学习内核---Linux网卡驱动分析 PDF转换可能丢失图片或格式，建议阅读原文
https://www.100test.com/kao_ti2020/267/2021_2022__E5_AD_A6_E4_B9_A0_E5_86_85_E6_c103_267174.htm 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 (&amp.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的内存并进行了初步的初始化。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). 