



Linux 设备驱动开发

www.farsight.com.cn

今天的内容

- ❖ Linux 设备驱动的现状
- ❖ 从 non-os 驱动到 Linux 驱动
- ❖ 内核设施
 - 自旋锁、信号量、互斥量、完成量
 - 异步通知、信号
 - 阻塞与非阻塞
 - 内存与 I/O 操作, DMA
 - 中断, top half/bottom half
- ❖ 字符设备驱动
- ❖ 复杂设备驱动的框架
 - LCD 设备 FRAMEBUFFER
 - FLASH 设备 MTD
 - TTY 设备
 - 块设备
- ❖ 用户空间的设备驱动
- ❖ 设备驱动开发流程
 - 开发环境建设
 - 调试手段
 - 用户空间测试
- ❖ 设备驱动的学习方法

Linux 设备驱动的现状

❖ 高需求

- Linux 内核的绝大多数代码为设备驱动
- 新设备、新芯片、新驱动的需求

❖ 高门槛

- 涉及到大量硬件操作
- 涉及到内核基础知识
- 涉及到并发控制与同步
- 复杂的软件结构框架

❖ 高回报

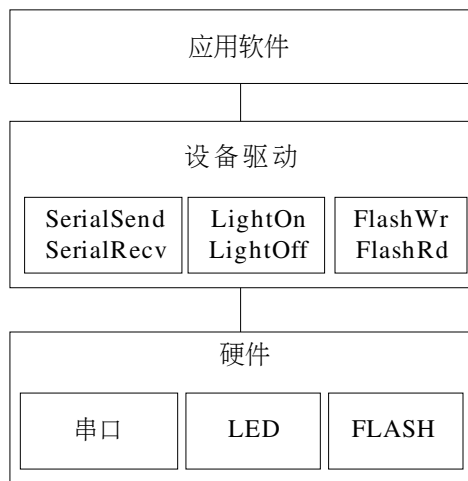
从 non-os 驱动到 Linux 驱动

❖ non-os 驱动

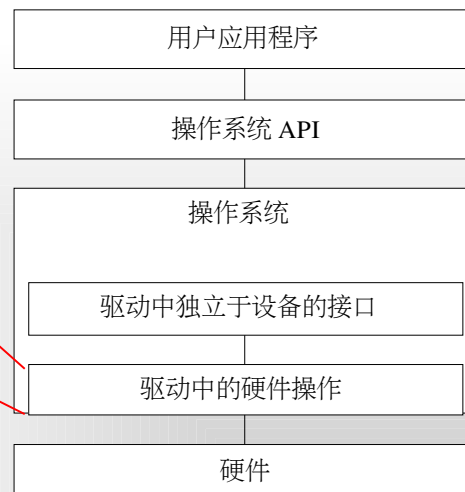
- 单刀直入 简单 直接提供 API

❖ Linux 驱动

- 兵团战役 复杂 间接提供 API



non-os 驱动与应用



on-os 驱动与应用

- 并发和竞态:

- 对称多处理器（SMP）的多个 CPU
- 单 CPU 内进程与抢占它的进程
- 中断（硬中断、软中断、Tasklet、底半部）与进程之间

- 处理思路:

lock() // 锁定，拿虎符

...

critical section // 临界区，调动军队

...

unlock() // 解锁定，归还虎符

- 常用方法:

- 中断屏蔽
- 原子操作
- 自旋锁
- 信号量
- 互斥体

接口

整型原子操作

□ 设置原子变量的值

`void atomic_set(atomic_t *v, int i);` // 设置原子变量的值为 `i`

`atomic_t v = ATOMIC_INIT(0);` // 定义原子变量 `v` 并初始化为 `0`

□ 获取原子变量的值

`atomic_read(atomic_t *v);` // 返回原子变量的值

□ 原子变量加 / 减

`void atomic_add(int i, atomic_t *v);` // 原子变量增加 `i`

`void atomic_sub(int i, atomic_t *v);` // 原子变量减少 `i`

⌘ 原子变量自增 / 自减

`void atomic_inc(atomic_t *v);` // 原子变量增加 `1`

`void atomic_dec(atomic_t *v);` // 原子变量减少 `1`

⌘ 操作并测试

`int atomic_inc_and_test(atomic_t *v);`

`int atomic_dec_and_test(atomic_t *v);`

`int atomic_sub_and_test(int i, atomic_t *v);`

⌘ 操作并返回

`int atomic_add_return(int i, atomic_t *v);`

`int atomic_sub_return(int i, atomic_t *v);`

`int atomic_inc_return(atomic_t *v);`

`int atomic_dec_return(atomic_t *v);`

位原子操作

⌘ 设置 / 清除 / 反转位

`void set_bit(nr, void *addr);`

`void clear_bit(nr, void *addr);`

`void change_bit(nr, void *addr);`

□ 测试位

`test_bit(nr, void *addr);`

⌘ 测试并操作位

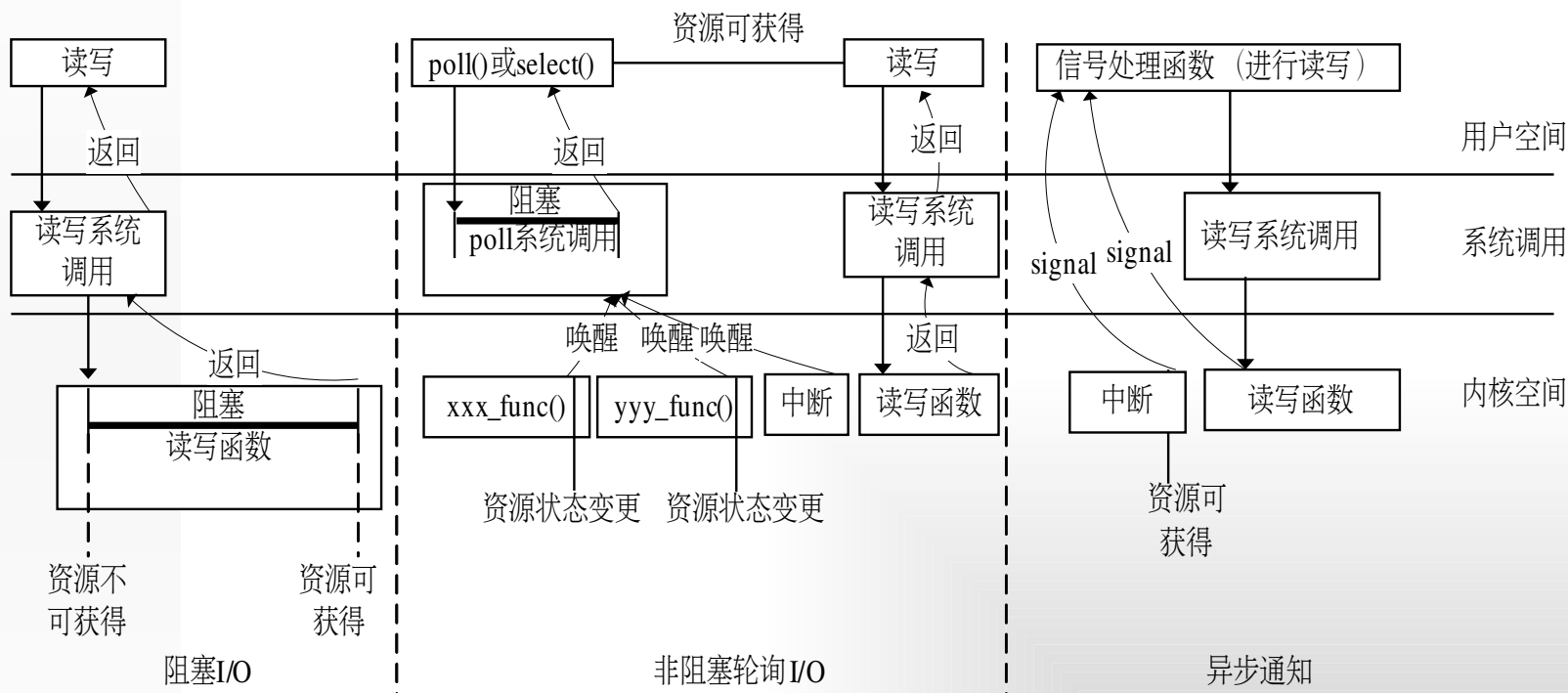
`int test_and_set_bit(nr, void *addr);`

`int test_and_clear_bit(nr, void *addr);`

`int test_and_change_bit(nr, void *addr);`

自旋锁 VS 信号量

- 自旋锁:
 - 忙等待, 无调度开销
 - 进程抢占被禁止
 - 锁定期间不能睡觉
 - `spinlock_t lock;`
 - `spin_lock_init(&lock);`
 - `spin_lock(&lock);` // 获取自旋锁, 保护临界区
 - `... // 临界区`
 - `spin_unlock(&lock);` // 解锁
- 信号量
 - 拿不到就切换进程, 有调度开销
 - 锁定期间可以睡觉, 不用于中断上下文
 - `// 定义信号量`
 - `DECLARE_MUTEX(mount_sem);`
 - `down(&mount_sem);` // 获取信号量, 保护临界区
 - `...`
 - `critical section` // 临界区
 - `...`
 - `up(&mount_sem);` // 释放信号量



- 等待队列：进程等待被唤醒的一种机制
- 阻塞与非阻塞使用模板

```

1 static ssize_t xxx_write(struct file *file, const char *buffer, size_t count,
2   loff_t *ppos)
3 {
4   ...
5   DECLARE_WAITQUEUE(wait, current); // 定义等待队列
6   add_wait_queue(&xxx_wait, &wait); // 添加等待队列
7
8   ret = count;
9   /* 等待设备缓冲区可写 */
10  do
11  {
12    avail = device_writable(...);
13    if (avail < 0)
14      __set_current_state(TASK_INTERRUPTIBLE); // 改变进程状态
15
16    if (avail < 0)
17    {
18      if (file->f_flags & O_NONBLOCK) // 非阻塞
19      {
20        if (!ret)
21          ret = -EAGAIN;
22        goto out;
23      }
24      schedule(); // 调度其他进程执行
25      if (signal_pending(current)) // 如果是因为信号唤醒
26      {
27        if (!ret)
28          ret = -ERESTARTSYS;
29        goto out;
30      }
31    }
32  } while (avail < 0);
33
34  /* 写设备缓冲区 */
35  device_write(...)
36  out:
37  remove_wait_queue(&xxx_wait, &wait); // 将等待队列移出等待队列头
38  set_current_state(TASK_RUNNING); // 设置进程状态为 TASK_RUNNING
39  return ret;
40 }

```

- 驱动中 POLL 模板

```

1 static unsigned int xxx_poll(struct file *filp, poll_table *wait)
2 {
3     unsigned int mask = 0;
4     struct xxx_dev *dev = filp->private_data; /* 获得设备结构体指针 */
5     ...
6     ...
7     ...
8     poll_wait(filp, &dev->wait, wait);
9     ...
10    if (...)// 可读
11    {
12        mask |= POLLIN | POLLRDNORM; /* 标示数据可获得 */
13    }
14    ...
15    if (...)// 可写
16    {
17        mask |= POLLOUT | POLLWRNORM; /* 标示数据可写入 */
18    }
19    ...
20    ...
21    return mask;
22 }

```

- 用户空间 POLL 模板

```

fd_set fds;
FD_ZERO(&fds);
FD_SET(fd, &fds);
select(fd + 1, &rfd, &wfd, NULL, NULL);
if (FD_ISSET(fd, &fds))
{
    printf("Poll monitor:can be access\n");
}

```

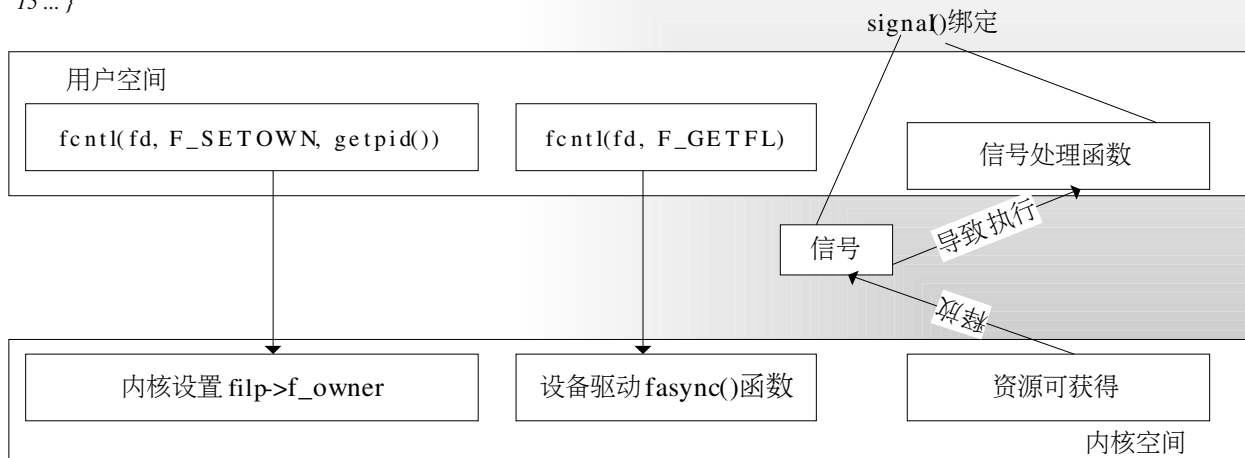
❖ 信号：软件意义上的“中断”

➤ 驱动发出信号

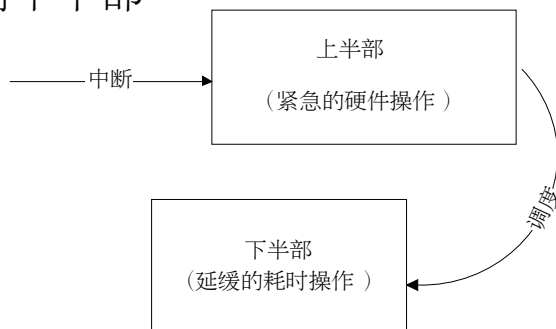
- `kill_fasync(&dev->async_queue, SIGIO, POLL_IN);`

➤ 用户空间应用程序处理信号

- `24 signal(SIGIO, input_handler);`
- `25 fcntl(STDIN_FILENO, F_SETOWN, getpid());`
- `26 oflags = fcntl(STDIN_FILENO, F_GETFL);`
- `27 fcntl(STDIN_FILENO, F_SETFL, oflags | FASYNC);`
- `8 void input_handler(int num)`
- `9 { ...`
- `14 len = read(STDIN_FILENO, &data, MAX_LEN);`
- `15 ... }`



❖ 两个半部



❖ 机制

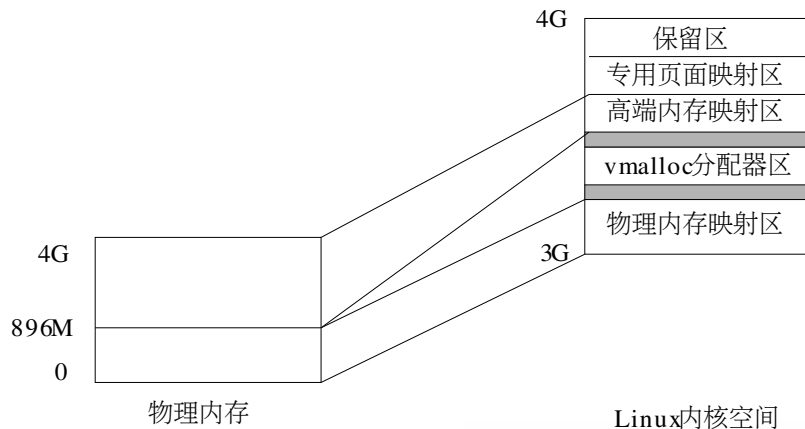
➤ tasklet

➤ 工作队列

- 1 /* 定义 tasklet 和底半部函数并关联 */
- 2 void xxx_do_tasklet(unsigned long);
- 3 DECLARE_TASKLET(xxx_tasklet, xxx_do_tasklet, 0);
- 4
- 5 /* 中断处理底半部 */
- 6 void xxx_do_tasklet(unsigned long)
- 7 { ...
- 8 }
- 9 }
- 10
- 11 /* 中断处理顶半部 */
- 12 irqreturn_t xxx_interrupt(int irq, void *dev_id, struct pt_regs *regs)
- 13 { ...
- 14
- 15 tasklet_schedule(&xxx_tasklet);
- 16 ... }

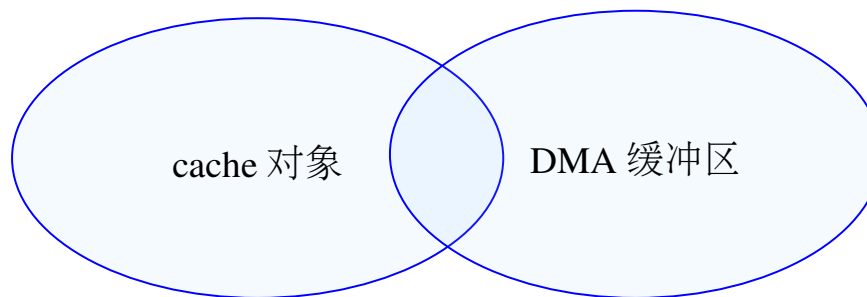
内存与 I/O 访问

- ❖ 内存空间与 I/O 空间
- ❖ Linux 内核地址空间



- ❖ 内存申请
 - kmalloc, get_free_pages: 物理连续, 线性映射
 - vmalloc: 物理非连续, 非线性映射
- ❖ 物理/虚拟地址映射
 - 静态映射
 - ioremap, ioremap_nocache
- ❖ mmap: 映射到用户空间

❖ cache 一致性问题



❖ 内存地址 / 总线地址

❖ DMA 缓冲区

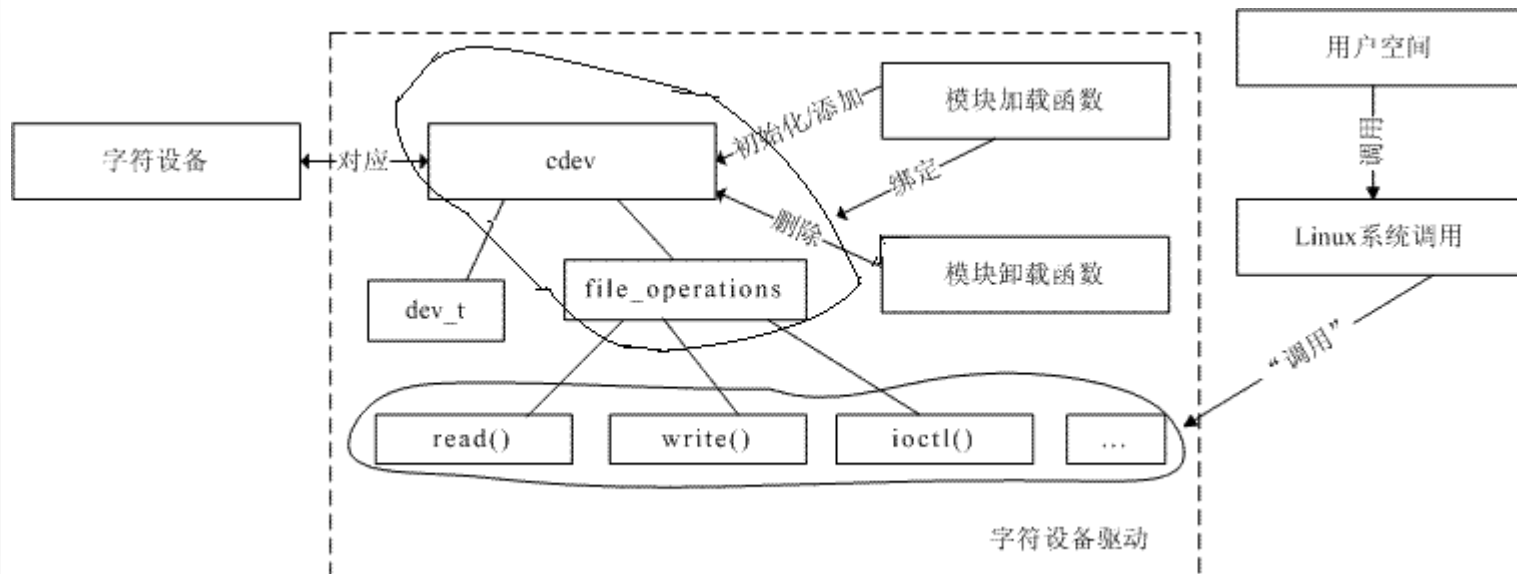
➤ 一致性缓冲区

- `void * dma_alloc_coherent(struct device *dev, size_t size, dma_addr_t *handle, gfp_t gfp);`
- `void dma_free_coherent(struct device *dev, size_t size, void *cpu_addr, dma_addr_t handle);`

➤ 流式 DMA 映射

- `dma_addr_t dma_map_single(struct device *dev, void *buffer, size_t size, enum dma_data_direction direction);`
- `void dma_unmap_single(struct device *dev, dma_addr_t dma_addr, size_t size, enum dma_data_direction direction);`

❖ 结构

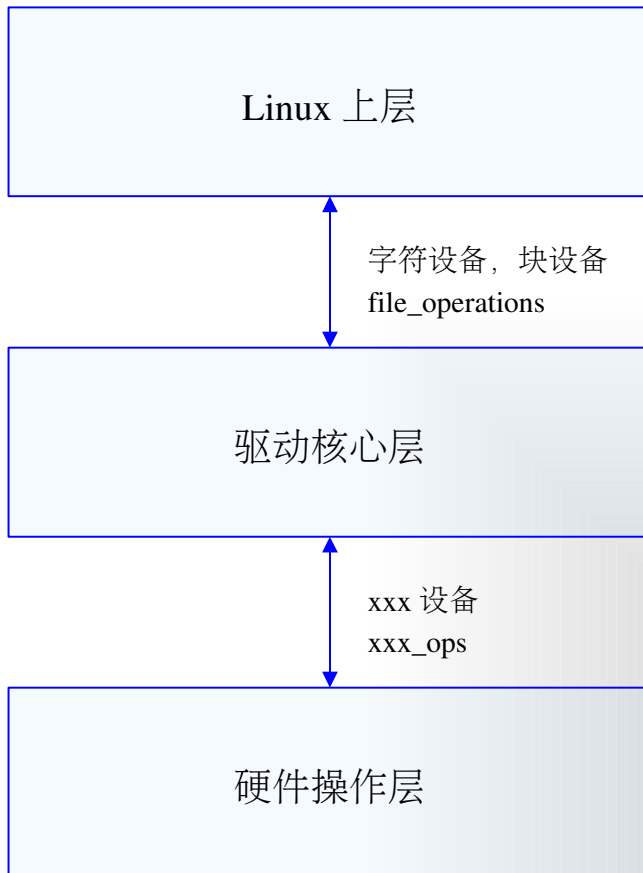


❖ file_operations

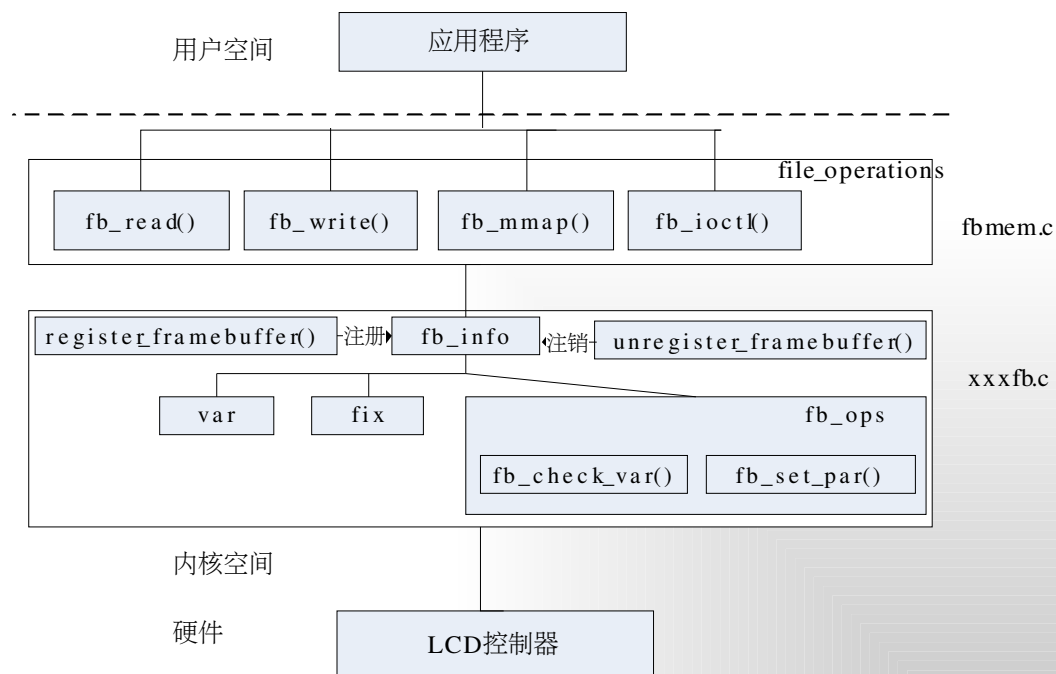
- 1 struct file_operations xxx_fops =
- 2 {
- 3 .owner = THIS_MODULE,
- 4 .read = xxx_read,
- 5 .write = xxx_write,
- 6 .ioctl = xxx_ioctl,
- 7 ...
- 8 };

❖ 复杂设备驱动的 framework

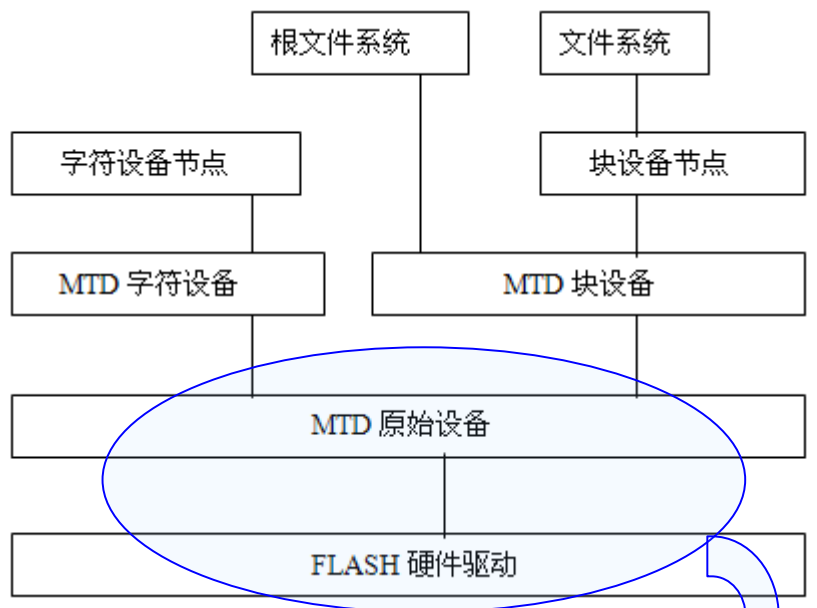
- 层次化
- 结构化
- 上层不依赖于具体硬件，下层与硬件接口



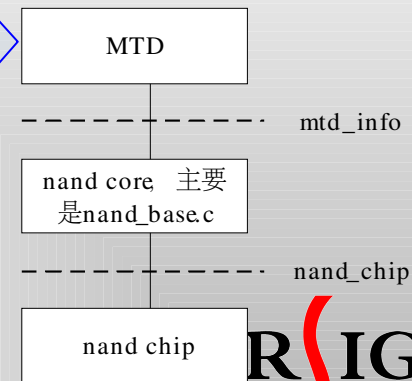
- ❖ 从硬件无关到硬件相关:
- file_operations->fb_info->fb_ops
- ❖ 从注册 cdev 到注册 framebuffer



❖ 层次结构

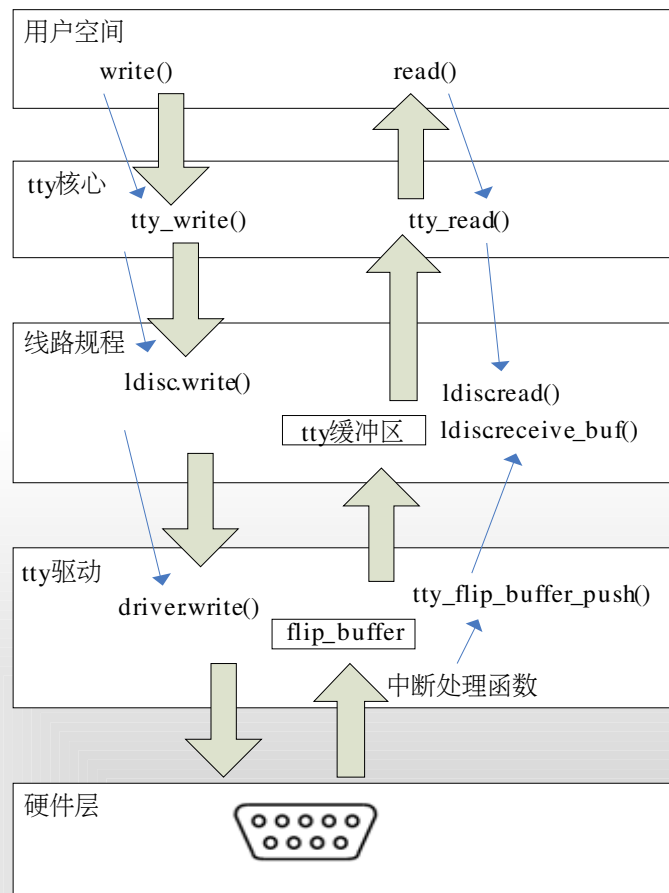
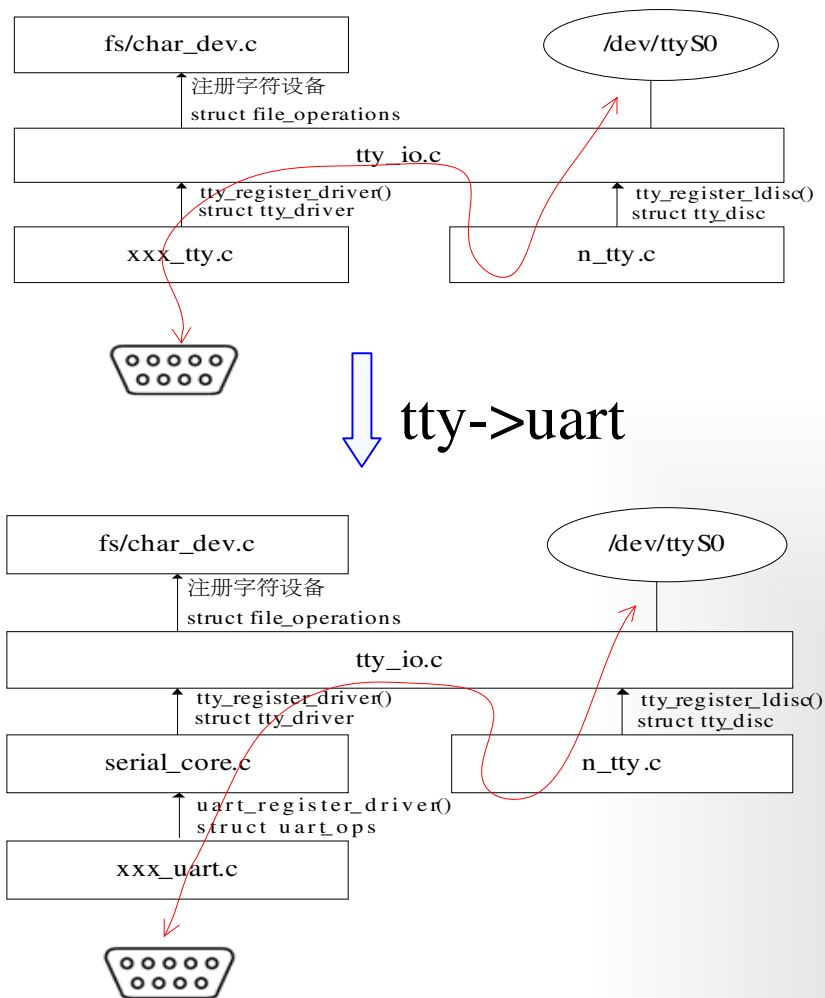


❖ 字符/块设备 -> mtd_info -> nand_chip

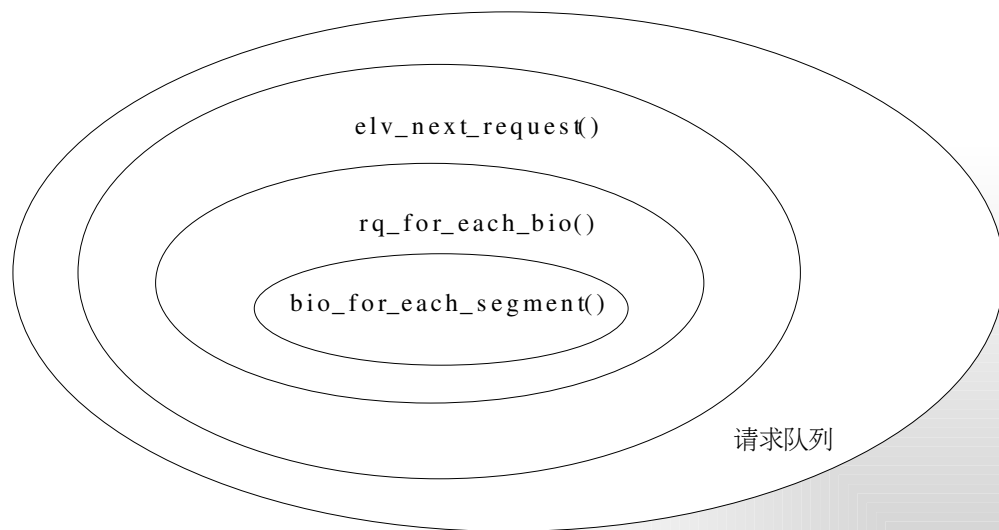


TTY 设备驱动

❖ 层次结构与数据流向

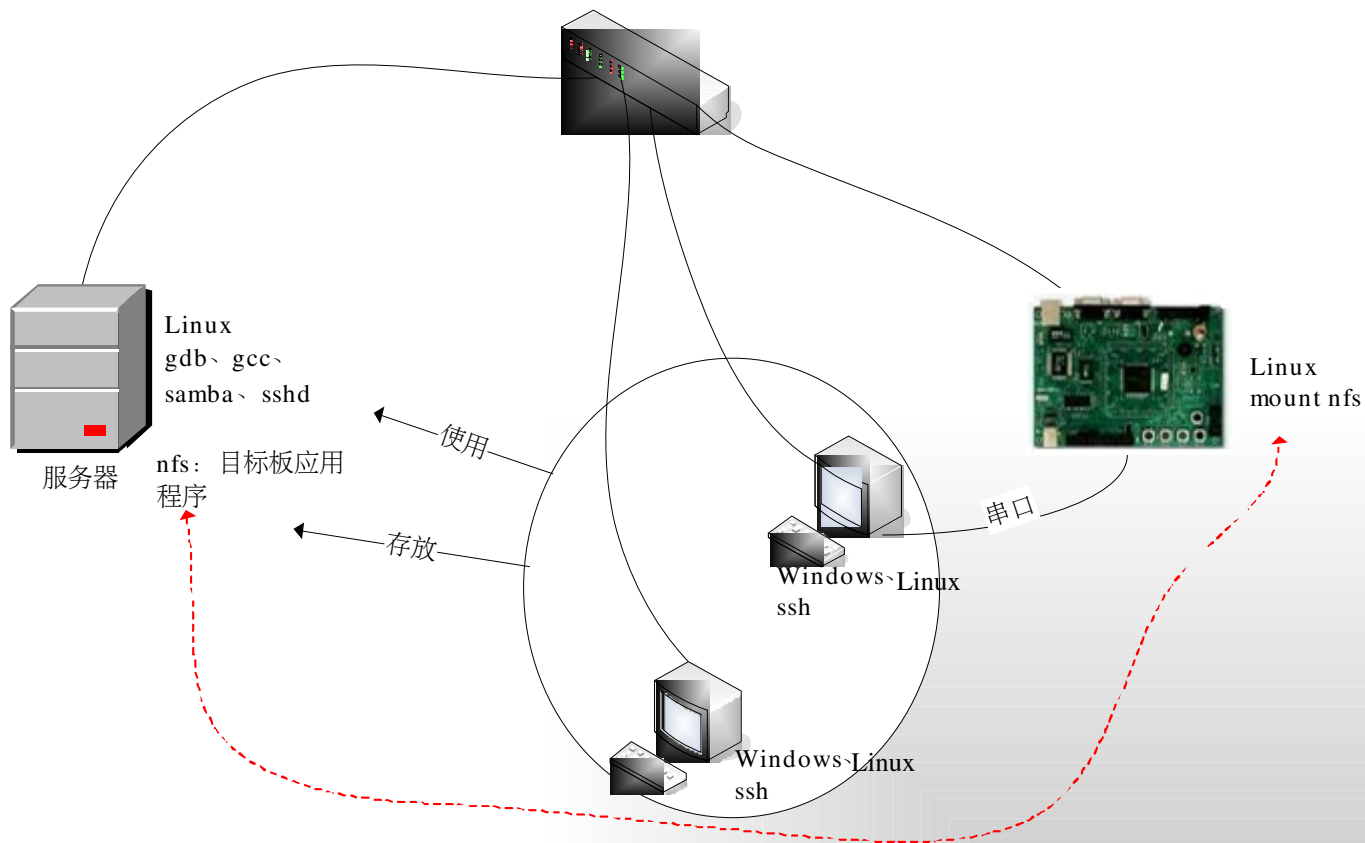


- ❖ 数据结构
 - `block_device_operations`
 - `gendisk`
 - `request` 与 `bio` : 表征等待进行的 I/O 请求
- ❖ I/O 请求



用户空间的设备驱动

- ❖ 从用户空间访问内存和 I / O
- ❖ userspace 接口
 - User Mode SCSI
 - User Mode USB
 - User Mode I2C
 - UIO:drivers/uio/





华清远见

驱动调试

- printk()
- /proc
- oops
- 监视工具
- kcore
- kdb
- kgdb
- 仿真器

- 最简单，最常用的方法
 - 调整打印级别： `# echo 5 > /proc/sys/kernel/printk`
 - 使用宏：通过 `make menuconfig` 选择是否包含打印信息
- `#ifdef CONFIG_XXXDEBUG`
- `#xxx_debug(fmt,arg...) printk(KERN_DEBUG fmt,##arg)`
- `#else`
- `#xxx_debug(fmt,arg...)`
- `#endif`

- 从用户空间获取内核信息的方法
- 7 ssize_t simple_proc_read(char *page, char **start, off_t off, int count,
- 8 int*eof, void *data)
- 23 ssize_t simple_proc_write(struct file *filp, const char __user *buff, unsigned
- 24 long len, void *data)
- 55 int __init simple_proc_init(void)
- 56 {
- 57 proc_entry = create_proc_entry("sim_proc", 0666, NULL); // 创建 /proc
- 58 if (proc_entry == NULL)
- 59 {...
- 62 }
- 63 else
- 64 {
- 65 proc_entry->read_proc = simple_proc_read;
- 66 proc_entry->write_proc = simple_proc_write;
- 67 proc_entry->owner = THIS_MODULE;
- 68 }
- 72 ...}
- 使用方法: cat, echo

```

8 static ssize_t oopsexam_write(struct file
*filp, const char *buf, size_t len, loff_t
•
9 *off)
•
10 {
•
11 int *p=0;
•
12 *p = 1; // 故意访问 0 地址
•
13 return len;
•
14 }

```

- Unable to handle kernel NULL pointer dereference at virtual address 00000000
- printing eip:
- c381a013
- *pde = 00000000
- Oops: 0002 [#1]
- PREEMPT SMP
- Modules linked in: oops_example
- CPU: 0
- EIP: 0060:[<c381a013>] Not tainted VLI
- EFLAGS: 00010286 (2.6.15.5)
- EIP is at oopsexam_write+0x4/0x11 [oops_example]
- eax: 00000002 ebx: c2b35480 ecx: 00000000 edx: c381a00f
- esi: 00000002 edi: 080e9408 ebp: c2007fa4 esp: c2007f68
- ds: 007b es: 007b ss: 0068
- Process bash (pid: 2453, threadinfo=c2006000 task=c2021570)
- Stack: c015e036 c2b35480 080e9408 00000002 c2007fa4 00000000 c2b35480 ffffffff
- 080e9408 c2006000 c015e1d1 c2b35480 080e9408 00000002 c2007fa4 00000000
- 00000000 00000000 00000001 00000002 c0102f9f 00000001 080e9408 00000002
- Call Trace:
- [<c015e036>] vfs_write+0xc5/0x18f
- [<c015e1d1>] sys_write+0x51/0x80
- [<c0102f9f>] sysenter_past_esp+0x54/0x75
- Code: Bad EIP value.

```

4 main()
5 {
6   int fd, num, pos;
7   char wr_ch[200] = "This is a test of globalmem";
8   char rd_ch[200];
9   // 打开 /dev/globalmem
10  fd = open("/dev/globalmem", O_RDWR, S_IRUSR | S_IWUSR);
11  if (fd != -1 )
12  {
13    // 清除 globalmem
14    if(ioctl(fd, MEM_CLEAR, 0) < 0)
15    {
16      printf("ioctl command failed\n");
17    }
18    // 读 globalmem
19    num = read(fd, rd_ch, 200);
20    printf("%d bytes read from globalmem\n", num);
21
22    // 写 globalmem
23    num = write(fd, wr_ch, strlen(wr_ch));
24    printf("%d bytes written into globalmem\n", num);
25    ...
26    close(fd);
27  }
28 }

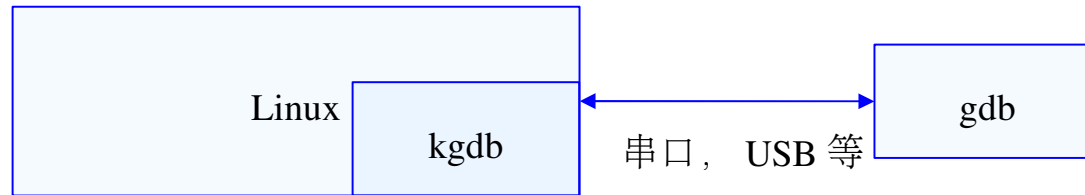
```

```

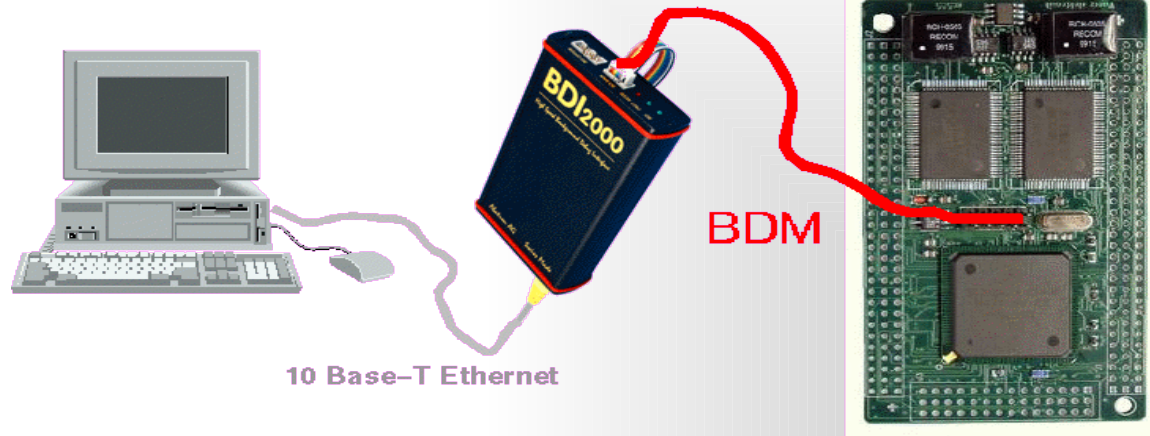
•   execve("./globalmem_test", ["/globalmem_test"], [/* 24 vars */]) = 0
•   ...
•   open("/dev/globalmem", O_RDWR)      = 3      ----- 打开
    的 /dev/globalmem 的 fd 是 3
•   ioctl(3, FIBMAP, 0)                  = 0
•   read(3, 0xbff17920, 200)             = 200     ----- 读取到 200 个
    字节
•   ...
•   write(1, "200 bytes read from globalmem\n", 30200 bytes read from
    globalmem
•   ) = 30                               ----- 向标准输出设备 (fd 为 1) 写入
    printf 中的字符串
•   write(3, "This is a test of globalmem", 27) = 27
•   write(1, "27 bytes written into globalmem\n", 3227 bytes written into
    globalmem
•   ) = 32
•   ...

```

- ❖ 目标机“插桩”：
- ❖ 打上 kgdb 补丁，这样主机上的 gdb 可与目标机的 kgdb 通过串口或网口通信。



- ❖ 使用仿真器：
- 仿真器可直接连接目标机的 JTAG/BDM，这样主机上的 gdb 就可以通过与仿真器的通信来控制目标机



用户空间测试

- ❖ 编写用户空间的程序访问设备驱动
 - read
 - write
 - ioctl
- ❖ 做成含测试菜单的程序
 - 每一个菜单体现一种功能

设备驱动学习方法

- ❖ 牢固掌握内核编程基础知识
 - 并发、同步
 - 内存与 I / O 访问
 - 中断两个半部
- ❖ 理解复杂设备驱动架构的特点
- ❖ 参考同类设备驱动的源代码
- ❖ 动手实践
 - 在 PC 上实践 globalmem 和 globalfifo
 - 在开发板上学习真实设备的驱动
 - 在工作中学习

Linux 设备驱动开发详解

❖ 主要出发点:

- 力求用最简单的实例讲解复杂的知识点，以免实例太复杂搅浑读者（驱动理论部分）
- 对 **Linux** 设备驱动多种复杂设备的框架结构进行了全面的介绍（驱动框架部分）
- 更面向实际的嵌入式工程，讲解开发必备的软硬件基础，及开发手段（调试与移植部分）
- 提供讨论与交流平台（华清远见， www.linuxdriver.cn，数个 QQ 群）

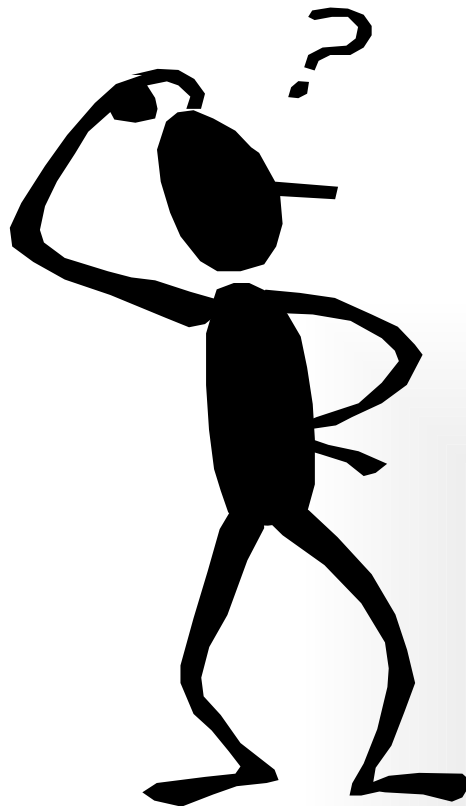


华清远见 Linux 驱动课程

- ❖ 嵌入式 Linux 驱动初级班
- ❖ 通过本课程的学习，学员可以掌握 Linux 下字符设备、块设备、网络设备的驱动程序开发，同时掌握嵌入式 Linux 的系统开发和分析方法。
- ❖ 嵌入式 Linux 驱动开发高级班
- ❖ 本课程以案例教学为主，系统地介绍 Linux 下有关 FrameBuffer、MMC 卡、USB 设备的驱动程序开发。
- ❖ 班级规模及环境
- ❖ 为了保证培训效果，增加互动环节，我们坚持小班授课，每期报名人数限 15 人，多余人员安排到下一期进行。人手一套开发板和开发用的 PC 主机。

华清远见

让我们一起讨论！



FAR  SIGHT



www.farsight.com.cn

谢谢！