分类：
Linux网卡驱动分析
学习应该是一个先把问题简单化，在把问题复杂化的过程。一开始就着手处理复杂的问题，难免让人有心惊胆颤，捉襟见肘的感觉。读Linux网卡驱动也是一样。那长长的源码夹杂着那些我们陌生的变量和符号，望而生畏便是理所当然的了。不要担心，事情总有解决的办法，先把一些我们管不着的代码切割出去，留下必须的部分，把框架掌握了，哪其他的事情自然就水到渠成了，这是笔者的心得。
一般在使用的Linux网卡驱动代码动辄3000行左右，这个代码量以及它所表达出来的知识量无疑是庞大的，我们有没有办法缩短一下这个代码量，使我们的学习变的简单些呢，经过笔者的不懈努力，在仍然能够使网络设备正常工作的前提下，把它缩减到了600多行，我们把暂时还用不上的功能先割出去。这样一来，事情就简单多了，真的就剩下一个框架了。下面我们就来剖析这个可以执行的框架。
限于篇幅，以下分析用到的所有涉及到内核中的函数代码，我都不予列出，但给出在哪个具体文件中，请读者自行查阅。
首先，我们来看看设备的初始化。当我们正确编译完我们的程序后，我们就需要把生成的目标文件加载到内核中去，我们会先ifconfig eth0 down和rmmod 8139too来卸载正在使用的网卡驱动，然后insmod 8139too.o把我们的驱动加载进去（其中8139too.o是我们编译生成的目标文件）。就像C程序有主函数main（）一样，模块也有第一个执行的函数，即module_init(rtl8139_init_module);在我们的程序中，rtl8139_init_module（）在 insmod之后首先执行，它的代码如下：
```
static int __init rtl8139_init_module (void)
{
return pci_module_init (&rtl8139_pci_driver);
}
```
它直接调用了pci_module_init（），这个函数代码在Linux/drivers/net/eepro100.c中，并且把 rtl8139_pci_driver（这个结构是在我们的驱动代码里定义的，它是驱动程序和PCI设备联系的纽带）的地址作为参数传给了它。 rtl8139_pci_driver定义如下：
```
static struct pci_driver rtl8139_pci_driver = {
name: MODNAME,
id_table: rtl8139_pci_tbl,
probe: rtl8139_init_one,
remove: rtl8139_remove_one,
};
```
pci_module_init（）在驱动代码里没有定义，你一定想到了，它是Linux内核提供给模块是一个标准接口，那么这个接口都干了些什么，笔者跟踪了这个函数。里面调用了pci_register_driver（），这个函数代码在Linux/drivers/pci/pci.c中， pci_register_driver做了三件事情。
①是把带过来的参数rtl8139_pci_driver在内核中进行了注册，内核中有一个PCI设备的大的链表，这里负责把这个PCI驱动挂到里面去。
②是查看总线上所有PCI设备（网卡设备属于PCI设备的一种）的配置空间如果发现标识信息与rtl8139_pci_driver中的id_table相同即rtl8139_pci_tbl，而它的定义如下：
```
static struct pci_device_id rtl8139_pci_tbl[] __devinitdata = {
{0x10ec, 0x8129, PCI_ANY_ID, PCI_ANY_ID, 0, 0, 1},
{PCI_ANY_ID, 0x8139, 0x10ec, 0x8139, 0, 0,0 },
{0,}
};
```
, 那么就说明这个驱动程序就是用来驱动这个设备的，于是调用rtl8139_pci_driver中的probe函数即rtl8139_init_one, 这个函数是在我们的驱动程序中定义了的，它是用来初始化整个设备和做一些准备工作。这里需要注意一下pci_device_id是内核定义的用来辨别不同PCI设备的一个结构，例如在我们这里0x10ec代表的是Realtek公司，我们扫描PCI设备配置空间如果发现有Realtek公司制造的设备时，两者就对上了。当然对上了公司号后还得看其他的设备号什么的，都对上了才说明这个驱动是可以为这个设备服务的。
③是把这个rtl8139_pci_driver结构挂在这个设备的数据结构（pci_dev）上，表示这个设备从此就有了自己的驱动了。而驱动也找到了它服务的对象了。
PCI是一个总线标准，PCI总线上的设备就是PCI设备，这些设备有很多类型，当然也包括网卡设备，每一个PCI设备在内核中抽象为一个数据结构 pci_dev,它描述了一个PCI设备的所有的特性，具体请查询相关文档，本文限于篇幅无法详细描述。但是有几个地方和驱动程序的关系特别大，必须予以说明。PCI设备都遵守PCI标准，这个部分所有的PCI设备都是一样的，每个PCI设备都有一段寄存器存储着配置空间，这一部分格式是一样的，比如第一个寄存器总是生产商号码，如Realtek就是10ec，而Intel则是另一个数字，这些都是商家像标准组织申请的，是肯定不同的。我就可以通过配置空间来辨别其生产商，设备号，不论你什么平台，x86也好，ppc也好，他们都是同一的标准格式。当然光有这些PCI配置空间的统一格式还是不够的，比如说人类，都有鼻子和眼睛，但并不是所有人的鼻子和眼睛都长的一样的。网卡设备是PCI设备必须遵守规则，在设备里集成了PCI配置空间，但它是一个网卡就必须同时集成能控制网卡工作的寄存器。而寄存器的访问就成了一个问题。在Linux里面我们是把这些寄存器映射到主存虚拟空间上的，换句话说我们的CPU访存指令就可以访问到这些处于外设中的控制寄存器。总结一下PCI设备主要包括两类空间，一个是配置空间，它是操作系统或BIOS控制外设的统一格式的空间，CPU指令不能访问，访问这个空间要借助BIOS功能，事实上Linux的访问配置空间的函数是通过CPU指令驱使BIOS来完成读写访问的。而另一类是普通的控制寄存器空间，这一部分映射完后CPU可以访问来控制设备工作。
现在我们回到上面pci_register_driver的第二步，如果找到相关设备和我们的pci_device_id结构数组对上号了，说明我们找到服务对象了，则调用rtl8139_init_one,它主要做了七件事：
① 建立net_device结构，让它在内核中代表这个网络设备。但是读者可能会问，pci_dev也是代表着这个设备，那么两者有什么区别呢，正如我们上面讨论的，网卡设备既要遵循PCI规范，也要担负起其作为网卡设备的职责，于是就分了两块，pci_dev用来负责网卡的PCI规范，而这里要说的 net_device则是负责网卡的网络设备这个职责。
```
dev = init_etherdev (NULL, sizeof (*tp));
if (dev == NULL) {
printk ("unable to alloc new ethernet/n");
```

```
return -ENOMEM;
}
tp = dev->priv;
```
init_etherdev函数在Linux/drivers/net/net_init.c中，在这个函数中分配了net_device的内存并进行了初始化。这里值得注意的是net_device中的一个成员priv，它代表着不同网卡的私有数据，比如Intel的网卡和Realtek的网卡在内核中都是以net_device来代表。但是他们是有区别的，比如Intel和Realtek实现同一功能的方法不一样，这些都是靠着priv来体现。所以这里把拿出来同net_device相提并论。分配内存时，net_device中除了priv以外的成员都是固定的，而priv的大小是可以任意的，所以分配时要把priv的大小传过去。
②开启这个设备（其实是开启了设备的寄存器映射到内存的功能）
```
rc = pci_enable_device (pdev);
if (rc)
goto err_out;
```
pci_enable_device 也是一个内核开发出来的接口，代码在drivers/pci/pci.c中，笔者跟踪发现这个函数主要就是把PCI配置空间的Command域的0位和1 位置成了1，从而达到了开启设备的目的，因为rtl8139的官方datasheet中，说明了这两位的作用就是开启内存映射和I/O映射，如果不开的话，那我们以上讨论的把控制寄存器空间映射到内存空间的这一功能就被屏蔽了，这对我们是非常不利的，除此之外，pci_enable_device还做了些中断开启工作。
③获得各项资源
```
mmio_start = pci_resource_start (pdev, 1);
mmio_end = pci_resource_end (pdev, 1);
mmio_flags = pci_resource_flags (pdev, 1);
mmio_len = pci_resource_len (pdev, 1);
```
读者也许疑问我们的寄存器被映射到内存中的什么地方是什么时候有谁决定的呢。是这样的，在硬件加电初始化时，BIOS固件同统一检查了所有的PCI设备，并统一为他们分配了一个和其他互不冲突的地址，让他们的驱动程序可以向这些地址映射他们的寄存器，这些地址被BIOS写进了各个设备的配置空间，因为这个活动是一个PCI的标准的活动，所以自然写到各个设备的配置空间里而不是他们风格各异的控制寄存器空间里。当然只有BIOS可以访问配置空间。当操作系统初始化时，他为每个PCI设备分配了pci_dev结构，并且把BIOS获得的并写到了配置空间中的地址读出来写到了pci_dev中的resource 字段中。这样以后我们在读这些地址就不需要在访问配置空间了，直接跟pci_dev要就可以了，我们这里的四个函数就是直接从pci_dev读出了相关数据，代码在include/linux/pci.h中。定义如下：
```
#define pci_resource_start(dev,bar) ((dev)->resource[(bar)].start)
#define pci_resource_end(dev,bar) ((dev)->resource[(bar)].end)
```
这里需要说明一下，每个PCI设备有0-5一共6个地址空间，我们通常只使用前两个，这里我们把参数1传给了bar就是使用内存映射的地址空间。
④把得到的地址进行映射
```
ioaddr = ioremap (mmio_start, mmio_len);
if (ioaddr == NULL) {
printk ("cannot remap MMIO, aborting/n");
rc = -EIO;
goto err_out_free_res;
}
```
ioremap是内核提供的用来映射外设寄存器到主存的函数，我们要映射的地址已经从pci_dev中读了出来（上一步），这样就水到渠成的成功映射了而不会和其他地址有冲突。映射完了有什么效果呢，我举个例子，比如某个网卡有100 个寄存器，他们都是连在一块的，位置是固定的，加入每个寄存器占4个字节，那么一共400个字节的空间被映射到内存成功后，ioaddr就是这段地址的开头（注意ioaddr是虚拟地址，而mmio_start是物理地址，它是BIOS得到的，肯定是物理地址，而保护模式下CPU不认物理地址，只认虚拟地址），ioaddr+0就是第一个寄存器的地址，ioaddr+4就是第二个寄存器地址（每个寄存器占4个字节），以此类推，我们就能够在内存中访问到所有的寄存器进而操控他们了。
⑤重启网卡设备
重启网卡设备是初始化网卡设备的一个重要部分，它的原理就是向寄存器中写入命令就可以了（注意这里写寄存器，而不是配置空间，因为跟PCI没有什么关系），代码如下：
```
writeb ((readb(ioaddr+ChipCmd) & ChipCmdClear) | CmdReset,ioaddr+ChipCmd);
```
是我们看到第二参数ioaddr+ChipCmd，ChipCmd是一个位移，使地址刚好对应的就是ChipCmd哪个寄存器，读者可以查阅官方datasheet得到这个位移量，我们在程序中定义的这个值为：ChipCmd = 0x37；与datasheet是吻合的。我们把这个命令寄存器中相应位（RESET）置1就可以完成操作。
⑥获得MAC地址，并把它存储到net_device中。
```
for(i = 0; i < 6; i++) { /* Hardware Address */
dev->dev_addr[i] = readb(ioaddr+i);
dev->broadcast[i] = 0xff;
}
```
我们可以看到读的地址是ioaddr+0到ioaddr+5，读者查看官方datasheet会发现寄存器地址空间的开头6个字节正好存的是这个网卡设备的MAC地址，MAC地址是网络中标识网卡的物理地址，这个地址在今后的收发数据包时会用的上。
⑦向net_device中登记一些主要的函数
```
dev->open = rtl8139_open;
dev->hard_start_xmit = rtl8139_start_xmit;
dev->stop = rtl8139_close;
```
由于dev（net_device）代表着设备，把这些函数注册完后，rtl8139_open就是用于打开这个设备， rtl8139_start_xmit就是当应用程序要通过这个设备往外面发数据时被调用，具体的其实这个函数是在网络协议层中调用的，这就涉及到 Linux网络协议栈的内容，不再我们讨论之列，我们只是负责实现它。rtl8139_close用来关掉这个设备。

好了，到此我们把rtl8139_init_one函数介绍完了，初始化个设备完了之后呢，我们通过ifconfig eth0 up命令来把我们的设备激活。这个命令直接导致了我们刚刚注册的rtl8139_open的调用。这个函数激活了设备。这个函数主要做了三件事。
①注册这个设备的中断处理函数。当网卡发送数据完成或者接收到数据时，是用中断的形式来告知的，比如有数据从网线传来，中断也通知了我们，那么必须要有一个处理这个中断的函数来完成数据的接收。关于Linux的中断机制不是我们详细讲解的范畴，有兴趣的可以参考《Linux内核源代码情景分析》，但是有个非常重要的资源我们必须注意，那就是中断号的分配，和内存地址映射一样，中断号也是BIOS在初始化阶段分配并写入设备的配置空间的，然后Linux在建立pci_dev时从配置空间读出这个中断号然后写入pci_dev的irq成员中，所以我们注册中断程序需要中断号就是直接从pci_dev里取就可以了。
```
retval = request_irq (dev->irq, rtl8139_interrupt, SA_SHIRQ, dev->name, dev);
if (retval) {
return retval;
}
```
我们注册的中断处理函数是rtl8139_interrupt，也就是说当网卡发生中断（如数据到达）时，中断控制器8259A把中断号发给CPU，CPU根据这个中断号找到处理程序，这里就是rtl8139_interrupt，然后执行。rtl8139_interrupt也是在我们的程序中定义好了的，这是驱动程序的一个重要的义务，也是一个基本的功能。request_irq 的代码在arch/i386/kernel/irq.c中。
②分配发送和接收的缓存空间
根据官方文档，发送一个数据包的过程是这样的：先从应用程序中把数据包拷贝到一段连续的内存中（这段内存就是我们这里要分配的缓存），然后把这段内存的地址写进网卡的数据发送地址寄存器(TSAD)中,这个寄存器的偏移量是TxAddr0 = 0x20。在把这个数据包的长度写进另一个寄存器（TSD）中，它的偏移量是TxStatus0 = 0x10。然后就把这段内存的数据发送到网卡内部的发送缓冲中(FIFO),最后由这个发送缓冲区把数据发送到网线上。
好了现在创建这么一个发送和接收缓冲内存的目的已经很显然了。
```
tp->tx_bufs = pci_alloc_consistent(tp->pci_dev, TX_BUF_TOT_LEN,
&tp->tx_bufs_dma);
tp->rx_ring = pci_alloc_consistent(tp->pci_dev, RX_BUF_TOT_LEN,
&tp->rx_ring_dma);
```
tp 是net_device的priv的指针，tx_bufs是发送缓冲内存的首地址，rx_ring是接收缓存内存的首地址，他们都是虚拟地址，而最后一个参数tx_bufs_dma和rx_ring_dma均是这一段内存的物理地址。为什么同一个事物，既用虚拟地址来表示它还要用物理地址呢，是这样的， CPU执行程序用到这个地址时，用虚拟地址，而网卡设备向这些内存中存取数据时用的是物理地址（因为网卡相对CPU属于头脑比较简单型的）。 pci_alloc_consistent的代码在Linux/arch/i386/kernel/pci-dma.c中。
③发送和接收缓冲区初始化和网卡开始工作的操作
RTL8139有4个发送描述符（包括4个发送缓冲区的基地址寄存器（TSAD0-TSAD3）和4个发送状态寄存器(TSD0-TSD3)。也就是说我们分配的缓冲区要分成四个等分并把这四个空间的地址都写到相关寄存器里去，下面这段代码完成了这个操作。
```
for (i = 0; i < NUM_TX_DESC; i++)
((struct rtl8139_private*)dev->priv)->tx_buf[i] =
&((struct rtl8139_private*)dev->priv)->tx_bufs[i * TX_BUF_SIZE];
```
上面这段代码负责把发送缓冲区虚拟空间进行了分割。
```
for (i = 0; i < NUM_TX_DESC; i++)
{
writel(tp->tx_bufs_dma+(tp->tx_buf[i]tp->tx_bufs),ioaddr+TxAddr0+(i*4));
readl(ioaddr+TxAddr0+(i * 4));
}
```
上面这段代码负责把发送缓冲区物理空间进行了分割，并把它写到了相关寄存器中，这样在网卡开始工作后就能够迅速定位和找到这些内存并存取他们的数据。
```
writel(tp->rx_ring_dma,ioaddr+RxBuf);
```
上面这行代码是把接收缓冲区的物理地址写到了相关寄存器中，这样网卡接收到数据后就能准确的把数据从网卡中搬运到这些内存空间中，等待CPU来领走他们。
```
writeb((readb(ioaddr+ChipCmd) & ChipCmdClear) |
CmdRxEnb | CmdTxEnb,ioaddr+ChipCmd);
```
重新RESET设备后，我们要激活设备的发送和接收的功能，上面这行代码就是向相关寄存器中写入相应值，激活了设备的这些功能。
```
writel ((TX_DMA_BURST << TxDMAShift),ioaddr+TxConfig);
```
上面这行代码是向网卡的TxConfig （位移是0x44）寄存器中写入TX_DMA_BURST << TxDMAShift这个值，翻译过来就是6<<8，就是把第8到第10这三位置成110，查阅管法文档发现6就是110代表着一次DMA的数据量为1024字节。
另外在这个阶段设置了接收数据的模式，和开启中断等等，限于篇幅由读者自行研究。
下面进入数据收发阶段：
当一个网络应用程序要向网络发送数据时，它要利用Linux的网络协议栈来解决一系列问题，找到网卡设备的代表net_device，由这个结构来找到并控制这个网卡设备来完成数据包的发送，具体是调用net_device的hard_start_xmit成员函数，这是一个函数指针，在我们的驱动程序里它指向的是 rtl8139_start_xmit，正是由它来完成我们的发送工作的，下面我们就来剖析这个函数。它一共做了四件事。
①检查这个要发送的数据包的长度，如果它达不到以太网帧的长度，必须采取措施进行填充。
```
if( skb->len < ETH_ZLEN ){//if data_len < 60
if( (skb->data + ETH_ZLEN) <= skb->end ){
memset( skb->data + skb->len, 0x20, (ETH_ZLEN - skb->len) );
skb->len = (skb->len >= ETH_ZLEN) ? skb->len : ETH_ZLEN;}
else{
printk("%s:(skb->data+ETH_ZLEN) > skb->end/n",__FUNCTION__);
}
}
```

skb->data和skb->end就决定了这个包的内容，如果这个包本身总共的长度(skb->end- skb->data)都达不到要求，那么想填也没地方填，就出错返回了，否则的话就填上。
②把包的数据拷贝到我们已经建立好的发送缓存中。
memcpy (tp->tx_buf[entry], skb->data, skb->len);
其中skb->data就是数据包数据的地址，而tp->tx_buf[entry]就是我们的发送缓存地址，这样就完成了拷贝，忘记了这些内容的回头看看前面的介绍。
③光有了地址和数据还不行，我们要让网卡知道这个包的长度，才能保证数据不多不少精确的从缓存中截取出来搬运到网卡中去，这是靠写发送状态寄存器（TSD）来完成的。
writel(tp->tx_flag | (skb->len >= ETH_ZLEN ? skb->len : ETH_ZLEN),ioaddr+TxStatus0+(entry * 4));
我们把这个包的长度和一些控制信息一起写进了状态寄存器，使网卡的工作有了依据。
④判断发送缓存是否已经满了，如果满了在发就覆盖数据了，要停发。
if ((tp->cur_tx - NUM_TX_DESC) == tp->dirty_tx)
netif_stop_queue (dev);
谈完了发送，我们开始谈接收，当有数据从网线上过来时，网卡产生一个中断，调用的中断服务程序是rtl8139_interrupt，它主要做了三件事。
①从网卡的中断状态寄存器中读出状态值进行分析，status = readw(ioaddr+IntrStatus);
if ((status &(PCIErr | PCSTimeout | RxUnderrun | RxOverflow |
RxFIFOOver | TxErr | TxOK | RxErr | RxOK)) == 0)
goto out;
上面代码说明如果上面这9种情况均没有的表示没什么好处理的了，退出。
② if (status & (RxOK | RxUnderrun | RxOverflow | RxFIFOOver))/* Rx interrupt */
rtl8139_rx_interrupt (dev, tp, ioaddr);
如果是以上4种情况，属于接收信号，调用rtl8139_rx_interrupt进行接收处理。
③ if (status & (TxOK | TxErr)) {
spin_lock (&tp->lock);
rtl8139_tx_interrupt (dev, tp, ioaddr);
spin_unlock (&tp->lock);
}
如果是传输完成的信号，就调用rtl8139_tx_interrupt进行发送善后处理。
下面我们先来看看接收中断处理函数rtl8139_rx_interrupt，在这个函数中主要做了下面四件事
①这个函数是一个大循环，循环条件是只要接收缓存不为空就还可以继续读取数据，循环不会停止，读空了之后就跳出。
int ring_offset = cur_rx % RX_BUF_LEN;
rx_status = le32_to_cpu (*(u32 *) (rx_ring + ring_offset));
rx_size = rx_status >> 16;
上面三行代码是计算出要接收的包的长度。
②根据这个长度来分配包的数据结构
skb = dev_alloc_skb (pkt_size + 2);
③如果分配成功就把数据从接收缓存中拷贝到这个包中
eth_copy_and_sum (skb, &rx_ring[ring_offset + 4], pkt_size, 0);
这个函数在include/linux/etherdevice.h中，实质还是调用了memcpy（）。
static inline void eth_copy_and_sum(struct sk_buff*dest, unsigned char *src, int len, int base)
{
memcpy(dest->data, src, len)；
}
现在我们已经熟知，&rx_ring[ring_offset + 4]就是接收缓存，也是源地址，而skb->data就是包的数据地址，也是目的地址，一目了然。
④把这个包送到Linux协议栈去进行下一步处理
skb->protocol = eth_type_trans (skb, dev);
netif_rx (skb);
在netif_rx（）函数执行完后，这个包的数据就脱离了网卡驱动范畴，而进入了Linux网络协议栈里面，把这些数据包的以太网帧头，IP头，TCP 头都脱下来，最后把数据送给了应用程序，不过协议栈不再本文讨论范围内。netif_rx函数在net/core/dev.c,中。
而rtl8139_remove_one则基本是rtl8139_init_one的逆过程。
到此，本文已经将Linux驱动程序的框架勾勒了出来