网名“鱼树”的学员聂龙浩，

学习“韦东山 Linux 视频第 2 期”时所写的笔记很详细,供大家参考。

也许有错漏,请自行分辨。

目录

网卡驱动程序框架

网卡驱动程序“收发功能”：

只要把上层的数据发给网卡，从网卡来的数据构造成包给上层即可。网卡只需要“socket”编程，不需要打开某设备。
驱动程序都是以面向对象的思想写的，都有相关的结构体。

编程步骤

1，分配某结构体：net_device
2，设置结构体。
①，提供一个发包函数：hard_start_xmit()
②，提供收包的功能 ： net_interrupt(int irq, void *dev_id)-->netif_rx(skb);
收到数据后，网卡里面一般都有中断程序。在中断程序中有一个上报数据的函数。
3，注册结构体:register_netdev(dev) 真实驱动中使用的是此注册函数。
4，硬件相关操作。

看内核中的“cs89x.c”这个真实的网卡驱动程序：

```
int __init init_module(void)
-->分配一个 net_device 结构体
    struct net_device *dev = alloc_etherdev(sizeof(struct net_local));
    --->alloc_netdev(sizeof_priv, "eth%d", ether_setup); 分配时用了eth%d这样的名字。
```

-->设置 net_device 结构 ：

MAC 地址，硬件相关操作

```
dev->dev_addr[0] = 0x08;
dev->dev_addr[1] = 0x89;
dev->dev_addr[2] = 0x89;
dev->dev_addr[3] = 0x89;
dev->dev_addr[4] = 0x89;
dev->dev_addr[5] = 0x89; //以上为MAC地址。
```

--> 硬件相关设置

```
-->ret = cs89x0_probe1(dev, io, 1);
    -->net_device结构中有open,read等函数。
  dev->open                                    = net_open;
  dev->stop                                    = net_close;
  dev->tx_timeout                              = net_timeout;
  dev->watchdog_timeo                  = HZ;
  dev->hard_start_xmit    = net_send_packet; //硬件启动传输。这是发包函数。
  dev->get_stats                               = net_get_stats;
  dev->set_multicast_list = set_multicast_list;
  dev->set_mac_address                 = set_mac_address;
```

```
    -->retval = register_netdev(dev);
```

接收到数据要做的事情：

```
irqreturn_t net_interrupt(int irq, void *dev_id)
-->net_rx(dev);
    -->从硬件芯片里读出来
        status = readword(ioaddr, RX_FRAME_PORT);
  length = readword(ioaddr, RX_FRAME_PORT);
    -->skb = dev_alloc_skb(length + 2);  分配一个skb缓冲。
    -->netif_rx(skb);
    -->netif_wake_queue(dev);  发送完数据后就唤醒队列
```

发包函数：

```
int net_send_packet(struct sk_buff *skb, struct net_device *dev)
```

接收到包后是上报了一个“sk_buff”缓冲（netif_rx(skb)）

sk_buff 结构 是纽带，运用“hard_start_xmit()”和“netif_rx()”：
应用层构造好一个包后，放到“skb_buff”结构交给网卡驱动，调用“hard_start_xmit()”来发送。
网卡在中断程序中收到数据后，从芯片里把数据读出来构造一个“skb_buff”结构数据，调用“netif_rx()”上报数据给应用层。

```
-->netif_stop_queue(dev);  先停止队列
-->将“skb_buff”中数据写到网卡芯片：
    writeword(dev->base_addr, TX_CMD_PORT, lp->send_cmd);
    writeword(dev->base_addr, TX_LEN_PORT, skb->len);
    writewords(dev->base_addr, TX_FRAME_PORT,skb->data,(skb->len+1) >>1);
-->dev_kfree_skb (skb);  然后释放skb_buff。
```

写一个虚拟网卡驱动：

1，分配 net_device 结构：

```
net_device *alloc_netdev(int sizeof_priv, const char *name,void (*setup)(struct net_device *))
```

//1,分配 net_device 结构体

```
        vnet_dev = alloc_netdev(0, "vnet%d", ether_setup);
```

直接用“alloc_netdev()”自忆定义网卡的名字为“vent%d”.其中“ether_setup”是默认的设置函数。
sizeof_priv 是私有数据，我们的这里定义私有数据为 0.

内核中经常会在只分配一个结构体的时候多分配一个内存。这块内存就是用来存放自己的“私有数据”。

这里不需要私有信息，所以直接分配"net_device"结构中没有分配“私有数据空间”，上面指为了“0”。

```
//1.1, net_device 结构变量 vnet_dev.
static net_device *vnet_dev;

static int virt_net_init(void)
{
    //1,分配 net_device 结构体
    vnet_dev = alloc_netdev(0, "vnet%d", ether_setup); //alloc_etherdev()
    //2,设置

    //3,注册
    register_netdevice (vnet_dev);
    return 0;
}

static void virt_net_exit(void)
{
    unregister_netdevice (vnet_dev);
    free_netdev(vnet_dev);
}

module_init (virt_net_init);
module_exit (virt_net_exit);

MODULE_AUTHOR ("ian1900");
MODULE_LICENSE("GPL");
```

这就是最简单的网卡驱动程序。

```
int register_netdev(struct net_device *dev)
{
    int err;

    rtnl_lock();

    /*
     * If the name is a format string the caller wants us to do a
     * name allocation.
     */
    if (strchr(dev->name, '%')) {
        err = dev_alloc_name(dev, dev->name);
        if (err < 0)
            goto out;
    }

    err = register_netdevice(dev);
out:
    rtnl_unlock();
    return err;
} ? end register_netdev ?
```

要用上面的“register_netdev()”来注册，里面获得锁后用“register_netdevice（）”来注册。

测试：

```
book@book-desktop:/work/drivers_and_test/16th_virt_net/1th$ make
make -C /work/system/linux-2.6.22.6 M=`pwd` modules
make[1]: Entering directory `/work/system/linux-2.6.22.6'
  CC [M]  /work/drivers_and_test/16th_virt_net/1th/virt_net.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /work/drivers_and_test/16th_virt_net/1th/virt_net.mod.o
  LD [M]  /work/drivers_and_test/16th_virt_net/1th/virt_net.ko
make[1]: Leaving directory `/work/system/linux-2.6.22.6'
book@book-desktop:/work/drivers_and_test/16th_virt_net/1th$
```

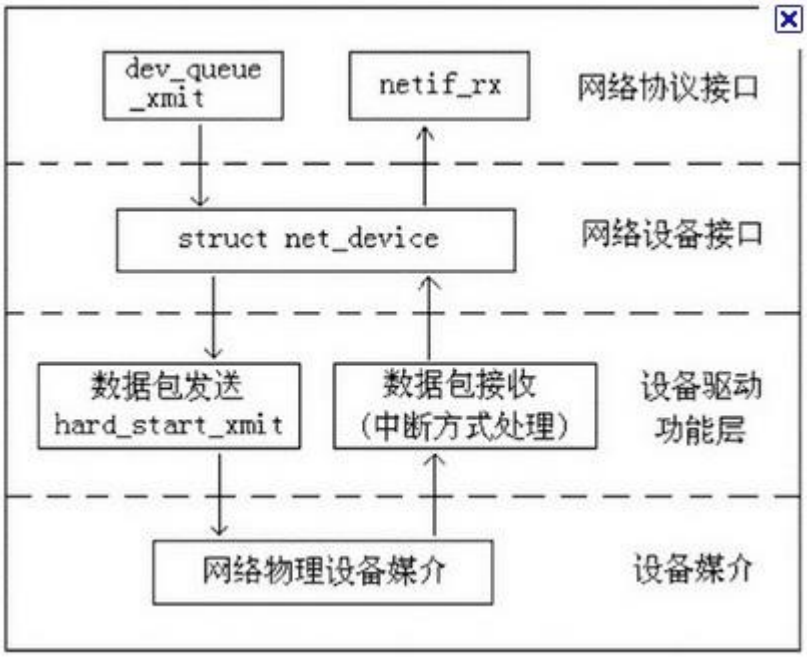
网卡驱动程序框架：

app: socket 编程

```
          --------------
          --------------- 若干层网络协议--纯软件（网络七层模型）
          --------------
          --------------
hard_start_xmit||  /\
             \/  ||  netif_rx   sk_buff
          --------------
```

硬件相关的驱动程序(要提供 hard_start_xmit，有数据时要用 netif_rx 上报)

硬件



怎么写网卡驱动程序？

1. 分配一个 net_device 结构体
2. 设置：
2.1 发包函数：hard_start_xmit
2.2 收到数据时(在中断处理函数里)用 netif_rx 上报数据
2.3 其他设置
3. 注册：register_netdev

测试 1th/2th:

1. insmod virt_net.ko

```
# insmod virt_net.ko
RTNL: assertion failed at net/core/dev.c (3065)
[<c002ede8>] (dump_stack+0x0/0x14) from [<c023fd88>] (register_netdevice+0x50/0x348)
[<c023fd38>] (register_netdevice+0x0/0x348) from [<bf002028>] (virt_net_init+0x28/0x3c [virt_net])
 r6:bf002460 r5:bf002460 r4:00000000
[<bf002000>] (virt_net_init+0x0/0x3c [virt_net]) from [<c006285c>] (sys_init_module+0x1424/0x1514)
[<c0061438>] (sys_init_module+0x0/0x1514) from [<c002aea0>] (ret_fast_syscall+0x0/0x2c)
RTNL: assertion failed at net/ipv4/devinet.c (1054)
```

```
int register_netdevice(struct net_device *dev)
{
    struct hlist_head *head;
    struct hlist_node *p;
    int ret;

    BUG_ON(dev_boot_phase);
    ASSERT_RTNL();
```

3065 行错误：

```
#define ASSERT_RTNL() do { \
     if (unlikely(rtnl_trylock())) { \
          rtnl_unlock(); \
          printk(KERN_ERR "RTNL: assertion failed at %s (%d)\n", \
                 __FILE__,  __LINE__); \
          dump_stack(); \
     } \
} while(0)
```

想获得一把锁，但没有成功。而真实的注册函数“register_netdev()”中有获得锁和解锁的过程。

```
ASSERT_RTNL() 相当于下面的代码：不停的获得锁。
 do
 {
     if (unlikely(rtnl_trylock()))  //若
     {
       rtnl_unlock();
       printk(KERN_ERR "RTNL: assertion failed at %s (%d)\n", __FILE__,  __LINE__);
       dump_stack();
     }
  } while(0)
```

提示在哪个文件哪一行

2. ifconfig vnet0 3.3.3.3

ifconfig // 查看

```
# ifconfig vnet0 3.3.3.3
# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:60:6E:33:44:55
          inet addr:192.168.1.17  Bcast:192.168.1.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1676 errors:0 dropped:0 overruns:0 frame:0
          TX packets:689 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:2139996 (2.0 MiB)  TX bytes:109814 (107.2 KiB)
          Interrupt:51 Base address:0xa000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

vnet0     Link encap:Ethernet  HWaddr 00:00:00:00:00:00
          inet addr:3.3.3.3  Bcast:3.255.255.255  Mask:255.0.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

#
```

3. ping 3.3.3.3 // 成功

```
# ping 3.3.3.3
PING 3.3.3.3 (3.3.3.3): 56 data bytes
64 bytes from 3.3.3.3: seq=0 ttl=64 time=0.783 ms
64 bytes from 3.3.3.3: seq=1 ttl=64 time=0.416 ms
64 bytes from 3.3.3.3: seq=2 ttl=64 time=0.405 ms
64 bytes from 3.3.3.3: seq=3 ttl=64 time=0.410 ms
64 bytes from 3.3.3.3: seq=4 ttl=64 time=0.418 ms
```

ping 应用程序可以 ping 通自己，说明 ping 从应用程序中进来，没有经过网卡驱动层直接从应用程序返回了。
这说明“IP” 是纯软件的概念。
ping 3.3.3.4 // 死机

```
Backtrace:
[<c023ddb0>] (dev_hard_start_xmit+0x0/0x240) from [<c024ae40>] (__qdisc_run+0xb0/0x198)
 r8:03030303 r7:c3d15c2c r6:c3cab980 r5:00000000 r4:c3d15c00
```

ping 时会调用“dev_hard_start_xmit”函数。但这个函数没有提供，故死机了。

加上发包函数后，会发现可以 ping 了：
//2,设置

```
vnet_dev->hard_start_xmit = virt_net_sendpacket;


static int virt_net_sendpacket (struct sk_buff *skb,struct net_device *dev)
{//里面什么也没做，只是打印出此函数被调用的次数。
      static int cnt = 0;
      printk("virt_net_sendpacket = %d\n",++cnt);
      return 0;
}
```


重新编译驱动后再 ping：

```
# ping 3.3.3.4
PING 3.3.3.4 (3.3.3.4): 56 data bytes
virt_net_send_packet cnt = 1
virt_net_send_packet cnt = 2
virt_net_send_packet cnt = 3
virt_net_send_packet cnt = 4
virt_net_send_packet cnt = 5
virt_net_send_packet cnt = 6
```

上面发送了 6 个数据包，但统计信息里显示还是“0”个。
在“net_dev”结构体中有统计信息成员：

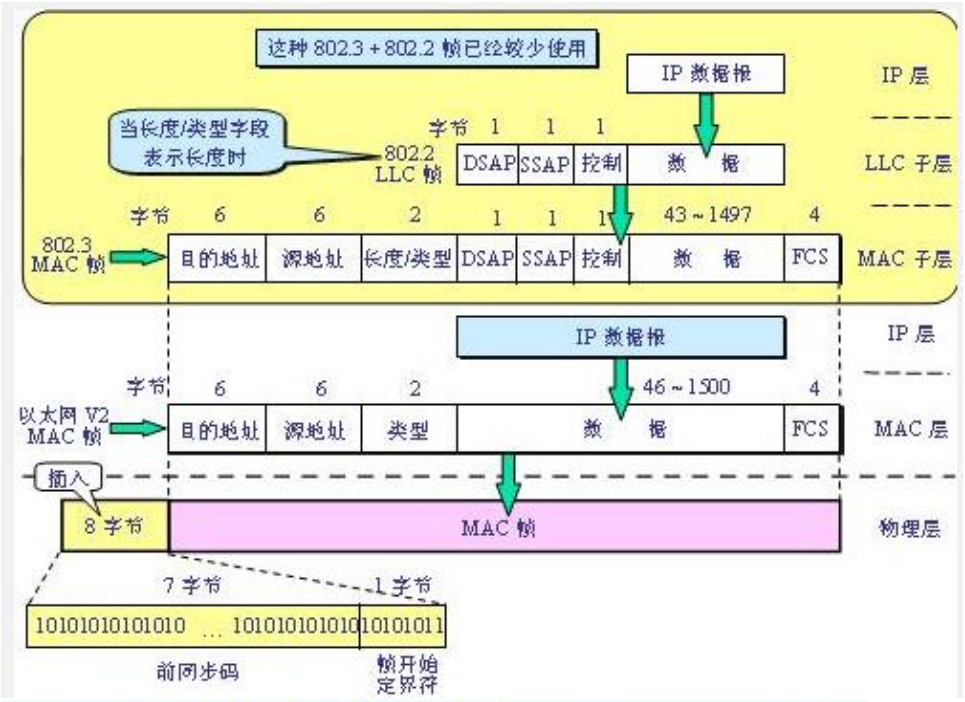
```
struct net_device   {
          …  …
struct net_device_stats* (*get_stats)(struct net_device *dev);
struct net_device_stats          stats;


          …  …
 }
```

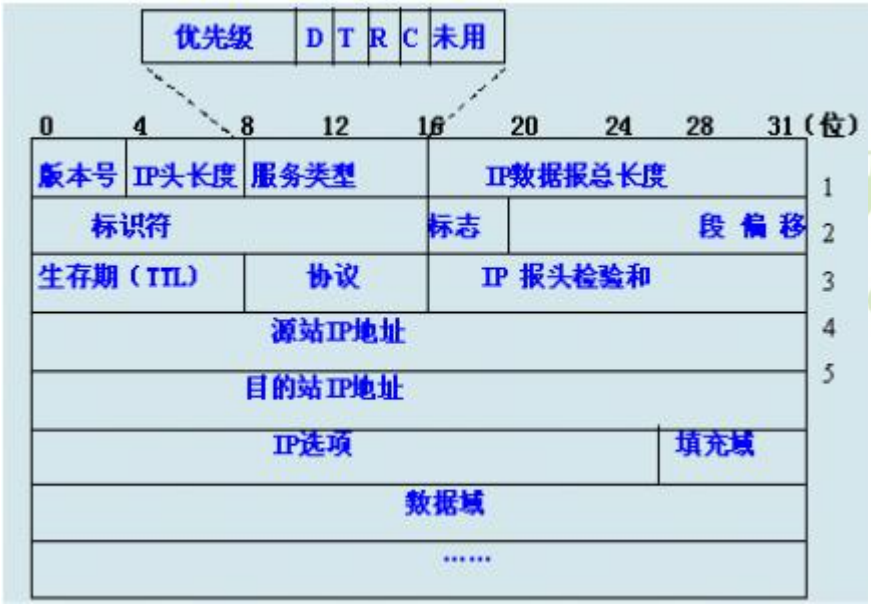
```
struct net_device_stats
{
    unsigned long rx_packets;       /* total packets received   */
    unsigned long tx_packets;       /* total packets transmitted    */
    unsigned long rx_bytes;         /* total bytes received     */
    unsigned long tx_bytes;         /* total bytes transmitted  */
    unsigned long rx_errors;        /* bad packets received     */
    unsigned long tx_errors;        /* packet transmit problems */
    unsigned long rx_dropped;       /* no space in linux buffers    */
    unsigned long tx_dropped;       /* no space available in linux  */
    unsigned long multicast;        /* multicast packets received   */
    unsigned long collisions;

    /* detailed rx_errors: */
    unsigned long rx_length_errors;
    unsigned long rx_over_errors;       /* receiver ring buff overflow  */
    unsigned long rx_crc_errors;        /* recved pkt with crc error    */
    unsigned long rx_frame_errors;      /* recv'd frame alignment error */
    unsigned long rx_fifo_errors;       /* recv'r fifo overrun      */
    unsigned long rx_missed_errors;     /* receiver missed packet   */

    /* detailed tx_errors */
    unsigned long tx_aborted_errors;
    unsigned long tx_carrier_errors;
    unsigned long tx_fifo_errors;
    unsigned long tx_heartbeat_errors;
    unsigned long tx_window_errors;

    /* for cslip etc */
    unsigned long rx_compressed;
    unsigned long tx_compressed;
} ? end net_device_stats ? ;
```

再编译后测试：

```
vnet0     Link encap:Ethernet  HWaddr 08:89:89:89:89:11
          inet addr:3.3.3.3  Bcast:3.255.255.255  Mask:255.0.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:9 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:378 (378.0 B)
```

把数据写入网卡后，网卡并不是立即全部发送出去。网卡发送完数据后会产生一个中断。





一开始是 MAC 头：收发数据包的 MAC 头要对调。

```
struct ethhdr {
    unsigned char    h_dest[ETH_ALEN];    /* destination eth addr   */
    unsigned char    h_source[ETH_ALEN]; /* source ether addr */
    __be16           h_proto;            /* packet type ID field   */
} __attribute__((packed));
```

/ 从硬件读出/保存数据

/* 对调"源/目的"的 mac 地址 */

```
ethhdr = (struct ethhdr *)skb->data; //数据放在skb的data中。
memcpy(tmp_dev_addr, ethhdr->h_dest, ETH_ALEN); //将目的MAC拷贝到临时数组中.
memcpy(ethhdr->h_dest, ethhdr->h_source, ETH_ALEN);//将源、目的MAC对调。
memcpy(ethhdr->h_source, tmp_dev_addr, ETH_ALEN); //对调 源、目的MAC。
```

接着 IP 头也要对调，将 ping 包的类型修改为回复：

```
struct iphdr {
#if defined(__LITTLE_ENDIAN_BITFIELD)
     __u8     ihl:4,
          version:4;
#elif defined (__BIG_ENDIAN_BITFIELD)
     __u8     version:4,
          ihl:4;
#else
#error  "Please fix <asm/byteorder.h>"
#endif
     __u8     tos;
     __be16   tot_len;
     __be16   id;
     __be16   frag_off;
     __u8     ttl;
     __u8     protocol;
     __sum16  check;
     __be32   saddr;
     __be32   daddr;
     /*The options start here. */
} ? end iphdr ? ;
```

/* 对调"源/目的"的 ip 地址 */

```
ih = (struct iphdr *)(skb->data + sizeof(struct ethhdr));
saddr = &ih->saddr;
daddr = &ih->daddr;

tmp = *saddr;           //源IP放到tmp
*saddr = *daddr;      //指针所指内容交换
*daddr = tmp;

//((u8 *)saddr)[2] ^= 1; /* change the third octet (class C) */
//((u8 *)daddr)[2] ^= 1;
type = skb->data + sizeof(struct ethhdr) + sizeof(struct iphdr);
//printk("tx package type = %02x\n", *type);
//  修改类型，原来0x8表示ping
*type = 0; /* 0表示reply */

重新计算校验码：
ih->check = 0;                                  /* and rebuild the checksum (ip needs it) */
ih->check = ip_fast_csum((unsigned char *)ih,ih->ihl);   //校验码要重新计算
```

重新编译并测试：

```
# ping 3.3.3.4
PING 3.3.3.4 (3.3.3.4): 56 data bytes
virt_net_send_packet cnt = 1
64 bytes from 3.3.3.4: seq=0 ttl=64 time=0.809 ms
virt_net_send_packet cnt = 2
64 bytes from 3.3.3.4: seq=1 ttl=64 time=0.454 ms
virt_net_send_packet cnt = 3
64 bytes from 3.3.3.4: seq=2 ttl=64 time=0.456 ms
virt_net_send_packet cnt = 4
64 bytes from 3.3.3.4: seq=3 ttl=64 time=0.450 ms
```

网卡驱动程序编写

测试 DM9000C 驱动程序:

1. 把 dm9dev9000c.c 放到内核的 drivers/net 目录下
2. 修改 drivers/net/Makefile

把

```
obj-$(CONFIG_DM9000) += dm9000.o
```

改为

```
obj-$(CONFIG_DM9000) += dm9dev9000c.o
```

3. make uImage

使用新内核启动

4. 使用 NFS 启动

或

```
ifconfig eth0 192.168.1.17
ping 192.168.1.1
```