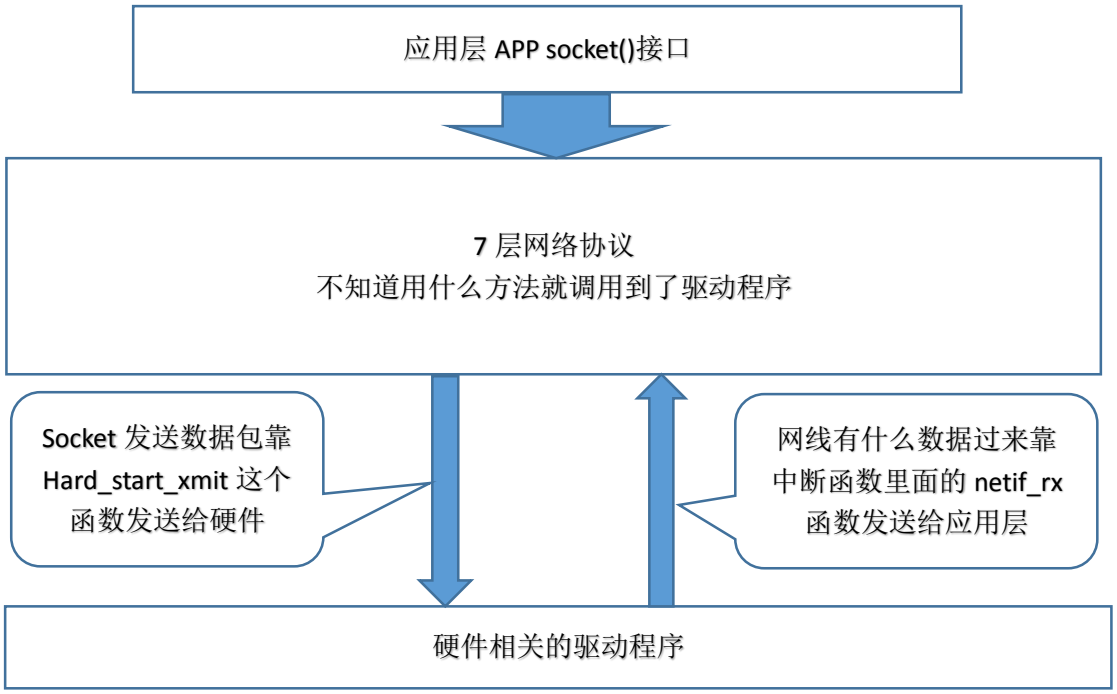
Linux 网络驱动框架

作者：向仔州

基于 IMX6 平台测试，对比 2440 开发平台
用 linux2.6 内核思维去开发 linux3.14.28 内核网卡驱动

网络驱动流程



驱动注册流程

```c
static struct net_device *vnet_dev;

static int virt_net_init(void)
{
        int ret;
        /*1.分配一个net_device结构体*/
        vnet_dev=alloc_netdev(0,"vnet%d",ether_setup);
        /*2.设置*/
        /*3.注册*/
        ret=register_netdev(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);
```

设置设备节点，ifconfig 的时候就可以看到这个节点名，这个名字可以自己取

这个注册函数在 3.14 内核上，开发板上电加载这个驱动会报错，所以这里写的代码仅供参考

网络包发送

```
static struct net_device *vnet_dev;

static int virt_net_send_packet(struct sk_buff *skb, struct net_device *dev)
{
        printf("这是发送网络包函数，ping另外一个IP应该没有问题，ping就是给另外一台服务器发送请求");
        return 0;
}


static int virt_net_init(void)
{
        int ret;
        /*1.分配一个net_device结构体*/
        vnet_dev=alloc_netdev(0,"vnet%d",ether_setup);
        /*2.设置*/

        vnet_dev->hard_start_xmit = virt_net_send_packet;
        /*3.注册*/
        ret=register_netdev(vnet_dev);
        return 0;
}
```

网络包发送函数

这是 2.6 内核给发送函数发送网络包的方式

```
In function 'virt_net_send_packet':
5:2: error: implicit declaration of function 'printf' [-Werror=implicit-function-declar
5:2: warning: incompatible implicit declaration of built-in function 'printf' [enabled

In function 'virt_net_init':
7:10: error: 'struct net_device' has no member named 'hard_start_xmit'
```

用 3.14 新内核去编译 2.6 老内核写的驱动是会报错的，因为 net_device 数据结构变了。

```
static struct net_device *vnet_dev;

static int virt_net_send_packet(struct sk_buff *skb, struct net_device *dev)
{
        printk("这是发送网络包函数，ping另外一个IP应该没有问题，ping就是给另外一台服务器发送请求");
        return 0;
}

static const struct net_device_ops dm9000_netdev_ops = {
        .ndo_start_xmit= virt_net_send_packet,
};

static int virt_net_init(void)
{
        int ret;
        /*1.分配一个net_device结构体*/
        vnet_dev=alloc_netdev(0,"xzznet%d",ether_setup);
        /*2.设置*/

//      vnet_dev->netdev_ops= &dm9000_netdev_ops;
        /*3.注册*/
        ret=register_netdev(vnet_dev);
        return 0;
}
```

这样编译就不会出错

```
root@imx6qdlsolo:/mnt# insmod 3.14virt_net.ko
Unable to handle kernel NULL pointer dereference at virtual address 00000000
pgd = 86ecc000
[00000000] *pgd=16911831, *pte=00000000, *ppte=00000000
Internal error: Oops: 17 [#1] PREEMPT SMP ARM
Modules linked in: 3.14virt_net(O+) volume_key(O) AP_imx6_switch(O) RGB_switch(O) rgb
) power_led(O) bh1750md(O) temphumi_sencer(O) dm1302web(O) lhm_crimagelmd(O)
CPU: 0 PID: 878 Comm: insmod Tainted: G           O 3.14.28+g91cf351 #299
task: 86d29680 ti: 86b80000 task.ti: 86b80000
PC is at register_netdevice+0x54/0x430
LR is at 0x6e7a78
pc : [<805917ac>]    lr : [<6e7a78>]    psr: 60030013
sp : 86b81dd0  ip : 00000030  fp : 7f0290f4
r10: 7f029000  r9 : 86b81f60  r8 : 7f0290ac
r7 : 00000001  r6 : 7f0290b8  r5 : 00000000  r4 : 86e6d800
r3 : 00000000  r2 : 80000000  r1 : 86b81db7  r0 : 00000000
Flags: nZCv  IRQs on  FIQs on  Mode SVC_32  ISA ARM  Segment user
Control: 10c53c7d  Table: 16ecc059  DAC: 00000015
Process insmod (pid: 878, stack limit = 0x86b80238)
Stack: (0x86b81dd0 to 0x86b82000)
1dc0:                                     86d29680 86e6d800 00000000 7f0290b8
1de0: 00000001 7f0290ac 86b81f60 7f029000 7f0290f4 805918dc 86b80000 7f02903c
1e00: 00000001 00000000 00000000 80008758 00000000 86bab100 8040003f 7f029000
1e20: 00000001 806caf34 00000000 86bab240 8040003e 8009d394 80d5c2bc 8bb273c0
1e40: 00000000 8040003e 8bb273c0 8040003f 00000000 00000000 8bd2e560 86001f00
1e60: 7f0290f4 8040003e 00000001 86bab240 00022a64 800cc23c 80d5c2ec 86b81f58
1e80: 00000001 7f0290b8 00000001 7f0290ac 86b81f60 86b66540 7f0290f4 80081d10
1ea0: 7f0290b8 00007fff 8007f984 00000000 86b81f58 8007f168 7f0291f4 7f0290ac
1ec0: 00000000 86b80000 86f00540 a09e5910 00000000 7f0290b8 6e72656b 00006c65
```

但是 insmod 内核的时候会崩

```c
static struct net_device *vnet_dev;

static int virt_net_send_packet(struct sk_buff *skb, struct net_device *dev)
{
        printk("这是发送网络包函数，ping另外一个IP应该没有问题，ping就是给另外一台服务器发送请求");
        return 0;
}

static const struct net_device_ops dm9000_netdev_ops = {
        .ndo_start_xmit= virt_net_send_packet,
};

static int virt_net_init(void)
{
        int ret;
        /*1.分配一个net_device结构体*/
        vnet_dev=alloc_netdev(0,"xzznet%d",ether_setup);
        /*2.设置*/

//      vnet_dev->netdev_ops= &dm9000_netdev_ops;
        /*3.注册*/
        ret=register_netdev(vnet_dev);
        return 0;
}
```

那是因为网卡驱动加载内核的时候就会自动去执行发包函数，去请求网线自动分配 IP 地址，但是我这里没有写这个发包函数所以内核崩了

```c
static struct net_device *vnet_dev;

static int virt_net_send_packet(struct sk_buff *skb, struct net_device *dev)
{
        printk("这是发送网络包函数，ping另外一个IP应该没有问题，ping就是给另外一台服务器发送请求");
        return 0;
}

static const struct net_device_ops dm9000_netdev_ops = {
        .ndo_start_xmit= virt_net_send_packet,
};

static int virt_net_init(void)
{
        int ret;
        /*1.分配一个net_device结构体*/
        vnet_dev=alloc_netdev(0,"xzznet%d",ether_setup);
        /*2.设置*/

        vnet_dev->netdev_ops= &dm9000_netdev_ops;
        /*3.注册*/
        ret=register_netdev(vnet_dev);
        return 0;
}
```

将发包函数加进来

```
root@imx6qdlsolo:/mnt# insmod 3.14virt_net.ko
root@imx6qdlsolo:/mnt#
```

加载虚拟网卡驱动没有问题了

```
root@imx6qdlsolo:/mnt# ifconfig xzznet0 up
root@imx6qdlsolo:/mnt# 这是发送网络包函数，ping另外一个IP应该没有问题，ping就是给另外一台服务器发送请求
```

启动我自己设置名字的网卡，驱动程序就会去调用发包函数。

```
xzznet0   Link encap:Ethernet  HWaddr 00:00:00:00:00:00
          inet6 addr: fe80::200:ff:fe00:0/64 Scope:Link
          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)

root@imx6qdlsolo:/mnt#
```

我们现在将发包函数里面的打印改成能正常测试的一些东西

```
static struct net_device *vnet_dev;

static int virt_net_send_packet(struct sk_buff *skb, struct net_device *dev)
{
        static int cnt=0;
        printk("发包次数  %d\n",++cnt);
        return 0;
}

static const struct net_device_ops dm9000_netdev_ops = {
        .ndo_start_xmit= virt_net_send_packet,
};

static int virt_net_init(void)
{
        int ret;
        /*1.分配一个net_device结构体*/
        vnet_dev=alloc_netdev(0,"xzznet%d",ether_setup);
        /*2.设置*/

        vnet_dev->netdev_ops= &dm9000_netdev_ops;
        /*3.注册*/
        ret=register_netdev(vnet_dev);
        return 0;
}
```

修改发包函数
计算发包次数

```
root@imx6qdlsolo:/mnt# ifconfig xzznet0 up
root@imx6qdlsolo:/mnt# 发包次数  1
发包次数  2
发包次数  3
发包次数  4
发包次数  5
发包次数  6
发包次数  7
发包次数  8
```

网卡启动也要
发 8 个包
一包数据 8 个
字节

启动自己名字的网卡会出现发包

```
root@imx6qdlsolo:/mnt# ifconfig xzznet0 3.3.3.3
root@imx6qdlsolo:/mnt# 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=0.188 ms
64 bytes from 3.3.3.3: seq=1 ttl=64 time=0.133 ms
64 bytes from 3.3.3.3: seq=2 ttl=64 time=0.095 ms
64 bytes from 3.3.3.3: seq=3 ttl=64 time=0.085 ms
64 bytes from 3.3.3.3: seq=4 ttl=64 time=0.079 ms
^C
--- 3.3.3.3 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max = 0.079/0.116/0.188 ms
```

8 X 8=64
也就是一个包 8 字节，8 个包 64 位

网卡 ping 自己也是 8 个包，那么我们 ping 下别人

```
root@imx6qdlsolo:/mnt# ping 3.3.3.4
PING 3.3.3.4 (3.3.3.4): 56 data b发包次数  9
ytes
发包次数  10
发包次数  11
```

ping 别人就是执行发包函数

真实网卡发送函数是怎么样的？

```c
static int virt_net_send_packet(struct sk_buff *skb, struct net_device *dev)
{
        static int cnt=0;
        printk("发包次数  %d\n",++cnt);
        /*真实网卡会把skb接收到应用层的数据在这里发送给网卡硬件*/
        return 0;
}

static const struct net_device_ops dm9000_netdev_ops = {
        .ndo_start_xmit= virt_net_send_packet,
};
```

比如像 3.14.28 内核 DM9000 网卡这样的做法

```c
dm9000_start_xmit(struct sk_buff *skb, struct net_device *dev)
{
    unsigned long flags;
    board_info_t *db = netdev_priv(dev);

    dm9000_dbg(db, 3, "%s:\n", __func__);

    if (db->tx_pkt_cnt > 1)
        return NETDEV_TX_BUSY;

    spin_lock_irqsave(&db->lock, flags);

    /* Move data to DM9000 TX RAM */
    writeb(DM9000_MWCMD, db->io_addr);

    (db->outblk)(db->io_data, skb->data, skb->len);
    dev->stats.tx_bytes += skb->len;

    db->tx_pkt_cnt++;
    /* TX control: First packet immediately send, second packet queue */
    if (db->tx_pkt_cnt == 1) {
        dm9000_send_packet(dev, skb->ip_summed, skb->len);
```

```c
static void dm9000_send_packet(struct net_device *dev,
                   int ip_summed,
                   u16 pkt_len)
{
    board_info_t *dm = to_dm9000_board(dev);

    /* The DM9000 is not smart enough to leave fragmented packets alone. *
    if (dm->ip_summed != ip_summed) {
        if (ip_summed == CHECKSUM_NONE)
            iow(dm, DM9000_TCCR, 0);
        else
            iow(dm, DM9000_TCCR, TCCR_IP | TCCR_UDP | TCCR_TCP);
        dm->ip_summed = ip_summed;
    }

    /* Set TX length to DM9000 */
    iow(dm, DM9000_TXPLL, pkt_len);
    iow(dm, DM9000_TXPLH, pkt_len >> 8);

    /* Issue TX polling command */
    iow(dm, DM9000_TCR, TCR_TXREQ); /* Cleared after TX complete */
} ? end dm9000_send_packet ?
```

```c
iow(board_info_t *db, int reg, int value)
{
    writeb(reg, db->io_addr);
    writeb(value, db->io_data);
}
```

这就是发送给网卡寄存器

我们现在发现个问题，就是发送数据包之后没有统计信息

```
root@imx6qdlsolo:/mnt# ping 3.3.3.4
PING 3.3.3.4 (3.3.3.4): 56 data b发包次数  9
ytes
发包次数  10
发包次数  11
发包次数  12
发包次数  13
发包次数  14
发包次数  15
发包次数  16
发包次数  17
发包次数  18
^C
xzznet0   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
          inet6 addr: fe80::200:ff:fe00:0/64 Scope:Link
          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)
```

没有发送包数据统计

这个时候我们要修改.ndo_start_xmit= virt_net_send_packet,发送函数里面的代码

```
static int virt_net_send_packet(struct sk_buff *skb, struct net_device *dev)
{
        static int cnt=0;
        printk("发包次数  %d\n",++cnt);
        /*真实网卡会把skb接收到应用层的数据在这里发送给网卡硬件*/
        dev->stats.tx_packets++; //发送的包
        dev->stats.tx_bytes += skb->len;//每包的字节数
        return 0;
}
```

```
root@imx6qdlsolo:/mnt# ifconfig xzznet0 up
root@imx6qdlsolo:/mnt# 发包次数  1
发包次数  2
发包次数  3
发包次数  4
发包次数  5
发包次数  6
发包次数  7
发包次数  8

root@imx6qdlsolo:/mnt# ifconfig
eth0      Link encap:Ethernet  HWaddr CA:5D:E0:78:33:5B
          UP BROADCAST 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)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:10 errors:0 dropped:0 overruns:0 frame:0
          TX packets:10 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:700 (700.0 B)  TX bytes:700 (700.0 B)

xzznet0   Link encap:Ethernet  HWaddr 00:00:00:00:00:00
          inet6 addr: fe80::200:ff:fe00:0/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:648 (648.0 B)
```

网卡启动发了 8 个包

统计为 8 个包

8 个包 648 个字节

```
root@imx6qdlsolo:/mnt# ifconfig xzznet0 3.3.3.3
root@imx6qdlsolo:/mnt# ping 3.3.3.4
PING 3.3.3.4 (3.3.3.4): 56 data b发包次数  9
ytes
发包次数  10
发包次数  11
发包次数  12
发包次数  13
发包次数  14
发包次数  15
发包次数  16
^C
--- 3.3.3.4 ping statistics ---
8 packets transmitted, 0 packets received, 100% packet loss
root@imx6qdlsolo:/mnt# 发包次数  17

root@imx6qdlsolo:/mnt# ifconfig
eth0      Link encap:Ethernet  HWaddr CA:5D:E0:78:33:5B
          UP BROADCAST 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)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:18 errors:0 dropped:0 overruns:0 frame:0
          TX packets:18 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1596 (1.5 KiB)  TX bytes:1596 (1.5 KiB)

xzznet0   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
          inet6 addr: fe80::200:ff:fe00:0/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:17 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:1026 (1.0 KiB)
```

我 ping 外网有接着发包，现在发了 17 个包，证明数据发送包统计功能搞定。
每次发完一个包之后我们网卡都要停止队列，释放 skb_buff，为下一次发送留空间

```
static int virt_net_send_packet(struct sk_buff *skb, struct net_device *dev)
{
        static int cnt=0;
        printk("发包次数  %d\n",++cnt);
        /*真实网卡会把skb接收到应用层的数据在这里发送给网卡硬件*/
        dev->stats.tx_packets++; //发送的包
        dev->stats.tx_bytes += skb->len;//每包的字节数

        netif_stop_queue(dev);//停止队列
        dev_kfree_skb(skb);//释放sk_buff
        return 0;
}
```

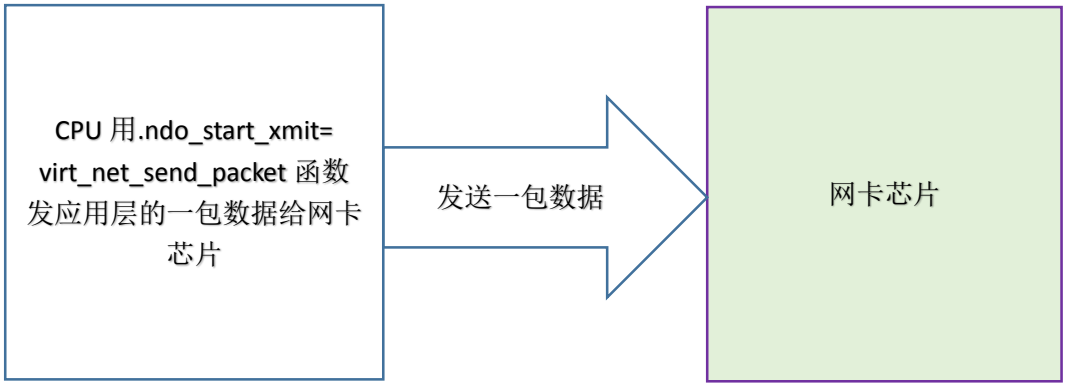
增加这两段代码

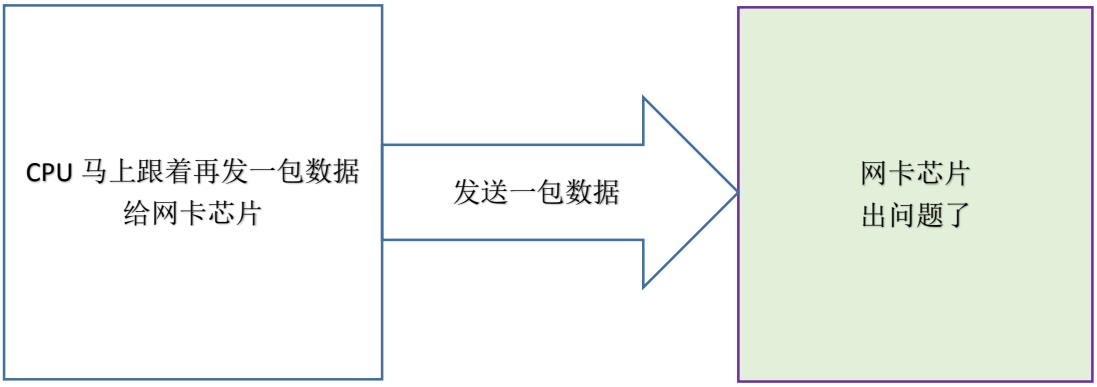
```
root@imx6qdlsolo:/mnt# ifconfig xzznet0 up
root@imx6qdlsolo:/mnt# 发包次数  1

root@imx6qdlsolo:/mnt# ifconfig xzznet0 3.3.3.3
root@imx6qdlsolo:/mnt# ping 3.3.3.4
PING 3.3.3.4 (3.3.3.4): 56 data bytes
```

但是停止队列之后，发送函数只执行一次然后就再也无法重复执行了，上面那种启动网卡发送 8 个字节才是正常的啊，这里为什么出现这种情况？

那是因为你发送一包数据后停止队列，但是你没有再次唤醒队列，所以发送函数无法执行

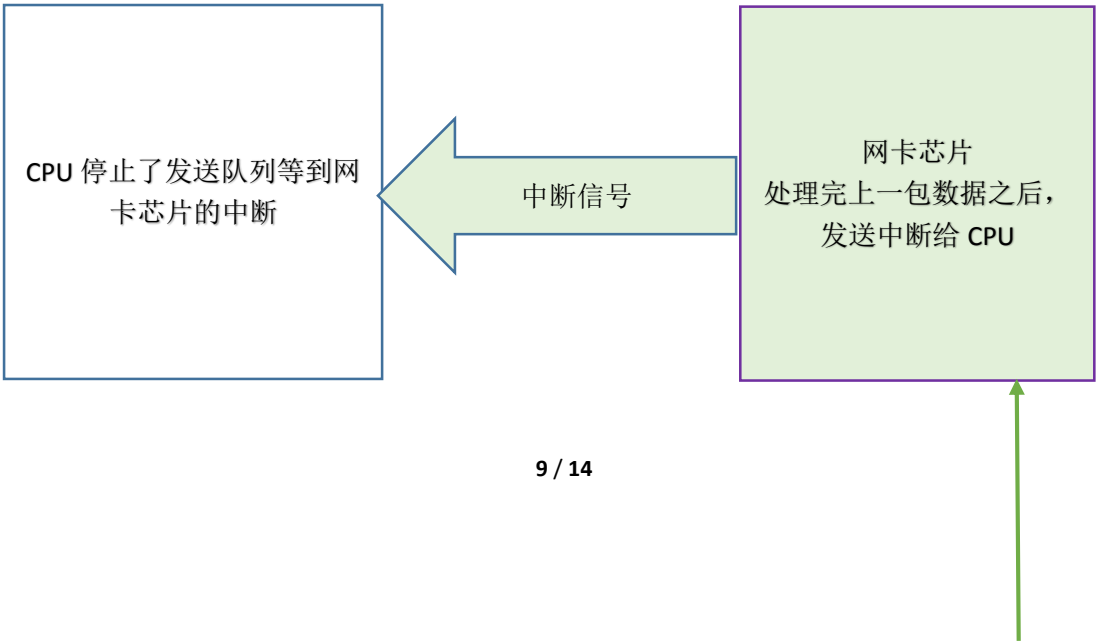




为什么第二包数据网卡芯片出问题呢？因为你 CPU 根本不知道网卡芯片接受 CPU 上一包发来的数据处理完没有，又鼓捣给别个网卡再发一包，你说网卡芯片涨不涨得死。

```
netif_stop_queue(dev);//停止队列
```

所以停止队列让 CPU 稍等一下



```c
static int virt_net_send_packet(struct sk_buff *skb, struct net_device *dev)
{
        static int cnt=0;
        printk("发包次数  %d\n",++cnt);
        /*真实网卡会把skb接收到应用层的数据在这里发送给网卡硬件*/
        dev->stats.tx_packets++; //发送的包
        dev->stats.tx_bytes += skb->len;//每包的字节数

        /*.....在这个位置把接受到的包发给网卡芯片.....*/
        /*.....*/

        netif_stop_queue(dev);//然后停止队列
        dev_kfree_skb(skb);//释放sk_buff

        netif_wake_queue(dev);//唤醒发送队列
        return 0;
}
```

这里是为了方便虚拟网卡实验，实际这个唤醒队列代码要放到网卡发给 CPU 的中断函数里面

```
root@imx6qdlsolo:/mnt# ifconfig xzznet0 up
root@imx6qdlsolo:/mnt# 发包次数  1
发包次数  2
发包次数  3
发包次数  4
发包次数  5
发包次数  6
发包次数  7
发包次数  8

root@imx6qdlsolo:/mnt# ifconfig xzznet0 3.3.3.3
root@imx6qdlsolo:/mnt# ping 3.3.3.4
PING 3.3.3.4 (3.3.3.4): 56 data b发包次数  9
ytes
发包次数  10
发包次数  11
发包次数  12
发包次数  13
发包次数  14
发包次数  15
发包次数  16
^C
--- 3.3.3.4 ping statistics ---
8 packets transmitted, 0 packets received, 100%
```

这里记录每次 ping 的发包数量，而不是总共发了多少包的数量

网络包接受

```c
static int virt_net_send_packet(struct sk_buff *skb, struct net_device *dev)
{
        static int cnt=0;
        printk("发包次数  %d\n",++cnt);
        /*真实网卡会把skb接收到应用层的数据在这里发送给网卡硬件*/
        dev->stats.tx_packets++; //发送的包
        dev->stats.tx_bytes += skb->len;//每包的字节数

        /*.....在这个位置把接受到的包发给网卡芯片.................*/
        /*.........................................................*/
        /* 构造一个假的sk_buff,上报 */
        emulator_rx_packet(skb, dev);

        netif_stop_queue(dev);//然后停止队列
        dev_kfree_skb(skb);//释放sk_buff

        netif_wake_queue(dev);//唤醒发送队列
        return 0;
}
```

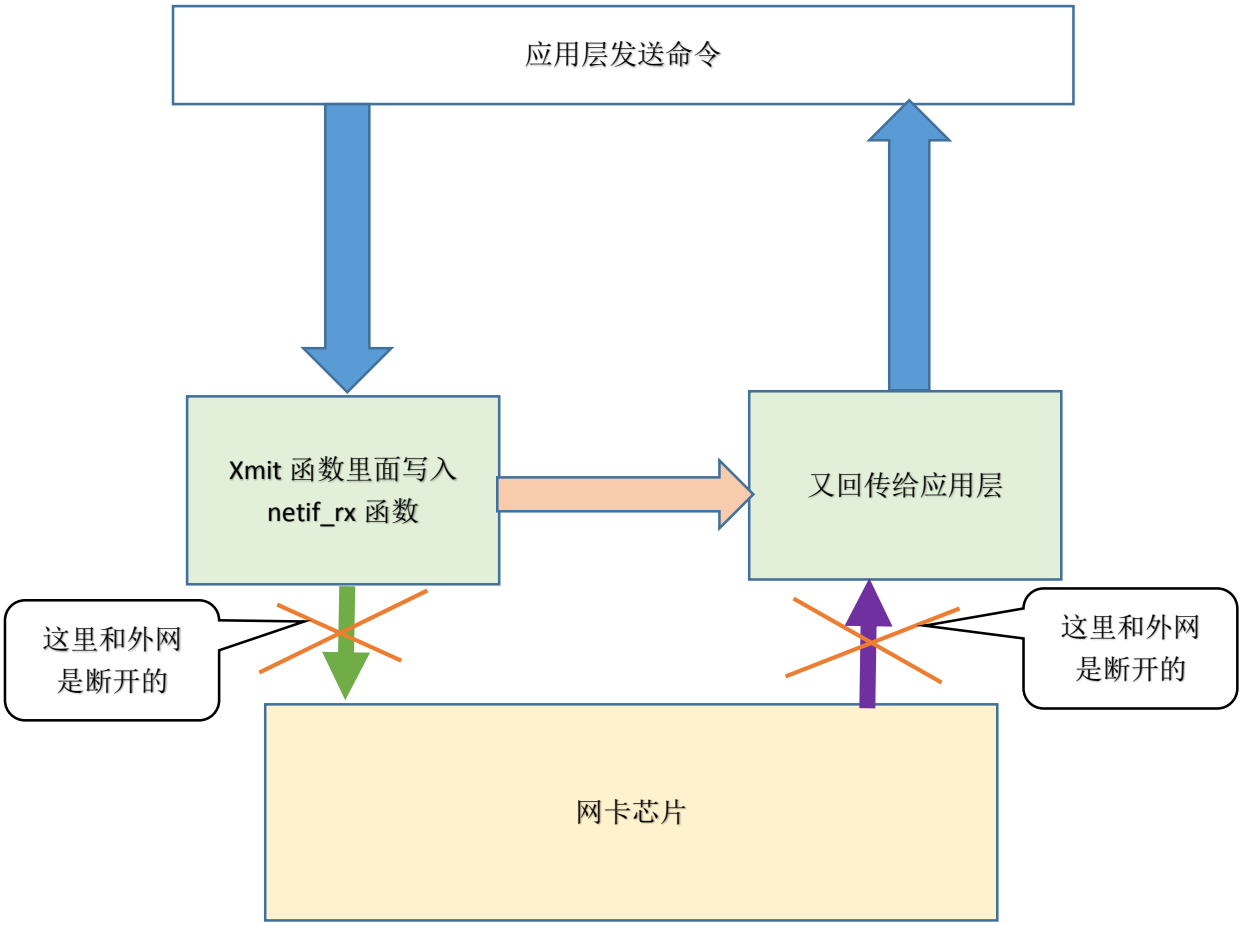
我们假装把发出去的数据包 skb 向接受包里面扔

```c
static void emulator_rx_packet(struct sk_buff *skb, struct net_device *dev)
{
        unsigned char *type;
        struct iphdr *ih;
        __be32 *saddr, *daddr, tmp;
        unsigned char tmp_dev_addr[ETH_ALEN];
        struct ethhdr *ethhdr;
        struct sk_buff *rx_skb;
        /***********从硬件读出/保存数据***********/
        ethhdr = (struct ethhdr *)skb->data;
        memcpy(tmp_dev_addr, ethhdr->h_dest, ETH_ALEN);
        memcpy(ethhdr->h_dest, ethhdr->h_source, ETH_ALEN);
        memcpy(ethhdr->h_source, tmp_dev_addr, ETH_ALEN);
        /* 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;
        *saddr = *daddr;
        *daddr = tmp;
        type = skb->data + sizeof(struct ethhdr) + sizeof(struct iphdr);
        *type = 0; /* 0表示reply */
        ih->check = 0;   /* and rebuild the checksum (ip needs it) */
        ih->check = ip_fast_csum((unsigned char *)ih,ih->ihl);
        // 构造一个sk_buff
        rx_skb = dev_alloc_skb(skb->len + 2);
        skb_reserve(rx_skb, 2); /* align IP on 16B boundary */
        memcpy(skb_put(rx_skb, skb->len), skb->data, skb->len);
        /* Write metadata, and then pass to the receive level */
        rx_skb->dev = dev;
        rx_skb->protocol = eth_type_trans(rx_skb, dev);
        rx_skb->ip_summed = CHECKSUM_UNNECESSARY; /* don't check it */
        dev->stats.rx_packets++;
        dev->stats.rx_bytes += skb->len;
        // 提交sk_buff
        netif_rx(rx_skb);
}
```

接受程序提取 skb 给 rx_skb

将 rx_skb 发给应用层，其实 rx_skb 就是发送出去的 skb

其实这个发包收包模拟过程是这样的




```
xzznet0   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
          inet6 addr: fe80::200:ff:fe00:0/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:863 errors:0 dropped:0 overruns:0 frame:0
          TX packets:863 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:36558 (35.7 KiB)  TX bytes:36558 (35.7 KiB)

root@imx6qdlsolo:/mnt#
```

这样只是为了能让 RX 接受字节数量也能统计出来，其实没有任何意义。

实际的网卡驱动里面是不能在 xmit 函数里面写 netif_rx 函数的

所以我们这里实现的网络接受就是将发出去的数据又接受回来，模拟老一个过程。实际过程中 netif_rx()函数绝对是放在芯片接受中断类函数里面的。

比如 3.14.28 内核的 DM9000 网卡驱动

```
static const struct net_device_ops dm9000_netdev_ops = {
    .ndo_open       = dm9000_open,
    .ndo_stop       = dm9000_stop,
    .ndo_start_xmit     = dm9000_start_xmit,
    .ndo_tx_timeout     = dm9000_timeout,
    .ndo_set_rx_mode    = dm9000_hash_table,
    .ndo_do_ioctl       = dm9000_ioctl,
    .ndo_change_mtu     = eth_change_mtu,
    .ndo_set_features   = dm9000_set_features,
    .ndo_validate_addr  = eth_validate_addr,
    .ndo_set_mac_address    = eth_mac_addr,
#ifdef CONFIG_NET_POLL_CONTROLLER
    .ndo_poll_controller    = dm9000_poll_controller,
#endif
};
```

```
static void dm9000_poll_controller(struct net_device *dev)
{
    disable_irq(dev->irq);
    dm9000_interrupt(dev->irq, dev);
    enable_irq(dev->irq);
}
```

```
static irqreturn_t dm9000_interrupt(int irq, void *dev_id)
{
```

```
/* Received the coming packet */
if (int_status & ISR_PRS)
    dm9000_rx(dev);
```

```
dm9000_rx(struct net_device *dev)
{
    board_info_t *db = netdev_priv(dev);
    struct dm9000_rxhdr rxhdr;
    struct sk_buff *skb;
    u8 rxbyte, *rdptr;
    bool GoodPacket;
    int RxLen;
```

```
            netif_rx(skb);
            dev->stats.rx_packets++;

        } ? end if GoodPacket&&((skb=net... ?  else {
```

IMX6 以太网 AR8031 降频处理

因为硬件问题，导致 AR8031 网口端也就是水晶头端无法实现 1000M 传输，但是 100M 传输是很正常的，所以为了让开发板插入 1000M 路由器，也能识别成 100M，需要修改内核代码。

`fsl-linux/drivers/net/ethernet/freescale` 在这个路径下

`fec_main.c` 打开这个 C 文件

```
1954          if (fep->quirks & FEC_QUIRK_HAS_GBIT) {
1955                  phy_dev->supported &= PHY_BASIC_FEATURES;
1956          //      phy_dev->supported &= PHY_GBIT_FEATURES;
1957          //      phy_dev->supported &= ~SUPPORTED_1000baseT_Half;
1958  #if !defined(CONFIG_M5272)
1959                  phy_dev->supported |= SUPPORTED_Pause;
1960  #endif
1961          }
1962          else
1963                  phy_dev->supported &= PHY_BASIC_FEATURES;
```

将 1956 和 1957 行代码屏蔽掉，把 1963 行的代码复制上去。这样就实现了进入 1000M 也初始化 100M 的代码，进入 100M 也是初始化 100M 的代码。

然后编译内核，烧写。