tea or ......

drink beer ?

3 daemon:x:2:2:daemon:/sbin:/sbin/nologin

.....(后面省略).....

每一行之间都必须要以反斜杠『 \ 』来进行新行的添加喔！所以，上面的例子中，我们可以发现在第一行的最后面就有 \ 存在。

以行为单位的替换与显示

将第2-5行的内容取代成为『No 2-5 number』呢？

[root@www ~]# nl /etc/passwd | sed '2,5c No 2-5 number'

1 root:x:0:0:root:/root:/bin/bash

No 2-5 number

6 sync:x:5:0:sync:/sbin:/bin/sync

.....(后面省略).....

透过这个方法我们就能够将数据整行取代了！

仅列出 /etc/passwd 文件内的第 5-7 行

[root@www ~]# nl /etc/passwd | sed -n '5,7p'

5 lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin

6 sync:x:5:0:sync:/sbin:/bin/sync

7 shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown

可以透过这个 sed 的以行为单位的显示功能， 就能够将某一个文件内的某些行号选择出来显示。

数据的搜寻并显示

搜索 /etc/passwd有root关键字的行

nl /etc/passwd | sed '/root/p'

1 root:x:0:0:root:/root:/bin/bash

1 root:x:0:0:root:/root:/bin/bash

2 daemon:x:1:1:daemon:/usr/sbin:/bin/sh

3 bin:x:2:2:bin:/bin:/bin/sh

4 sys:x:3:3:sys:/dev:/bin/sh

5 sync:x:4:65534:sync:/bin:/bin/sync

....下面忽略

如果root找到，除了输出所有行，还会输出匹配行。

使用-n的时候将只打印包含模板的行。

nl /etc/passwd | sed -n '/root/p'

1 root:x:0:0:root:/root:/bin/bash

数据的搜寻并删除

删除/etc/passwd所有包含root的行，其他行输出

nl /etc/passwd | sed '/root/d'

2 daemon:x:1:1:daemon:/usr/sbin:/bin/sh

3 bin:x:2:2:bin:/bin:/bin/sh

....下面忽略

#第一行的匹配root已经删除了

数据的搜寻并执行命令

搜索/etc/passwd,找到root对应的行，执行后面花括号中的一组命令，每个命令之间用分号分隔，这里把bash替换为blueshell，再输出这行：

nl /etc/passwd | sed -n '/root/{s/bash/blueshell/;p;q}'

1 root:x:0:0:root:/root:/bin/blueshell

最后的q是退出。

数据的搜寻并替换

除了整行的处理模式之外， sed 还可以用行为单位进行部分数据的搜寻并取代。基本上 sed 的搜寻与替代的与 vi 相当的类似！他有点像这样：

sed 's/要被取代的字串/新的字串/g'

先观察原始信息，利用 /sbin/ifconfig 查询 IP

[root@www ~]# /sbin/ifconfig eth0

eth0 Link encap:Ethernet HWaddr 00:90:CC:A6:34:84

inet addr:192.168.1.100 Bcast:192.168.1.255 Mask:255.255.255.0

inet6 addr: fe80::290:ccff:fea6:3484/64 Scope:Link

UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1

.....(以下省略).....

本机的ip是192.168.1.100。

将 IP 前面的部分予以删除

[root@www ~]# /sbin/ifconfig eth0 | grep 'inet addr' | sed 's/^.\*addr://g'

192.168.1.100 Bcast:192.168.1.255 Mask:255.255.255.0

接下来则是删除后续的部分，亦即： 192.168.1.100 Bcast:192.168.1.255 Mask:255.255.255.0

将 IP 后面的部分予以删除

[root@www ~]# /sbin/ifconfig eth0 | grep 'inet addr' | sed 's/^.\*addr://g' | sed 's/Bcast.\*$//g'

192.168.1.100

多点编辑

一条sed命令，删除/etc/passwd第三行到末尾的数据，并把bash替换为blueshell

nl /etc/passwd | sed -e '3,$d' -e 's/bash/blueshell/'

1 root:x:0:0:root:/root:/bin/blueshell

2 daemon:x:1:1:daemon:/usr/sbin:/bin/sh

-e表示多点编辑，第一个编辑命令删除/etc/passwd第三行到末尾的数据，第二条命令搜索bash替换为blueshell

直接修改文件内容(危险动作)

sed 可以直接修改文件的内容，不必使用管道命令或数据流重导向！ 不过，由於这个动作会直接修改到原始的文件，所以请你千万不要随便拿系统配置来测试！ 我们还是使用文件 regular\_express.txt 文件来测试看看吧！

regular\_express.txt 文件内容如下：

[root@www ~]# cat regular\_express.txt

runoob.

google.

taobao.

facebook.

zhihu-

weibo-

利用 sed 将 regular\_express.txt 内每一行结尾若为 . 则换成 !

[root@www ~]# sed -i 's/\.$/\!/g' regular\_express.txt

[root@www ~]# cat regular\_express.txt

runoob!

google!

taobao!

facebook!

zhihu-

weibo- :q:q

利用 sed 直接在 regular\_express.txt 最后一行加入 # This is a test:

[root@www ~]# sed -i '$a # This is a test' regular\_express.txt

[root@www ~]# cat regular\_express.txt

runoob!

google!

taobao!

facebook!

zhihu-

weibo-

# This is a test

由於 $ 代表的是最后一行，而 a 的动作是新增，因此该文件最后新增 # This is a test！

sed 的 -i 选项可以直接修改文件内容，这功能非常有帮助！举例来说，如果你有一个 100 万行的文件，你要在第 100 行加某些文字，此时使用 vim 可能会疯掉！因为文件太大了！那怎办？就利用 sed 啊！透过 sed 直接修改/取代的功能，你甚至不需要使用 vim 去修订！

### 七、Linux awk命令详解

awk是一个强大的文本分析工具，相对于grep的查找，sed的编辑，awk在其对数据分析并生成报告时，显得尤为强大。简单来说awk就是把文件逐行的读入，以空格为默认分隔符将每行切片，切开的部分再进行各种分析处理。

使用方法 ： awk '{pattern + action}' {filenames}

尽管操作可能会很复杂，但语法总是这样，其中 pattern 表示 AWK 在数据中查找的内容，而 action 是在找到匹配内容时所执行的一系列命令。花括号（{}）不需要在程序中始终出现，但它们用于根据特定的模式对一系列指令进行分组。 pattern就是要表示的正则表达式，用斜杠括起来。

awk语言的最基本功能是在文件或者字符串中基于指定规则浏览和抽取信息，awk抽取信息后，才能进行其他文本操作。完整的awk脚本通常用来格式化文本文件中的信息。通常，awk是以文件的一行为处理单位的。awk每接收文件的一行，然后执行相应的命令，来处理文本。

awk内置变量

ARGC 命令行参数个数

ARGV 命令行参数排列

ENVIRON 支持队列中系统环境变量的使用

FILENAME awk浏览的文件名

FNR 浏览文件的记录数

FS 设置输入域分隔符，等价于命令行 -F选项

NF 浏览记录的域的个数

NR 已读的记录数

OFS 输出域分隔符

ORS 输出记录分隔符

RS 控制记录分隔符

$0变量是指整条记录。$1表示当前行的第一个域,$2表示当前行的第二个域,......以此类推。

$NF是number finally,表示最后一列的信息，跟变量NF是有区别的，变量NF统计的是每行列的总数

常用的命令展示

**awk擅长列输出**

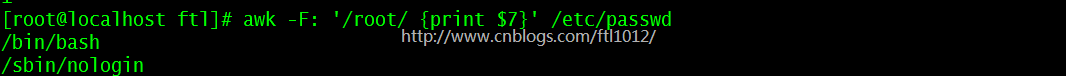
搜索/etc/passwd有root关键字的所有行

awk '/root/' /etc/passwd 【这种是pattern的使用，匹配了pattern(这里是root)的行才会执行action(没有指定action，默认输出每行的内容)】



搜索/etc/passwd有root关键字的所有行，并显示对应的shell

awk -F: '/root/ {print $7}' /etc/passwd



统计/etc/passwd:文件名，每行的行号，每行的列数，对应的完整行内容:

awk -F ':' '{print "filename:" FILENAME ",linenumber:" NR ",columns:" NF ",linecontent:"$0}' /etc/passwd



使用printf替代print,可以让代码更加简洁，易读

awk -F: '{printf ("filename:%10s, linenumber:%3s,column:%3s,content:%3f\n",FILENAME,NR,NF,$0)}' /etc/passwd



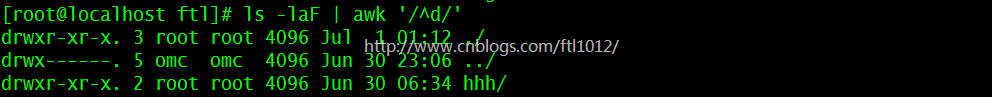
打印/etc/passwd/的第二行信息

awk -F: 'NR==2{print "filename: "FILENAME, $0}' /etc/passwd

image

awk的过滤使用方法

ls -lF | awk '/^d/'



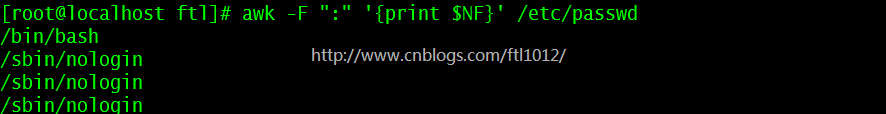
指定特定的分隔符，查询第一列

awk -F ":" '{print $1}' /etc/passwd



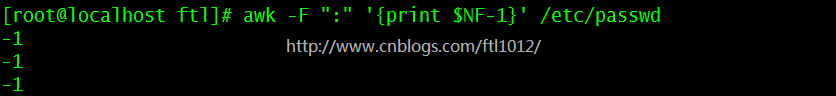
指定特定的分隔符，查询最后一列

awk -F ":" '{print $NF}' /etc/passwd



指定特定的分隔符，查询倒数第二列

awk -F ":" '{print $NF-1}' /etc/passwd



获取第12到31行的第一列的信息

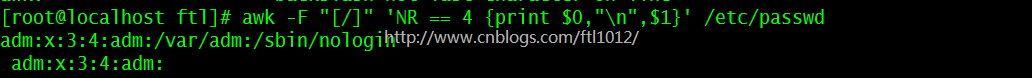
awk -F ":" '{if(NR<31 && NR >12) print $1}' /etc/passwd



多分隔符的使用：

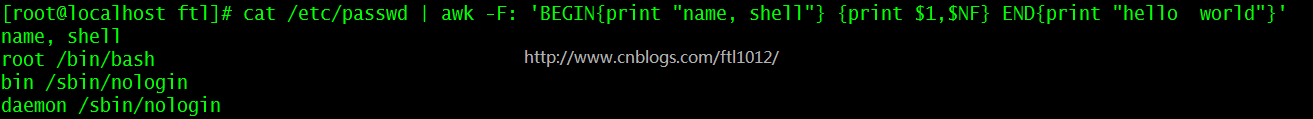
[root@localhost ftl]# awk -F "[/]" 'NR == 4 {print $0,"\n",$1}' /etc/passwd

这里以/为分隔符，多个分隔符利用[]然后在里面写分隔符即可



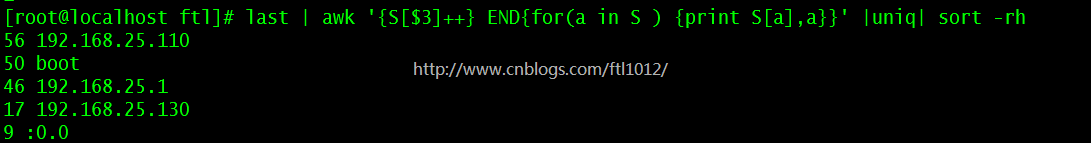
添加了BEGIN和END

[root@localhost ftl]# cat /etc/passwd | awk -F: 'BEGIN{print "name, shell"} {print $1,$NF} END{print "hello world"}'



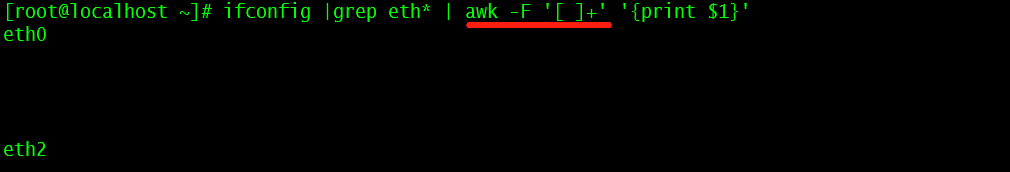
查看最近登录最多的IP信息

[root@localhost ftl]# last | awk '{S[$3]++} END{for(a in S ) {print S[a],a}}' |uniq| sort –rh



利用正则过滤多个空格

[root@localhost ~]# ifconfig |grep eth\* | awk -F '[ ]+' '{print $1}'<br><br>



**awk编程--变量和赋值**

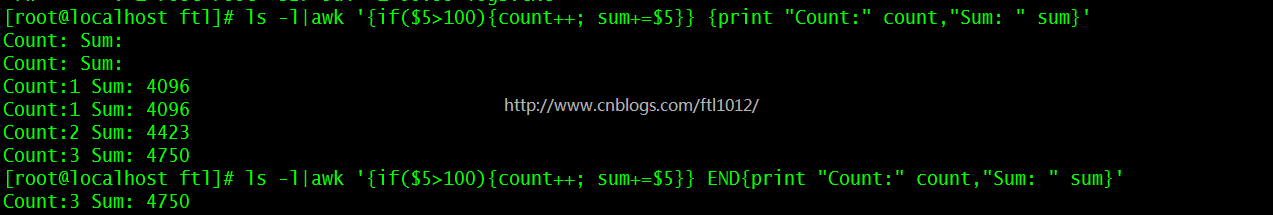
除了awk的内置变量，awk还可以自定义变量, awk中的循环语句同样借鉴于C语言，支持while、do/while、for、break、continue，这些关键字的语义和C语言中的语义完全相同。

统计某个文件夹下的大于100k文件的数量和总和

ls -l|awk '{if($5>100){count++; sum+=$5}} {print "Count:" count,"Sum: " sum}' 【因为awk会轮询统计，所以会显示整个过程】

ls -l|awk '{if($5>100){count++; sum+=$5}} END{print "Count:" count,"Sum: " sum}' 【天界END后只显示最后的结果】

<strong>备注：</strong>count是自定义变量。之前的action{}里都是只有一个print,其实print只是一个语句，而action{}可以有多个语句，以;号隔开

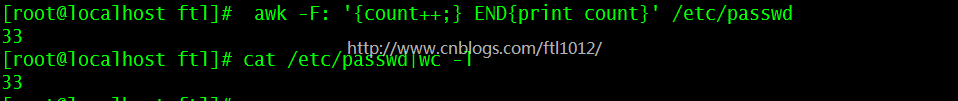


统计显示/etc/passwd的账户

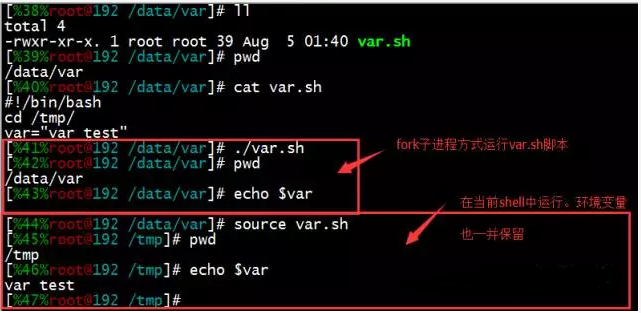
awk -F: '{count++;} END{print count}' /etc/passwd

cat /etc/passwd|wc -l

awk -F ':' 'BEGIN {count=0;} {name[count] = $1;count++;}; END{for (i = 0; i < NR; i++) print i, name[i]}' /etc/passwd



## 9、source 和 ./ 执行的区别



如图大家可以看到用./方式执行完脚本后，脚本内的所有环境变量均不影响当时的SHELL环境变量，而使用source则完全相反的效果。

1、source命令用法：

source FileName

作用:

在当前bash环境下读取并执行FileName中的命令。该filename文件可以无执行权限”

注：该命令通常用命令“.”来替代。

如：source .bash\_profile

. .bash\_profile两者等效。

source(或点)命令通常用于重新执行刚修改的初始化文档。

source命令(从 C Shell 而来)是bash shell的内置命令。

点命令，就是个点符号，(从Bourne Shell而来)。

3、./的命令用法：

./FileName

作用:

打开一个子shell来读取并执行FileName中命令，需要有执行权限。

注：运行一个shell脚本时会启动另一个命令解释器.

每个shell脚本有效地运行在父shell(parent shell)的一个子进程里. 这个父shell是指在一个控制终端或在一个xterm窗口中给你命令指示符的进程.shell脚本也可以启动他自已的子进程. 这些子shell(即子进程)使脚本并行地，有效率地地同时运行脚本内的多个子任务.

常用的shell的嵌入命令（内部命令）有如下这些，

: 空，永远返回为true

. 从当前shell中执行操作

break 退出for、while、until或case语句

cd 改变到当前目录

continue 执行循环的下一步

echo 反馈信息到标准输出

eval 读取参数，执行结果命令

exec 执行命令，但不在当前shell

exit 退出当前shell

export 导出变量，使当前shell可利用它

pwd 显示当前目录

read 从标准输入读取一行文本

readonly 使变量只读

return 退出函数并带有返回值

set 控制各种参数到标准输出的显示

shift 命令行参数向左偏移一个

test 评估条件表达式

times 显示shell运行过程的用户和系统时间

trap 当捕获信号时运行指定命令

ulimit 显示或设置shell资源

umask 显示或设置缺省文件创建模式

unset 从shell内存中删除变量或函数

wait 等待直到子进程运行完毕

直接执行enable命令也可以得到所有的内部命令列表。

## 10、LINUX 的一些宏的含义

### 1、EXPORT\_SYMBOL的作用

概念：EXPORT\_SYMBOL标签内定义的函数或者符号对全部内核代码公开，不用修改内核代码就可以在您的内核模块中直接调用，即使用EXPORT\_SYMBOL可以将一个函数以符号的方式导出给其他模块使用。

例子：

/\* LED flash control for high current capture mode\*/

ssize\_t strobe\_VDIrq(void)

{

return 0;

}

EXPORT\_SYMBOL(strobe\_VDIrq);

### 2、DIV\_ROUND\_CLOSEST函数（运算结果四舍五入）

定义：

该函数主要进行四舍五入计算使用。

例如：3/2=1.5，计算机中的整数运算结果为 3/2=1，经过DIV\_ROUND\_CLOSEST函数四舍五入运算后，3/2=2

代码(kernel.h)：

Linux内核中DIV\_ROUND\_CLOSEST函数的代码如下：

#define DIV\_ROUND\_CLOSEST(x, divisor)(                        \

{                                                                \

         typeof(divisor)\_\_divisor = divisor;            \

         (((x)+ ((\_\_divisor) / 2)) / (\_\_divisor));       \

}                                                                \

)

代码解读：

3/2我们需要进行四舍五入计算，而在计算机中3/2等于1.因为该运算为整数相除。因此如何使3/2中的余数大于被除数的一半，我们需要将结果加一。

例如：DIV\_ROUND\_CLOSEST（3,2）翻译为  (3+(2/2))/2在余数的基础上增加被除数的一半即可。(3+2/2)/2=2

## 11、自旋锁和互斥锁

**linux内核自锁旋spinlock常用宏解释**

自旋锁与互斥锁有点类似，只是自旋锁不会引起调用者睡眠，如果自旋锁已经被别的执行单元保持，调用者就一直循环在那里看是否该自旋锁的保持者已经释放了锁，"自旋"一词就是因此而得名。

　　由于自旋锁使用者一般保持锁时间非常短，因此选择自旋而不是睡眠是非常必要的，自旋锁的效率远高于互斥锁。

　　信号量和读写信号量适合于保持时间较长的情况，它们会导致调用者睡眠，因此只能在进程上下文使用（\_trylock的变种能够在中断上下文使用），而自旋锁适合于保持时间非常短的情况，它可以在任何上下文使用。

　　如果被保护的共享资源只在进程上下文访问，使用信号量保护该共享资源非常合适，如果对共巷资源的访问时间非常短，自旋锁也可以。但是如果被保护的共享资源需要在中断上下文访问（包括底半部即中断处理句柄和顶半部即软中断），就必须使用自旋锁。

　　自旋锁保持期间是抢占失效的，而信号量和读写信号量保持期间是可以被抢占的。自旋锁只有在内核可抢占或SMP的情况下才真正需要，在单CPU且不可抢占的内核下，自旋锁的所有操作都是空操作。

　　跟互斥锁一样，一个执行单元要想访问被自旋锁保护的共享资源，必须先得到锁，在访问完共享资源后，必须释放锁。如果在获取自旋锁时，没有任何执行单元保持该锁，那么将立即得到锁；如果在获取自旋锁时锁已经有保持者，那么获取锁操作将自旋在那里，直到该自旋锁的保持者释放了锁。

　　无论是互斥锁，还是自旋锁，在任何时刻，最多只能有一个保持者，也就说，在任何时刻最多只能有一个执行单元获得锁。

　　自旋锁的API有：

spin\_lock\_init(x)

　　该宏用于初始化自旋锁x。自旋锁在真正使用前必须先初始化。该宏用于动态初始化。

DEFINE\_SPINLOCK(x)

　　该宏声明一个自旋锁x并初始化它。该宏在2.6.11中第一次被定义，在先前的内核中并没有该宏。

SPIN\_LOCK\_UNLOCKED

　　该宏用于静态初始化一个自旋锁。

DEFINE\_SPINLOCK(x)等同于spinlock\_t x = SPIN\_LOCK\_UNLOCKED spin\_is\_locked(x)

　　该宏用于判断自旋锁x是否已经被某执行单元保持（即被锁），如果是，返回真，否则返回假。

spin\_unlock\_wait(x)

　　该宏用于等待自旋锁x变得没有被任何执行单元保持，如果没有任何执行单元保持该自旋锁，该宏立即返回，否则将循环在那里，直到该自旋锁被保持者释放。

spin\_trylock(lock)

　　该宏尽力获得自旋锁lock，如果能立即获得锁，它获得锁并返回真，否则不能立即获得锁，立即返回假。它不会自旋等待lock被释放。

spin\_lock(lock)

　　该宏用于获得自旋锁lock，如果能够立即获得锁，它就马上返回，否则，它将自旋在那里，直到该自旋锁的保持者释放，这时，它获得锁并返回。总之，只有它获得锁才返回。

spin\_lock\_irqsave(lock, flags)

　　该宏获得自旋锁的同时把标志寄存器的值保存到变量flags中并失效本地中断。

spin\_lock\_irq(lock)

　　该宏类似于spin\_lock\_irqsave，只是该宏不保存标志寄存器的值。

spin\_lock\_bh(lock)

　　该宏在得到自旋锁的同时失效本地软中断。

spin\_unlock(lock)

　　该宏释放自旋锁lock，它与spin\_trylock或spin\_lock配对使用。如果spin\_trylock返回假，表明没有获得自旋锁，因此不必使用spin\_unlock释放。

spin\_unlock\_irqrestore(lock, flags)

　　该宏释放自旋锁lock的同时，也恢复标志寄存器的值为变量flags保存的值。它与spin\_lock\_irqsave配对使用。

spin\_unlock\_irq(lock)

　　该宏释放自旋锁lock的同时，也使能本地中断。它与spin\_lock\_irq配对应用。

spin\_unlock\_bh(lock)

　　该宏释放自旋锁lock的同时，也使能本地的软中断。它与spin\_lock\_bh配对使用。

spin\_trylock\_irqsave(lock, flags)

　　该宏如果获得自旋锁lock，它也将保存标志寄存器的值到变量flags中，并且失效本地中断，如果没有获得锁，它什么也不做。

　　因此如果能够立即获得锁，它等同于spin\_lock\_irqsave，如果不能获得锁，它等同于spin\_trylock。如果该宏获得自旋锁lock，那需要使用spin\_unlock\_irqrestore来释放。

spin\_trylock\_irq(lock)

　　该宏类似于spin\_trylock\_irqsave，只是该宏不保存标志寄存器。如果该宏获得自旋锁lock，需要使用spin\_unlock\_irq来释放。

spin\_trylock\_bh(lock)

　　该宏如果获得了自旋锁，它也将失效本地软中断。如果得不到锁，它什么也不做。因此，如果得到了锁，它等同于spin\_lock\_bh，如果得不到锁，它等同于spin\_trylock。如果该宏得到了自旋锁，需要使用spin\_unlock\_bh来释放。

spin\_can\_lock(lock)

　　该宏用于判断自旋锁lock是否能够被锁，它实际是spin\_is\_locked取反。如果lock没有被锁，它返回真，否则，返回假。该宏在2.6.11中第一次被定义，在先前的内核中并没有该宏。

　　获得自旋锁和释放自旋锁有好几个版本，因此让读者知道在什么样的情况下使用什么版本的获得和释放锁的宏是非常必要的。

　　如果被保护的共享资源只在进程上下文访问和软中断上下文访问，那么当在进程上下文访问共享资源时，可能被软中断打断，从而可能进入软中断上下文来对被保护的共享资源访问，因此对于这种情况，对共享资源的访问必须使用spin\_lock\_bh和spin\_unlock\_bh来保护。

　　当然使用spin\_lock\_irq和spin\_unlock\_irq以及spin\_lock\_irqsave和spin\_unlock\_irqrestore也可以，它们失效了本地硬中断，失效硬中断隐式地也失效了软中断。但是使用spin\_lock\_bh和spin\_unlock\_bh是最恰当的，它比其他两个快。

　　如果被保护的共享资源只在进程上下文和tasklet或timer上下文访问，那么应该使用与上面情况相同的获得和释放锁的宏，因为tasklet和timer是用软中断实现的。

　　如果被保护的共享资源只在一个tasklet或timer上下文访问，那么不需要任何自旋锁保护，因为同一个tasklet或timer只能在一个CPU上运行，即使是在SMP环境下也是如此。实际上tasklet在调用tasklet\_schedule标记其需要被调度时已经把该tasklet绑定到当前CPU，因此同一个tasklet决不可能同时在其他CPU上运行。

　　timer也是在其被使用add\_timer添加到timer队列中时已经被帮定到当前CPU，所以同一个timer绝不可能运行在其他CPU上。当然同一个tasklet有两个实例同时运行在同一个CPU就更不可能了。

　　如果被保护的共享资源只在两个或多个tasklet或timer上下文访问，那么对共享资源的访问仅需要用spin\_lock和spin\_unlock来保护，不必使用\_bh版本，因为当tasklet或timer运行时，不可能有其他tasklet或timer在当前CPU上运行。

　如果被保护的共享资源只在一个软中断（tasklet和timer除外）上下文访问，那么这个共享资源需要用spin\_lock和spin\_unlock来保护，因为同样的软中断可以同时在不同的CPU上运行。

　　如果被保护的共享资源在两个或多个软中断上下文访问，那么这个共享资源当然更需要用spin\_lock和spin\_unlock来保护，不同的软中断能够同时在不同的CPU上运行。

　　如果被保护的共享资源在软中断（包括tasklet和timer）或进程上下文和硬中断上下文访问，那么在软中断或进程上下文访问期间，可能被硬中断打断，从而进入硬中断上下文对共享资源进行访问，因此，在进程或软中断上下文需要使用spin\_lock\_irq和spin\_unlock\_irq来保护对共享资源的访问。

　　而在中断处理句柄中使用什么版本，需依情况而定，如果只有一个中断处理句柄访问该共享资源，那么在中断处理句柄中仅需要spin\_lock和spin\_unlock来保护对共享资源的访问就可以了。

　　因为在执行中断处理句柄期间，不可能被同一CPU上的软中断或进程打断。但是如果有不同的中断处理句柄访问该共享资源，那么需要在中断处理句柄中使用spin\_lock\_irq和spin\_unlock\_irq来保护对共享资源的访问。

　　在使用spin\_lock\_irq和spin\_unlock\_irq的情况下，完全可以用spin\_lock\_irqsave和spin\_unlock\_irqrestore取代，那具体应该使用哪一个也需要依情况而定，如果可以确信在对共享资源访问前中断是使能的，那么使用spin\_lock\_irq更好一些。

　　因为它比spin\_lock\_irqsave要快一些，但是如果你不能确定是否中断使能，那么使用spin\_lock\_irqsave和spin\_unlock\_irqrestore更好，因为它将恢复访问共享资源前的中断标志而不是直接使能中断。

　　当然，有些情况下需要在访问共享资源时必须中断失效，而访问完后必须中断使能，这样的情形使用spin\_lock\_irq和spin\_unlock\_irq最好。

**互斥锁**

struct mutex ctp\_mutex; //定义

mutex\_init(&ctp\_mutex); //初始化

mutex\_lock(&ctp\_mutex); //上锁

mutex\_unlock(&ctp\_mutex); //解锁

## 12、Kconfig,Makefile 和 .config

分布在各目录下的Kconfig构成了一个分布式的内核配置数据库，每个Kconfig分别描述了所属目录源文件相关的内核配置菜单。在内核配置make menuconfig(或xconfig等)时，从Kconfig中读出配置菜单，用户配置完后保存到.config(在顶层目录下生成)中。在内核编译时，主Makefile调用这个.config，就知道了用户对内核的配置情况。Kconfig就是对应着内核的配置菜单。假如要想添加新的驱动到内核的源码中，可以通过修改Kconfig来增加对我们驱动的配置菜单，这样就有途径选择我们的驱动，假如想使这个驱动被编译，还要修改该驱动所在目录下的Makefile。

### 一、Kconfig

先来看下一个相对完整的Kconfig文件：

menuconfig MISC\_DEVICES

bool "Misc devices"

---help---

Say Y here to get to see options for device drivers from various

different categories. This option alone does not add any kernel code.

If you say N, all options in this submenu will be skipped and disabled.

if MISC\_DEVICES

config ST\_L3GD20\_GYR

tristate "L3GD20\_GYR gyroscope sensor support"

depends on I2C=y

help

If you say yes here you get support for ST's

gyroscope sensors L3GD20\_GYR.

choice

prompt "Preemption Model"

depends on SENSORS\_AFA750

default CALI\_NONE

config CALI\_NONE

bool "None"

help

Say yes here to disable calibration function for AFA750

config CALI\_POSITIVE

bool "positive calibration"

help

Say yes here when the afa750 and LCD are laid towared the same direction on your board

endchoice

config SENSORS\_LSM303D

tristate "LSM303 sensor driver"

depends on I2C=y

help

Say yes here to support the sensor

endif

1.语法：

config symbol

options

symbol是一个新的标记的菜单项，options是在这个新的菜单项下的属性和选项。

2.菜单结构：

配置文件描述了菜单选项，每行都是以一关键字开头(除了帮助信息)。下面的关键字结束一菜单选项：

- config ：config是关键字，表示一个配置选项的开始，是菜单menuconfig 的子菜单；紧跟着的TMPFS\_POSIX\_ACL是配置选项的名称，省略了前缀"CONFIG\_

- menuconfig : 表示一个菜单

- choice/endchoice

choice条目将多个类似的配置选项组合在一起，供用户单选或多选

choice

prompt "ARM system type"

default ARCH\_VERSATILE

config ARCH\_AAEC2000

.........

config ARCH\_REALVIEW

.........

endchoice

prompt "ARM system type"给出提示信息“ARM system type”，光标选中

后回车进入就可以看到多个config条目定义的配置选项

- comment ： comment条目用于定义一些帮助信息，出现在界面的第一行，如在arch/arm/Kconifg中有如下代码：

- menu/endmenu

menu条目用于生成菜单，其格式如下：

menu "Floating poing emulation"

config FPE\_NWFPE

..............

config FPE\_NWFPE\_XP

.............

endmenu

menu之后的Floating poing emulation是菜单名，menu和endmenu间有很多config条目，在配置界面中如下所示：

Floating poing emulation--->

[] FPE\_NWFPE

[] FPE\_NWFPE\_XP

- if/endif ：用作判断菜单是否存在

例如：

menuconfig INPUT\_TOUCHSCREEN

bool "Touchscreens"

help

Say Y here, and a list of supported touchscreens will be displayed.

This option doesn't affect the kernel.

If unsure, say Y.

if INPUT\_TOUCHSCREEN

- source ：内核源码目录树中每一个Kconfig都会用source引入其所有子目录下的Kconfig，从而保证了所有的Kconfig项目都被包含进menuconfig中。这个也说明了：如果你自己在linux内核中添加了一个文件夹，一定要在这个文件夹下创建一个Kconfig文件，然后在这个文件夹的上一层目录的Kconfig中source引入这个文件夹下的Kconfig文件。

2.options类型定义：

每个config菜单项都要有类型定义：bool布尔类型、 tristate三态（内建、模块、移除）、 string字符串、 hex十六进制、 integer整型。

例如：

[html] view plain copy

config CALI\_NONE

bool "None"

bool类型的只能选中或不选中，tristate类型的菜单项多了编译成内核模块的选项，如果选择编译成内核模块，则会在.config中生成一个CONFIG\_CALI\_NONEE=m的配置，如果选择内建，就是直接编译成内核影响，就会在.config中生成一个CONFIG\_CALI\_NONE=y的配置.

3.依赖型定义depends on或requires

指此菜单的出现与否依赖于另一个定义

config SENSORS\_LSM303D

tristate "LSM303 sensor driver"

depends on I2C=y

这个例子表明SENSORS\_LSM303D这个菜单项只I2C有效。

4.select与depends on是相反的逻辑关系。

A depends on B

那么只有在B选中才能选A

A select B

那么只要选中A就会选中B

5.帮助性定义

只是增加帮助用关键字help或者---help---，"---help---" 和 "help" 在实现的作用上没有区别，"---help---" 有助于将文件中的配置逻辑与给开发人员的提示分开。

6.prompt --输入提示

### 二、Makefile

1.顶层的Makefile文档读取 .config文档的内容，并总体上负责build内核和模块。

2.Arch Makefile则提供补充体系结构相关的信息。

3.scripts目录下的Makefile文档包含了任何用来根据kbuild Makefile 构建内核所需的定义和规则。

其中.config的内容是在make menuconfig的时候，通过Kconfig文档配置的结果，在/Documentation/kbuild目录下有详细的介绍有关kernel makefile的知识。

**举个例子：**

假设想把G-sensor LSM303D驱动code加载到工程中，配置内核时该怎么办呢？

1：将您写的lsm303d.c 文档添加到/driver/misc/ 目录下。

2：修改/driver/misc/ 目录下的kconfig文档：

[html] view plain copy

config SENSORS\_LSM303D

tristate "LSM303 sensor driver"

depends on I2C=y

help

Say yes here to support the sensor

3：修改该目录下makefile文档。

添加code：

[html] view plain copy

obj-$(CONFIG\_SENSORS\_LSM303D) += lsm303d.o

从上述分析知道CONFIG\_SENSORS\_LSM303D 是从.config 中读出的。

4.配置kernel下configs/XXXX\_defconfig文件

添加code：

[html] view plain copy

CONFIG\_SENSORS\_LSM303D=y

当您编译内核时，将会读取.config文档，当发现CONFIG\_SENSORS\_LSM303D=y，系统在调用/driver/misc下的makefile 时，将会把 lsm303d.o 加入到内核中。即可达到您的目的。

Foreach :

Makefile中的foreach函数几乎是仿照于Unix标准Shell （/bin/sh）中的for语句，或是C-Shell（/bin/csh）中的foreach语句而构建的。它的语法是：

$(foreach <var>,<list>,<text>)

这个函数的意思是，把参数<list>;中的单词逐一取出放到参数<var>;所指定的变量中，然后再执行< text>;所包含的表达式。每一次<text>;会返回一个字符串，循环过程中，<text>;的所返回的每个字符串会以空格分隔，最后当整个循环结束时，<text>;所返回的每个字符串所组成的整个字符串（以空格分隔）将会是foreach函数的返回值。

所以，<var>;最好是一个变量名，<list>;可以是一个表达式，而<text>;中一般会使用<var>;这个参数来依次枚举<list>;中的单词。举个例子：

names := a b c d

files := $(foreach n,$(names),$(n).o)

上面的例子中，$(name)中的单词会被挨个取出，并存到变量“n”中，“$(n).o”每次根据“$(n)”计算出一个值，这些值以空格分隔，最后作为foreach函数的返回，所以，$(files)的值是“a.o b.o c.o d.o”。

注意，foreach中的<var>;参数是一个临时的局部变量，foreach函数执行完后，参数<var>;的变量将不在作用，其作用域只在foreach函数当中。

**一、Makefile基本语法**

**1、Makefile基本格式：**

make所看到的第一项规则会被当做默认规则使用。

一个规则可分成三个部分：

目标：依赖1 依赖2。。。

命令

注意：命令前必须使用有一个制表符(<TAB>)。

例：

foo.o: foo.c foo.h

gcc -c foo.c

当要求make执行一条规则时，它首先找到目标和前提条件指出的文件。如果没有任何前提条件都有一条关联规则，那么make将试图先更新性前提条件。接着，更性目标文件。

如果任何前提条件都比目标要新，目标将通过执行命令得以更新。

每条命令都传给shell并在它的子shell中执行。如果任何一条命令产生错误，目标的创建将被终止，随后make退出。

如果一个文件最近被修改，那就认为它比其他的文件新。

GNU make 规则：

target ... : prerequisites ...

command

....

....

target — 目标文件， 可以是Object File 也可以是可执行文件，还可也是标签Label（标签内容在“伪目标”章节）；

prerequisites—生成target所需的文件或目标；

command—make需要执行的命令，可以是任何shell命令。

**2、一个简单的例子**

创建一个名为count\_word.c的文件，代码如下

[cpp] view plain copy

#include <stdio.h>

extern int fee\_count, fie\_count, foe\_count, fum\_count;

extern int yylex( void );

int main( int argc, char \*\* argv ){

yylex( );

printf( "%d %d %d %d\n", fee\_count, fie\_count, foe\_count, fum\_count );

return( 0 );

}

另外创建一个lexer.l文件，其中所有的空白均为tab键

int fee\_count = 0;

int fie\_count = 0;

int foe\_count = 0;

int fum\_count = 0;

%%

fee fee\_count++;

fie fie\_count++;

foe foe\_count++;

fum fum\_count++;

最后创建makefile文件，内容为：

[cpp] view plain copy

count\_words: count\_words.o lexer.o -lfl

gcc count\_words.o lexer.o -lfl -o count\_words

count\_words.o: count\_words.c

gcc -c count\_words.c

lexer.o: lexer.c

gcc -c lexer.c

lexer.c: lexer.l

flex -t lexer.l > lexer.c

clean:

rm lexer.c lexer.o count\_words.o count\_words

以上内容保存在Makefile或者是makefile都可以，直接输入make命令就可以生成可执行文件count\_words了；

gcc -c count\_words.c

flex -t lexer.l > lexer.c

gcc -c lexer.c

gcc count\_words.o lexer.o -lfl -o count\_words

如果要删除执行文件和中间的目标文件，那么就执行一下make clean。

注意1： 当依赖关系定好后，下面一行就是如何生成目标文件的操作系统命令了，一定要以一个Tab键开头。 另外，make会比较targets文件和prerequisites文件的修改日期，如果prerequisites文件的日期比targets文件新，或者targets不存在，那么make就会执行这下面一行的系统命令。

注意2： clean不是一个文件，它是一个动作名，冒号后面什么都没有，make就不会自动去找它的依赖性，也不会执行它后面的系统命令。因此，要执行clean就需要显式的指出make clean。

注意3： 如果报错，可能需要先安装flex：

sudo apt-get install flex

注意4： 运行count\_works后，它会回显你的输出并统计'fee','fie','foe','fum'的次数。结束统计需要按Ctrl+d，然后会输出四个单词出现的次数。

**3、make 如何工作**

直接输入make，则

（1）make会在当前的目录下找到名为“Makefile”或者“makefile”的文件。

（2）如果找到，它会把文件中第一个target作为最终的目标文件（如上面例子中的count\_words）。

1）首先，make会检查目标count\_words的prerequisite文件count\_words.o, lexer.o 和 -lfl。

2）count\_words.o通过编译count\_words.c生成

3）lexer.o通过编译lexer.c 生成，但是lexer.c 并不存在，因此会继续寻找lexer.c的生成方式，并找到了通过flex程序将lexer.l生成为lexer.c。

4）最后，make会检查-lfl，-l是gcc的一个命令选项，表示将系统库链接到程序。而"fl"对应的是libfl.a的库。（GNU make 可以识别这样的命令，当一个prerequisite是以这种-l<name>的形式表示出来的时候，make会自己搜索lib<name>.so的库文件，如果没找到则继续搜索lib<name>.a的库文件）。这里make找到的是/usr/lib/libfl.a文件，并将它与程序进行连接。

（3）如果count\_words文件不存在，或者count\_words所依赖的后面的.o文件的修改时间比count\_words本身更加新，那么，它会执行后面定义的命令来生成这个count\_words文件。如果count\_words所依赖的.o文件也不存在，那么make会继续按照前面的方式生成.o文件。

（4）找到相应的.c和.h，用来生成.o，然后再用.o完成make的最终任务。

关于依赖关系

make会一层一层的去找文件的依赖关系，最终编译出第一个目标文件。

关于重新编译

只要任何prerequisite 比 target新，那么这个目标文件就会被下面的命令重新生成。每一个命令都会被传递到shell中，并在自己的子shell里面执行。

关于错误

如果在寻找过程中出现错误，如文件找不到，则make会直接退出并报错。对于所定义的命令错误或者编译不成功，make是不会理会的，它只负责文件的依赖性。

**4、变量使用**

上面的例子可以看到，文件或者目标的名字几乎都毫无例外的出现了至少两次，甚至如果算上clean的内容，有些文件名出现了三次。然而，在一个大型的工程中这种情况会更加复杂，任何不经意的错误都会导致编译失败。为了让makefile更容易维护，在makefile中我们可以使用变量，或者更确切的说是一个字符串，类似c语言中的宏。例如：

[cpp] view plain copy

CC = gcc

object = lexer.o count\_words.o

count\_words: $(object) -lfl

$(CC) $(object) -lfl -o count\_words

count\_words.o: count\_words.c

$(CC) -c count\_words.c

lexer.o: lexer.c

$(CC) -c lexer.c

lexer.c: lexer.l

flex -t lexer.l > lexer.c

clean:

rm lexer.c $(object) count\_words

**5、自动推导依赖关系**

GNU make可以根据.o文件的文件名自动推导出同名的.c文件并加入依赖关系，不需要我们手动注明。并且gcc -c也会被自动推导出来，于是我们的makefile就变成了

[cpp] view plain copy

CC = gcc

object = lexer.o count\_words.o

count\_words: $(object) -lfl

$(CC) $(object) -lfl -o count\_words

count\_words.o:

lexer.o:

lexer.c: lexer.l

flex -t lexer.l > lexer.c

clean:

rm lexer.c $(object) count\_words

这种方法也叫“隐式规则”。

**6、关于Clean**

一个好习惯是每个makefile都要写clean规则，这样不仅可以方便重编译，也有利于保持文件路径的清洁。一般的风格是：

[cpp] view plain copy

clean:

rm lexer.c $(object) count\_words

但是，更为稳妥的做法是：

[cpp] view plain copy

.PHONY: clean

clean:

-rm lexer.c $(object) count\_words

.PHONY表示clean是一个“伪目标”，而rm命令前面的减号则表示，不管出现什么问题都要继续做后面的事情。

注意：clean规则不要放在makefile的开头，不然就会变成make的默认目标了。

**二、显式规则（Explicit Rules）**

通常在写makefile时使用的都是显式规则，这需要指明target和prerequisite文件。一条规则可以包含多个target，这意味着其中每个target的prerequisite都是相同的。当其中的一个target被修改后，整个规则中的其他target文件都会被重新编译或执行。

**1、通配符（Wildcards）**

make支持的通配符与Bourne shell基本相同，包括~, \*, ?, [...], [^...]。

\*.\* 表示了所有文件；

? 表示任意单个字符；

[...] 表示一个字符类；

[^...] 表示相反的字符类。

~ 表示当前用户的/home路径，~加用户名可以表示该用户的/home路径。

注意： make会在读取makefile的时候就自动补充好通配符替代的内容，而shell则是在执行命令的时候才会进行通配符替代，在某些复杂情况，这两种方式会有极大的区别。

**2、伪目标（Phony Targets）**

多数情况下makefile中的target目标文件是像前面提到的那样带有指定的prerequisite文件，但也有一些target仅仅是作为一个标签，代表了一条命令，这种不代表任何文件的目标就被称为伪目标。常见的伪目标例如在makefile开头部分的第一个目标 all， 以及前面例子中见到的 clean：

clean:

rm -f \*.o lexer.c

但是，make本身是无法区别出目标文件和伪目标的，如果碰巧在编译路径下有一个与伪目标同名的文件存在，那么make会在依赖关系图中把这个文件与伪目标名相关联。而再运行make clean 命令则会因为clean文件存在且如果没有被更新过，则makefile中的clean对应的命令将不会被执行。

为了避免这种情况，GNU make提供了一种特殊目标： “.PHONY”，用来表示目标文件不是真正的文件，即伪目标。clean命令可以被写作：

.PHONY: clean

clean:

rm -f \*.o lexer.c

这样，即使再有名为clean的文件存在，make也会执行clean后面的命令。

通常不会将一个伪目标的prerequisite设置为真是存在的文件，因为.PHONY会让他后面的文件在每次make时都进行重新编译。伪目标可以被认为是内嵌在makefile中的shell脚本。

优势： 通过使用伪目标在编译过程中进行屏幕输出，可以使make的可读性增加。例如：

$(Program): build\_msg $(OBJECTS) $(BUILTINS\_DEP) $(LIBDEP)

$(RM) $@

$(CC) $(LDFLAGS) -o $(Program) $(OBJECTS) $(LIBS)

ls -l $(Program)

size $(Program)

.PHONY: build\_msg

build\_msg:

@printf "#\n# Building $(Program)\n#\n"

这里将printf命令作为伪目标，可以使make在更新任何prerequisite之前就将指定的编译信息输出。

同时，伪目标也可也作为makefile的默认目标，放在文件的最前端，由于伪目标的特性，他指出的所有prerequisite都会被重新编译。这样可以用来同时生成多个目标。另外，与普通目标文件一样，伪目标也可也使用依赖关系，例如：

.PHONY: clean cleano cleanc

clean: cleano cleanc

-rm $(program)

cleano:

-rm \*.o

cleanc:

-rm lexer.c

这样就可以对不同类型的文件进行单独删除。

**3、空目标（Empty Targets）**

空目标是伪目标的一种变形形式，通常情况下通过创建一个空文件来实现。例如：

size: count\_words.o

size $^

touch size

这样，空文件size就被make当作时间戳，只有当count\_words.o被更新时，size里面的命令才会再次被执行。另外，所有加载了size作为prerequisite的目标，都不会因为size被编译而强制编译，他们的其他prerequisite目标被更新。

**三、变量**

变量最简单的形式就是：

$(variable\_name)

变量可以包含几乎所有的字符包括标点符号。一般情况下，变量名需要被$( )所包裹，但是当变量名只有一个字符时，括号可以省略。makefile可以定义很多变量，但同时make本身也定义了一些自动变量。

自动变量

自动变量是make自动根据规则生成的，不需要用户显式的指出相应的文件或目标名称。以下就是七个最核心的自动变量：

$@ 目标文件的文件名；

$% 仅当目标文件为归档成员文件（.lib 或者 .a）时，显示文件名，否则为空；

$< 依赖（prerequisite）列表里面的第一个文件名；

$? 所有在prerequisite列表里面比当前目标新的文件名，用空格隔开；

$^ 所有在prerequisite列表中的文件，用空格隔开； 如果有重复的文件名（包含扩展名），会自动去除重复；

$+ 与$^相似，也是prerequisite列表中的文件名用空格隔开，不同的是这里包含了所有重复的文件名；

$\* 显示目标文件的主干文件名，不包含后缀部分。

此外，上面的每个变量都带有两个不同的变种，用于适应不同种类的make。分别是在后面附加一个“D”或者“F”。例如，$(^D)就是代表所有依赖文件的路径，$(<F)表示依赖文件第一个的文件部分的值。使用上述内容前面的makefile可以重写为：

[plain] view plain copy

CC = gcc

object = lexer.o count\_words.o

program = count\_words

$(program): size $(object) -lfl

$(CC) $(object) -lfl -o $@

count\_words.o: count\_words.c

$(CC) -c $^

lexer.o: lexer.c

$(CC) -c $^

lexer.c: lexer.l

flex -t $^ > $@

size: count\_words.o

size $^

touch size

.PHONY: clean cleano cleanc

clean: cleano cleanc

-rm $(program)

cleano:

-rm \*.o

cleanc:

-rm lexer.c

**四、查找文件（VPATH）**

上一篇所使用的例子中，makefile和源文件都是在同一个简单目录下，但真正的程序往往会复杂很多。让我们重新修改整个程序，添加一个叫做counter的函数，同时添加counter.c:

#include <lexer.h>

#include <counter.h>

void counter( int counts[4]) {

while ( yylex() )

;

counts[0] = fee\_count;

counts[1] = fie\_count;

counts[2] = foe\_count;

counts[3] = fum\_count;

}

为了使这个库函数具有复用性，再添加一个counter.h作为头文件声明：

#ifndef COUNTER\_H\_

#define COUNTER\_H\_

extern void counter( int counts[4]);

#endif

同样的，也可以为lexer.l创建一个lexer.h的头文件：

#ifndef LEXER\_H\_

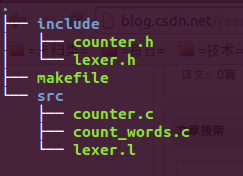
#define LEXER\_H\_

extern int fee\_count, fie\_count, foe\_count, fum\_count;

extern int yylex( void );

#endif

如果将这些文件都放在根目录下，显然比较混乱。通常情况下，头文件会放到include/下，源文件被放到src/。最后，将makefile放在根目录下，整个文件系统如下所示



为了让make能够找到相应的位置，需要在makefile开头添加VPATH参数，显式的指出源文件和头文件的路径：

VPATH = src include

此外，不仅make需要知道路径，gcc同样需要，通过添加编译选项 -I 的方式，显式的告诉gcc头文件的位置：

CPPFLAGS = -I include

最终，makefile为：

VPATH=src include

CC = gcc

CPPFLAGS = -I include

count\_words: count\_words.o counter.o lexer.o -lfl

$(CC) $^ -o $@

count\_words.o: count\_words.c counter.h

$(CC) $(CPPFLAGS) -c $<

counter.o: counter.c counter.h lexer.h

$(CC) $(CPPFLAGS) -c $<

lexer.o: lexer.c include/lexer.h

$(CC) $(CPPFLAGS) -c $<

lexer.c: lexer.l

flex -t $< > $@

.PHONY: clean

clean:

rm \*.o lexer.c count\_words

运行make的结果为：

gcc -I include -c src/count\_words.c;

gcc -I include -c src/counter.c

flex -t src/lexer.l > lexer.c

gcc -I include -c lexer.c

gcc count\_words.o counter.o lexer.o /usr/lib/x86\_64-linux-gnu/libfl.so -o count\_words

注意1： 如果是因为make的相关路径配置错误，终端会输出例如：

make: \*\*\* No rule to make target `count\_words.c', needed by `count\_words.o'. Stop.

但如果是因为gcc的头文件路径配置错误，在终端会提示，例如：

src/counter.c:1:19: fatal error: lexer.h: No such file or directory

compilation terminated.

注意2： 在UNIX系统中，路径列表可以被空格或者冒号分隔开，在Windows中则是用空格或者分号。（既然两种系统都用空格，那最好就使用空格）

注意3： make会在每次需要文件的时候搜索VPATH列表中的路径，如果有两个不同路径下文件重名，则make只会使用顺序查找到的第一个。

更加准确的方式是使用 vpath 变量，它的语法是：

vpath pattern directory-list

因此，上面makefile中的VPATH可以写做：

vpath %.c src

vpath %.l src

vpath %.h include

这样就告诉了make去src/中寻找.c和.l文件，去include中寻找.h文件。

**五、模式匹配规则**

通常情况下，编译器会将带有它可以识别后缀名的文件编译成相应的目标文件。例如，C语言的编译器会将.c后缀名的文件编译成带有.o后缀名的目标文件。再比如，前面的用到过的flex使用.l后缀名文件作为输入，输出则是.c的文件。事实上，这样一些约定可以根据文件名模式，通过内建规则来进行处理。例如，用内建规则，之前的makefile可以简写做：

VPATH=src include

CC = gcc

CPPFLAGS = -I include

count\_words: counter.o lexer.o -lfl

count\_words.o: counter.h

counter.o: counter.h lexer.h

lexer.o: lexer.h

.PHONY: clean

clean:

rm \*.o lexer.c count\_words

所有的内建规则都是模式匹配规则的实例，这个makefile之所以可以使用，是因为三个内建规则

规则一： 从.c到.o

%.o: %.c

$(COMPILE.c) $(OUTPUT\_OPTION) $<

规则二： 从.l 到.c

%.c: %.l

@$(RM) $@

$(LEX.l) $< > $@

规则三： 从.c到无后缀名

当生成目标没有后缀名的时候（通常是可执行文件）

%： %.c

$(LINK.c) $^ $(LOADLIBES) $(LDLIBS) -o $@

依照上述的模式匹配规则，make的生成过程如下：

gcc -I include -c -o count\_words.o src/count\_words.c

gcc -I include -c -o counter.o src/counter.c

lex -t src/lexer.l > lexer.c

gcc -I include -c -o lexer.o lexer.c

gcc count\_words.o counter.o lexer.o /usr/lib/x86\_64-linux-gnu/libfl.so -o count\_words

rm lexer.c

STEP 1： make根据makefile中的内容，将默认目标设置为count\_words（如果命令行中特别指出，则为其它，如clean）。根据依赖关系，分别是count\_words.o（虽然没有在makefile显式的指出，但make会根据隐式规则自动填充）， counter.o， lexer.o 和 -lfl。

STEP 2：根据依赖关系列表中的顺序，make会先找到count\_words.o，由于count\_words.o的依赖关系没有后续更新，因此make只需要找到count\_word.c并进行编译。在当前目录下，没有count\_word.c的情况下，make会根据VPATH变量继续寻找，直到在src/中找到。接下来，counter.o的编译过程也是一样的。

STEP3： 编译lexer.o的过程比前面多了一步：因为工程中并不存在lexer.c，于是make发现了从lexer.l生成lexer.c的模式匹配规则。

STEP4： make检查-lfl库的具体位置，本人用的是Ubuntu12.04 64bit， 因此对应的路径为: /usr/lib/x86\_64-Linux-gnu/libfl.so，这个路径跟操作系统和make的版本有关，其实它具体在哪都不影响make的编译(只要是make可以找到的地方)。

STEP5： make已经准备好了生成count\_words所需的所有依赖文件，生成。

STEP6：注意到，make创建的lexer.c是一个中间文件，makefile中并没有要生成它，因此在编译完成后将它删除。

DONE！

事实上，每一个makefile都有一个专有的内置规则库，在相应目录下可以使用下面的命令查看这个库（注意内容偏多，可以用more来分开看，或者重定向输出到文件）

make --print-data-base

模式匹配

模式匹配规则中使用的百分号“%”与UNIX shell里面的通配符 “\*”非常类似，它也可以代表任何长度的字符，并能被放在模式匹配中的任何位置，但在一个模式匹配中只能出现一次。

下面这些例子都是合法的模式匹配：

%,v

s%.o

wrapper\_%

当然，模式匹配中只包含一个百分号也是允许的，这种方式最常用的例子就是在UNIX下创建可执行文件。

静态模式规则

静态模式规则是只对一系列特定的目标生效的规则。

$(OBJECT): %.o: %.c

$(CC) -c $(CFLAGS) $< -o $@

与普通的模式匹配规则的唯一区别是：初始化中的“$(OBJECT):” ，这限定了文件列表。在$(OBJECT)中的每一个目标文件，会匹配到%.o，然后再通过%.c产生依赖关系。如果，目标的匹配不存在，则make会提示一个warning。

静态模式规则可以显式的指出匹配列表，而不用仅仅指出后缀等匹配模式。

后缀规则

后缀规则是一种定义隐式规则的传统方式，因为即便其他版本的make不会识别GNU make的一些模式规则语法，但后缀规则却依然会出现在其他的makefile中。因此，尽管GNU make是一个不错的选择，但我们还是应该掌握可以适应其他编译环境的makefile编写和理解方式。

后缀规则包含一到两个连接起来的后缀作为目标文件:

.c.o:

$(CC) $(OUTPUT\_OPTION) $<

注意：这里跟前面有一点不一样，首先写的.c实际上是依赖关系，.o才是目标文件。用前面的方式重写这段：

%.o: %.c

$(CC) $(OUTPUT\_OPTION) $<

只有当目标和依赖关系的两个文件后缀都在make的已知后缀列表中存在的时候，后缀规则才会生效。上面的后缀规则又叫双后缀规则，顾名思义它包含了两种后缀。另一种后缀规则是单后缀规则，它只包含了一个属于源文件的后缀，当然这个规则是用来创建可执行文件的.

定义后缀规则

后缀规则的定义就像一个特别的目标文件一样：

.SUFFIXES: .out .a .ln .o .c .cc .C .cpp .p .f .F .r .y .l

当然，你也可以自己定义其他的后缀规则，如pdf，html，xml等。要删除所有这些定义的后缀也很简单

.SUFFIXES:

也可以使用命令行参数： --no-builtin-rules或者-r。

**六、隐含规则数据库**

GNU make 3.80拥有90多个内建隐含规则。隐含规则即是模式匹配规则又是后缀规则。这些规则支持的语言有很多： C++, Pascal, FORTRAN, ratfor, Modula, Texinfo, TEX (包括Tangle 和 Weave), Emacs Lisp, RCS, SCCS等。但如果你想要编译JAVA或者XML，你可以自己编写规则。（别担心，事实上它们非常简单）

你可以通过--print-data-base或者-p参数来查看make的内建规则数据库（小心，输出有n多行）。

使用隐含规则

当处理一个目标文件没有发现显式规则时，make就会调用隐含规则。其实，只要不在makefile中目标文件的命令行程序部分添加任何内容，就可以调用隐含规则。

这种方式通常很好，在极特殊的情况下会导致一些问题。例如，在混编过程中使用Lisp和C两种语言，同一个路径下分别有editor.l和editor.c两个文件，使用make隐含规则编译的时候，make有可能将editor.l认做flex的文件，并将它编译成editor.c（正如前面（上）部分的例子）。于是，真正的editor.c就会被覆盖掉。要想避免这个问题，就需要将flex编译相关的两个内建规则删掉：

%.o: %.l

%.c: %.l

这样的模式规则不带有任何的命令，就可以将他们从make的数据库删除。尽管在实际操作中，这种规则导致的错误非常罕见，但是知道有这样一种情况总是会在不经意的时候对你有所帮助。

make的另一个强大之处在于，对于每一个符合模式匹配的目标文件，make会为它寻找相应的依附条件。如果找到了符合依附条件模式的源文件，这条规则才会生效。但当找不到时，make会再次查找所有的规则，并假设符合依附关系的源文件是另外的一个需要被生成的目标文件。这样，make会递归式的找到一个规则链用以更新目标文件（就像前面的例子一样，make可以根据规则链从lexer.l生成到lexer.o）。

例如一个名为a.o的文件的源文件可能是.c,.cpp,.cc,.p,.f,.r,.s,.mod等等。

规则结构

为了方便用户自定义，内建规则库都有标准的结构。以从C程序生成目标文件的规则为例：

%.o: %.c

$(COMPILE.c) $(OUTPUT\_OPTION) $<

用户自定义的部分完全取决于变量的使用，事实上这两个变量也是由其他多个变量和参数决定的：

COMPILE.c = $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET\_ARCH) -c

CC = gcc

OUTPUT\_OPTION = -o $@

需要注意的是，在makefile中设置参数时需要避免将这些变量赋值，如果在makefile中设置：

CPPFLAGS = -I include/

那么，当需要在生成过程中加入命令行参数

make CPPFLAGS=-DDEBUG

则-I选项和它的参数就会被取消掉。因为在命令行里面的变量将重写其他所有对变量的设置。因此，这样的设置将最终导致make找不到头文件的位置，而造成编译失败。

帮助命令

大型的makefile会包含大量的目标文件，并且非常不容易被记住，一个简单的解决方式就是为默认的目标文件设置帮助命令，然后手工方式维护这些命令又是相当复杂和繁琐的。因此，make的规则数据库提供了命令用于直接使用，下面的例子是使用了这些命令按顺序输出了所有目标列表（每行四个）：

.PHONY: help

help:

make --print-data-base --question | \

awk '/^[^.%][-A-Za-z0-9\_]\*:/ \

{ print substr(

1)-1) }' | \

sort | \

pr --omit-pagination --width=80 --columns=4

执行make help后就会在屏幕上看到所有的目标文件。

简单解释一下这个命令： 首先，使用--print-data-base查找出规则数据库的内容；然后使用awk命令从内容中抓取到目标文件信息，去掉以百分号和点号开头的文件（模式匹配规则和后缀规则文件），并删掉这一行多余的内容；最后将列表排序并按四个一行输出到屏幕。

**七、特殊目标文件**

特殊目标文件是一种改变make默认方式的内建伪目标。例如，.PHONY会声明一个文件不会依赖任何其他真实的文件，并且永远都需要更新。伪文件.PHONY是最常见的特殊目标文件，但是还有些其他特殊文件。特殊文件也遵循着target: prerequisite的语法规则，但目标文件并不是一个文件，他们更像是修改make内部算法的指令。

特殊文件共有十二个，分为三类：一类是为了改变make在更新目标时的动作；还有一类是作为全局标志的形式，编译或忽略他们的目标文件；最后一类是后缀名特殊目标，当指明了旧的后缀规则时使用。

最常用的目标修饰符有：

.INTERMEDIATE

这个特殊目标文件的依赖关系被视为中间文件，当make更新其他文件时创建了列表中的文，make会在结束时删除这些文件；但如果更新前这个文件已经存在，则make不会删除它。

.SECONDARY

依赖列表中的文件会被当作中间文件，但不会被自动删除。这个特殊目标最常见的地方是针对一些库文件，为了方便调试过程，开发期间使用的库文件尽管也是中间文件，但保留着它可以减少调试中的重复编译过程。

.PRECIOUS

当make在执行过程中被中断时，它会将所有这次更新过的目标文件删除。因此，make不会将半成品文件遗留在编译路径中。但是，当某些生成的文件相当大或者运算非常费时的结果。因此，如果将这类文件定义为PRECIOUS，则它们就不会在中断时被删除掉了。尽管.PRECIOUS不太常见，但是它经常会在需要的时候起到意想不到的效果。注意：make不会在发生错误时自动删除文件，只有当它被信号中断时才会。

.DELETE\_ON\_ERROR

这个正好和.PRECIOUS相反，它会使依赖关系列表中的文件在发生错误时被删除。

**八、自动生成依赖关系**

在通常情况下，手动添加目标文件和头文件之间的依赖关系几乎是不可能完成的。以C语言中最常见的stdio.h的头文件为例，它包含了15个其他的头文件，因此一一添加这些头文件以及它们互相之间的依赖关系必须依赖程序来实现。好在gcc提供了这样的一种方式。首先创建一个stdio.c的文件，包含了stdio.h的头文件声明：

echo “#include <stdio.h>" > stdio.c

然后，运行gcc的编译命令：

gcc -M stdio.c

屏幕会输出相关的头文件的路径：

stdio.o: stdio.c /usr/include/stdio.h /usr/include/features.h \

/usr/include/x86\_64-linux-gnu/bits/predefs.h \

/usr/include/x86\_64-linux-gnu/sys/cdefs.h \

/usr/include/x86\_64-linux-gnu/bits/wordsize.h \

/usr/include/x86\_64-linux-gnu/gnu/stubs.h \

/usr/include/x86\_64-linux-gnu/gnu/stubs-64.h \

/usr/lib/gcc/x86\_64-linux-gnu/4.6/include/stddef.h \

/usr/include/x86\_64-linux-gnu/bits/types.h \

/usr/include/x86\_64-linux-gnu/bits/typesizes.h /usr/include/libio.h \

/usr/include/\_G\_config.h /usr/include/wchar.h \

/usr/lib/gcc/x86\_64-linux-gnu/4.6/include/stdarg.h \

/usr/include/x86\_64-linux-gnu/bits/stdio\_lim.h \

/usr/include/x86\_64-linux-gnu/bits/sys\_errlist.h

这样就可以将这些需要的路径复制粘贴到makefile中了。但是，这种方法有点笨，对吧。

聪明的方法是：在makefile中添加一个include指示，但目前大多数版本的make都已经有include指示了，因此小技巧是设置一个depend目标

depend: count\_words.c lexer.c counter.c

$(CC) -M $(CPPFLAGS) $^ > $@

include depend

在运行make之前，先执行make depend命令。

如果我们把每一个源文件的依赖关系都写入它自己的依赖关系文件中，例如.d为后缀的同名文件，并将.d文件作为这个依赖规则的目标文件。当源文件改变时，make就会知道.d文件需要被更新了。以下代码可以实现这个规则：

%.d: %.c

$(CC) -M $(CPPFLAGS) $< > $@.

; \

sed 's,\.o[ :]\*,\1.o $@ : ,g' < $@.

> $@; \

rm -f $@.

在shell中$$表示当前进程的进程号，它会确立独一无二的文件名。先将依赖关系保存到这个特殊的文件中，然后使用 "sed" 命令将.d文件作为一个目标文件添加到规则中。sed命令包含了一个查找部分 \($\*\)\.o[ :]\* 和一个替代部分 \1.o $@,它们都被用逗号隔开。查找部分的文件名为$\*，它被包含在括号的正则表达式之中，并要求后缀名为.o。后面的[ :]\*表示零到多个空格或者冒号。替代部分前面的正则表达式替换为\1.o，并把当前目标文件添加到依赖文件。

于是，我们的makefile就变成了：

VPATH = src include

CPPFLAGS = -I include

CC = gcc

SOURCES = count\_words.c \

lexer.c \

counter.c

count\_words: counter.o lexer.o -lfl

count\_words.o: counter.h

counter.o: counter.h lexer.h

lexer.o: lexer.h

include $(subst .c,.d, $(SOURCES))

%.d: %.c

$(CC) -M $(CPPFLAGS) $< > $@.

; \

sed 's,\.o[ :]\*,\1.o $@ : ,g' < $@.

> $@; \

rm -f $@.

# help - The default goal

.PHONY: help

help:

make --print-data-base --question | \

awk '/^[^.%][-A-Za-z0-9\_]\*:/ \

{ print substr(

1)-1) }' | \

sort | \

pr --omit-pagination --width=80 --columns=4

.PHONY: clean

clean:

rm \*.o lexer.c count\_words \*.d

注意1： include必须放到手动设置的依赖关系之后，以防被手动设置的依赖关系覆盖。

注意2： include里面使用了subst函数，这是make的一个函数，它将$(SOURCE)文件里的所有.c替换成了.d，这两个后缀之间用逗号隔开且不能有空格。

运行make --just-print 会得到如下结果

makefile:12: count\_words.d: No such file or directory

makefile:12: lexer.d: No such file or directory

makefile:12: counter.d: No such file or directory

cc -M -I include src/counter.c > counter.d.$$; \

sed 's,\(counter\)\.o[ :]\*,\1.o counter.d : ,g' < counter.d.$$ > counter.d; \

rm -f counter.d.$$

lex -t src/lexer.l > lexer.c

cc -M -I include lexer.c > lexer.d.$$; \

sed 's,\(lexer\)\.o[ :]\*,\1.o lexer.d : ,g' < lexer.d.$$ > lexer.d; \

rm -f lexer.d.$$

cc -M -I include src/count\_words.c > count\_words.d.$$; \

sed 's,\(count\_words\)\.o[ :]\*,\1.o count\_words.d : ,g' < count\_words.d.$$ > count\_words.d; \

rm -f count\_words.d.$$

rm lexer.c

cc -I include -c -o count\_words.o src/count\_words.c

cc -I include -c -o counter.o src/counter.c

cc -I include -c -o lexer.o lexer.c

cc count\_words.o counter.o lexer.o /usr/lib/x86\_64-linux-gnu/libfl.so -o count\_words

前三行并不是报错，而是make的warning，make在所有路径和include里都找不到这几个文件。这些warning可以通过在include前面添加-（减号）消除掉。后面，make开始执行gcc -M来自动生成各种头文件依赖关系。

**九、库管理**

档案库（archive library）是一类特殊类型的文件，它用来归类相关的目标文件。make提供了对库文件的专门支持，包括创建，维护和引用。继续使用上面的例子，将counter.o和lexer.o作为库文件，创建的命令为：

ar rv libcounter.a counter.o lexer.o

参数rv表示我们想用列表里面的目标文件替换掉库文件里面的相同内容，如果库中不存在则添加进去，并要求ar显示整个过程。即使库文件不存在，这个参数也可以使用。参数后面的第一个文件名就是库文件名（有些其他版本的ar需要一个参数C，来显式的创建库文件）。

使用库文件的方式十分简单，通常就是在加在命令的编译列表里面，编译器和连接器会自动根据后缀名识别：

cc count\_words.o libcounter.a /lib/libfl.a -o count\_words

事实上，cc会自动识别出libcounter.a 和 /lib/libfl.a 是库文件，并且也会根据定义的库文件位置搜索它们，因此，也可以使用编译器的-l 参数，直接引用库文件：

cc count\_words.o -lcounter -lfl -o count\_words

这样可以省略掉前面表示库文件的部分和后缀名。-l参数可以使编译器搜索系统的库文件路径，并且对于不同的系统都适用。此外，对于支持共享库的系统（在UNIX中扩展名为.so的库文件），连接器会自动的先查找共享库，而不需要明确指出（GNU 的编译器有这种效果）。查找路径可以通过添加-L参数进行修改，修改后的路径会在系统库之前加载，并可以被所有-l参数使用。

事实上，上面一条命令是不能执行的，因为当前工作路径并不是cc的搜索路径，它找不到counter这个库文件。因此，需要做以下修改

cc count\_words.o -L. -lcounter -lfl -o count\_words

创建和更新库

在makefile中，库的创建与一般文件没有什么区别，简单的方式例如：

libcounter.a: counter.o lexer.o

$(AR) $(ARFLAGS) $@ $^

这里使用了make对ar程序的内置定义和标准参数选项 rv。但是，每次编译库的时候都会将所有的依赖文件进行编译，为了节省时间，可以将$^改成$?，这样就只会更新比ar库新的目标文件。但是，部分更新库文件所要付出的时间成本通常是远远高于整个库文件的更新，尤其是当库文件的数量比较多的时候，完整更新库会显得更加划算。

在GNU make中，引用库文件里面的成员可以用以下方式：

libcounter.a(counter.o): counter.o

$(AR) $(ARFLAGS) $@ $<;

将上面的内容综合起来，makefile变成了下面的样子：

VPATH = src include

CPPFLAGS = -I include

CC = gcc

count\_words: libcounter.a -lfl

libcounter.a: libcounter.a(lexer.o) libcounter.a(counter.o)

libcounter.a(lexer.o): lexer.o

$(AR) $(ARFLAGS) $@ $<

libcounter.a(counter.o): counter.o

$(AR) $(ARFLAGS) $@ $<

count\_words.o: counter.h

counter.o: counter.h lexer.h

lexer.o: lexer.h

# help - The default goal

.PHONY: help

help:

make --print-data-base --question | \

awk '/^[^.%][-A-Za-z0-9\_]\*:/ \

{ print substr(

1)-1) }' | \

sort | \

pr --omit-pagination --width=80 --columns=4

.PHONY: clean

clean:

rm \*.o lexer.c count\_words \*.d

执行make的输出为：

注意到生成库文件时使用的“$@”表示的是libcounter.a 而不是libcounter.a(lexer.o)

当然，我们也可以将内建规则应用在这里，使makefile更加精简：

VPATH = src include

CPPFLAGS = -I include

CC = gcc

count\_words: libcounter.a -lfl

libcounter.a: libcounter.a(lexer.o) libcounter.a(counter.o)

count\_words.o: counter.h

counter.o: counter.h lexer.h

lexer.o: lexer.h

# help - The default goal

.PHONY: help

help:

make --print-data-base --question | \

awk '/^[^.%][-A-Za-z0-9\_]\*:/ \

{ print substr(

1)-1) }' | \

sort | \

pr --omit-pagination --width=80 --columns=4

.PHONY: clean

clean:

rm \*.o lexer.c count\_words \*.d

库文件作为依赖关系

在依赖关系中使用库文件的方式有两种，一种是直接使用绝对路径，另一种则是在库文件名前面加上一个-l参数。后者的好处在于，它会优先搜索共享路径，并且可以根据用户自定义搜索相应的路径。用户自定义的模式匹配的内容存放在.LIBPATTERNS中。

然而，并不是所有情况下-l参数都是有效的，例如：

count\_words: -lcounter -lfl

libcounter.a: counter.o lexer.o

$(AR) $(ARFLAGS) $@ $^

如果是第一次编译，这里面的-lcounter是无法被make找到的，因为对于在makefile中生成的库文件，他们的名字在make过程中还不能被查找到。但如果当前目录已经存在了这个库文件，或者使用库文件的全名时，就不会出现这个问题了。

注意： 通常情况下，库文件在依赖关系中的顺序很重要，如果其中一个库会引用另外一个库的成员，被引用的一定要在引用它的库之后出现，因为连接器是不会回述链接好的库文件的。例如，库文件A调用了库文件B中的一个成员，于是在依赖列表中一定要保证-lA -lB的顺序，但如果B中的另一个成员又引用了A中的成员，则形成了循环引用。这时需要将依赖关系写作：-lA -lB -lA。这种多次调用库文件的方式在大型工程中经常出现，甚至会重复多次。这提醒了我们，在用$^替代依赖关系时是有问题的，因为它会去掉所有重复的内容，因此，会使用$+作为依赖关系变量使用，它保留了重复的内容。

## 13、编译报错

（1）、

extern void \_\_init xprj\_earlyyyy\_debug(void);

static void earlycon\_owl\_write(struct console \*con, const char \*s, unsigned n)

{

#if 1

xprj\_earlyyyy\_debug();

while (1);

#endif

错误信息为：

WARNING: modpost: Found 2 section mismatch(es).

To see full details build your kernel with:

make CONFIG\_DEBUG\_SECTION\_MISMATCH=y'

FATAL: modpost: Section mismatches detected.

Set CONFIG\_SECTION\_MISMATCH\_WARN\_ONLY=y to allow them.

make[3]: \*\*\* [vmlinux.o] Error 1

make[2]: \*\*\* [vmlinux] Error 2

原因是我们在非\_\_init的section中（earlycon\_owl\_write）调用了\_\_init section（xprj\_earlyyyy\_debug）的代码，按照提示说的，打开CONFIG\_SECTION\_MISMATCH\_WARN\_ONLY配置项即可编译通过：

## 14、Linux相关网站

Linux源码下载：https://www.kernel.org/

## 15、Linux内核空间内存申请函数kmalloc、devm\_kzalloc、kzalloc、vmalloc的区别【转】

我们都知道在用户空间动态申请内存用的函数是 malloc()，这个函数在各种操作系统上的使用是一致的，对应的用户空间内存释放函数是 free()。注意：动态申请的内存使用完后必须要释放，否则会造成内存泄漏，如果内存泄漏发生在内核空间，则会造成系统崩溃。

　　那么，在内核空间中如何申请内存呢？一般我们会用到devm\_kzalloc（ kmalloc()、kzalloc()、vmalloc() 等，下面我们介绍一下这些函数的使用以及它们之间的区别。

**kmalloc()**

函数原型：void \*kmalloc(size\_t size, gfp\_t flags)；

　　kmalloc() 申请的内存位于物理内存映射区域，而且在物理上也是连续的，它们与真实的物理地址只有一个固定的偏移，因为存在较简单的转换关系，所以对申请的内存大小有限制，不能超过128KB。

较常用的 flags（分配内存的方法）：

GFP\_ATOMIC —— 分配内存的过程是一个原子过程，分配内存的过程不会被（高优先级进程或中断）打断；

GFP\_KERNEL —— 正常分配内存；

GFP\_DMA —— 给 DMA 控制器分配内存，需要使用该标志（DMA要求分配虚拟地址和物理地址连续）。

flags 的参考用法：

　|– 进程上下文，可以睡眠　　　　　GFP\_KERNEL

　|– 进程上下文，不可以睡眠　　　　GFP\_ATOMIC

　|　　|– 中断处理程序　　　　　　　GFP\_ATOMIC

　|　　|– 软中断　　　　　　　　　　GFP\_ATOMIC

　|　　|– Tasklet　　　　　　　　　GFP\_ATOMIC

　|– 用于DMA的内存，可以睡眠　　　GFP\_DMA | GFP\_KERNEL

　|– 用于DMA的内存，不可以睡眠　　GFP\_DMA |GFP\_ATOMIC

对应的内存释放函数为：

void kfree(const void \*objp);

**kzalloc()**

　　kzalloc() 函数与 kmalloc() 非常相似，参数及返回值是一样的，可以说前者是后者的一个变种，因为 kzalloc() 实际上只是额外附加了 \_\_GFP\_ZERO 标志。所以它除了申请内核内存外，还会对申请到的内存内容清零。

/\*\* \* kzalloc - allocate memory. The memory is set to zero. \* @size: how many bytes of memory are required. \* @flags: the type of memory to allocate (see kmalloc). \*/static inline void \*kzalloc(size\_t size, gfp\_t flags){ return kmalloc(size, flags | \_\_GFP\_ZERO);}

kzalloc() 对应的内存释放函数也是 kfree()。

**devm\_kzalloc()**

原型：static inline void \* devm\_kzalloc（struct device \* dev，size\_t size，gfp\_t gfp）

函数devm\_kzalloc（）和kzalloc（）一样都是内核内存分配函数，但是devm\_kzalloc（）是跟设备（装置）有关的，当设备（装置）被拆卸或者驱动（驱动程序）卸载（空载）时，内存会被自动释放。另外，当内存不在使用时，可以使用函数devm\_kfree（）释放。

而kzalloc（）则需要手动释放（使用kfree（）），但如果工程师检查不仔细，则有可能造成内存泄漏。

注：也就是在驱动的探针函数中调用devm\_kzalloc（），在除去函数中调用devm\_kfree（）函数

例如：

static int xxx\_probe(struct platform\_device \*pdev)

{

struct sprdbat\_drivier\_data \*data = NULL;

.

.

.

data = devm\_kzalloc(&pdev->dev, sizeof(\*data), GFP\_KERNEL);

if (data == NULL) {

ret = -ENOMEM;

goto err\_data\_alloc\_failed;

}

.

.

.

}

**vmalloc()**

函数原型：

void \*vmalloc(unsigned long size);

　　vmalloc() 函数则会在虚拟内存空间给出一块连续的内存区，但这片连续的虚拟内存在物理内存中并不一定连续。由于 vmalloc() 没有保证申请到的是连续的物理内存，因此对申请的内存大小没有限制，如果需要申请较大的内存空间就需要用此函数了。

对应的内存释放函数为：

void vfree(const void \*addr);

注意：vmalloc() 和 vfree() 可以睡眠，因此不能从中断上下文调用。

总结

kmalloc()、kzalloc()、vmalloc() 的共同特点是：

1、用于申请内核空间的内存；

2、内存以字节为单位进行分配；

3、所分配的内存虚拟地址上连续；

kmalloc()、kzalloc()、vmalloc() 的区别是：

1、kzalloc 是强制清零的 kmalloc 操作；（以下描述不区分 kmalloc 和 kzalloc）

2、kmalloc 分配的内存大小有限制（128KB），而 vmalloc 没有限制；

3、kmalloc 可以保证分配的内存物理地址是连续的，但是 vmalloc 不能保证；

4、kmalloc 分配内存的过程可以是原子过程（使用 GFP\_ATOMIC），而 vmalloc 分配内存时则可能产生阻塞；

5、kmalloc 分配内存的开销小，因此 kmalloc 比 vmalloc 要快；

一般情况下，内存只有在要被 DMA 访问的时候才需要物理上连续，但为了性能上的考虑，内核中一般使用 kmalloc()，而只有在需要获得大块内存时才使用 vmalloc()。例如，当模块被动态加载到内核当中时，就把模块装载到由 vmalloc() 分配的内存上。

## 16、Linux内核中的pinctrl子系统应用实例

转：https://blog.csdn.net/hanp\_linux/article/details/72818437

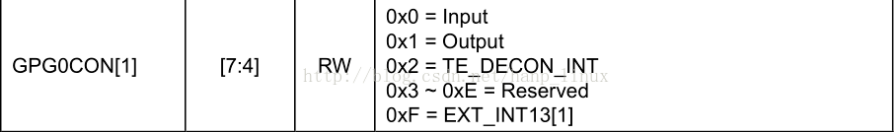
由于近期在做一个项目用到了pinctrl子系统，但是对pinctrl子系统了解又不是很多，所以遇到了麻烦，但是找度娘发现很少有同行对pinctrl的具体用法做出说明，所以只能自己去搞了，在经过一段时间对Linux内核源码的折腾，最终搞定，并将我所应用的实例给展示一下，希望对大家有所帮助。

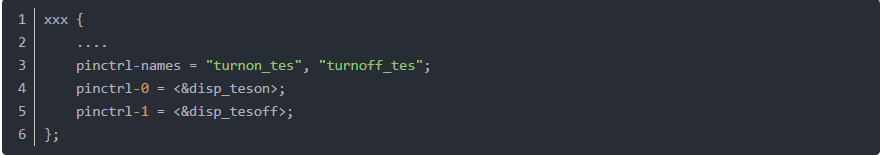
关于pinctrl是什么，为什么要用pinctrl，源码深度剖析我在这就不赘述了，有位博友总结的非常好，大家可以参考http://www.wowotech.net/sort/gpio\_subsystem。

下面我介绍一下如何去使用内核中的pinctrl子系统以device tree设备树为例，当你需要控制某些pin的时候，你首先要在devicetree中去按照pinctrl的规则去描述它，然后才能在driver中去使用：

案例1：

xxx这个设备要用到gpg0\_1这个pin的TE\_DECON\_INT功能，并分别将这两个状态取了个名字turnon\_tes和turnoff\_tes.这个名字是随便起的。重点是看pinctrl-0和pinctrl-1，根据示例，它们分别引用了disp\_teson和disp\_tesoff这两个节点。



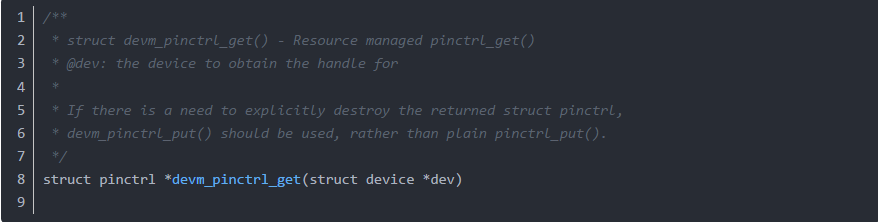


两个重要的属性必须有：pins 和 pin-function分别是pin的名字和要把pin配置成什么功能，还有gpg0属于pinctrl\_2,所以这个地方引用的是pinctrl\_2而不是其他。

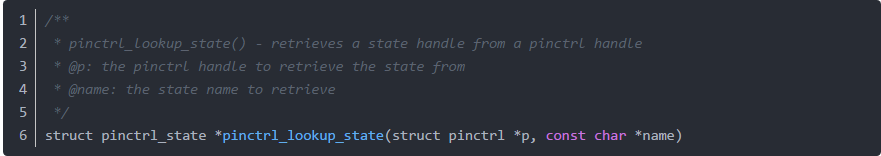


那么driver如何去操作这个pin呢？首先需要大家熟悉几个内核的API：

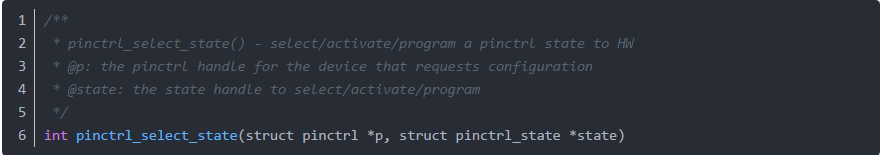
1. 获取一个pinctrl句柄，参数是dev是包含这个pin的device结构体即xxx这个设备的device



2. 获取这个pin对应pin\_state（引脚状态-turnon\_tes/turnoff\_tes）



3. 设置引脚为为某个stata -- turnon\_tes/turnoff\_tes



具体操作：

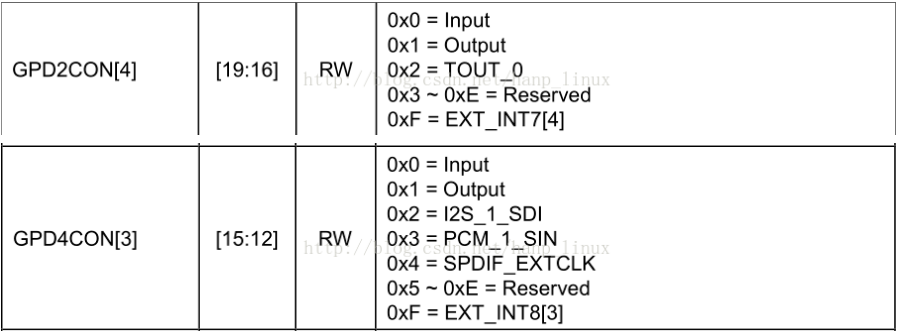


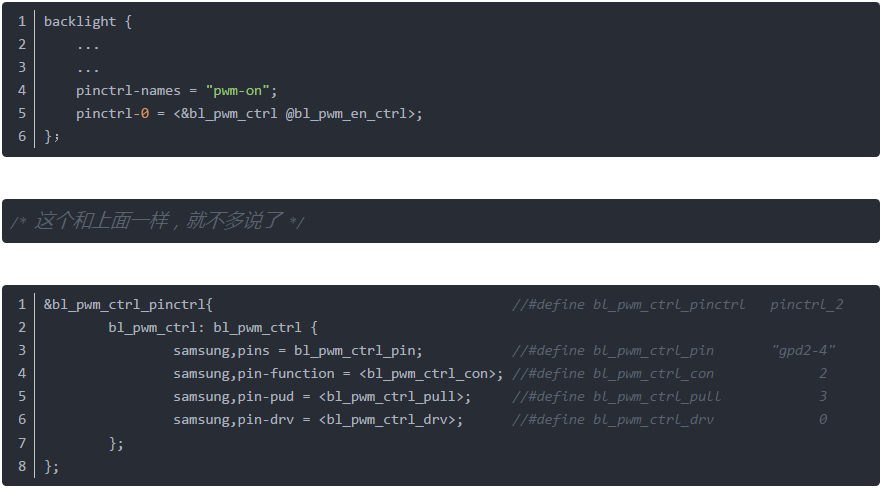
经过以上操作，gpg\_1引脚对应的con寄存器的对应的位域被配置成2，即0x2 = TE\_DECON\_INT功能。同理，根据此方法也可以设置turnoff\_tes的状态。

案例2 -- 一个背光灯device需要使用pwm的输出pin：

device tree：

背光系统中要用到gpd2\_4这个pin的TOUT\_0功能和gpd4\_3这个pin的输出功能并输出1，需要在backlight这个node中做以下描述，这两个pin只有一个状态（pwm-on），同样，这个名字也是可以随便起的。bl\_pwm\_ctrl和bl\_pwm\_en\_ctrl分别是对这两个pin的描述。

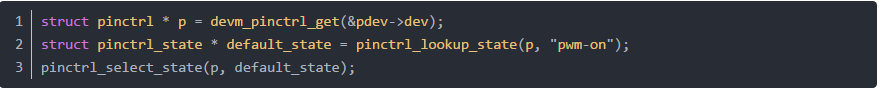




这个描述比上面多了个pin-val，因为这个引脚不仅要配置成输出功能，还要输出1，所以pin-val = 1。

driver的操作：

在backlight的driver的probe中：



执行完以上操作，可以发现gpd2\_4引脚被配置成了TOUT\_0功能，gpd4\_3引脚被配置成为了输出功能，并且输出1（高电平）。

以上就是pinctrl子系统的应用实例。如果有解释不太正确的地方请指教。

应用实例：6739 指纹 tp

## 17、linux内核几种打印调试信息的方法

**（1）、pr\_debug需要怎样设置才会打印消息（一般打印调试时候才需要的信息）**

1）在Makefile里面增加 EXTRA\_CFLAGS += -DDEBUG 即可

可以有如下用法“

#define fpsensor\_debug(fmt, args...) pr\_debug("[DBG] "fmt, ##args)

**（2）、pr\_info**

#define pr\_info(fmt, ...) \

printk(KERN\_INFO pr\_fmt(fmt), ##\_\_VA\_ARGS\_\_)

**（3）、pr\_err**

#define pr\_err(fmt, ...) \

printk(KERN\_ERR pr\_fmt(fmt), ##\_\_VA\_ARGS\_\_)

## 18、SELinux学习笔记

**小知识：**

1、 SELinux：SELinux(Security-Enhanced Linux) 是美国国家安全局（NSA）对于强制访问控制的实现，是 Linux历史上最杰出的新安全子系统。NSA是在Linux社区的帮助下开发了一种访问控制体系，在这种访问控制体系的限制下，进程只能访问那些在他的任务中所需要文件。SELinux 默认安装在 Fedora 和 Red Hat Enterprise Linux 上，也可以作为其他发行版上容易安装的包得到。

2、除了SELinux之外还有另一种对权限的控制（自主访问控制(DAC)），其实这个就是我们创建文件的时候设置的权限，还可以手动使用chmod 777 文件 修改权限。系统的访问权限是由DAC和SElinux共同完成缺谁都访问不了。

**3、参考博客：https://blog.csdn.net/gulinxieying/article/details/78677139**

**使用adb命令开关SELinux：**

adb shell setenforce 0 关闭SElinux

adb shell setenforce 1 开启SElinux

**使用adb命令确认SElinux是否开启：**

adb shell getenforce 返回 Enforcing 表示开启

返回 Permissive 表示关闭

其中enforcing,permissive 分别对应 1, 0

**语法规则：**

**（1）、访问向量规则**，访问向量规则又称AV规则，属于SElinux策略语言的控制部分。常用的通用AV规则有如下4个(详细解释https://blog.csdn.net/gulinxieying/article/details/78677139)：

allow：表示允许主体对客体执行允许的操作

dontaudit：表示不记录违反规则的决策信息，且违反规则不影响运行(允许操作且不记录)

auditallow：表示允许操作并记录访问决策信息(允许操作且记录)

neverallow：表示不允许主体对客体执行指定的操作

完整的基本安全控制语句格式为：

AV规则 主体 客体：客体类别 许可

eg: allow netd proc:file write

**（2）、SElinux策略文件（SElinux的策略文件主要集中在external\sepolicy下 展讯：device\sprd\sharkl3\common\sepolicy）：**

这里也是我们添加修改安全策略的主要地方我们会发现里面有很多.te文件，这些te文件就是我们定义的策略文件，除了策略文件之外，还有很多没有后缀的，这些没有后缀的文件就是专门定义策略语句元素的。由于所有的te文件中的规则语句都是基于这些语句元素组成的，这里我们先来看下这些定义性文件

所有的安全策略在编译后最终都会汇集到，out/target/product/generic/obj/ETC/sepolicy\_intermediates/policy.conf文件中（Android 8.0的位置变掉），

1）、定义性文件

users ：该文件中定义了SElinux中唯一的用户u：

roles ：该文件定义了SElinux中唯一的角色r：

security\_classes ：该文件定义了所有通用客体类别：

attributes ：该文件定义了所有SContext中的通用type：

access\_vectors ：该文件定义了资源的所有通用许可：

file.te ：该文件自定义了file\_contexts中的所有文件类型的type：

device.te ：该文件自定义了file\_contexts中的所有设备节点的type：

genfs\_contexts ：定义了虚拟文件的SContext：

fs\_use ：定义了所有文件系统类型的SContext：

service.te :：定义了所有service\_contexts中SContext的type：

service\_contexts ：定义了所有所有系统service的SContext：

mac\_permissions.xml ：与seapp\_contexts一起负责为接下来安装的不同类型的app添加SContext。

seapp\_contexts ：与mac\_permissions.xml一起负责为接下来安装的不同类型的app添加SContext。

成对出现的有：

\*device类型：

-- 类型定义: external/sepolicy/device.te; device/mediatek/common/sepolicy/device.te

-- 类型绑定: external/sepolicy/file\_contexts;device/mediatek/common/sepolicy/file\_contexts

\*File类型:

-- 类型定义: external/sepolicy/file.te;device/mediatek/common/sepolicy/file.te

-- 绑定类型: external/sepolicy/file\_contexts;device/mediatek/common/sepolicy/file\_contexts

\*虚拟File类型:

-- 类型定义: external/sepolicy/file.te;device/mediatek/common/sepolicy/file.te

-- 绑定类型: external/sepolicy/genfs\_contexts;device/mediatek/common/sepolicy/genfs\_contexts

\*Service类型:

-- 类型定义: external/sepolicy/service.te; device/mediatek/common/sepolicy/service.te

-- 绑定类型：external/sepolicyservice\_contexts;device/mediatek/common/sepolicy/service\_contexts

\*Property类型:

-- 类型定义: external/sepolicy/property.te;device/mediatek/common/sepolicy/property.te

-- 绑定类型: external/sepolicy/property\_contexts;device/mediatek/common/sepolicy/property\_contexts;

2）、其他te文件

剩余的.te文件中主要是各个进程的具体策略控制。

**（3）、如何配置SElinux**

在支持SElinux的Android版本中，有关selinux策略文件有两处，一个是google原生的，另一个是产品平台配置的，他们分别对应如下目录：

google原生目录：s912/external/sepolicy

平台配置目录：s912/device/amlogic/common/sepolicy

在编译时, 系统会以合并的方式，将平台配置目录下的policy 附加到Google 原生的policy 上，而非替换。一般情况下，不建议修改google原生目录下的策略控制。接下来看下如何配置平台的selinux。

找到平台下的BoardConfig.mk，其相关变量配置就是BOARD\_SEPOLICY\_DIRS，目的就是指定平台sepolicy的目录。BoardConfig.mk中直接包含了sepolicy的mkfile文件： include device/amlogic/common/sepolicy.mk打开该sepolicy.mk文件可以看到BOARD\_SEPOLICY\_DIRS变量的配置：BOARD\_SEPOLICY\_DIRS := device/amlogic/common/sepolicy

**（4）、SElinux调试方法**

**1）、selinux的编译**

首先我们关注下selinux的编译，大家知道编译整个Android工程是个很耗时的工作。为了避免耗时，在我们修改或者添加了selinux策略之后，建议先独自对sepolicy模块进行一次编译，如果通过了，然后再进行整个Android的编译。具体指令如下：如：mmm external/sepolicy

**2）、如何快速确认是selinux引起？**

目前所有的SELinux check 失败，在kernel log 或者android log(L版本后)中都有对应的" avc: denied"的LOG 与之对应。

标志性 log: avc: denied { 操作权限 } for pid=7201 comm=“进程名” scontext=u:r:源类型:s0 tcontext=u:r:目标类型:s0 tclass=访问类型 permissive=0

其中：

进程的scontext 结构： user:role:type[:range]

注释：

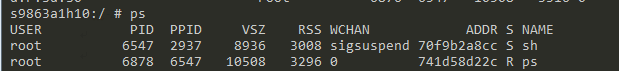
u——user，用户，SEAndroid中仅定义了一个SELinux用户，值为u。

r——role，role是角色之意，它是SELinux中一种比较高层次，更方便的权限管理思路，即Role Based Access Control（基于角色的访问控制，简称为RBAC）。简单点说，一个u可以属于多个role，不同的role具有不同的权限。RBAC我们到最后再讨论。

init——tyoe，代表该进程所属的Domain为init。MAC的基础管理思路其实不是针对上面的RBAC，而是所谓的Type Enforcement Accesc Control（简称TEAC，一般用TE表示）。对进程来说，Type就是Domain。比如init这个Domain有什么权限，都需要通过[例子1]中allow语句来说明。

S0——和SELinux为了满足军用和教育行业而设计的Multi-Level Security（MLS）机制有关。简单点说，MLS将系统的进程和文件进行了分级，不同级别的资源需要对应级别的进程才能访问。限于篇幅，这里不做过多探讨。

ps -Z



文件的scontext 结构： user:role:type[:range]

其中：

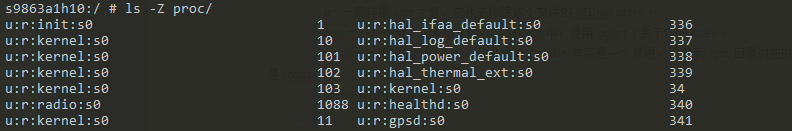
u——同样是user之意，它代表创建这个文件的SELinux user。

object\_r——role，这里指文件，在SELinux中，使用object\_r表示它的role。

rootfs——死的东西的Type，和进程的Domain其实是一个意思。它表示root目录对应的Type是rootfs。

s0——MLS的级别。

ls –Z



在相应源类型.te 文件，添加如下格式的一行语句：(结尾别忘了分号) 格式：“allow 源类型 目标类型:访问类型 {操作权限};” 或者叫做 “AV规则 主体 客体：客体类别 许可”

例如：

07-10 14:06:32.780 9043 9043 W com.filemanager: type=1400 audit(0.0:104): avc: denied { read } for name="u:object\_r:curef\_set\_prop:s0" dev="tmpfs" ino=1334 scontext=u:r:platform\_app:s0:c512,c768 tcontext=u:object\_r:curef\_set\_prop:s0 tclass=file permissive=0

可以得到如下 sepolicy：添加到 platform\_app.te 文件 allow platform\_app curef\_set\_prop : file { read };

**（5）、常用指令**

SELinux 是个经过安全强化的Linux操作系统，实际上，基本上原来的运用软件没有必要修改就能在它上面运行。真正做了特别修改的RPM包只要50多个。像文件系统EXT3都是经过了扩展。对于一些原有的命令也进行了扩展，另外还增加了一些新的命令，接下来我们就来看看这些命令。

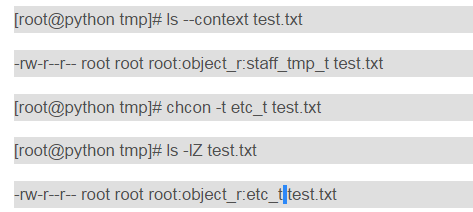
1）、文件操作

**ls命令（可以看到文件或者文件夹的标签）：**

在命令后加个 －Z 或者加 –context



**chcon**更改文件的标签



**restorecon** 当这个文件在策略里有定义是，可以恢复原来的 文件标签。

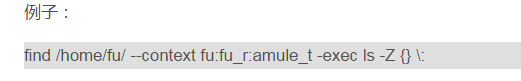
**setfiles**  跟chcon一样可以更改一部分文件的标签，不需要对整个文件系统重新设定标签。

**fixfiles**  一般是对整个文件系统的， 后面一般跟 relabel,对整个系统 relabel后，一般我们都重新启动。如果，在根目录下有.autorelabel空文件的话，每次重新启动时都调用 fixfiles relabel

**star**就是tar在SELinux下的互换命令，能把文件的标签也一起备份起来。

**cp**可以跟 -Z, --context=CONTEXT 在拷贝的时候指定目的地文件的security context

**find**可以跟 –context 查特定的type的文件。



**run\_init**在sysadm\_t里手动启动一些如Apache之类的程序，也可以让它正常进行，domain迁移。

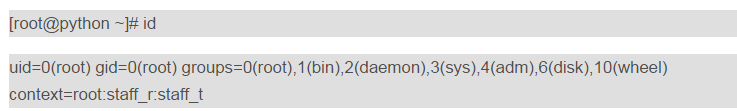
2）、进程domain的确认

程序现在在那个domain里运行，我们可以在ps 命令后加 －Z

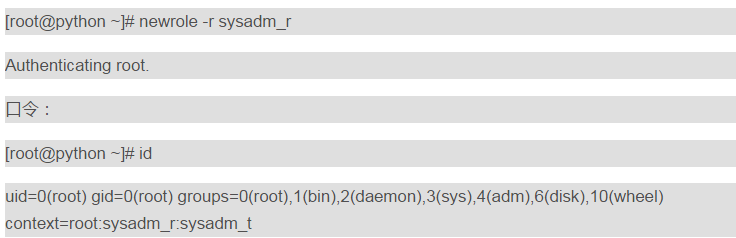


3）、ROLE的确认和变更

命令id能用来确认自己的 security context

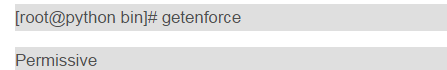


这里，虽然是ROOT用户，但也只是在一般的ROLE和staff\_t里运行，如果在enforcing模式下，这时的ROOT对于系统管理工作来说什么也干不了。



4）、模式切换

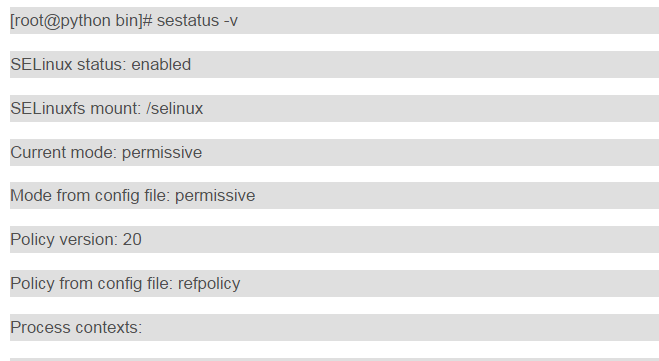
**getenforce**得到当前的SELINUX值



**setenforce**更改当前的SELINUX值 ，后面可以跟 enforcing,permissive 或者 1, 0。



**sestatus**显示当前的 SELinux的信息

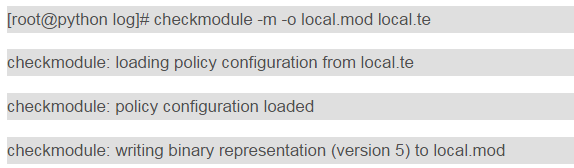


5）、其他重要命令

**Audit2allow**很重要的一个以python写的命令，主要用来处理日志，把日志中的违反策略的动作的记录，转换成 access vector，对开发安全策略非常有用。在refpolicy里，它的功能比以前有了很大的扩展。



**checkmodule**编译模块



**semodule\_package**创建新的模块



**semodule**可以显示，加载，删除 模块



**semanage**  这是一个功能强大的策略管理工具，有了它即使没有策略的源代码，也是可以管理安全策略的。因为我主要是介绍用源代码来修改策略的，详细用法大家可以参考它的man页。

**（6）、参考文献**

1. Security-Enhanced Linux in Android

<https://source.android.com/security/selinux/index.html>

2. 深入理解SELinux SEAndroid

<http://blog.csdn.net/innost/article/details/19299937>

3.SElinux详解.pdf

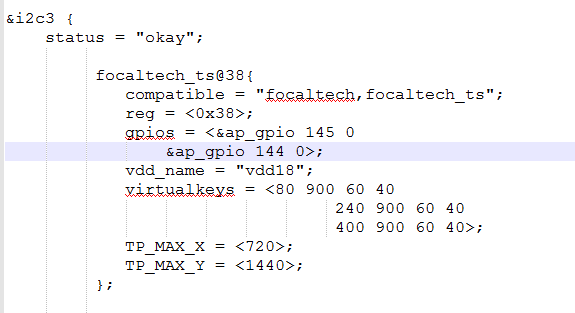
4.类型强制（TE编写规则）

http://blog.csdn.net/myarrow/article/details/10105961

## 19、dts学习笔记

### （1）、dts配置和实际驱动应用

eg1:



struct ft5x0x\_ts\_platform\_data{

int irq\_gpio\_number;

int reset\_gpio\_number;

const char \*vdd\_name;

u32 virtualkeys[12];

int TP\_MAX\_X;

int TP\_MAX\_Y;

};

static struct ft5x0x\_ts\_platform\_data \*ft5x0x\_ts\_parse\_dt(struct device \*dev)

{

struct ft5x0x\_ts\_platform\_data \*pdata;

struct device\_node \*np = dev->of\_node;

int ret;

pdata = kzalloc(sizeof(\*pdata), GFP\_KERNEL);

if (!pdata) {

dev\_err(dev, "Could not allocate struct ft5x0x\_ts\_platform\_data");

return NULL;

}

pdata->reset\_gpio\_number = of\_get\_gpio(np, 0); //得到145

if(pdata->reset\_gpio\_number < 0){

dev\_err(dev, "fail to get reset\_gpio\_number\n");

goto fail;

}

pdata->irq\_gpio\_number = of\_get\_gpio(np, 1); //得到144

if(pdata->reset\_gpio\_number < 0){

dev\_err(dev, "fail to get reset\_gpio\_number\n");

goto fail;

}

ret = of\_property\_read\_string(np, "vdd\_name", &pdata->vdd\_name);

if(ret){

dev\_err(dev, "fail to get vdd\_name\n");

goto fail;

}

ret = of\_property\_read\_u32\_array(np, "virtualkeys", pdata->virtualkeys,12);

if(ret){

dev\_err(dev, "fail to get virtualkeys\n");

goto fail;

}

ret = of\_property\_read\_u32(np, "TP\_MAX\_X", &pdata->TP\_MAX\_X);

if(ret){

dev\_err(dev, "fail to get TP\_MAX\_X\n");

goto fail;

}

ret = of\_property\_read\_u32(np, "TP\_MAX\_Y", &pdata->TP\_MAX\_Y);

if(ret){

dev\_err(dev, "fail to get TP\_MAX\_Y\n");

goto fail;

}

return pdata;

fail:

kfree(pdata);

return NULL;

}

### （2）、常用的API

在Linux的BSP和驱动代码中经常会使用到一些设备树的API， 这些API通常被冠以of\_前缀， 它们的

实现代码位于内核的drivers/of目录下

**1）、API判断具体的电路板是什么**

of\_machine\_is\_compatible（const char\*compat）

**2）、判断设备节点的兼容属性是否包含compat指定的字符串。 这个API多用于一个驱动支持两个以上设备的时候。**

int of\_device\_is\_compatible(const struct device\_node \*device,const char \*compat);

**（**

一个驱动兼容多个设备除了以上的方法之外

还可以采用在驱动的of\_device\_id表中填充.data成员的形式。 譬如， arch/arm/mm/cache-l2x0.c支持“arm， l210-cache”“arm， pl310-cache”“arm， l220-cache”等多种设备

eg:

#define L2C\_ID(name, fns) { .compatible = name, .data = (void \*)&fns }

static const struct of\_device\_id l2x0\_ids[] \_\_initconst = {

L2C\_ID("arm,l210-cache", of\_l2c210\_data),

L2C\_ID("arm,l220-cache", of\_l2c220\_data),

L2C\_ID("arm,pl310-cache", of\_l2c310\_data),

L2C\_ID("brcm,bcm11351-a2-pl310-cache", of\_bcm\_l2x0\_data),

L2C\_ID("marvell,aurora-outer-cache", of\_aurora\_with\_outer\_data),8 L2C\_ID("marvell,aurora-system-cache", of\_aurora\_no\_outer\_data),

L2C\_ID("marvell,tauros3-cache", of\_tauros3\_data),

/\* Deprecated IDs \*/

L2C\_ID("bcm,bcm11351-a2-pl310-cache", of\_bcm\_l2x0\_data),

{}

}

在驱动中， 通过以下方法拿到了对应于L2缓存类型的.data成员， 其中主要用到了of\_match\_node（） 这个API。

int \_\_init l2x0\_of\_init(u32 aux\_val, u32 aux\_mask)

{

const struct l2c\_init\_data \*data;

struct device\_node \*np;

np = of\_find\_matching\_node(NULL, l2x0\_ids);

if (!np)

return -ENODEV;

…

data = of\_match\_node(l2x0\_ids, np)->data;

}

如果电路板的.dts文件中L2缓存是arm， pl310-cache， 那么上述代码第10行找到的data就是

of\_l2c310\_data， 它是l2c\_init\_data结构体的一个实例。 l2c\_init\_data是一个由L2缓存驱动自定义的数据结构， 在其定义中既可以保护数据成员， 又可以包含函数指针， 如以下代码清单：

struct l2c\_init\_data {

const char \*type;

unsigned way\_size\_0;

unsigned num\_lock;

void (\*of\_parse)(const struct device\_node \*, u32 \*, u32 \*);

void (\*enable)(void \_\_iomem \*, u32, unsigned);

void (\*fixup)(void \_\_iomem \*, u32, struct outer\_cache\_fns \*);

void (\*save)(void \_\_iomem \*);

struct outer\_cache\_fns outer\_cache;

};

通过这种方法， 驱动可以把与某个设备兼容的私有数据寻找出来， 如此体现了一种面向对象的设计思

想， 避免了大量的if， else或者switch， case语句。

**)**

**3）、寻找节点**

struct device\_node \*of\_find\_compatible\_node(struct device\_node \*from,const char \*type, const char \*compatible);

根据兼容属性， 获得设备节点。 遍历设备树中的设备节点， 看看哪个节点的类型、 兼容属性与本函数

的输入参数匹配， 在大多数情况下， from、 type为NULL， 则表示遍历了所有节点。

**4）、读取属性值（整型）**

int of\_property\_read\_u8\_array(const struct device\_node \*np,const char \*propname, u8 \*out\_values, size\_t sz);

int of\_property\_read\_u16\_array(const struct device\_node \*np,const char \*propname, u16 \*out\_values, size\_t sz);

int of\_property\_read\_u32\_array(const struct device\_node \*np,const char \*propname, u32 \*out\_values, size\_t sz);

int of\_property\_read\_u64(const struct device\_node \*np, const char\*propname, u64 \*out\_value);

读取设备节点np的属性名， 为propname， 属性类型为8、 16、 32、 64位整型数组。 对于32位处理器来讲， 最常用的是of\_property\_read\_u32\_array（） 。

of\_property\_read\_u32\_array(np, "arm,data-latency",data, ARRAY\_SIZE(data));

在arch/arm/boot/dts/vexpress-v2p-ca9.dts中， 对应的含有"arm， data-latency"属性的L2cache节点如下：

L2: cache-controller@1e00a000 {

compatible = "arm,pl310-cache";

reg = <0x1e00a000 0x1000>;

interrupts = <0 43 4>;

cache-level = <2>;

arm,data-latency = <1 1 1>;

arm,tag-latency = <1 1 1>;

}

在有些情况下， 整型属性的长度可能为1， 于是内核为了方便调用者， 又在上述API的基础上封装出

了更加简单的读单一整形属性的API， 它们为int of\_property\_read\_u8（） 、 of\_property\_read\_u16（） 等， 实现于include/linux/of.h中，如下代码：

static inline int of\_property\_read\_u8(const struct device\_node \*np,const char \*propname,u8 \*out\_value)

{

return of\_property\_read\_u8\_array(np, propname, out\_value, 1);

}

static inline int of\_property\_read\_u16(const struct device\_node \*np,const char \*propname, u16 \*out\_value)

{

return of\_property\_read\_u16\_array(np, propname, out\_value, 1);

}

static inline int of\_property\_read\_u32(const struct device\_node \*np,const char \*propname,u32 \*out\_value)

{

return of\_property\_read\_u32\_array(np, propname, out\_value, 1);

}

**5）、读取属性值（字符型）**

int of\_property\_read\_string(struct device\_node \*np, const char\*propname,const char \*\*out\_string);//读取字符串属性。

int of\_property\_read\_string\_index(struct device\_node \*np, const char\*propname,int index, const char \*\*output);// 读取字符串数组属性中的第index个字符串,例如：

const char \*of\_clk\_get\_parent\_name(struct device\_node \*np, int index)

{

struct of\_phandle\_args clkspec;

const char \*clk\_name;

int rc;

if (index < 0)

return NULL;

rc = of\_parse\_phandle\_with\_args(np, "clocks", "#clock-cells", index&clkspec);

if (rc)

return NULL;

if (of\_property\_read\_string\_index(clkspec.np, "clock-output-names", clkspec.args\_count clkspec.args[0]: 0,&clk\_name) < 0)

clk\_name = clkspec.np->name;

of\_node\_put(clkspec.np);

return clk\_name;

}

EXPORT\_SYMBOL\_GPL(of\_clk\_get\_parent\_name);

**6）、读取属性值（布尔型）**

static inline bool of\_property\_read\_bool(const struct device\_node \*np,const char \*propname);

如果设备节点np含有propname属性， 则返回true， 否则返回false。 一般用于检查空属性是否存在。

**7）、内存映射**

void \_\_iomem \*of\_iomap(struct device\_node \*node, int index);

上述API可以直接通过设备节点进行设备内存区间的ioremap（） ， index是内存段的索引。 若设备节点

的reg属性有多段， 可通过index标示要ioremap（） 的是哪一段， 在只有1段的情况， index为0。 采用设备树后， 一些设备驱动通过of\_iomap（） 而不再通过传统的ioremap（） 进行映射， 当然， 传统的ioremap（）的用户也不少。

int of\_address\_to\_resource(struct device\_node \*dev, int index,struct resource \*r);

上述API通过设备节点获取与它对应的内存资源的resource结构体。 其本质是分析reg属性以获取内存

基地址、 大小等信息并填充到struct resource\*r参数指向的结构体中。

**8）、解析中断**

unsigned int irq\_of\_parse\_and\_map(struct device\_node \*dev, int index);

通过设备树获得设备的中断号， 实际上是从.dts中的interrupts属性里解析出中断号。 若设备使用了多

个中断， index指定中断的索引号。

**9）、获取与节点对应的platform\_device**

struct platform\_device \*of\_find\_device\_by\_node(struct device\_node \*np);

在可以拿到device\_node的情况下， 如果想反向获取对应的platform\_device， 可使用上述API。当然， 在已知platform\_device的情况下， 想获取device\_node则易如反掌， 例如：

static int sirfsoc\_dma\_probe(struct platform\_device \*op)

{ s

truct device\_node \*dn = op->dev.of\_node;

…

}

## 20、如何制作静态库和使用静态库

### （1）编译出一个.o文件

gcc -c test.c -o test.o

### （2）、生成静态库

ar rc libname.a test.o

### （3）、Makefile配置静态库

PLATFORM\_LIBS += -L$(srctree)/drivers/udc/ -lname

## 21、container\_of函数的作用

**（1）、作用：**通过ptr（结构体某个成员的实际地址）计算的到相结构体的实际首地址（进而可以得到整个结构体）。

**（2）函数原型：**

#define container\_of(ptr,type,member) ({ \

const typeof( ((type \*)0)->member ) \*\_\_mptr = (ptr); \

(type \*)( (char \*)\_\_mptr - offsetof(type,member) );})

函数解析：

第二行的主要作用是用作校验ptr的类型就是type，如果ptr的类型不是type的话编译就会有警告。

第三行的主要思想是：用已知结构体成员的实际地址减去他相对于结构体首地址的偏移量。其中的难点在于计算偏移量，其中sffsetof就是用来计算偏移量的。

sffsetof的原型是 #define offsetof(TYPE, MEMBER) &((TYPE \*)0)->MEMBER

解析：由于定义0指针结构体他的首地址是0所以可以很容易得到成员相对于0地址的偏移量。这个偏移量也就上面所说的偏移量了。

（3）、实例列举

struct test

{

int a;

int b;

}

struct test \*temp;

int c=temp->b;

temp = container\_of(&temp->b,struct test,b);

## 22、GPIO操作

### 1、linux的默认操作API

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* dts配置\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

ap\_gpio: gpio-controller@402c0000 {

compatible = "sprd,gpio-plus";

reg = <0x0 0x402c0000 0x0 0x1000>;

gpio-controller;

#gpio-cells = <2>;

sprd,gpiobase = <0>;

sprd,ngpios = <256>;

interrupt-controller;

#interrupt-cells = <2>;

interrupts = <GIC\_SPI 130 IRQ\_TYPE\_LEVEL\_HIGH>;

};

focaltech\_ts@38{

compatible = "focaltech,focaltech\_ts";

reg = <0x38>;

gpios = <&ap\_gpio 145 0

&ap\_gpio 144 0>;

。

。

。

};

int reset\_gpio\_number;

int irq\_gpio\_number;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*从dts中取出相应的值\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

reset\_gpio\_number = of\_get\_gpio(np, 0);

if(reset\_gpio\_number < 0){

dev\_err(dev, "fail to get reset\_gpio\_number\n");

goto fail;

}

irq \_gpio\_number = of\_get\_gpio(np, 1);

if(irq \_gpio\_number < 0){

dev\_err(dev, "fail to get irq \_gpio\_number\n");

goto fail;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*初始化和操作相应的GPIO口\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**gpio口用作为irq类型：**

int irq;

int err=0;

gpio\_request(irq\_gpio\_number, "ts\_irq\_pin"); //声明

gpio\_direction\_input(irq\_gpio\_number); //设置为输入

irq = gpio\_to\_irq(irq\_gpio\_number); //由相应的IO口获取到中断号

#if USE\_THREADED\_IRQ

err = request\_threaded\_irq(irq, NULL, ft5x0x\_ts\_interrupt,

IRQF\_TRIGGER\_FALLING | IRQF\_ONESHOT, “name”, ft5x0x\_ts); //创建中断服务函数（这种创建方式分为上半部中断和底半部中断，底半部中断是通过创建一个线程实现的；其中ft5x0x\_ts\_interrupt为创建的中断服务函数是一个线程，IRQF\_TRIGGER\_FALLING | IRQF\_ONESHOT为上升沿触发不进入休眠，关于这个函数的使用下面有描述）

#else

err = request\_irq(rq, ft5x0x\_ts\_interrupt,

IRQF\_TRIGGER\_FALLING | IRQF\_ONESHOT | IRQF\_NO\_SUSPEND, client->name, ft5x0x\_ts); //（创建普通的中断函数没有上半部底半部的区分）

#endif

if (err < 0) {

goto exit\_irq\_request\_failed;

}

static irqreturn\_t ft5x0x\_ts\_interrupt(int irq, void \*dev\_id) //实现中断服务函数

{

。

。

。

}

enable\_irq(irq); //使能中断

disable\_irq(irq); //关闭中断

enable\_irq\_wake(irq);//使能中断，在安卓系统中在休眠的时候如果需要中断唤醒需要这么写

disable\_irq\_wake(irq); //关闭中断

irq\_set\_irq\_type(this\_client->irq,IRQF\_TRIGGER\_LOW|IRQF\_NO\_SUSPEND); //设置中断的电平触发条件

备注： /kernel/include/linux/interrupt.h

#define IRQF\_TRIGGER\_RISING //上升沿触发

#define IRQF\_TRIGGER\_FALLING //下降沿触发

#define IRQF\_TRIGGER\_HIGH //高电平触发

#define IRQF\_TRIGGER\_LOW //低电平触发

#define IRQF\_ONESHOT //防止中断嵌套

#define IRQF\_NO\_SUSPEND //防止在系统进入suspend的时候中断会进入休眠（注意此标志位只能确保有中断过来的时候能够触发中断，但是并不一定能够唤醒系统如果想要唤醒系统还是得声明enable\_irq\_wake(irq);）

**gpio类型：**

gpio\_request(reset\_gpio\_number, "ts\_rst\_pin"); //注册GPIO口

gpio\_direction\_output(reset\_gpio\_number, 1); //相应的GPIO模式被设置为输出同时默认值为高电平。

gpio\_set\_value(reset\_gpio\_number, 0); //设置GPIO的极性，其中0为低电平 1为高电平

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*注销GPIO口\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

gpio\_free(reset\_gpio\_number); //释放GPIO口

## 23、Ubuntu 16.0加一个新的硬盘

操作环境：

系统：Ubuntu 16.04

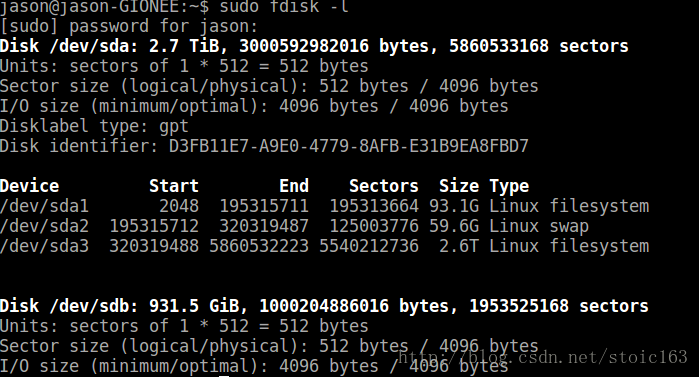
硬件：硬盘 1T（全新未使用过的WD蓝盘）

操作过程：

1.拆开电脑机箱，安装硬盘，具体操作省略。

2.开机查看系统硬盘分区情况：

命令：sudo fdisk -l

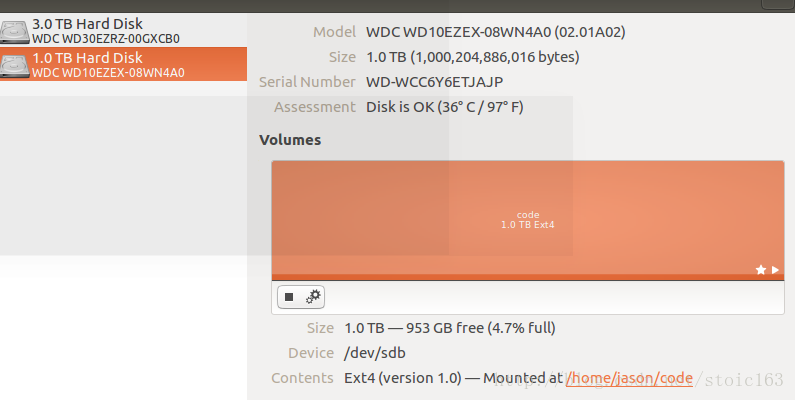


得到如上图：可以知道第二块硬盘:/dev/sdb

3.格式化硬盘

命令：sudo mkfs.ext4 /dev/sdb

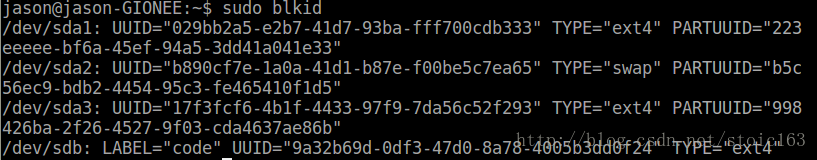
还有一种方法是通过Ubuntu系统中搜索框搜索disk软件查看硬盘情况，并通过该软件图形化进行格式化操作



鼠标点击两个齿轮的位置，第一选项就是格式化硬盘，或者使用crtl+shift+F进行格式化操作，可以个格式化硬盘设置一个标签，如：code

4.查看硬盘分区的UUID

命令：sudo blkid



从上图可以清楚的看到，该硬盘已经格式化为ext4，标签为code

5.在已有的user目录下新建一个挂在点

         命令：mkdir xxx

         例如：我自己在已有的目录下创建一个code目录挂载新的硬盘分区，如：mkdir /home/jason/code

6.编辑系统挂载配置文件/etc/fstab

        命令：sudo  vim etc/fstab



 按照已有的格式，将新加的硬盘分区信息添加到结尾；

注：格式为 设备名称 挂载点 分区类型 挂载选项 dump选项 fsck选项

dump选项–这一项为0，就表示从不备份。如果上次用dump备份，将显示备份至今的天数。

fsck选项 –启动时fsck检查的顺序。为0就表示不检查，（/）分区永远都是1，其它的分区只能从2开始，当数字相同就同时检查（但不能有两1）】

7.修改对应的用户和权限

    命令： sudo chown -R jason:jason /home/jason/code

                 sudo chmod -R 4755 /home/jason/code

注：chmod使用的数字的意思： 读（r=4），写（w=2），执行（x=1）可读可写为4+2=6 依次内存 755表示的是文件所有者权限7（三者权限之和），与所有者同组用户5（读+执行），其他用户同前一个5，这里的4的意思是（其他）用户执行拥有所有者相同的文件权限（对于其他要使用的文件）

8.重启电脑

命令：sudo reboot

## 24、在ubuntu64位PC机上面运行32位的程序

安装32位的库：

sudo apt-get install ia32-libs

sudo apt-get install lib32c-dev

## 25、ubuntu 16.0.4连接不到网络

（1）、定位问题

1）、如果 ping 8.8.8.8 可以ping通 而 ping www.baidu.com ping不通屏表示网络可以正常连接但是可能DNS没有配置好。

2）、如果ping 8.8.8.8 都ping不通那可能是由于IP和网关没有设置好

（2）、配置IP和网关

sudo vim /etc/network/interfaces //执行命令打开interfaces文件

配置内容如下：

auto lo

　　iface lo inet loopback

　　iface eth0 inet static //其中eth 0是网卡的名字可以通过ifconfig看到一定要对应配置不然开机不会挂//载static是表示设置为静态IP

　　address 192.168.1.183 //配置的IP地址

　　netmask 255.255.255.0 //子网掩码

　　gateway 192.168.1.1 //网关

auto eth0 //开机自动连接

**配置完成记得重启**

（3）、配置DNS

sudo vim /etc/resolvconf/resolv.conf.d/base //执行命令打开文件

配置如下：

nameserver 8.8.8.8

nameserver 8.8.4.4

//多写几个DNS所谓

**配置完成记得重启**

重启之后可以执行命令 cat /etc/resolv.conf 看是否配置成功

nameserver 8.8.8.8

nameserver 8.8.4.4

## 26、shell 脚本

### （1）、问题现象：在window编写的shell脚本linux运行老是报错

原因：

就是由于window和linux的换行符不一样

window下默认是 \r\n

linux下是\n

unix下是\r

解决方法：

使用vim打开文件执行set ff=unix 系统会自动转换。

拓展：

可以通过vim打开文件然后输入命令 :set ff? ，看返回值是什么类型的，比如fileformat=dos

# Android知识

## 开机的时候一直在动画界面开不了机。

### 平台：SC7731C、SC9832 、SC9850

查找问题方法：通过usb进去adb，通过打印底层log或者上层的log，看是否有异常的问题存在。

解决方法：通过查看上层log发现一直在报错说找不到任何的PackageInstaller，所以app当然也就装不上；所以Config.mk中加上 ZYT\_GMS\_PACKAGES=GooglePackageInstaller （当然不一定要是谷歌的，360或者原生的皆可以。）既可以解决问题。

## Android 编译与下载

### Android 编译：

IDH\_PROP\_ZIP=xxxx.zip make -j24

. build/envsetup.sh

lunch

kheader

IDH\_PROP\_ZIP=xxxx.zip make -j24

### SO编译：

编译

修改哪个文件夹就往外退，直到看到Android.mk文件的目录，在这个目录下面执行 mm进行编译。

编译生成路径

idh.code\out\target\product\sp9850ka\_1h10\system\app

idcode\out\target\product\sp9850ka\_1h10\system\priv-app

一般情况下是在这两个目录中的一个apk所在目录的名字和刚刚编译的时候Android.mk所以的目录一致。

下载

adb root

adb remount

adb push apk路径

adb reboot

## 3、发送广播

adb shell am broadcast -a com.dfl.receive.on

adb shell am broadcast -a com.dfl.receive.off

## 4、Android 编译环境搭建——build/envsetup.sh

准备好编译环境后，编译Rom的第一步是 source build/envsetup.sh，该步骤把e\nvsetup.sh里的函数声明为当前会话终端可用的命令。这些命令能让我们切换目录，提交代码，编译Rom更方便。如果记不住所有命令,只要你记住hmm就可以了，也可通过hmm命令看到支持的命令列表。

**1. 命令分类:**

1.1 编译用的命令



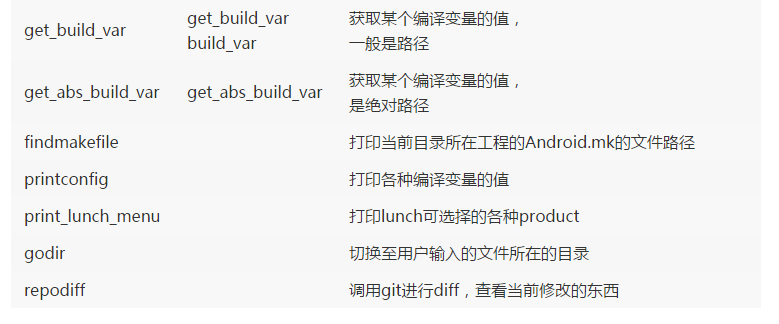






1.2 查看代码时的辅助命令





1.3 辅助函数

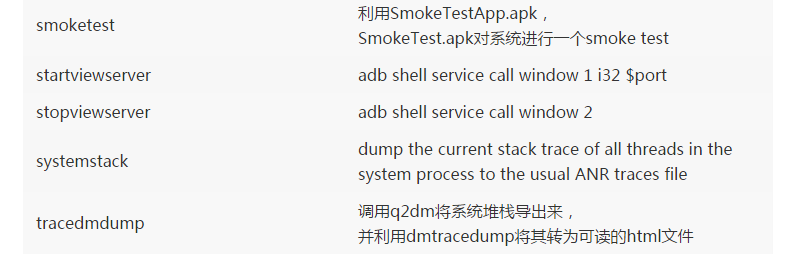






1.4 调试相关





1.5 提交代码相关命令



**2. source build/envsetup.sh 执行流程**

envsetup.sh 定义了很多函数，除此之外还执行了其它操作，代码如下:

VARIANT\_CHOICES=(user userdebug eng)，# TARGET\_BUILD\_VARIANT变量的可能值

#LUNCH\_MENU\_CHOICES是供用户选择的prodcut列表,

#每次source build/envsetup.sh时需重置变量LUNCH\_MENU\_CHOICES

#不然后续的include vendor/cm/vendorsetup.sh时会继续添加产品至变量LUNCH\_MENU\_CHOICES里，

#导致出现很多重复产品

unset LUNCH\_MENU\_CHOICES

add\_lunch\_combo full-eng #默认添加full-eng，full\_mips-eng等4个产品

add\_lunch\_combo full\_x86-eng

add\_lunch\_combo vbox\_x86-eng

add\_lunch\_combo full\_mips-eng

alias bib=breakfast#给breakfast起别名

complete -F \_lunch lunch #给lunch添加tab提示

case `uname -s` in #定义sgrep函数 在所有工程类型代码里搜索

Darwin)

function sgrep()

{

find -E . -name .repo -prune -o -name .git -prune -o -type f \

-iregex '.\*\.(c|h|cpp|S|java|xml|sh|mk)' -print0 \

| xargs -0 grep --color -n "$@"

}

;;

\*)

function sgrep()

{

find . -name .repo -prune -o -name .git -prune -o -type f \

-iregex '.\*\.\(c\|h\|cpp\|S\|java\|xml\|sh\|mk\)' -print0 \

| xargs -0 grep --color -n "$@"

}

;;

esac

export -f cmremote

export -f aospremote

alias mmp='dopush mm' #定义更多编译后push到设备的函数的操作

alias mmmp='dopush mmm'

alias mkap='dopush mka'

alias cmkap='dopush cmka'

if [ "x$SHELL" != "x/bin/bash" ]; then #只支持Bash终端

case `ps -o command -p $$` in

\*bash\*)

;;

\*)

echo "WARNING: Only bash is supported," \

"use of other shell would lead to erroneous results"

;;

esac

fi

#Execute the contents of any vendorsetup.sh files we can find.

#source vendor和device下能找到的所有vendorsetup.sh

for f in `/bin/ls vendor/\*/vendorsetup.sh vendor/\*/\*/vendorsetup.sh device/\*/\*/vendorsetup.sh 2> /dev/null`

do

echo "including $f"

. $f

done

unset f

#source目录 sdk/bash\_completion vendor/cm/bash\_completion下的bash脚本，

#能提供tab提示

addcompletions

export ANDROID\_BUILD\_TOP=$(gettop)

2.1 执行的vendorsetup.sh有：

在envsetup.sh里将执行vendor和device目录及各自子目录下所有的vendorsetup.sh，这些vendorsetup.sh做的事情是调用add\_lunch\_combo将它们各自的产品添加到 LUNCH\_MENU\_CHOICES 变量里

#执行cm的vendorsetup.sh将从网上下载cm支持的产品列表，并添加至LUNCH\_MENU\_CHOICES

vendor/cm/vendorsetup.sh

#将添加mini\_armv7a\_neon产品 add\_lunch\_combo mini\_armv7a\_neon-userdebug

device/generic/armv7-a-neon/vendorsetup.sh

#add\_lunch\_combo mini\_armv7a-userdebug

device/generic/armv7-a/vendorsetup.sh

#add\_lunch\_combo mini\_mips-userdebug

device/generic/mips/vendorsetup.sh

#add\_lunch\_combo mini\_x86-userdebug

device/generic/x86/vendorsetup.sh

#add\_lunch\_combo cm\_jflteatt-eng

device/samsung/jflteatt/vendorsetup.sh

# add\_lunch\_combo full\_panda-userdebug

device/ti/panda/vendorsetup.sh

# add\_lunch\_combo zte\_blade-eng

#add\_lunch\_combo zte\_blade-userdebug

device/zte/blade/vendorsetup.sh

2.2 执行的completion bash有：

在envsetup.sh里将执行sdk/bash\_completion和vendor/cm/bash\_completion目录下的bash脚本，这些bash脚本主要是为命令提供tab支持，有了这些tab支持，输入命令后如果某个选项忘记了，只需要敲tab键，就能获得提示，使用命令更加方便

including sdk/bash\_completion/adb.bash

including vendor/cm/bash\_completion/git.bash

including vendor/cm/bash\_completion/repo.bash

分别对应adb，git，repo的tab提示

## 5、logcat为什么可以输出上层的log

背景：在default状态下调用printf等std C/C++接口输出的log不会被打印到eclipse的logcat中，但是android提供了\_\_android\_log\_print这个函数可以将log重定向到eclipse的logcat中。

1)

在对应的mk文件中加入:LOCAL\_LDLIBS := -llog

2)

在要使用LOG的cpp文件中加入：

#include <android/log.h>

使用方法如下;

\_\_android\_log\_print(ANDROID\_LOG\_DEBUG,"Tag", \_\_VA\_ARGS\_\_)

ANDROID\_LOG\_DEBUG: log的level

Tag：module的tag

\_\_VA\_ARGS\_\_:格式化参数列表

例子：

\_\_android\_log\_print(ANDROID\_LOG\_DEBUG,"libAirplay", "service URI : %s", "www.163.com.video.xixi.mp4");

3)

直接使用并不是个好的想法，如果能够可以和eclipse一样实现分等级打印，那还不错；实现如下：

#define LOGV(...)\_\_android\_log\_print(ANDROID\_LOG\_VERBOSE, " Tag", \_\_VA\_ARGS\_\_) // VERBOSE

#define LOGD(...) \_\_android\_log\_print(ANDROID\_LOG\_DEBUG , " Tag ", \_\_VA\_ARGS\_\_) // DEBUG

#define LOGI(...) \_\_android\_log\_print(ANDROID\_LOG\_INFO , " Tag ",\_\_VA\_ARGS\_\_) // INFO

#define LOGW(...) \_\_android\_log\_print(ANDROID\_LOG\_WARN , " Tag ", \_\_VA\_ARGS\_\_) //WARN

#define LOGE(...) \_\_android\_log\_print(ANDROID\_LOG\_ERROR , " Tag ",\_\_VA\_ARGS\_\_) // ERROR

注意：括号中是省略号，不用改为参数.

在需要的地方直接使用

LOGV(“www.163.com.video.xixi.mp4”);

或者LOGD(“www.163.com.video.xixi.mp4”); 就好

## 6、服务器开启ccache 加快编译速度

参考文档：https://www.jianshu.com/p/43561051d360

第一步、安装ccache工具

在Android源代码目录prebuilts/misc/linux-x86/ccache下面有ccache工具的源代码包拷出来(建议根目录)，把它解压，然后参考里面的INSTALL.txt文档。操作如下命令(进入到解压的文件里面执行)：

./configure

make

sudo make install

第2步：配置ccache环境

在~/.bashrc中添加(或者/etc/profile文件中)：

export USE\_CCACHE=1

export CCACHE\_DIR=~/.ccache

注：配置.bashrc后注意source该文件，否则cache（缓存）会保存在~/.ccache目录下，而不是你设置的目录。

source ~/.bashrc

第3步： 使用Android源码prebuilts目录下面的ccache工具初始化该文件夹

ccache -M 60G

注：这里的大小设置跟代码的份数相关，建议一份代码保留20G空间，这里60G就是三份代码。

第4步：你可以查看ccahe使用情况，看看上面的设置是否有效

ccache -s

注：如果满了可以用ccache -C清除所有。在满了状态下编译新codebase，将会旧的那个清除，这会降低效率，所以cache一定要设置正确。如果编译中发现速度异常，请关注一下cache的大小。

第5步：开启kernel ccache

目前我尝试开启后没有任何提升，可以先不管，40%已经提升很大了。后续有人研究好了可以再通报出来。

第6步：多人共用的一台服务器时的处理

以上配置是针对一个登陆用户有效的方案，如果比如大服务器上有多个人用不同的用户登陆，处理有所不同。由于多人同时写ccache缓存可能引起IO带宽不够反而降低效率，强烈建议多人共用一台服务器时，一个人来写缓存内容，其他共享用户仅只读缓存内容。

## 7、Adroid.mk的编写

### 一、Android.mk介绍

Android.mk是Android提供的一种makefile文件，用来指定诸如编译生成so库名、引用的头文件目录、需要编译的.c/.cpp文件和.a静态库文件等。要掌握jni，就必须熟练掌握Android.mk的语法规范。

举一个简单的例子：

LOCAL\_PATH := $(call my-dir)

include $(CLEAR\_VARS)

................

LOCAL\_xxx := xxx

LOCAL\_MODULE := hello-jni

LOCAL\_SRC\_FILES := hello-jni.c

LOCAL\_xxx := xxx

................

include $(BUILD\_SHARED\_LIBRARY)

### 二、一些常用宏的含义

**LOCAL\_PATH** := $(call my-dir)

每个Android.mk文件必须以定义LOCAL\_PATH为开始。它用于在开发tree中查找源文件。宏my-dir 则由Build System提供。返回包含Android.mk的目录路径。

**include $(CLEAR\_VARS)**

CLEAR\_VARS 变量由Build System提供。并指向一个指定的GNU Makefile，由它负责清理很多LOCAL\_xxx.

例如：LOCAL\_MODULE, LOCAL\_SRC\_FILES, LOCAL\_STATIC\_LIBRARIES等等。但不清理LOCAL\_PATH.

这个清理动作是必须的，因为所有的编译控制文件由同一个GNU Make解析和执行，其变量是全局的。所以清理后才能避免相互影响。

**LOCAL\_MODULE**  := hello-jni

LOCAL\_MODULE模块必须定义，以表示Android.mk中的每一个模块。名字必须唯一且不包含空格。Build System会自动添加适当的前缀和后缀。例如，foo，要产生动态库，则生成libfoo.so. 但请注意：如果模块名被定为：libfoo.则生成libfoo.so. 不再加前缀

**LOCAL\_MODULE\_PATH** :=$(TARGET\_ROOT\_OUT) 指定最后生成的模块的目标地址

TARGET\_ROOT\_OUT:根文件系统，路径为out/target/product/generic/root

TARGET\_OUT:system文件系统，路径为out/target/product/generic/system

TARGET\_OUT\_DATA:data文件系统，路径为out/target/product/generic/data

除了上面的这些，NDK还提供了很多其他的TARGET\_XXX\_XXX变量，用于将生成的模块拷贝到输出目录的不同路径默认是TARGET\_OUT

**LOCAL\_MODULE\_RELATIVE\_PATH** := npidevice 指定最后生成的模块的目标地址

LOCAL\_MODULE\_RELATIVE\_PATH 是指路径 ：out/target/product/s9863a1h10/vendor/lib/

后面接的npidevice既会在lib下面创建一个文件名字叫做 npidevice然后把生成的so放到这个目录下面 ：

例如：out/target/product/s9863a1h10/vendor/lib/npidevice/autotestfinger.so

**LOCAL\_SRC\_FILES :**= hello-jni.c

LOCAL\_SRC\_FILES变量必须包含将要打包如模块的C/C++ 源码。不必列出头文件，build System 会自动帮我们找出依赖文件。缺省的C++源码的扩展名为.cpp. 也可以修改，通过LOCAL\_CPP\_EXTENSION

**include $(BUILD\_SHARED\_LIBRARY)**

BUILD\_SHARED\_LIBRARY：是Build System提供的一个变量，指向一个GNU Makefile Script。

它负责收集自从上次调用 include $(CLEAR\_VARS) 后的所有LOCAL\_XXX信息。并决定编译为什么。

BUILD\_STATIC\_LIBRARY ：编译为静态库。

BUILD\_SHARED\_LIBRARY ：编译为动态库

BUILD\_EXECUTABLE ：编译为Native C可执行程序

BUILD\_PREBUILT ：该模块已经预先编译

NDK还定义了很多其他的BUILD\_XXX\_XXX变量，它们用来指定模块的生成方式。

**LOCAL\_CFLAGS += -DXXX** 相当于在所有源文件中增加一个宏定义#define XXX

例如：

Android.mk中增加

ifeq ($(PRODUCT\_MODEL),XXX\_A)

LOCAL\_CFLAGS += -DBUILD\_MODEL1

endif

所编译的Cpp中如下使用：

#ifdef BUILD\_MODEL1

....

#endif

**LOCAL\_C\_INCLUDES** 用来设置头文件的include目录

一般写成：

LOCAL\_C\_INCLUDES := $(LOCAL\_PATH)/../../Classes \

$(LOCAL\_PATH)/../../Classes/game \

$(LOCAL\_PATH)/../../Classes/logic \

$(LOCAL\_PATH)/../../Classes/view

小技巧：

有一个目录就要写一行, 实在繁琐, 有没有写法可以把源码目录下的所有子目录都引入呢, 看下面

LOCAL\_C\_INCLUDES := $(LOCAL\_PATH)/../../Classes

LOCAL\_C\_INCLUDES += $(shell ls -FR $(LOCAL\_C\_INCLUDES) | grep $(LOCAL\_PATH)/$ )

LOCAL\_C\_INCLUDES := $(LOCAL\_C\_INCLUDES:$(LOCAL\_PATH)/%:=$(LOCAL\_PATH)/%)

即可把$(LOCAL\_PATH)/../../Classes目录和子目录全部包含进来

**LOCAL\_SHARED\_LIBRARIES**:

表示模块在运行时要依赖的共享库（动态库），在链接时就需要，以便在生成文件时嵌入其相应的信息。

**LOCAL\_STATIC\_LIBRARIES ：**

应该链接到这个模块的静态库列表（使用 BUILD\_STATIC\_LIBRARY 生成），这仅仅对共享库模块才有意义。

**LOCAL\_COPY\_HEADERS\_TO 和 LOCAL\_COPY\_HEADERS**

在使用Android編譯的庫，需要include這個庫的頭文件

Android編譯時會包含out/target/product/generic/obj/include/ 下的頭文件，LOCAL\_COPY\_HEADERS\_TO和LOCAL\_COPY\_HEADERS的作用就是把需要暴露給外部的頭文件copy到這個路徑下

其中

LOCAL\_COPY\_HEADERS\_TO 指定out/target/product/generic/obj/include/ 下的一個目錄

LOCAL\_COPY\_HEADERS 指定要 copy 的頭文件

**LOCAL\_MODULE\_TAGS**

给他赋予的值有以下几种含义：

user: 指该模块只在user版本下才编译

eng: 指该模块只在eng版本下才编译

tests: 指该模块只在tests版本下才编译

optional:指该模块在所有版本下都编译

# 通讯协议

## 1、MIPI相关知识

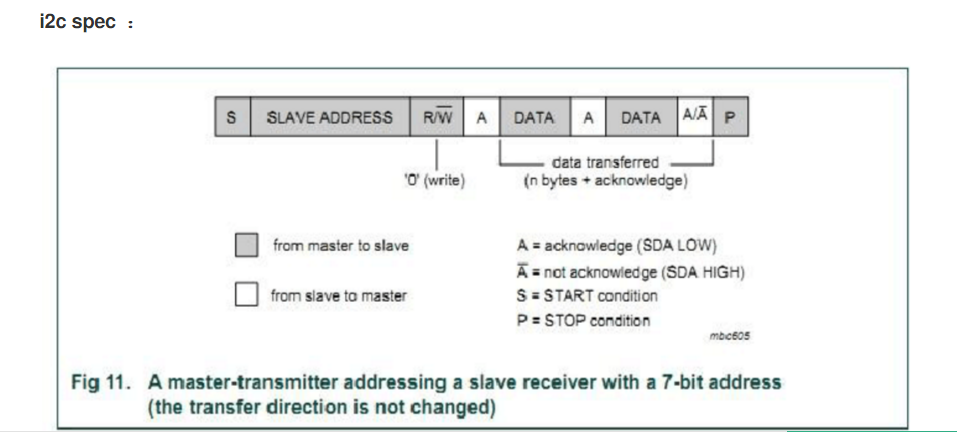
（1）、CSI和DPI分别代表什么：

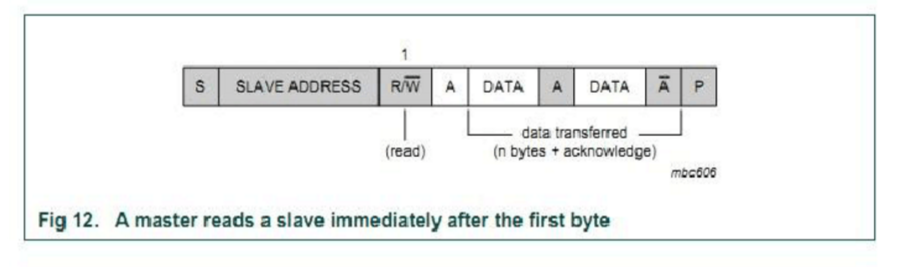
MIPI CSI和DPI是MIPI标准的一种，因为MIPI在移动领域应用范围太广了，各种外围设备都可以用它来传输信息，所以，MIPI联盟给不同的外设接口定义了版本名，CSI是for Camera的，DPI是for Display的。而且这些标准不单包括物理层的时序定义，还包括上层的传输协议/数据处理协议和应用层协议。

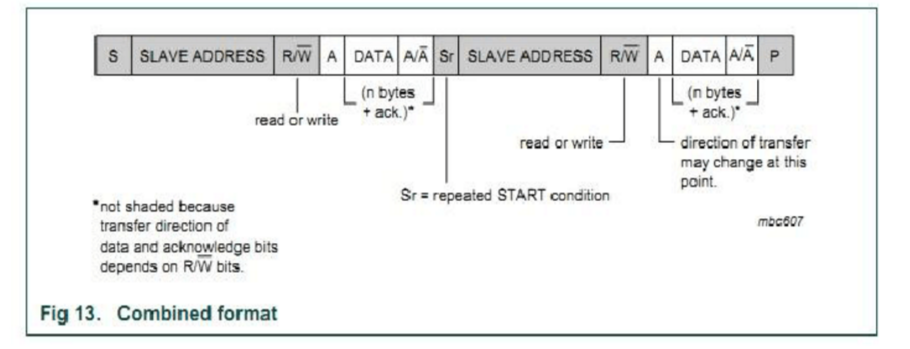
## 2、IIC通讯协议

**1）、I2C通讯及波形讲解**

**概念：**







The Acknowledge signal is defined as follows: the transmitter releases the SDA line (Wei: put SDA at 1 state)  during the acknowledge clock pulse so the receiver can pull the SDA line LOW and it

remains stable LOW during the HIGH period of this clock pulse  (Wei: if the high of clock is hold on, SDA is kept at LOW!! )(see Figure 4). Set-up and

hold times (specified in Section 6) must also be taken into account.

**I2C设备调试及波形分析**

最新在开发OLED屏驱动，需要用到I2C总线，下面大体上讲解一下I2C设备的调试及波形分析，为大家做一些参考，由于刚涉及这部分内容，因此有什么错误的，还请赐教~

 一、概要

I2C总线只需要两条线，一条SDA数据线，一条SCL时钟线；根据这两条线的高低电平、上升沿、下降沿就可以实现主机与I2C设备的通讯；其中有：

（1）I2C总线相关

传输开始条件：SCL处于高电平，SDA下降沿时； 传输接收条件：SCL处于高电平，SDA上升沿时；

传输数据：开始传输后，SCL处于高电平时，SDA的数据为所传输的数据；

回应：当传输完一个字节后，I2C设备需要回应一个ACK，这样主机才继续发送；因此回应信号是在传输完8bit后的下一个数据位（SDA值），当SDA为0表示有回应，为1表示没回应；

正常I2C总线的数据是：Start + I2C devece id + R/W + ACK + Data（first byte）+ ACK + ... + Data（n）+ ACK + Stop

（2）I2C设备相关

设备地址：有7位和10位两种，具体见I2C设备芯片的DataSheet，由于目前用到的是7位，因此下面主要针对7位讲述；在讲I2C设备地址是有可能有两种说法，主要是用8位表示还是用7为表示，比如对于我的OLED来说，当用8位表示时则为0x78地址，当用7位时则为0x3c（即0x78右移1为），在驱动中用0x78还是用0x3c要看具体平台的I2C总线驱动，我在AMLOGIC平台上用的是0x78，而在MV平台上用的是0x3c；

寄存器reg：一般的I2C设备芯片都有带reg，一般在传输正式的数据之前需要先传输reg地址，比如我的OLED来说，在传控制命令时需要先发送0x00的reg地址，在传输数据时需要发送0x40的reg地址；

**二、调试及波形分析**

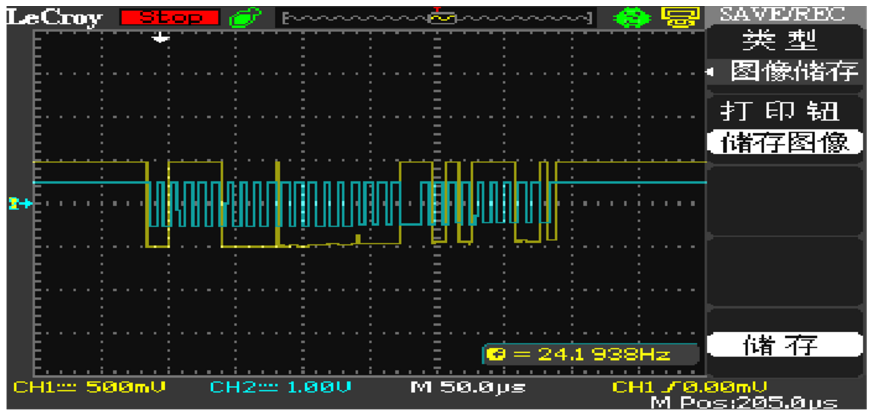
一般当我们拿到一个I2C设备时，就必须涉及到驱动的编写，就比如对于OLED来说，就要用编写OLED驱动，这样我们才能控制它，对于OLED屏来说，第一步也是最重要的一步就是点亮它；当我们做完这一步，那后面剩下的就只是细节问题了；”万事开头难“，这句话真的不假，对于OLED来说，如何才能点亮，我们该怎么调试呢？当我们写完OLED驱动，但OLED屏还是不亮，可能问题会出现在哪？是硬件问题还是软件问题？；若为软件问题，那会是I2C总线驱动问题，还是我们I2C设备驱动有问题？那么我们该如何判断问题出现位置呢？这就需要我们对I2C总线上的数据进行分析；那么下面我将详细讲述如何获取和分析I2C总线上的数据；

1）示波器

对于I2C总线的数据，我们要用到示波器，这样我们才能抓取到信号，而且必须同时采集SDA和SCL的数据；该如何抓取呢？我这边的方式是将示波器调成边下降沿触发模式（因为开始信号是SDA下降沿），并且设置成单次模式（这样抓取完一次就会stop，便于我们数据分析）；

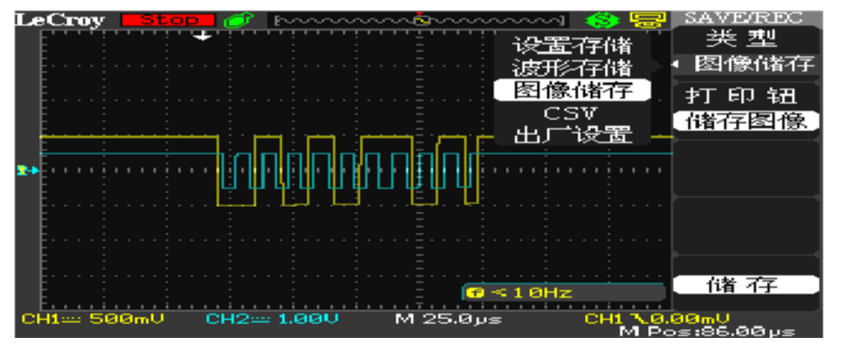
（2）波形

由于我的OLED设备的地址是0x78(8位)，而第一次我必须将OLED设置成off状态，通过命令表可以查到，必须发生0xAE，而刚刚有讲到在发送命令前必须先发送寄存器reg地址，即0x00；于是我发送的数据为0x78+0x00+0xAE；下面是我用示波器抓取到的波形：



分析（其中黄色的为SDA信号，浅蓝色为SCL信号）：从上述波形中我们可以读取到数据依次是：01111000 0 00000000 101011100；即：0111100(7位OLED设备地址) + 0( 读写为，0为写，1为读)+0（ACK回应）+00000000（寄存器reg）+ 10101110(0xAE OLED off命令)；（这里还有一点我还没弄明白就是reg后没有回应，还望大家不吝赐教）

下面当我来随便发送一个地址时，比如当我发送0x56（8位地址），测到的波形如下：



分析：从波形可看出数据依次为：010101101 即0101011（设备地址）+0（读写位）+1（NACK无应答），由于没有相应的I2C设备因此无应答ACK信号，于是就停止传输；

三、总结

由以上可知，可以通过查看I2C总线数据来确认跟踪I2C设备，已更好地确定是软件问题还是硬件问题；

**用示波器对单片机I2C时序进行图形波形分析的试验小结**

I2C要求要有一个主设备，负责发起请求和控制时钟；其它为从设备，通过设备ID地址来识别并响应主设备请求。主从设备要轮流控制SDA。一开始我没搞明白这一点，直接加了写I2C数据代码，然后用示波器在SDA和SCL脚测量，却只能找到些凌乱的波形，没有预期的效果。后来把从设备接上，两边写好代码，互相有了响应，这才在示波器上看到波形。

这里我找了一个主设备往从设备写数据的例子，代码如下：

char buf[128]; int len;

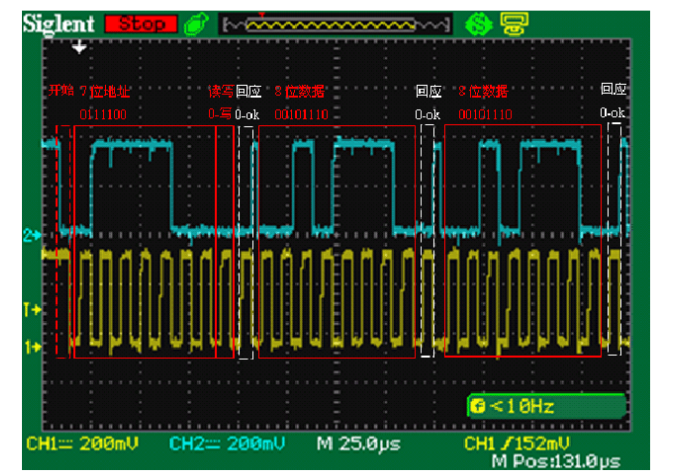
strcpy(buf,"..huz\_hello\_i2c/n"); len=strlen(buf);

//deviceid: 0x3c

write\_i2c(0x3c, buf , len);

接收端的代码比较简单，就不贴了。

将示波器的X和Y分别接到SDA和SCL，得到波形并分析如图：



从图中可知时序如下：

1. 由主机发起，在SCL为高电平时，SDA由高到低切变，形成开始信号；

2.

接着是7位地址和一位读写标志，这里7位地址为0111100，即0x3c，正是我们代码中设置的地址ID；最后一位为0表示写操作；

3. 接着在下一个时钟，主机以高电平状态释放SDA，这时从机响应，将SDA拉低了； 4. 接着是两个8位数据00101110与响应，即0x2E，正是“.”号的ASCII码，符合预期输出； 5.

还有其它数据和最后的停止位，图中被截掉了。

从图中可知，纵向一格是200mV，则SDA和SCL的电平大概就是350mV；由于信号笔上设置了信号x10，因此实际电平应该大概是3.5V（理论上应该是3.3V）。横向一格是25us，10个时钟周期大概用了4格，即4x25us=100us，平均每个时钟周期是10us，可算出传输频率为1/10us=100,000/s，即100k bps。

另外，对于读从设备内容，基本流程是主设备先往从设备写一个命令，然后再输出读取命令，然后才由从设备发送数据。过程类似，不再具体分析了。

下图示例中，主机先向从机写了一个地址命令，然后重新开始并进入读取周期。



# C语言知识

## （1）sprintf

注意：会在末尾自动加上\0

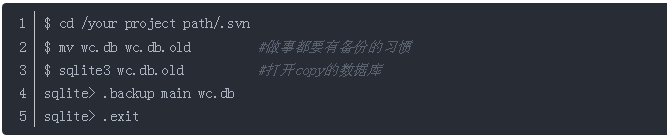
（2）对于宏的操作

1)、#undef A 不定义宏A

# SVN使用

## svn 报错

### （1）、svn: E200033: database is locked



cd /your project path/.svn

mv wc.db wc.db.old

sqlite3 wc.db.old

sqlite> .backup main wc.db

sqlite> .exit

（2）、

## 2、当前版本回滚到指定版本：

svn merge -r 7008:7007 .

# GitHub知识小计

## 下载方法：

Mac：https://sourceforge.net/projects/git-osx-installer/

Windows：https://git-for-windows.github.io/

Linux：apt-get install git

## 1、相关文档

learn-github-from-zero.pdf

## 知名GIT账号

Google: https://github.com/google

苹果: https://github.com/apple

Facebook: https://github.com/facebook

T witter：https://github.com/twitter

微软：https://github.com/microsoft

Square：https://github.com/square

阿里：https://github.com/alibaba

Linux：https://github.com/torvalds/linux

Rails：https://github.com/rails/rails

Nodejs：https://github.com/nodejs/node

Swift：https://github.com/apple/swift

CoffeeScript：https://github.com/jashkenas/coffeescript

Ruby：<https://github.com/ruby/ruby>

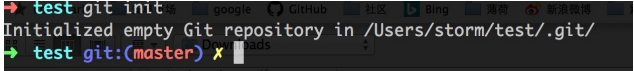
Linux 发明者 Linus T orvalds：https://github.com/torvalds

安卓JakeWharton：https://github.com/JakeWharton

Rails 创始人 DHH：https://github.com/dhh

## 3、相关命令

**git init**  初始化 git 仓库



可以看到初始化成了，至此 test 目录已经是一个 git 仓库了。

**git status** 查看状态 相当于svn的svn status

**git add 文件** 把相应的文件加入到git的本地仓中

**git commit** git commit -m 'first commit' ，这个命令什么意思呢？ commit 是提交的意思，-m 代表是提交信息，执行了以上命令代表我们已经正式进行了第一次提交。

## 4、与远程库之间的操作

**（1）、设置邮箱和用户名（没有设置提交的时候有问题）：**

git config --global user.name "yourname" 注释：这个名字要和GitHub上一样

git config --global user.email“your@email.com" 注释：邮箱设置一个自己的邮箱即可

如果忘记自己的用户名和邮箱可以通过指令查看：

git config --global user.name

git config --global user.email

**（2）、创建公钥私钥**

ssh-keygen -t rsa -C "username" (注：username为你git上的用户名)

注释：

Generating public/private rsa key pair.

Enter file in which to save the key (/Users/username/.ssh/id\_rsa): //这里的username是电脑上的用户名，这个地址也是文件的存储地址，然后我们按

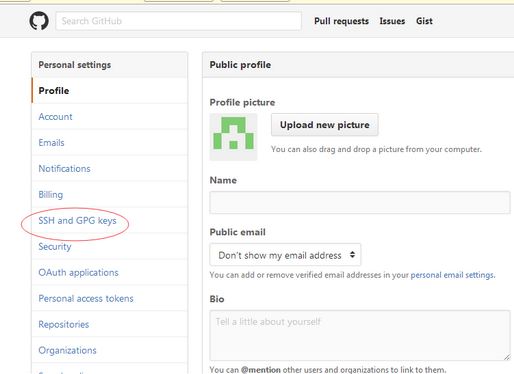
回车，如果你以前有存储地址会返回/Users/your username/.ssh/id\_rsa already exists.Overwrite (y/n)?直接输入y回车。如果以前没有储存地址就会出现

Enter passphrase(empty for no passphrase);也直接回车，两种情况回车后都会出现 Enter same passphrase again 然后接着回车会显示一长串内容其中

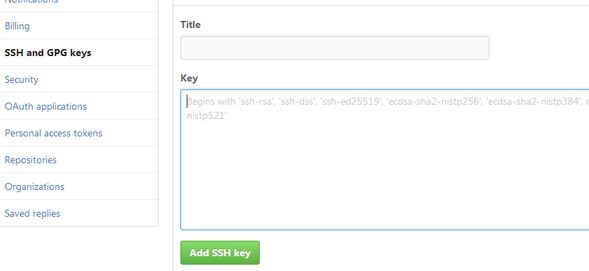
还有一些..o.. o oo .oS. 之类的代码，这说明SSH key就已经生成了。文件目录就是：username/.ssh/id\_rsa.pub.

**（3）、然后找到.ssh文件夹（一般路径：/Users/username/.ssh 也可以看提示）；在.ssh文件夹下面会有两个文件** **id\_rsa和****id\_rsa.pub 其中 id\_rsa是私钥 id\_rsa.pub 是公钥；打开 id\_rsa.pub 复制里面的公钥。**

**（4）、打开https://github.com/，登陆你的账户，进入设置（Settings）找到**



然后将你复制的内容粘贴到key中，其中title可以不用写



再点击Add SSH Key

**（5）、测试是否连接GitHub成功**

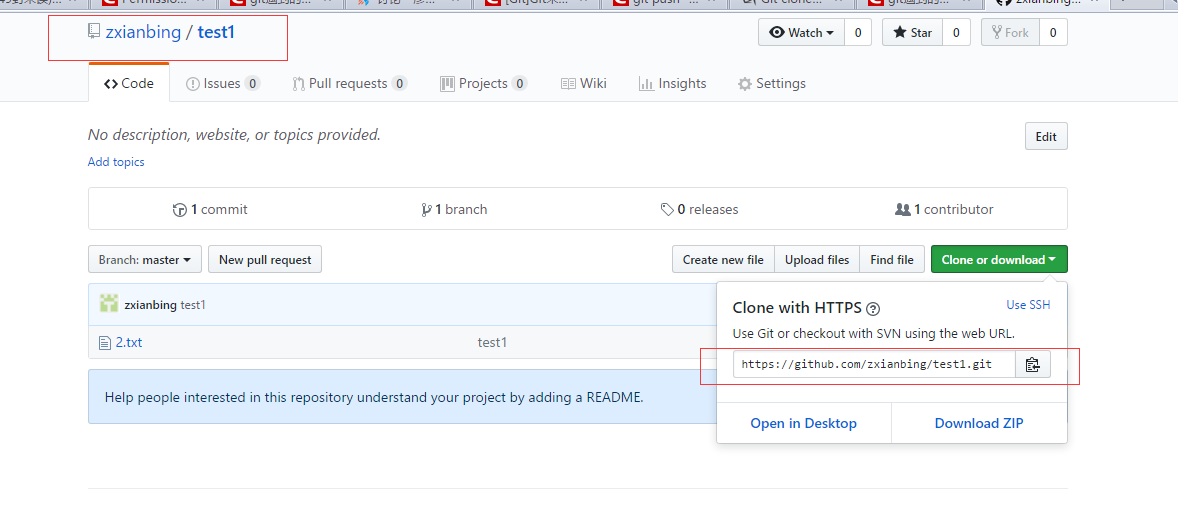
ssh -T [git@github.com](mailto:git@github.com) 注意:执行这个语句有的是直接提示成功，有些会让你输入yes或no 直接输入yes即可。

**（6）、建立本地库和远程库的连接 （执行这步后会在.ssh文件夹下生成****文件）**

1）从GitHub直接克隆一个库到本地

git clone [git@github.com:stormzhang/test.git](mailto:git@github.com:stormzhang/test.git)

注释：其中后面的是相应项目的地址



经过验证从里面得到的地址是<https://github.com/zxianbing/test1.git> 需要手动改为 [git@github.com:zxianbing/test1.git](mailto:git@github.com:zxianbing/test1.git) 格式

2）把本地已有的代码提交到远程库

第一步就是在 GitHub 上建一个 test 项目。

第二步把本地 test2 项目与 GitHub 上的 test 项目进行关联，切换到 test2 目录，执行如下命

令：

git remote add origin [git@github.com:stormzhang/test.git](mailto:git@github.com:stormzhang/test.git) 同样后面的地址和上面的获取方法是一样的。当关联后就可以做push pull的相关操作了。

（7）、如果怀疑自己公钥有问题，需要先删除文件，然后按上面步骤重新再来一次。

## 5、git 的一些配置

（1）、git log 配置为中文

git config --global i18n.commitencoding utf-8

git config --global i18n.logoutputencoding utf-8

（2）、git status 中午乱码

git config --global core.quotepath false

# cmder 使用

## 1、快捷键

（1）、win +Alt+p 打开setting

（2）Ctrl+T 建立新页签；

（3）Ctrl+W 关闭页签;

（4）Ctrl+Tab 切换页签

（5）Alt+F4 关闭所有页签

（6）Alt+Shift+1 开启cmd.exe

（7）Alt+Shift+2 开启powershell.exe

（8）Alt+Shift+3 开启powershell.exe (系统管理员权限)

（9）Ctrl+1 快速切换到第1个页签

（10）Ctrl+n 快速切换到第n个页签( n值无上限)

（11）Alt + enter 切换到全屏状态

（12）Ctr+r  历史命令搜索;

（13）End, Home, Ctrl : Traversing text with as usual on Windows

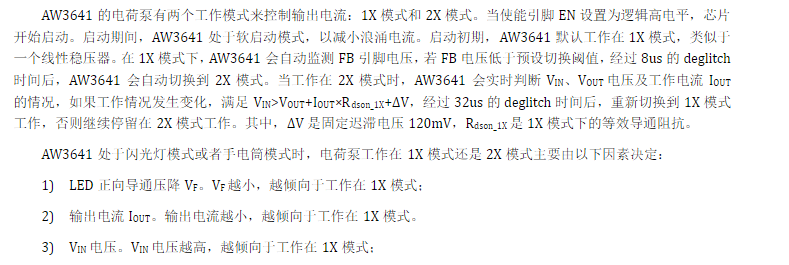
# 展讯MTK通用

## 1、闪光灯 AW3641

### （1）、引脚定义及功能 ：



### （2）、自适应电荷泵



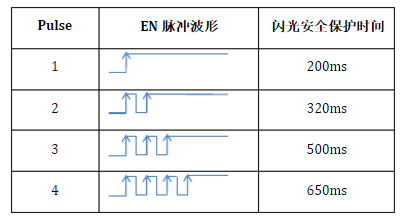
### （3）、PWM调光



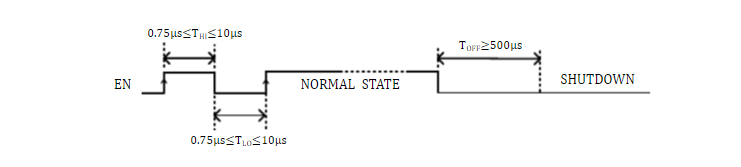


### （4）、闪光安全保护时间设置

在闪光灯模式下，为了防止故障情况下闪光灯持续大电流工作造成可靠性风险，同时保证不同应用条件下闪光灯的曝光时间满足应用要求，AW3641内置了闪光安全保护时间调节功能。闪光灯模式下，默认的闪光时间设置是200ms。在EN引脚输入一线脉冲信号，可以调节闪光安全保护时间。一线脉冲上升沿的个数和闪光安全保护时间的对应关系如下表



下图为一线脉冲的时序要求，其中THI为脉冲的高电平宽度，推荐值为2μs；TLO为脉冲的低电平宽度，推荐值为2μs；为一线脉冲的时序要求，其中THI为脉冲的高电平宽度，推荐值为2μs；TLO为脉冲的低电平宽度，推荐值为2μs；为一线脉冲的时序要求，其中THI为脉冲的高电平宽度，推荐值为2μs；TLO为脉冲的低电平宽度，推荐值为2μs；



### （5）、过温保护



# 360相关

## （1）、360推送更新代码指令

# VS2019相关技术（C、C++）

## 1、快捷键

ctrl + F7 编译

## 2、编译报错

### （1）、关于SDL会提示的相关报错（安全开发生命周期（SDL）检查）

|  |  |  |
| --- | --- | --- |
| **/sdl 启用警告** | **等效的命令行开关** | **描述** |
| [C4146](http://msdn.microsoft.com/zh-cn/library/4kh09110.aspx) | /we4146 | 一元负运算符应用于无符号类型，从而导致无符号结果。 |
| [C4308](http://msdn.microsoft.com/zh-cn/library/5ft2cz8d.aspx) | /we4308 | 一个负整型常数转换为无符号类型，从而导致一个可能无意义结果。 |
| [C4532](http://msdn.microsoft.com/zh-cn/library/h6b9te7b.aspx) | /we4532 | **\_\_finally** /**finally**中的关键词，使用**continue**， **break** 或 **goto**在异常终止块未定义行为。 |
| [C4533](http://msdn.microsoft.com/zh-cn/library/f7687yks.aspx) | /we4533 | 初始化变量的代码不会执行。 |
| [C4700](http://msdn.microsoft.com/zh-cn/library/axhfhh6x.aspx) | /we4700 | 使用未初始化的局部变量。 |
| [C4703](http://msdn.microsoft.com/zh-cn/library/jj851030.aspx) | /we4703 | 对一个潜在的未初始化的局部指针变量的使用。 |
| [C4789](http://msdn.microsoft.com/zh-cn/library/w0c0bww3.aspx) | /we4789 | 当使用时，请缓冲区溢出特定 C 运行时 (CRT) 函数。 |
| [C4995](http://msdn.microsoft.com/zh-cn/library/8wsycdzs.aspx) | /we4995 | 使用函数的标[deprecated](http://msdn.microsoft.com/zh-cn/library/c8xdzzhh.aspx)。 |
| [C4996](http://msdn.microsoft.com/zh-cn/library/ttcz0bys.aspx) | /we4996 | 使用函数的标记作为[deprecated](http://msdn.microsoft.com/zh-cn/library/044swk7y.aspx)。 |

**如何关闭SDL：** 点击项目->点击相应工程的属性（看你建的工程是哪一个了） -> C/C++ ->SDL检查改为否