# 一个内核模块至少包含两个函数

初始化函数，在模块加载到内核时被调用。通过宏 module\_init()声明初始化函数。

卸载函数，在内核模块被卸载时被调用。 通过宏module\_exit()声明卸载函数。

# Makefile的编写

obj-m+= module.o //添加需要编译的模块

make -C /home/sice/workspace/linux-3.5.0 M=`pwd` modules

(内核所需的位置) (需要编译模块的位置)

# Linux内核链表

Linux内核的标准链表就是采用环形双向链表形式实现的

# Linux内核链表和普通链表的区别

普通链表是 将数据结构塞入到节点指针中

Linux内核链表是 将链表节点塞入到数据结构中

# Linux内核链表的实现

struct list\_head{

struct list\_head \*next; //next指针指向下一个链表节点

struct list\_head \*prev; //prev指针指向前一个链表节点

};

struct name{

数据

struct list\_head list;

};

# Linux链表的几个函数和宏

|  |  |
| --- | --- |
| 函数原型 | container\_of(ptr, type, member) |
| 函数作用 | 可以很方便地从链表指针找到父结构中包含的任何变量 |
| 参数的含义 | ptr: list\_head节点指针 |
| type:父结构的类型 |
| member：list\_head结构在夫结构中的成员名 |
| 使用实例 | struct fox \*p = container\_of(ptr,struct fox,list);  已知一个ptr指针指向节点 |

|  |  |
| --- | --- |
| 宏原型 | LIST\_HEAD(name); |
| 宏作用 | 创建链表 |
| 参数的含义 | ptr: list\_head节点指针 |

|  |  |
| --- | --- |
| 函数原型 | 将new链表添加到head中  void list\_add(struct list\_head \*new,struct list\_head \*head);  将new链表添加到head的尾部中  void list\_add\_tail(struct list\_head \*new, struct list\_head \*head); |
| 函数作用 | 添加节点 |
| 参数的含义 | new是要添加的节点 |
| head是被添加的链表 |
| 使用实例 | /\* 将one 添加到链表student\_list中 \*/  list\_add(&one->list,&student\_list);  /\* 将one添加到链表student\_list的尾部 \*/  list\_add\_tail(&one->list,&student\_list); |

|  |  |
| --- | --- |
| 函数原型 | 被删除的节点的指针域还指向相应的前、后节点  void list\_del(struct list\_head \*entry);  删除节点后，初始化节点的指针域，使之不再指向链表  void list\_del\_init(struct list\_head \*entry); |
| 函数作用 | 删除节点 |
| 参数的含义 | entry指向要删除的节点 |
| 使用实例 | list\_del\_init(student\_list.prev); |

|  |  |
| --- | --- |
| 函数原型 | void list\_replace(struct list\_head \*old,struct list\_head \*new);  void list\_replace\_init(struct list\_head \*old,struct list\_head \*new); |
| 函数作用 | 替换节点 |
| 参数的含义 | old指向旧节点（被替换的） |
| new指向新节点 |

|  |  |
| --- | --- |
| 函数原型 | void list\_move(struct list\_head \*list, struct list\_head \*head)  void list\_move\_tail(struct list\_head \*list,struct list\_head \*head); |
| 函数作用 | 移动节点 |
| 参数的含义 | 节点list移动到链表head中 |

|  |  |
| --- | --- |
| 函数原型 | int list\_empty(struct list\_head \*head); |
| 函数作用 | 判断链表是否为空 |
| 参数的含义 | head: list\_head节点指针 |
| 使用实例 | list\_empty(&student\_list) |

|  |  |
| --- | --- |
| 函数原型 | int list\_is\_singular(struct list\_head \*head); |
| 函数作用 | 判断链表是否只有一个元素 |
| 参数的含义 | head: list\_head节点指针 |

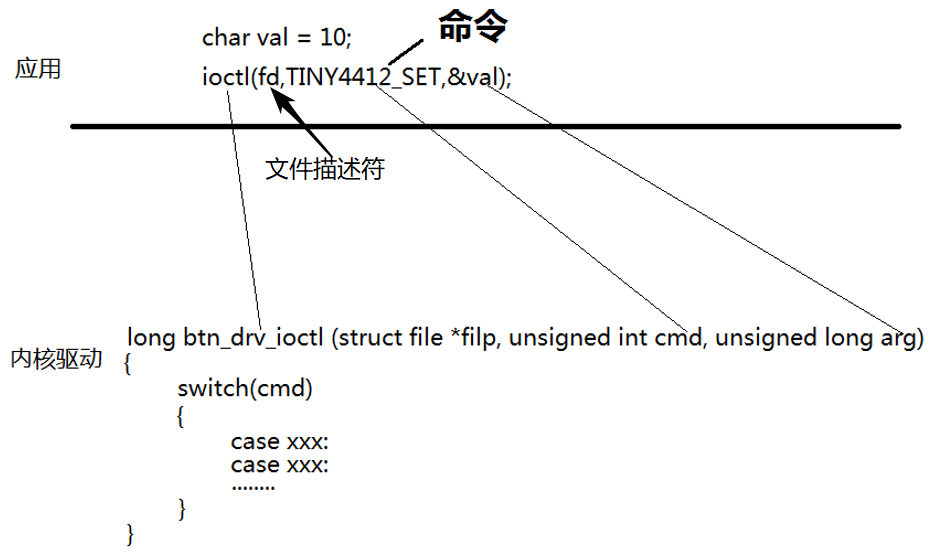
|  |  |
| --- | --- |
| 函数原型 | 将new链表添加到head中  void list\_add(struct list\_head \*new,struct list\_head \*head);  将new链表添加到head的尾部中  void list\_add\_tail(struct list\_head \*new, struct list\_head \*head); |
| 函数作用 | 添加节点 |
| 参数的含义 | new是要添加的节点 |
| head是被添加的链表 |
| 使用实例 | /\* 将one 添加到链表student\_list中 \*/  list\_add(&one->list,&student\_list);  /\* 将one添加到链表student\_list的尾部 \*/  list\_add\_tail(&one->list,&student\_list); |

|  |  |
| --- | --- |
| 宏原型 | list\_for\_each(pos, head)  list\_for\_each\_entry(pos, head, member) |
| 宏作用 | 遍历节点 |
| 参数的含义 | pos表示需要遍历的链表 |
| head表示需要遍历链表的表头 |
| member表示pos中需要遍历的子项 |
| 使用实例 | list\_for\_each\_entry(one,&student\_list,list) |
| 两者区别 | list\_for\_each这个宏必须要把list\_head放在数据结构的第一项成员，而list\_for\_each\_entry可以把list\_head放在任意位置上。  list\_for\_each\_entry这是宏是遍历链表中的某一个子项 |

# 什么时候使用ioctl

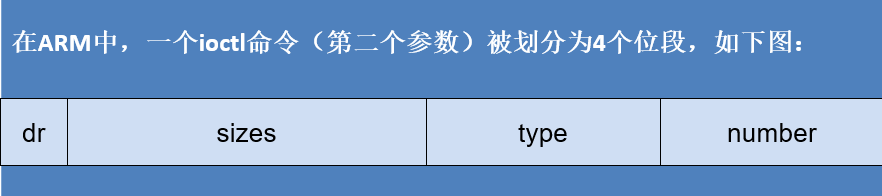
通过设备驱动程序执行各种类型的硬件控制，用户空间经常会请求设备锁门、弹出介质、报告错误信息、改变波特率或者执行破坏等等。

# ioctl参数传递图



|  |  |
| --- | --- |
| 函数原型 | int ioctl(int fd,int cmd, ...); |
| 函数作用 | 从用户空间传递数据到内核空间 |
| 参数的含义 | fd:文件描述符 |
| cmd:控制命令 |
| …：传递给内核的数据 |
| 使用实例 | ioctl(fd,TINY4412\_BTN\_IOGET,&val); |

# ioctl命令编号的构造



**构造命令编号**

|  |  |
| --- | --- |
| 构造无参数的命令号 | \_IO(type,nr) |
| 构造从驱动程序中读取数据的命令号 | \_IOR(type,nr,datetype) |
| 构造向驱动程序中写入数据的命令号 | \_IOW(type,nr,datatype) |
| 构造双向传输的命令号 | \_IORW(type,nr,datatype) |
| //定义一个幻数（type）  #define TINY4412\_BTN\_TYPE 'f'  //分别定义reset，get，set命令  #define TINY4412\_BTN\_IORESET \_IO(TINY4412\_BTN\_TYPE,0)  #define TINY4412\_BTN\_IOGET \_IOR(TINY4412\_BTN\_TYPE,1,char)  #define TINY4412\_BTN\_IOSET \_IOW(TINY4412\_BTN\_TYPE,2,char)  #define TINY4412\_BTN\_IOSETI \_IOW(TINY4412\_BTN\_TYPE,3,int) | |

**解开字段**

|  |  |
| --- | --- |
| 获得传输方向位段的值 | \_IOC\_DIR(cmd)  \_IOC\_WRITE应用层写数据掩码，\_IOC\_READ应用层读数据掩码 |
| 获得类型的值 | \_IOC\_TYPE(cmd) |
| 获得编号的值 | \_IOC\_NR(cmd) |
| 获得大小的值 | \_IOC\_SIZE(cmd) |
| 验证地址是否合法 | int access\_ok(int type,const void \*addr,unsigned long size); |
| //从用户空间读取数据  \_IOC\_DIR(cmd) & \_IOC\_WRITE  access\_ok(VERIFY\_READ,(void \*)arg,sizeof(char))  //向用户空间写入数据  \_IOC\_DIR(cmd) & \_IOC\_READ  access\_ok(VERIFY\_WRITE,(void \*)arg,sizeof(char)) | |

# 定义一个等待队列

DECLARE\_WAIT\_QUEUE\_HEAD（）宏可以作为定义并初始化等待队列头部的“快捷方式”

DECLARE\_WAIT\_QUEUE\_HEAD (name)

# I/O复用poll的使用

|  |  |
| --- | --- |
| 函数原型 | int poll(struct pollfd fds[], nfds\_t nfds, int timeout)； |
| 函数作用 | 能够同时操作多个设备的方法，及时处理多个设备的数据。 |
| 参数的含义 | fds：是一个struct pollfd结构类型的数组，用于存放需要检测其状态的文件描述符集 |
| nfds：用于标记数组fds中的结构体元素的总数量 |
| timeout：是poll函数调用阻塞的时间，单位：毫秒；如果timeout==0，那么poll() 函数立即返回而不阻塞，如果设置为负数，那么poll() 函数会一直阻塞下去，直到所检测的文件描述符上的感兴趣的事件发生是才返回。 |
| 函数返回值 | >0：数组fds中准备好读、写或出错状态的那些文件描述符的总数量 |
| ==0：此时poll超时 |
| -1： poll函数调用失败，同时会自动设置全局变量errno |
| 使用实例 | struct pollfd fds[2];poll(fds, 2, -1) |

**struct pollfd结构体**

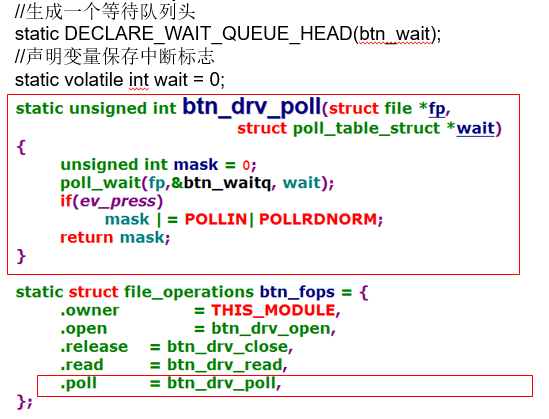
struct pollfd {

int fd; /\*文件描述符\*/

short events; /\* 等待的需要测试事件 通常是**POLLIN**\*/

short revents; /\* 实际发生了的事件，也就是返回结果 \*/

};



# 常用的信号

|  |  |  |
| --- | --- | --- |
| **信号名** | **含义** | **默认操作** |
| SIGHUP | 该信号在用户终端连接(正常或非正常)结束时发出，通常是在终端的控制进程结束时，通知同一会话内的各个作业与控制终端不再关联。 | 终止 |
| SIGINT | 该信号在用户键入INTR字符(通常是Ctrl-C)时发出，终端驱动程序发送此信号并送到前台进程中的每一个进程。 | 终止 |
| SIGKILL | 该信号用来立即结束程序的运行，并且不能被阻塞、处理和忽略。 | 终止 |

# 应用层发信号的三个函数

|  |  |
| --- | --- |
| 函数原型 | int kill(pid\_t pid, int sig); |
| 函数作用 | 发送信号 |
| 参数的含义 | pid: 正数：要接收信号的进程的进程号  0：信号被发送到所有和pid进程在同一个进程组的进程  -1：信号发给所有的进程表中的进程 |
| sig:信号 |
| 使用实例 | kill(pid, SIGKILL); |

|  |  |
| --- | --- |
| 函数原型 | int raise(int sig); |
| 函数作用 | 发送信号 |
| 参数的含义 | sig:信号 |
| 使用实例 | raise(SIGSTOP); |

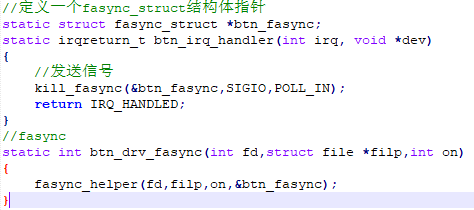
|  |  |
| --- | --- |
| 函数原型 | void (\*signal(int signum, void (\*handler)(int)))(int); |
| 函数作用 | 发送信号 |
| 参数的含义 | signum：指定信号 |
| handler: SIG\_IGN：忽略该信号  SIG\_DFL：采用系统默认方式处理信号  自定义的信号处理函数指针 |
| 使用实例 | signal(SIGIO,signal\_handler); |

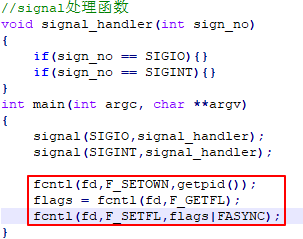
# 设备驱动程序如何实现异步信号

当应用层F\_SETFL 被执行来打开 FASYNC, 驱动的 fasync 方法被调用.

当数据到达, 驱动使用kill\_fasync()函数向进程发出一个 SIGIO 信号

# 实现一部信号驱动层和应用层的函数设置





# misc（杂项设备）

主设备号为10，***MISC\_MAJOR的值是10；***

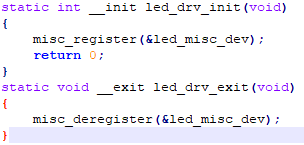
把没有规律的设备归纳为misc设备，led、watchdog、beep等都可以归纳为杂项设备

# misc设备驱动编写

创建一个miscdevice结构变量，初始化其minor、 name、fops成员；



然后使用misc\_register注册该结构体即可；



# input子系统的组成

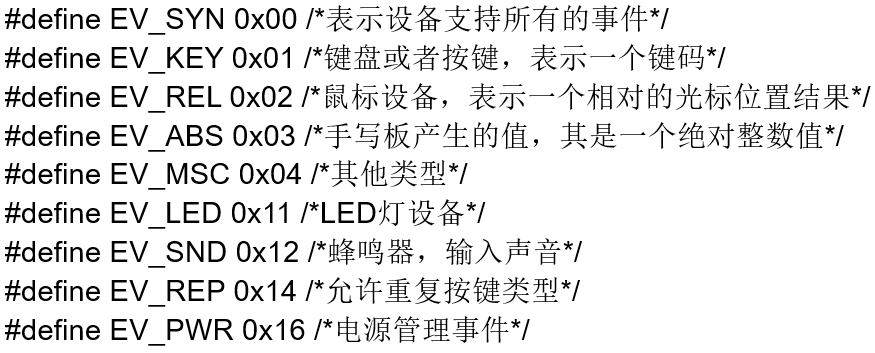
设备驱动层

核心层

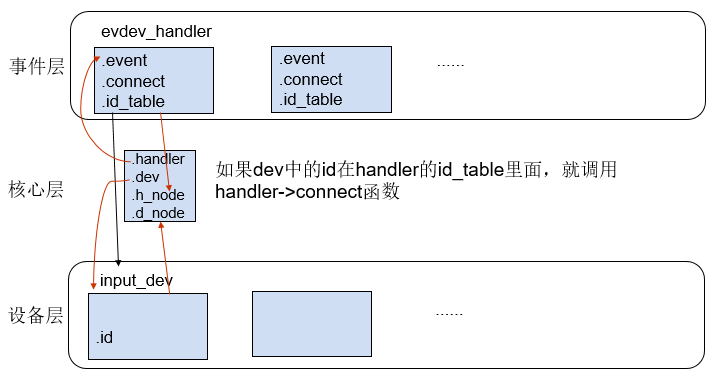
事件处理层

input子系统的主设备号是13，INPUT\_MAJOR宏的值为 13

# input子系统的事件类型



# input子系统框架



**编写符合input子系统的设备驱动程序**

初始化一个input\_dev结构

并且使用input\_register\_device函数将其注册到链表input\_dev\_list中即可

在中断处理函数中调用input\_event()函数上报事件即可

# platform（bus-device-driver）驱动模型

设备是数据，驱动是代码

device和driver是通过名字或者ID进行关联。

**bus总线下的关键函数（理解）**

**match() 匹配函数：**

判定设备和驱动是否匹配，是总线体系相关的。驱动核心通过match和probe两个函数来完成匹配每当有设备添加到总线时，驱动核心遍历总线上的驱动链表查找设备驱动；每当有驱动添加到总线时，驱动核心遍历总线上的设备链表查找驱动可操控的设备。

当前，match一般只执行总线特定的匹配处理，而在probe中，通过回调设备驱动probe，完成设备特定的匹配、设备初始化等。

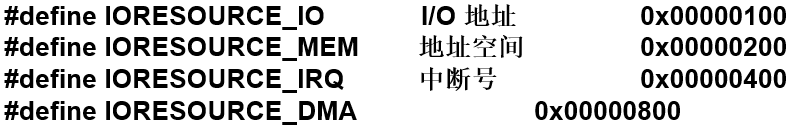
match匹配成功则返回1，失配返回0。

**probe()探测函数:**

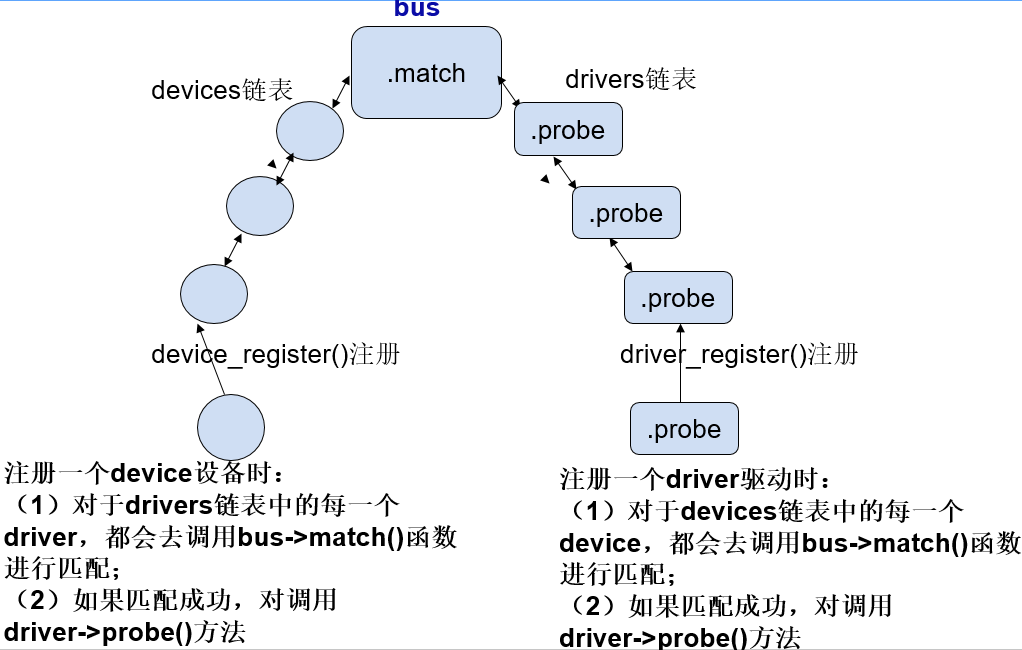
probe执行设备相关的匹配探测、设备初始化、资源分配等。

对于一次遍历匹配而言，如果match和probe均成功，则结束匹配成功；如果match成功而probe失配，继续遍历查找匹配；如果遍历结束而没有找到成功的匹配，对于驱动而言表示没有可操控设备，对于设备而言表示没有适当的驱动。

Platform驱动的资源一般驱动根据硬件来调整的参数。主要是中断号和寄存器地址。



# 设备与驱动注册的示意图



# I2C体系结构

I2C包含数据线（SDA），时钟线（SCL）

开始信号（S）：SCL为高电平时，SDA由高变低电平

结束信号（P）：SCL为高电平时，SDA由低变高电平

**Linux的I2C体系结构分为3个组成部分**

**I2C核心层**

提供了I2C总线驱动和设备驱动的注册、注销方法，I2C通信方法（即Algorithm）上层的与具体适配器无关的代码以及探测设备、检测设备地址的上层代码等

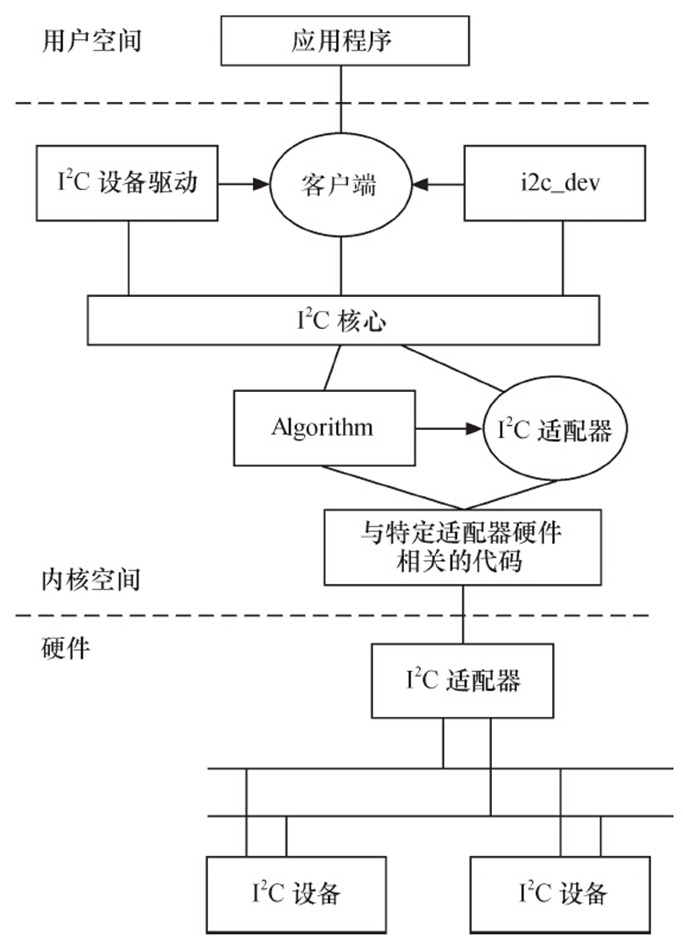
**I2C适配器层**

对I2C硬件体系结构中适配器端的实现，适配器可由CPU控制，甚至可以直接集成在CPU内部

**I2C设备驱动层**

I2C设备驱动（也称为客户驱动）是对设备端的实现，设备一般挂接在受CPU控制的I2C适配器上，通过I2C适配器与CPU交换数据。

# I2C驱动模型和编写方法

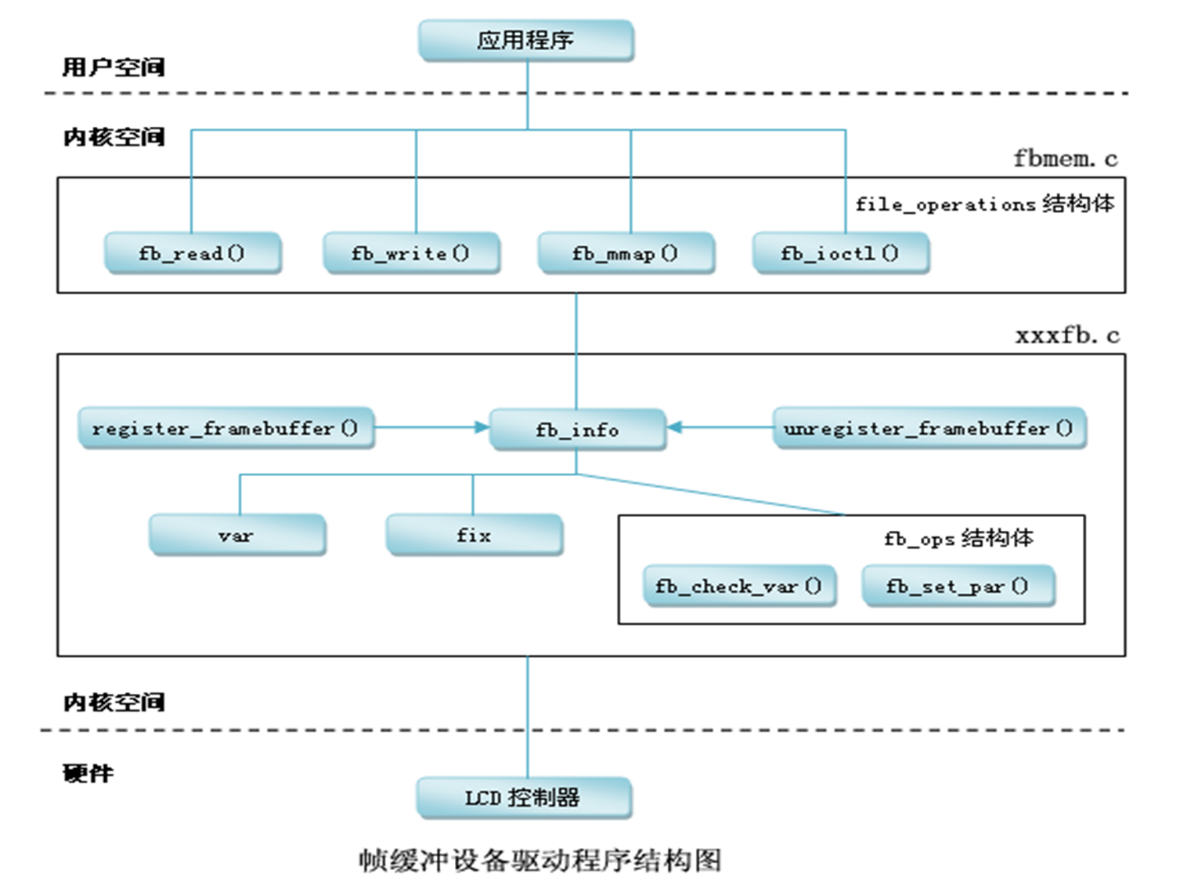


通过i2c\_new\_device()函数创建dev

通过i2c\_register\_driver()函数创建drv

# framebuffer设备

主设备号为29，对应于/dev/fb％d设备文件



**编写framebuffer驱动的主要步骤：**

初始化一个struct fb\_info结构；

将这个结构注册到系统中（register\_framebuffer）；

# 块设备的基本概念(理解)

**扇区(Sectors)**：任何块设备硬件对数据处理的基本单位。通常，1个扇区的大小为512byte。

**块(Blocks)**：由Linux制定对内核或文件系统等数据处理的基本单位。通常，1个块由1个或多个扇区组成。

**段(Segments)**：由若干个相邻的块组成。

块设备子系统可以分为文件系统层、页面缓冲层、通用块层、I/O调度层

# 块设备的驱动框架和编写流程

1、确定主设备号和次设备号

2、确定设备名称

3、创建设备文件

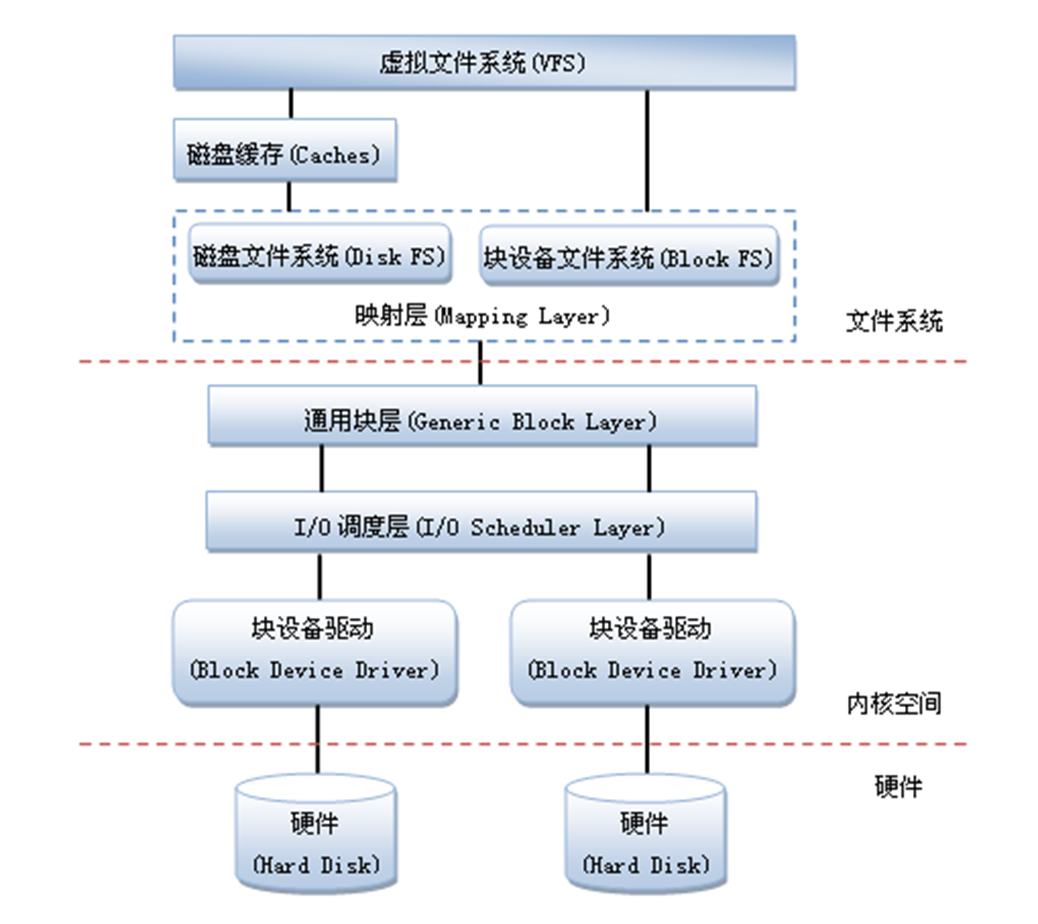
4、实现块设备驱动程序

4-1、实现block\_device\_operations结构体

4-2、实现request（请求）

4-3、实现初始化函数，注册块设备

4-4、实现销毁函数，取消块设备



# 网络体系结构

两类非常重要的体系结构：OSI与TCP/IP

# Linux网络设备框架

