第十四章 网络驱动程序

我们已经讨论了字符设备和块设备驱动程序，接着要讨论的是迷人的网络世界。网络接口是Linux 设备中的第三标准类，这一章就是讲述它们是如何与核心的其余部分交互的。
网络接口并不象字符和块设备那样存在于文件系统。相反，它在核心层处理包的发送和接收，并不与进程中的某个打开的文件绑定在一起。
网络接口在文件系统中的角色就象被安装的块设备。一个块设备在 blk_dev 数组和其它核心结构中注册它的特征，接着按照要求通过它的 *request_fn* 函数“发送”和“接收”块。类似地，一个网络接口必须在特定的数据结构中注册自己，从而在与外部世界交换包时可以被调用。
安装的磁盘与包发送接口有几个重要的不同。首先，磁盘以一个结点的形式存在于*/dev* 目录，而网络接口并不在文件系统中出现。不过两者之间最大的不同在于：磁盘是被请求向核心发送一个缓冲区，而网络接口则是请求向核心推送进来的包。
Linux 的网络子系统被设计成完全协议无关的。这对网络协议（IP vs. IPX 或其它协议）和硬件协议（以太网 vs.令牌环等）都是如此。网络驱动程序和核心之间的交互一次处理一个网络包；这允许协议可以干净地对驱动程序隐`藏起来，而物理传输则可以对协议隐藏起来。
本章描述网络接口如何与核心的其它部分紧密合作，并给出一个基于内存的模块化的网络接口，称之为（你可能已经猜到了）*snull*。为简化讨论，这个接口使用以太网硬件协议并传送 IP 包。通过 *snull* 获得的知识可以很好地应用于 IP 以外的协议，从以太网移到其它硬件协议只要求你对使用的物理协议有所了解。
snull 的另一个限制是它不能在 Linux1.2 中编译。再说一遍，这样做只是为了保持代码简单，并避免在 *snull* 中加入一些另人厌倦的条件。不过，本章将会提到与网络驱动程序相关的可移植性问题。
本章并不介绍 IP 的编号原则，网络协议，以及其它普通的网络概念。这个主题与驱动程序作者无关，而且以不到几百页的篇幅想对网络技术有一个令人满意的概述是不可能的。感兴趣的读者可以参考一些讲述网络问题的书。
在讨论网络设备之前，我想提醒你网络事务中的原子数据项被称做一个八元组（octet），由八个数据位组成。在本章中我都这样使用。网络文档从不使用术语“字节”。

Snull 如何设计

本节讨论与 *snull* 网络接口有关的一些设计概念。尽管这些信息可能显得用处并不大，但如果不理解它则可能在研究示例代码时遇到一些困难。
第一位的设计决定（也是最重要的）是示例接口不应绑定于任何实际硬件。实际接口不依赖于所传送的协议，*snull* 的这个限制并不影响本章给出的示例代码，因为它是协议无关的。IP 限制的唯一影响是地址分配----我们将位示例接口分配 IP 地址。

分配 IP 号码

snull 模块生成两个接口。这样的接口与简单的环回（loopback）并不一样，这里你从一个接口传送的包总是环回到另一个接口，而不是它自己。它看起来好象是你有两个外部链接，但实际上你的计算机只是应答自己。

不幸的是，这个效果并不能仅仅通过 IP 号码分配来达到，因为核心不会从接口 A 发送一个指向它自己接口 B 的包。相反，这时它会使用环回通道，从而根本不通过 *snull*。为了能建立一个通过 *snull* 接口的通信，源和目的地址必须在数据传送的时候修改一下。换句话说，从一个接口发出的包应能被另一个接口接收，但外出包的接收者不能被认为是本机。这也适用于收到包的源地址。

为了收到这种“隐藏环回”的效果，*snull* 接口反转一下源和目的地址的第三个八元组的最低位。其效果就是发向网络 A（连在接口 sn0 上）的包在 sn1 接口上好象是属于网络 B。

为了避免和太多的号码打交道，我们给用到的 IP 号码分配一些符号名：

- snullnet0 是一个连接在 sn0 接口上的一个 C 类网络。类似地，snullnet1 是连在 sn1 上的网络。这两个网络的地址仅在第三个八元组的最低位不同。
- local0 是分配给接口 sn0 的 IP 地址；它属于 snullnet0。与 sn1 相关联的地址是 local1。local0 和 local1 的第三和第四个八元组必须都不相同。
- remote0 是 snullnet0 中的一个主机，它的第四个八元组与 local1 相同。所有发向 remote0 的包在其 C 类地址被接口代码修改后将到达 local1。主机 remote1 属于 snullnet1，并且它的第四个八元组与 local0 相同。

snull 接口的操作见图 14-1，图中与接口相关联的主机名印在接口名旁边。

下面是几个可能的网络号码。一旦你把这几行写到*/etc/networks*，你就可以用名字来称呼这些网络。这些值是从保留私用的号码范围中选取的。

```
snullnet0     192.168.0.0
snullnet1     192.168.1.0
```

下面是写入*/etc/hosts* 的可能的主机号码：

```
192.168.0.88        local0
192.168.0.99        remote0
192.168.1.99        local1
192.168.1.88        remote1
```

(图 14-1 Page304)

不过如果你的计算机已经连到了一个网络上，那么一定要注意。你选择的号码有可能是实际的 Internet 或 intranet 的号码，把它们分配给你的接口可能会妨碍与真正主机的通信。而且，尽管我给出的这些号码不是实际的 Internet 号，但也有可能被你的私用网所适用，如果它处于防火墙之后的话。

不论你选择什么号码，你可以通过发出下面的命令来正确地设置接口：

（代码 304 #1）

到此为止，接口的“远”端已经可以到达了。下面的屏幕快照显示了我的主机是如何通过 *snull* 到达 remote0 和 remote1 的。

（代码 304 #2）

注意你不可能达到属于这两个网络的其它主机，因为在包的地址被改变并被接收到后，你的计算机会把它丢弃。

包的物理传送

至于数据传送，*snull* 属于以太网一类。示例代码使用了核心的以太网支持。这使我们不必去实现网络设备一些令人厌倦的细节。

我选择以太网是因为现存网络的主体----至少与工作站相连的这一部分-----都是基于以太网技术，不论是 10base2，10baseT，还是 100baseT。另外，核心还提供了对以太网设备的一般化

的支持，因此没有理由拒绝使用。以太网设备的优势如此明显，连 *plip* 接口（一类使用打印机端口的接口）都自称是以太网设备。
在 *snull* 中使用以太网设置的最后一个优势是你可以在接口上运行 *tcpdump*。不过，如果你想这样做，你需要把接口称做 eth*x*，而不是 sn*x*。*snull* 模块已经准备好将自己声明为 eth*x*。如果在 *insmode* 命令行中指定 eth=1，你就选择了这种行为。如果你忘了为 *snull* 请求 eth 命名，*tcpdump* 会拒绝倾倒这个接口，而是返回一个“未知的物理层类型”错。
snull 接口的另一个设计决定是只处理 IP 协议，本章的讨论也仅限于 IP。不过要注意，接口驱动程序本身并不依赖于它所处理的底层协议；网络驱动程序根本不查看它所传送的包。关于多协议传送将在后面的“非以太网包头”中详细介绍。
不过说实话，*snull* 还是查看包内容的，甚至还要修改它们，因为这是为保证代码工作要求的。代码修改每个 IP 包头的源，目的，以及校验和，但不检查它是否真地携带了 IP 信息。这种快而脏的数据修改会破坏非 IP 包。如果你想让 *snull* 处理其它协议，你必须修改这个模块的源码。不过，这种需求不太可能增长，因为每个拥有 Linux 盒的人都运行 IP，而其它协议则是可选的。

与核心相连

我们将通过拆解 *snull* 源码来看看网络驱动程序的结构。保证有几个驱动程序的源码在手边会很有助于你理解我们的讨论。我个人推荐 *loopback.c*，*plip.c*，以及 *3c509.c*，以逐渐增加的复杂性排序。有 *skeleton.c* 在手边也很有帮助，尽管这个示例驱动程序并不能真正运行。所有这些文件都居于核心源码树的 *drivers/net* 下。

模块加载

当一个模块被加载到运行的核心时，它要请求一些资源，并提供一些方便的功能；这已不再新鲜。另外请求资源的方式也不新鲜。驱动程序要探测它的设备及硬件位置（I/O 端口和 IRQ 线）----但并不注册它们----就象在第九章中断处理中“安装一个中断处理程序”一节中介绍的一样。网络驱动程序通过它的函数 *init_module* 进行注册的方法与字符或块设备驱动程序不一样。与请求一个主设备号不同，驱动程序为每个新检测到的接口在一个网络设备的全局列表中插入一个数据结构。
每个接口用一个 device 结构描述。sn0 和 sn1 这两个 *snull* 接口的结构如下所示：
（代码 306）
注意第一个域，既名字域指向一个静态缓冲区，它在加载时将被填充。通过这个方法，可以晚点儿选择接口名，下面会给出解释。如果你想在这个结构中使用一个显式缓冲区，如“01234567”，我要警告你那样可能导致代码不能可靠地工作。这是因为编译器会将两个重复的串折叠；因此你得到的会是一个缓冲区和两个指向它的缓冲区。而且，编译器有可能将常量串存在只读内存中，这显然不是你想要的。
在下节之前我不想完整描述结构 device，因为它是一个庞大的结构，太早地肢解它没有什么好处。我想在驱动程序中使用这个结构，并在每个域被使用时再解释它。
前面的代码显式地使用了 device 结构中的 name 和 init 域。name 是第一个域，含有接口名（识别接口的字符串）。驱动程序可以将接口名硬写在程序中，也允许动态赋值，其工作方式如下：如果名字的第一个字符是个空或者空格，那么设备注册项就使用第一个可用的 eth*n* 名。这样第一个以太网接口就被称做 eth0，其它的按序号类推。*snull* 接口则被缺省地称为 sn0 和 sn1。不过，如果在加载时指定 eth=1，那么 *init_module* 将使用动态赋值。缺省名由 *init_module* 给出：

（代码 307 #1）
init 域是个函数指针。任何时候当你注册一个设备时，核心要求驱动程序初始化自己。初始化就是指探测物理接口，用正确的数值填充 device 结构，下一节将给予描述。如果初始化失败，这个结构就不能被链入网络设备的全局列表。这种特别的设置的方法在系统引导时特别有效；每个驱动程序都试图注册它自己的设备，但只有确实存在的设备才被链入列表。这与字符和块设备驱动程序不同，它们被组织成一个两级树，由主设备号和次设备号索引。
由于真正的初始化在别的地方完成，*init_module* 要做的工作非常少，只需一句如下：
（代码 307 #2）

初始化每个设备

设备的探测在接口的 init 函数里完成，它通常被称做“探测”函数。init 收到的唯一的参数是一个指向正被初始化的设备的指针，其返回值是 0 或者一个负的错误代码----通常是-ENODEV。
对 *snull* 接口并没有进行实际的探测，因为它未绑定到任何硬件上。当你为一个实际的接口写实际的驱动程序时，探测字符设备的原则仍然适用：在使用 I/O 端口之前先检测它们，在检测期间不要向它写。另外，你还要避免在此时注册 I/O 端口和中断线。真正的注册应该推迟到设备打开时；这个非常重要，特别是当中断线被其它设备共享时。每次当别的设备触发中断线时，你的接口当然不希望被调用，而应简单地回答：不，它不是我的。
实际上，在加载时进行设备探测对 ISA 设备并不鼓励，因为这有可能很危险----ISA 体系结构在容错方面名声不佳。由于这个原因，大多数网络驱动程序在以模块的方式加载时拒绝为其硬件探测，核心也只探测第一个网络接口，在一个网络设备检测出来后不再进行任何硬件测试。通常 dev->base_addr----当前设备的 I/O 基地址----决定了要做什么：

- 如果 dev->base_addr 是一个有效的设备 I/O 地址，将不再探测其它 I/O 位置，而是使用这个值。如果这个值在加载时被赋值，这种情况就会发生。
- 如果 dev->base_addr 是 0，那么探测设备是可以接受的。拥护可以通过在加载时置这个 I/O 地址为 0 来请求探测。
- 其它情况下，不进行探测。核心使用 0xffe0 来阻止探测，但其实任何无效值都行。这需要依赖于驱动程序来无声地拒绝 base_addr 中的一个无效地址。一个模块应该缺省地置这个地址为无效值来防止不期望的探测。注意查看 PCI 设备总是安全的，因为它并不牵扯任何探测（见第 15 章，外围总线概述）

正如你可能已经注意到的，用一个加载时的设置来控制探测与我们在 *skull* 中使用的技术是一样的。
当从 dev->init 中退出时，dev 结构应该用正确的值填充。初始化例程的主要工作就是填充这个结构。幸运的是，核心通过函数 *ether_setup* 填充结构 device 负责了一些以太网的缺省设置。
snull_init 的核心是：
（代码 308）

```
        /* keep the default flags, just add NOARP */
        dev->flags           |=IFF_NOARP;
```

这段代码唯一不寻常的特征是在标志中设置 IFF_NOARP。这指明接口不能使用 ARP，即“地址解析协议”。ARP 是一个低级的以太网协议；每个真实的以太网接口都懂得 ARP，因此不需要设置这个标志。有趣的是注意到一个接口在没有 ARP 时仍能工作。例如，*plip* 接口就是没有 ARP 支持的以太网类接口，与 *snull* 相似。这个主题将在后面的“地址解析”中详细讨论，device 结构将在下一节肢解。

现在我想介绍结构 device 的另一个域 priv。它的作用类似与我们在字符设备驱动程序中用过的 private_data 指针。与 fops->private_data 不同的是，priv 指针是在初始化时分配，而不是在打开时，因为 priv 所指向的数据项包含有接口的统计信息。有一点很重要，就是统计信息要保证总是可用的，即使在接口宕掉时，因为用户可能在任何时候通过调用 *ifconfig* 来显式统计信息。在初始化时而不是打开时分配 priv 浪费的内存是无关紧要的，因为多数被探测到的接口保持一直在系统中运行。*snull* 模块为 priv 声明了一个数据结构 snull_priv。这个结构包含了结构 enet_statistics，它是存放接口信息的标准地方。
下面这几条 *snull_init* 中的语句分配 dev->priv：
（代码 309 #1）

模块卸载

当模块被卸载时没有什么特殊的事情发生。函数 *cleanup_module* 在释放了与私有结构相关的内存后，只需将接口从列表中取消即可。
（代码 309 #2）

模块化的和非模块化的驱动程序

尽管在对字符设备和块设备来说，模块化和非模块化的驱动程序并没有什么引人注意的区别，但对网络驱动程序来说，情况并非如此。
如果一个驱动程序做为主流 Linux 核心的一部分发行的话，它并不声明自己的 device 结构，而是使用在 *drivers/net/Space.c* 中声明的结构。*Space.c* 声明了所有网络设备的链表，即包括 plip1 一类驱动程序特定的结构，也包括通用目的的 eth 设备。以太网根本不关心它们的 device 结构，因为它们使用通用目的的结构。这种通用的 eth 设备结构声明 *ethif_probe* 为它们的 init 函数。程序员要想在主流核心中插入一个新的以太网接口只需要在 *ethif_probe* 中加入一个对驱动程序初始化函数的调用。另一方面，非 eth 驱动程序的作者需要在 *Space.c* 中插入它们的 device 结构。在两种情况下，如果驱动程序必须被链到核心，只需要修改源文件 *Space.c*。
在系统引导时，网络初始化代码循环遍历所有的 device 结构，调用它们的探测函数（dev->init），向它们传递一个指向设备本身的指针。如果探测函数成功了，*Space.c* 初始化 device 结构。这种设置驱动程序的方式允许渐增地将设备赋予名字 eth0，eth1，依次类推，而不需要改变每个设备的 name 域。
另一方面，当加载一个模块化的驱动程序时，它声明它自己的 device 结构（入我们在本章中已经看到的那样），即使它控制的接口是以太网接口。
好奇的读者可以查看 *Space.c* 和 *net_init.c* 来得到更多关于接口初始化的信息。这里对驱动程序设置的介绍只是为了强调 init 设备方法的重要性。如果一个驱动程序模块包含了预填好的设备结构，那么它将不适合主流核心的初始化技术，并且如果结构 device 中引入新的域，会使它变的不能向前兼容。

设备结构的细节

device 结构居于网络驱动程序的真正核心，值得完全的描述。第一次阅读本书的读者可以跳过本节，因为开始时不需要对这个结构有详细的理解。下面的列表描述所有的域，但主要目的是提供一个参考而不是要被记住。本章的其余部分在一个域被示例代码用到的时候会简单地描述一下，所以你不必不停地回头来参考本节。

结构 device 在结构上可以分为两个部分：“可见的”和“不可见的”。可见部分由那些在静态 device 结构中显式赋值的域组成，象前面给出的在 *snull* 中出现的两项。其余的域内部使用。有些被驱动程序访问（例如在初始化时被赋值的那些），而有些不能动。本章在版本 2.0.30 前都是完全的。

可见的头

结构 device 的第一部分由下列域组成，按序为：

char *name;

设备名。如果第一个字符是 0（NULL 字符）或空格，*register_netdev* 给它分配名字 eth*n*，n 取合适的值。

unsigned long rmem_end;

unsigned long rmem_start;

unsigned long mem_end;

unsigned long mem_start;

这些域存有设备使用的共享内存的开始和结束地址。如果设备有不同的发送和接收内存，那么 mem 域就用做发送内存，而 rmem 用做接收内存。mem_end 和 mem_start 可以在系统引导时在核心命令行指定，它们的值由 *ifconfig* 获取。Rmem 域在驱动程序以外不会被引用。一般地，end 域被设置成使得 end-start 为板上可用内存量。

unsigned long base_addr;

I/O 基地址。这个域，和前面的一样，在设备检测时被赋值。*ifconfig* 可以用来显示和修改当前值。base_addr 可以在系统引导或加载时在核心命令行显式赋值。

unsigned char irq;

被赋予的中断号。当接口被列出时 dev->irq 由 *ifconfig* 打印出来。这个值通常在引导或加载时被设置，以后可以用 *ifconfig* 修改。

unsigned char start;

unsigned char interrupt;

这些域是二进制标志。start 通常在设备打开时设置，在关闭时清楚。在接口准备号运行时它是非零。interrupt 是用来告诉代码的高层一个中断到达接口，并正在处理中。

unsigned long tbusy;

这个域表明“传送忙”。当驱动程序不能再接收新的包发送时（既所有的输出缓冲区都满了），它应该为非零。使用 long 类型而不是 char 是因为有时要使用原子的位操作以避免竞争条件。注意在核心 1.2，tbusy 的确是个八位的域，向后可移植的驱动程序应该注意这一点。原子的位操作在第九章的“使用锁变量”一节中介绍过。

struct device *next;

用来维护链表；任何驱动程序都不能动这个域。

int (*init)(struct device *dev);

初始化函数。这个域通常是 device 结构中显式列出的最后一个域。

隐藏的域

device 结构包含几个额外的域，通常在设备初始化时被赋值。这些域中的一些携带了接口的信息，一些存在只是为了方便驱动程序（也就是说，核心并不使用它们）；还有一些域，最引人注意的是一些设备方法，它们是核心和驱动程序的接口。

我想分别列为三组，与域的实际顺序无关，那并不重要。

接口信息

多数接口信息都由函数 *ether_setup* 来正确设置。以太网卡在大部分域都可以依赖这个通用目的的函数，但 flags 和 dev_addr 域是设备特定的，必须在初始化时显式地赋值。
一些非以太网的接口可以使用类似于 *ether_setup* 的助手函数。*driver/net/net_init.c* 引出 *tr_setup*(令牌环)和 *fddi_setup*。如果你的设备不属于这些类中的一种，你需要自己为所有的域赋值。

unsigned short hard_header_len;

“硬件包头长”。发送包头中 IP 头（或其它协议信息）之前那部分的八元组个数。对以太网接口来说，这个值是 14。

unsigned short mtu;

“最大传送单元”。在包传输时，这个域由网络层使用。以太网的 MTU 为 1500 个八元组。

__u32 tx_queue_len;

在设备传送队列中可以排队的最大祯数。*ether_setup* 将这个值设为 100，不过你可以改变它。例如，*plip* 使用 10 以避免浪费系统内存（*plip* 比实际的以太网接口吞吐率要低）。

unsigned short type;

接口的硬件类型。这个域被 ARP 使用以判断接口支持的硬件地址类型。以太网接口把它设为 ARPHRD_ETHER----*ether_setup* 为你做这件事。

unsigned char addr_len;

unsigned char broadcast[MAX_ADDR_LEN];

unsigned char dev_addr[MAX_ADDR_LEN];

以太网地址长为六个八元组（我们是指接口板的硬件标志），播送地址由六个 0xff 八元组组成；*ether_setup* 负责这些值的正确设置。另一方面，设备地址必须以设备特定的方式从接口板中读出，驱动程序应把它复制到 dev_addr。这个硬件地址用来在把包交给驱动程序传送前产生正确的以太网包头。*snull* 并不使用物理接口，它生成一个它自己的物理地址。

unsigned short family;

接口的地址族，通常为 AF_INET。接口并不常查看这个域或者向其赋值。

unsigned short pa_alen;

协议地址长。对 AF_INET 来说为四个八元组。接口不需要修改这个数。

unsigned long pa_addr;

unsigned long pa_brdaddr;

unsigned long pa_mask;

刻划接口的三个地址：接口地址，播送地址，及网络掩码。这些值是协议特定的（既它们是“协议地址”）；如果 dev->family 是 INET，则它们为 IP 地址。这些域由 *ifconfig* 赋值，对驱动程序是只读的。

unsigned long pa_dstaddr;

plip 和 ppp 一类点到点协议使用这个域记录连接另一侧的 IP 号码。和前面的域一样，它也是只读的。

unsigned short flags;

接口标志。这个域含有下列位值。前缀 IFF 意为接口标志（InterFace Flags）。有些标志由核心管理，有些则是在初始化时由接口设置，以确认接口的能力。有效的标志是：

IFF_UP

当接口是活跃的时，核心置上该标志。这个标志对驱动程序是只读的。

IFF_BROADCAST

这个标志表明接口的播送地址是有效的。以太网卡支持播送。

IFF_DEBUG

查错模式。这标志控制 *printk* 调用的唠叨，还用在其它一些查错目的。尽管目前没有官方驱动程序使用它，用户程序可以通过 *ioctl* 来对其置位或者清除，你的驱动程序可以使用它。*misc-progs/netifdebug* 程序可以用来将这个标志打开或关闭。

IFF_LOOPBACK

这个标志在环回接口中要被置位。核心检测这个标志而不是将名字 lo 作为特殊接口硬写入程序。

IFF_POINTOPOINT

点到点的初始化函数应置位这个标志。例如，*plip* 对它置位。*ifconfig* 工具也可以对其置位和清除。当它被置位时，dev->pa_dstaddr 应该指向连接的另一端。

IFF_NOARP

常规网络接口可以传送 ARP 包。如果接口不能进行 ARP，它必须置这个标志。例如，点到点接口并不需要运行 ARP，它只能增加额外的通信，却不能获取任何有用的信息。*snull* 不具有 ARP 能力，因此它要对其置位。

IFF_PROMISC

这个标志被置位以获得杂类操作。在缺省情况下，以太网接口使用硬件过滤器以保证它只收到播送包和指向其硬件地址的包。而象 *tcpdump* 一类包监视器则在接口上设置杂类模式，以获取经过接口传输介质的所有包。

IFF_MULTICAST

能进行选播传送的接口要置这个标志。*ether_setup* 在缺省情况下对其置位。所以如果你的驱动程序不支持选播，它必须在初始化时清除这个标志。

IFF_ALLMULTI

这个标志告诉接口接收所有的选播包。只有当 IFF_MULTICAST 被置位，而主机由进行选播路由时，核心对其置位。它对接口时只读的。IFF_MULTICAST 和 IFF_ALLMULTI 早在 1.2 版就已经定义了，但那时并未使用。在后面“选播”一节我们将看到它是如何使用的。

IFF_MASTER

IFF_SLAVE

这些标志被加载均衡代码使用。接口驱动程序不需要知道它们。

IFF_NOTRAILERS

IFF_RUNNING

这些标志在 Linux 中不使用，只是为了和 BSD 兼容而存在。

当一个程序改变 IFF_UP，*open* 和 *close* 方法会被调用。当 IFF_UP 或其它标志被修改时，*set_multicast_list* 方法被调用。如果驱动程序因为标志的修改而要执行一些动作，那么必须在 *set_multicast_list* 中进行。例如，当 IFF_PROMIS 被置位或清除时，板上硬件过滤器必须被通知。这个设备方法的责任将在后面的“选播”一节简单介绍。

设备方法

与字符设备和块设备的情况一样，每个网络设备要声明在其上操作的函数。可以在网络接口上进行的操作列在下面。一些操作可以留为 NULL，还有一些通常不去动它们，因为 *ether_setup* 给它们分配合适的方法。

一个网络接口的设备方法可以分为两类：基本的和可选的。基本的包括那些为访问接口所需要的；可选的方法实现一些并不严格要求的高级功能。下面是基本方法：

```
int (*open)(struct device *dev);
```

打开接口。只要 *ifconfig* 激活一个接口，它就被打开了。*open* 方法要注册它需要的所有资源（I/O 端口，IRQ，DMA，等），打开硬件，增加模块的使用计数。

int (*stop)(struct device *dev);

终止接口。接口在关闭时就终止了；在打开时进行的操作应被保留。

int (*hard_start_xmit)(struct sk_buff *skb, struct device *dev);

硬件开始传送。这个方法请求一个包的传送。这个包含在一个套接字缓冲区结构（sk_buff）中。套接字缓冲区在下面介绍。

int (*rebuild_header)(void *buf, struct device *dev, unsigned long raddr, struct sk_buffer *skb);

这个函数用来在一个包传送之前重构硬件包头。这个以太网设备使用的缺省包头用 ARP 向包中填入缺少的信息。*snull* 驱动程序实现了它自己的这个方法，因为 ARP 并不在 sn 接口上运行。（在本章的后面会介绍 ARP。）这个方法的参数是一些指针，分别指向硬件包头，设备，“路由器地址”（包的初始目的地），以及被传送的缓冲区。

int (*hard_header)(struct sk_buffer *skb, struct device *dev, unsigned short type,
void *daddr, void *saddr, unsigned len);

硬件包头。这个函数用以前获取的源和目的地址构造包头；它的任务是组织那些以参数的形式传给它的信息。*eth_header* 是以太网类接口的缺省函数， *ether_setup* 相应地对这个域赋值。给出的参数顺序适用于核心 2.0 或更高版本，但与 1.2 有所不同。这个改变对以太网驱动程序是透明的，因为它继承了 *eth_header* 的实现；其它驱动程序可能要处理一下这个不同，如果它们想保持向后兼容的话。

struct enet_statistics * (*get_stats)(struct device *dev);

当应用希望获得接口的统计信息时需要调用这个方法，例如，当运行 *ifconfig* 或 *netstat –i* 时。在 *snull* 中的一个示例实现将在后面“统计信息”中介绍。

int (*set_config)(struct device *dev, struct ifmap *map);

改变接口的配置。这个方法是配置驱动程序的入口点。设备的 I/O 地址和中断号可以在运行时用 *set_config* 改变。在接口不能探测到时，系统管理员可以适用这个能力。这个方法在后面的“运行时配置”中介绍。

其余的设备方法是被我称为可选的那些。传递给其中一些的参数在 Linux1.2 到 Linux2.0 的转变中改了好几次。如果你想写一个可以在两个版本核心都工作的驱动程序，你可以只为从 2.0 开始的版本实现这些操作。

int (*do_ioctl)(struct devices *dev, struct ifreg *ifr, int cmd);

执行接口特定的 *ioctl* 命令。这些命令的实现在后面的“自定义 *ioctl* 命令”中描述。这里给出的原形在 1.2 以上的核心都能工作。如果接口不需要任何接口特定的命令，那么结构 device 中相应的域可以留为 NULL。

void (*set_multicast_list)(struct device *dev);

当设备的选播列表改变和标志改变时，将调用这个方法。这里的参数传递与 1.2 版本不同。更多的细节和一个示例实现见“选播”一节。

int (*set_mac_address)(struct device *dev, void *addr);

如果接口支持改变硬件地址的能力，可实现这个函数。多数接口要么不支持这个能力，要么使用缺省的 *eth_mac_addr* 实现。这个原形与 1.2 版也不同。

#define HAVE_HEADER_CACHE

void (*header_cache_bind)(struct hh_cache **hhp, struct device *dev, unsigned short htype, __u32 daddr);

void (*header_cache_update)(struct hh_cache *hh, struct device *dev, unsigned char *haddr);

这些函数和宏在 Linux1.2 中没有。以太网驱动程序不必关心 header_cache 的问题，因为 *eth_setup* 会安排使用缺省的方法。

#define HAVE_CACHE_MTU

int (*change_mtu)(struct device *dev, int new_mtu);

如果接口的 MTU（最大传送单元）发生了改变，这个函数负责采取动作。这个函数和宏在 Linux1.2 中都没有。当 MTU 改变时，如果驱动程序要做一些特殊的事情，它应该声明它自己的函数，不然将由缺省函数来完成。如果你感兴趣，*snull* 有一个这个函数的模版。

工具域

其余的结构 device 中的域被接口用来保存一些有用的状态信息。其中一些被 *ifconfig* 和 *netstat* 用来向用户提供当前配置的信息。因此，接口应该对这些域赋值。

unsigned long trans_start;

unsigned long last_rx;

这两个域用来保存一些瞬间值。它们目前不用，但核心有可能将来使用这些计时提示。驱动程序负责在传送开始时和收到包时更新这些值。trans_start 域还可以被驱动程序用来检测锁定。驱动程序可以在等待一个“传送完成”的中断时用 trans_start 来检查超时。

void *priv

等价于 filp->private_data。驱动程序拥有这个指针，可以随意使用。通常这个私有数据结构含有一个 enet_statistics 结构项。这个域在以前的“初始化每个设备”中用过。

unsigned char if_prot;

这个域用来记录哪个硬件端口被接口使用（例如，BNC，AUI，TP）。任何数值都可以按需要赋给它。

unsigned char dma;

被接口使用的 DMA 通道。这个域被 *ioctl* 的 SIOCGIFMAP 命令使用。

struct dev_mc_list *mc_list;

int mc_count

这两个域被用来处理选播传送。Mc_count 是 mc_list 中项的个数。更多的细节见“选播”。

结构 device 中还有一些别的域，但驱动程序没有使用它们。

打开和关闭

我们的驱动程序可以在模块加载和核心引导时探测接口。下一步是给接口赋一个地址，这样驱动程序就可以通过它交换数据了。打开和关闭一个接口由 *ifconfig* 命令完成。

当使用 *ifconfig* 为一个接口赋地址时，它完成两项工作。第一，它通过 ioctl(SIOCSIFADDR)(即 Socket I/O Control Set InterFace ADDRess)来赋地址。接着它通过 ioctl(SIOCSIFFLAGS)(即 Socket I/O Control Set InterFace FLAGS) 对 dev->flag 中的 IFF_UP 置位来打开接口。

至于设备，ioctl(SIOCSIFADDR) 设置 dev->pa_addr，dev->family，dev->pa_mask，dev->pa_brdaddr，没有驱动程序函数被调用----这个任务是设备无关的，由核心来完成。不过，后一个命令 ioctl(SIOCSIFFLAGS)为设备调用 *open* 方法。

类似地，当一个接口关闭时，*ifconfig* 使用 ioctl(SIOCSIFFLAGS)来清除 IFF_UP，并且调用 *stop* 方法。
两个设备方法在成功时都返回 0，发生错误时，通常返回一个负值。
至于代码，驱动程序必须执行与字符和块设备同样的工作。*open* 请求它所需要的所有的系统资源，并告诉接口启动；*stop* 则关闭接口，并释放系统资源。
如果驱动程序不准备使用共享中断（例如，它不打算与旧的核心兼容），还有最后一步需要做。核心引出一个 irq2dev_map 阵列，它由 IRQ 号寻址，持有空指针；驱动程序也许想用这个数组将中断号映射到指向 device 结构的指针。这是在不使用接口处理程序的情况下，在一个驱动程序里支持一个以上接口的唯一方法。
另外，在接口可以和外界通信以前，硬件地址还必须从板上复制到 dev->dev_addr。硬件地址可以按驱动程序的意愿在探测时或打开时被赋值。*snull* 软件接口时从 *open* 里对其赋值；它用两个 ASCII 串伪造一个硬件号码。地址的第一个字节是个空字符（在后面的“地址解析”中解释）。
结果得到的 *open* 代码如下所示：
（代码 319）
（代码 320 #1）
正如你所看到的，device 结构中的几个域被修改了。start 表明接口已准备好， tbusy 断言发送者不忙（也就是说，核心可以发出一个包）。
stop 方法是 *open* 的操作的反转。由于这个原因，实现 *stop* 的函数通常调用 *close*。
（代码 320 #2）

包发送

网络接口执行的最重要的工作是数据发送和接收。我准备从发送开始，因为它相对比较简单。
当核心需要发送一个数据包时，它调用 *hard_start_transmit* 方法将数据放到一个输出队列。核心处理的每个包包含在一个套接字缓冲区结构（struct sk_buff）中，其定义见<linux/skbuff.h>。这个结构从 Unix 用来表示一个网络连接的抽象，即套接字得名。即使接口与套接字无关，每个网络包在较高的网络层中一定属于某个套接字，任何套接字的输入输出缓冲区都是 sk_buff 结构的列表。同样的 sk_buff 结构在整个 Linux 网络子系统中都被用来承载网络数据，但在考虑接口时，一个套接字缓冲区就是一个包。
指向 sk_buff 的指针通常被称做 skb，我将在示例和正文中都使用使用这个习惯。
套接字缓冲区是一个复杂的结构，核心提供一组函数来对其操作。这些函数在后面的“套接字缓冲区”中描述----目前，知道 sk_buff 的一些基本事实已足以写出可工作的驱动程序。另外，我习惯于在扎入另人讨厌的细节之前先弄明白是如何工作的。
传递给 *hard_start_xmit* 的套接字缓冲区含有物理包，它具有传输层的包头。接口不需要修改被发送的数据。skb->data 指向被发送的包，skb->len 是它的长度，以八元组为单位。
snull 的包发送代码如下所示；物理发送机制被隔离在另一个函数中，因为每个接口驱动程序必须按照被驱动的特定硬件来实现它。
（代码 321）
这样发送函数只进行一些清晰的对包的检查，并通过硬件相关的函数发送数据。在一个中断表明一个“发送结束”的条件时，dev->tbusy 被清除。

包接收

从网络中接收数据比发送要复杂一些，因为必须分配一个 sk_buff，并从一个中断处理程序中将其交递给高层----接收包的最好的办法是通过中断，除非接口是象 *snull* 一样是纯软件的，或是环回接口。尽管有可能写轮询的驱动程序，而且在正式的核心里也的确有几个，但中断驱动的要好的多，不管是在数据吞吐率还是计算需求上。由于绝大多数网络接口都是中断驱动的，我不打算谈论轮询实现，它只是利用了核心计时器。

snull 的实现是将硬件细节和设备无关的工作分离开的。这样，在硬件收到一个包后，*snull_rx* 被调用，它已经在计算机的内存中了。*snull_rx* 因此收到一个指向数据的指针和包的长度。。这个函数唯一的责任就是将包和一些额外信息发送到网络代码的高层。其代码与数据指针及长度获得的方法无关。

（代码 322）

这个函数足够通用，可以作为任何网络驱动程序模版，但在你有信心重用这个代码段之前还需要一些解释。

注意缓冲区分配函数需要知道数据长度。者避免了在调用 *kmalloc* 时浪费内存。*dev_alloc_skb* 以原子优先级调用分配函数，因此它也可以在中断时安全地使用。核心还提供了套接字缓冲区分配的其它一些接口，但不值得在这里介绍；套接字缓冲区在本章后面的“套接字缓冲区”中详细介绍。

一旦有了一个有效的 skb 指针，就可以通过调用 *memcpy* 将包数据复制到这个缓冲区。*skb_put* 更新缓冲区中数据尾的指针，并返回一个指向新生成空间的指针。

不幸的是，包头中没有足够的信息来正确处理网络层----在缓冲区向上层传递之前，dev 和 protocol 域必须被赋值。接着我们需要指定如何执行校验和（*snull* 不进行任何校验和）。skb->ip_summed 可能的策略为：

CHECKSUM_HW

板子用硬件执行校验和。一个硬件校验和的离子是 Sparc HME 接口。

CHECKSUM_NONE

校验和完全有软件完成。对新分配的缓冲区，这是缺省的策略。

CHECKSUM_UNNECESSARY

不做任何校验和。这是 *snull* 和环回接口的策略。

在 1.2 核心版本中没有校验和选项和 ip_summed。

最后，驱动程序更新它的统计计数器记录一个新包被收到了。统计结构有几个域组成，最重要的是 rx_packets 和 tx_packets，它们包含收到的和发送的包的个数。所有的域在后面的“统计信息”中给出一个彻底的描述。

包接收的最后一步由 *netif_rx* 完成，它将套接字缓冲区递交到上一层。

中断驱动的操作

大多数硬件接口以中断处理程序的方式控制。接口中断处理器表明两种事件中的一种：一个新包到达了或一个包发送完成了。这种一般化并不是总适用，但它基本上揭示了与异步包传送相关的问题。PLIP 和 PPP 是不适用这种一般化的例子。它们处理同样的事件，但低级中断处理略有不同。

一般的中断例程可以通过检查在硬件设备上的一个状态寄存器来分辨新包到达中断与完成发送的通知。*snull* 接口工作方式类似，但其状态字在 dev->priv 中。网络接口的中断处理程序看起来如下：

（代码 324）

处理程序的第一个任务是接收一个指向正确的 device 结构的指针。你可以用 irq2dev_map[]（假如你在打开是给它赋了一个值）或者接收到的 dev_id 指针作为一个参数。如果你希望驱动程序可以与新于 1.3.70 的核心工作，你必须使用 irq2dev_map[]，因为早期版本中没有 dev_id。
这个处理程序中有趣的部分是处理“发送完成”的部分。接口通过清除 dev->tbusy 并标志网络下半部例程来相应发送完成。如果 *net_bh* 的确运行了，它会试图发送所有等待的包。
另一方面，包接收并不需要任何特殊的中断处理。所有需要做的就是调用 *snull_rx*。
实际上，当 *netif_rx* 被接收函数调用时，它所进行的实际操作只有标志 *net_bh*。换句话说，核心在一个下半部处理程序中完成了所有网络相关的工作。因此，网络驱动程序应该总是宣称它的中断处理程序太慢，因为下半部将会更早地执行（见第九章中“下半部设计”）。

套接字缓冲区

我们已经讨论了于网络接口相关的多数内容。下面几节我们将更细地讨论 sk_buff 时如何设计的。这几节既介绍这个结构的主要域，也介绍在套接字缓冲区上操作的函数。
尽管并没有理解 sk_buff 内部的严格需要，但是如果能理解它的内容将会有助于你解决问题和优化代码。例如，如果你看了 *loopback.c*，你会发现一个基于 sk_buff 内部知识的优化。
我不打算在这里描述整个结构，而只是那些可能被驱动程序用到的域。如果你想知道更多，你可以看<linux/skbuff.h>，结构的定义和函数的原形都在那里定义。至于这些域和函数如何使用的细节可以通过浏览核心源码得到。

重要的域

出于我们的目的，结构里重要的域是那些驱动程序的作者可能要用到的域。它们如下所示，无特别顺序。

struct device *dev;
: 设备接收或者发送这个缓冲区。

__u32 saddr;
__u32 daddr;
__u32 raddr;
: 源地址，目的地址，和路由器地址，由 IP 协议使用。raddr 是包要到达其目的地的第一步。这些域在包被发送前被设置，收到之后就不必赋值了。到达 *hard_start_xmit* 方法的外出包已经有了一个合适的硬件包头设置反映了“第一步”信息。

unsigned char *head;
unsigned char *data;
unsigned char *tail;
unsigned char *end;
: 这些指针用来访问包中的数据。head 指向分配空间的开始，data 是有效八元组的开始（通常比 head 略大），tail 是有效八元组的结束， end 指向 tail 可以到达的最大地址。观察它们的另一个方法是：可用缓冲区空间为 skb->end-skb->head，当前使用的数据空间为 skb->tail-skb->data。这种处理内存区域的清晰方法在 1.3 开发时才实现。这是 *snull* 没有被移植在 Linux1.2 上编译的主要原因。

unsigned long len;
: 数据本身的长度（skb->tail-skb->head）。

unsigned char ip_summed;

这个域有驱动程序对进来包设置，由 TCP/UDP 校验和使用。它在前面的“包接收”中介绍过。

unsigned char pkt_type;

这个域被内部用来发送进来包。驱动程序负责将其设置为 PACKET_HOST（这个包是我的）， PACKET_BROADCAST，PACKET_MULTICAST，或是 PACKET_OTHERHOST（不，这个包不是我的） 。以太网驱动程序并不显式地修改 pkt_type，因为 eth_type_trans 会为它做这件事。

union { unsigned char *raw; […]} mac;

与 pkt_type 类似，这个域被用来处理进来包，必须在包接收时设置。函数 eth_type_trans 为以太网驱动程序负责这件事。非以太网驱动程序应设置 skb->mac.raw 指针，后面“非以太网包头”中将会提到。

结构中其余的域并无特别兴趣。它们被用来维护缓冲区列表，解释占有缓冲区的套接字的内存，等等。

在套接字缓冲区上操作的函数

使用 sock_buff 的网络设备通过正式的接口函数在这个结构上操作。有很多在套接字缓冲区上操作的函数，下面是最有趣的一些：

struct sk_buff *alloc_skb(unsigned int len, int priority);

struct sk_buff *dev_alloc_skb(unsigned int len);

分配一个缓冲区。*alloc_skb* 分配一个缓冲区并初始化 skb->data 和 skb->tail 到 skb->head。 *dev_alloc_skb* 函数（在 Linux1.2 中没有）一个快捷方式，它用 GFP_ATOMIC 优先级调用 *alloc_skb*，并反转 skb->head 和 skb->data 之间的 16 个字节。这个数据空间可以用来“推”硬件包头。

void kfree_skb(struct sk_buff *skb, int rw);

void dev_kfree)skb(struct sk_buff *skb, int rw);

释放一个缓冲区。*kfree_skb* 被核心内部使用。驱动程序应该使用 *dev_kfree_skb*，在拥有缓冲区的套接字需要再次使用它的情况下，它可以正确地处理缓冲区加锁。两个函数的 rw 参数是 FREE_READ 或 FREE_WRITE。这个值用来跟踪套接字的内存。外出缓冲区应用 FREE_WRITE 来释放，而进来的则使用 FREE_READ。

unsigned char *skb_put(struct sk_buff *skb, int len);

这个线入函数更新结构 sk_buff 的 tail 和 len 域，它被用来在缓冲区尾加入数据。其返回值是 skb->tail 以前的值（或者说，它指向刚生成的数据空间）。有些驱动程序通过调用 ins(ioaddr,skb_put(…))或 memcpy(skb_put(…), data,len)来使用这个返回值。这个函数及下面的一些在为 Linux1.2 构造模块是不存在。

unsigned char *skb_push(struct sk_buff *skb, int len);

这个函数减小 skb->data，增加 skb->len。类似于 skb_put，除了数据是加在包开始而不是结尾。返回值指向刚生成的空间。

int skb_tailroom(struct sk_buff *skb);

这个函数返回为在缓冲区中放置数据的可用空间量。如果驱动程序在缓冲区中放了多于它能承载的数据，系统可能回崩溃。你也许会反对并认为，用 *printk* 指出这个错误已经足够了，而内存崩溃对系统太有害了，开发者肯定要采取一些措施。但实际上，如果缓冲区被正确分配了，你根本不必检查可用空间。因为驱动程序通常在分配缓冲区之前获

得包大小，只有有严重缺陷的驱动程序才可能在缓冲区内放太多的数据，崩溃可以认为是应得的惩罚。

int skb_headroom(struct sk_buff *skb);

返回数据前面得可用空间量，也就是可以向缓冲区中“推”多少八元组。

void skb_reserve(struct sk_buff *skb, int len);

这个函数增加 data 和 tail。它可以用来在填充缓冲区前预留空间。大多数以太网接口在包前预留两个字节；这样 IP 头可以在一个 4 字节以太网头之后，在 16 字节边界对齐。*snull* 完成得很好，尽管在“包接收”中并未提到这一点，那主要是为了避免彼时引入过多得概念。

unsigned char *skb)pull(struct sk_buff *skb, int len);

从包头中删除数据。驱动程序并不用这个函数，但为了完整性也包含在这里。它减少 skb->len，增加 skb->data；这是从进来包的开始剥出以太网包头的方法。

核心还定义了几个在套接字缓冲区上操作得别的函数，但它们主要应用于网络代码得高层，驱动程序并不需要它们。

地址解析

以太网通信最急迫得问题之一是硬件地址（接口得唯一标志符）与 IP 号码之间的关联。大多数协议都有类似问题，但我只向重点讨论一下以太网类得情况。我力图给出一个全面得描述，因此我将显示三种情况：ARP，没有 ARP 的以太网头(象 *plip*)，以及非以太网包头。

在以太网上使用 ARP

地址解析得一般方法是 ARP，即地址解析协议。幸运的是，ARP 由核心管理，以太网接口不必为支持 ARP 做任何特殊工作。只要在打开时正确地设置了 dev->addr 和 dev->addr_len，驱动程序不需担心任何从 IP 号码到物理地址的转换；*ether_setup* 将正确的设备方法赋给 dev->hard_header 和 dev->rebuild_header。

当一个包被构造时，以太网包头由 dev->hard_header 来布局，并由 dev->rebuild_header 在后来填充，它使用 ARP 协议将未知的 IP 号码映射到地址上。驱动程序作者不必知道这个过程的细节去写一个可工作得驱动程序。

越过 ARP

简单得点到点网络接口如 *plip* 可以从以太网包头获益，但却要避免来回发送 ARP 包得开销。*snull* 中得示例代码就属于这一类网络设备。*snull* 不能使用 ARP，因为驱动程序修改被发送得包得 IP 地址，而 ARP 包也交换 IP 地址。

如果你的设备想用一般的硬件包头，却不想运行 ARP，你需要越过缺省的 dev->rebuild_header 方法。这就是 *snull* 实现的方法，这个简单的函数有三条语句：

（代码 329）

事实上，并没有指定 eth->h_source 和 eth->h_dest 内容的实际需要，因为这些值只被用来进行包得物理传送，而一个点到点得连接保证能将包发送到它的目的地，而与硬件地址无关。*snull* 重构包头的原因是向你演示，当 *eth_rebuild_header* 不可用时，一个真实的网络接口的重构函数是如何实现的，

当接口收到一个包时，硬件包头只被 *eth_type_trans* 使用。我们在 *snull_rx* 中已经见过这个调用：

```
skb->protocol=eth_type_trans(skb,dev);
```

这个函数从以太网包头中抽取协议标志符（在这里是 ETH_P_IP）；它还要赋值 skb->mac.raw，从包数据中删去硬件包头，并设置 skb->pkt_type。最后一项在 skb 分配时缺省为 PACKET_HOST（表明包被指向这个主机），当然它也可以改为符合以太网目的地址得其它值。

如果你的接口是点到点连接，你将无法收到未想到的选播包。为避免这个，你必须记住那些第一个八元组的最低位（LSB）是 0 的目的地址将被指向单个主机（也就是说，它是 PACKET_HOST 或 PACKET_OTHERHOST）。*plip* 驱动程序用 0xfc 作为它的硬件地址的第一个八元组，而 *snull* 用 0x00。这两个地址都导致一个可工作的以太网类的点到点连接。

非以太网包头

这节简要地介绍硬件包头是如何用来封装相关信息的。如果你希望了解细节，你可以从核心源码或特别传输介质的技术文档中得到。我们刚才已经看到硬件包头除了含有目的地址外，还有一些信息，其中最重要的是通信协议。

不过，并不是每个协议都要提供所有的信息。象 *plip* 或 *snull* 之类的点到点连接可以避免传送整个以太网包头，同时不失去一般性。*hard_header* 设备方法从核心接收传送信息----包括协议级和硬件地址。它也收到 16 位的协议号。例如 IP 由 ETH_P_IP 标志。驱动程序应能正确地象接收主机传送包数据和协议号。点到点连接可以在硬件包头中省略地址，只传送协议号，因为传送是有保证的，与源和目的地址无关。一个只有 IP 的连接甚至什么硬件头都不传送。两种情况下，所有的工作都由 *hard_header* 完成，*rebuild_header* 除了返回 0 外什么都不做。

当包在连接的另一端被捡起时，接收函数将正确地设置 skb->protocol，skb->pkt_type，和 skb->mac.raw。

skb->mac.raw 是一个被网络高层代码实现的地址解析机制使用的字符指针（例如，net/ipv4/arp.c）。它必须指向一个与 dev->type 匹配的机器地址。设备类型的可能值被定义在 <linux/if_arp.h>；以太网接口用 ARPHRD_ETHER。例如，下面是 *eth_type_trans* 处理收到包的以太网包头的方法：

```
skb->mac.raw=skb->data;
skb_pull(skb, dev->hard_header_len);
```

在最简单的情况下（无包头的点到点连接），skb->mac.raw 可以指向一个含有这个接口的硬件地址的静态缓冲区，protocol 可以被置为 ETH_P_IP，packet_type 仍维持其缺省值 PACKET_HOST。

加载时配置

用户可以用几个标准的关键字来配置接口。任何新的网络模块都应遵循这个标准：

io=
: 为接口设置 I/O 端口的基地址。如果系统中安装了不只一个接口，那么可以用一个由逗号分隔的列表来指定。

irq=
: 设置中断号。和上面一样，可以指定不止一个值。

换句话说，一个装了两个 own_eth 接口的 Linux 用户可能用下面的命令行来加载模块：

```
insmod own_eth.o io=0x300, 0x320 irq=5,7
```

如果指定 0 值，那么 io=和 irq=选项都要被探测。因此用户可以通过指定 io=0 来强制探测。如果用户名优指定任何选项，多数驱动程序通常都探测一个接口，但有时，模块可能被禁止探测。（见 ne.c 中关于 NE2000 设备的探测）。
设备驱动程序应该象刚才描述的这样工作。ISA 设备的典型实现如下所示，假设驱动程序最多可以支持四个接口：
（代码 331）
这段代码缺省探测一个板子，并总是自动探测中断，但用户可以改变这种行为。例如，io=0,0,0 将探测三块板子。
除了使用 io 和 irq 外，驱动程序的作者可以随意增加其它加载时配置参数。也没有已建立的命名标准。

运行时配置

用户有时可能希望在运行时改变接口的配置。例如，当中断号无法探测时，想对它正确配置的唯一办法就是“尝试—错误”技术。一个用户空间的程序可以获取设备的当前配置，并通过在一个打开的套接字上调用 *ioctl* 来设置一个新的配置。例如，应用 *ifconfig* 使用 *ioctl* 为接口设置 I/O 端口。
我们前面知道一个为网络接口定义的方法中的一个是 *set_config*。这个方法被用来在运行时设置或改变一些接口特征。
当一个程序询问当前配置时，核心从结构 device 中抽取相关信息，而不通知驱动程序；另一方面，当一个新的配置被传递给接口时，*set_config* 被调用，这样驱动程序就可以检查这些值并采取相应的动作。这个驱动程序方法对应下面的原形：

```
int (*set_config)(struct device *dev, struct ifmap *map);
```

map 参数指向一个由用户程序传递的结构的拷贝；这个拷贝已经在核心空间，所以驱动程序不需要调用 *memcpy_from _fs*。
结构 ifmap 的域是：

```
unsigned long mem_start;
unsigned long mem_end;
unsigned short base_addr;
unsigned char irq;
unsigned char dma;
```

这些域对应着结构 device 中的域。

```
Unsigned char prot;
```

这个域对应着 dev 中的 if_port。map->port 的含义是设备特定的。

当一个进程为设备发出 ioctl(SIOCSIFMAP)(即 Socket I/O Control Set InterFace MAP)时，*set_config* 设备方法被调用。这个进程在时强制使用新值之前，应该发出 ioctl(SIOCGIFMAP)(即 Socket I/O Control Get InterFace MAP)，这样驱动程序只需要查看 dev 和 ifmap 结构不匹配的地方。Map 中任何不被驱动程序使用的域均可以略过。例如，一个不使用 DMA 的网络设备可以忽略 map->dma。
snull 实现被设计成可以显示驱动程序是如何针对配置改变而动作的。对 *snull* 来说，没有一个域由物理意义。但出于说明的目的，代码禁止改变 I/O 地址，允许改变 IRQ 号，并忽略其它选项，从而显示这些改变是如何被响应、拒绝、或是忽略的。

（代码 333 #1）
这个方法的返回值被作为发出的 *ioctl* 系统调用的返回值，对于没有实现 *set_config* 的驱动程序则返回-EOPNOTSUPP。
如果你对接口配置如何从用户空间访问感到好奇，请看 *misc-progs/netifconfig.c*，它可以用来与 *set_config* 比较。下面是一个示例运行的输出：
（代码 333 #2）

自定义 *ioctl* 命令

我们已经看到 *ioctl* 系统调用是为套接字实现的。SIOCSIFADDR 和 SIOCSIFMAP 是“套接字 *ioctl*”的例子现在让我们看看这个系统调用的第三个参数是如何被网络代码使用的。
当 *ioctl* 系统调用在套接字上被调用时，其命令号是在<linux/sockios.h>定义的符号之一，并且函数 *sock_ioctl* 直接调用一个协议特定的函数（这里协议指使用的主要网络协议，如 IP 或 AppleTalk）。
任何协议层不认识的 *ioctl* 命令被传递给设备层。这些设备相关的 *ioctl* 命令从用户空间接收第三个参数，即结构 ifreq*；这个结构在<linux/if.h>中定义。SIOCSIFADDR 和 SIOCSIFMAP 命令实际上是工作在 ifreq 结构上。SIOCSIFMAP 的额外参数，尽管定义为 ifmap，是 ifreq 的一个域。
除了使用标准的调用，每个接口可以定义它自己的 *ioctl* 命令。例如 *plip* 接口允许通过 *ioctl* 修改其内部超时值。套接字的 *ioctl* 实现将 16 个命令看作对接口是私有的：从 SIOCDEVPRIVATE 到 SIOCDEVPRIVATE+15。
当这些命令中的一个被认识时，dev->do_ioctl 在相关的接口驱动程序里被调用。这个函数接收与通用目的的 *ioctl* 函数使用的一样的 ifreq*指针。

```
Int (*do_ioctl)(struct device *dev, struct ifreq *ifr, int cmd);
```

Ifr 指针指向核心空间的一个地址，放有被用户传来结构的一个拷贝。在 *do_ioctl* 返回后，这个结构又被拷贝回用户空间；这样，驱动程序可以使用私有命令来接收和返回数据。
设备特定的命令可以选择使用结构 ifreq 中的域，但它们已经带有标准的含义，驱动程序不太可能根据自己的需要适配这个结构。域 ifr_data 是个 caddr_t 项（一个指针），用于设备特定的需要。驱动程序和调用 *ioctl* 命令的程序应在 ifr_data 的使用上取得一致。例如，*pppstats* 使用设备特定的命令来从 *ppp* 接口驱动程序中获取信息。
在这里不值得给出 *do_ioctl* 的一个实现，但根据本章的信息和核心的例子，你应能在需要的时候写出一个。不过注意，*plip* 实现不正确地使用了 ifr_data，不应做为 *ioctl* 实现的一个例子。

统计信息

一个驱动程序需要的最后一个方法是 *get_stats*。这个方法返回指向设备统计信息的一个指针。它的实现相当容易：
（代码 335）
返回有意义的统计信息所需的实际工作散布在驱动程序中，不同的域分别被更新。下表给出 enet_statistics 结构中最有趣的几个域。
int rx_packets;

int tx_packets;

这两个域含有接口成功传送的进来和外出包的总数。

int rx_errors;

int tx_errors;

出错的接收和发送的数目。接收错可能是错误的校验和、错误的包大小，以及其它问题的结果。发送错误不太常见，一般都是线缆的问题。

int rx_dropped;

int tx_dropped;

在接收和发送时被丢弃的包的个数。当包数据没有可用内存时，包便被丢弃了。tx_dropped 很少使用。

这个结构还有几个域，可以用来细分发送和接收时发生的错误。感兴趣的读者可以看<linux/if_ether.h>中结构的定义。

选播

“选播”包是指一个网络包，它将要被多于一个，但又不是全部的主机接收。

这个功能是通过给一组主机赋以特殊的硬件地址来获得的。指向这些特殊地址中的一个的包将被这个组中所有的主机收到。在以太网的情况下，一个选播地址是将目的地址中第一个八元组的最低位设置而得到，而所有的设备板子都在其硬件地址中将这一位清除。

处理主机组以及硬件地址的棘手部分都有应用或核心完成了，接口驱动程序并不需要处理这些问题。

选播包的发送很简单，与其它包完全一样。接口在传输介质上发送它们，根本不管目的地址。核心必须设置一个正确的硬件目的地址；*rebuild_header* 设备方法（如果被定义）并不需要查看它整理的数据。

而另一方面，接收选播包需要设备的一些合作。当一个“有趣的”选播包被收到时（也就是一个包的目的地址确定一组主机，其中包含了这个接口），硬件应该通知操作系统。这意味着硬件过滤器应被设计为能够区别不同的选播地址。这个过滤器在接口的一般操作中，将网络包的地址与其自己的硬件地址进行匹配。

典型地，在考虑选播的情况下，硬件可分为一下三类：

- 不能处理选播的接口。这类接口要么接收指向自己硬件地址的包（包括播送包），要么节收所有的包。它们接收选播包是通过接收所有的包实现的，这样，操作系统中会充斥着大量“无意义”的包。一般我们不认为这类接口为支持选播的，驱动程序不在 dev->flags 中置 IFF_MULTICAST。

  点到点接口是一种特殊情况，它们通常接收所有的包，根本不进行任何硬件过滤。
- 能区别选播包和其它包（主机到主机或播送）的接口。这类接口可以被要求接收所有的选播包，然后用软件来判断自己是否是有效的接收者。这种情形引入的开销是可以接收的，因为一个典型的网络中选播包的数量都很少。
- 能够进行选播地址硬件检测的接口。可以给这类接口一组需要接收的选播地址，它们会忽略其它的选播包。这对核心来说是最优化的情况，因为不会浪费处理器事件去丢弃接口收到的“无意义”的包。

核心试图利用高级接口的能力，能最好地支持第三类接口（用途最广）。因此，当有效的选播地址被改变时核心应通知驱动程序，它把新的一组地址传给驱动程序，这样它可以按照新的信息更新硬件过滤器。

核心对选播的支持

下面是与驱动程序的选播能力相关的数据结构和函数的概括：

void (*dev->set_multicast_list)(struct device *dev)

当与设备相关的机器地址表改变时调用这个设备方法。当 dev->flags 被修改时它也被调用，因为有些标志也要求你重新配置硬件过滤器。这个方法接收一个指向 device 结构的指针作为参数，返回 void。对实现这个方法不感兴趣的驱动程序可以留域为 NULL。

struct dev_mc_list *dev->mc_list

这是域设备相关的所有选播地址的链表。这个结构的实际定义在本节结束时介绍。

int dev->mc_count

链表项数。这个信息有点冗余，但检查 mc_count 是否为 0 是优于列表检查的一个有用的快捷方式。

IFF_MULTICAST

除非驱动程序在 dev->flags 中设置了这个标志，接口将不必处理选播包。当 dev->flags 改变时，至少 set_multicast_list 会被调用。

IFF_ALLMULTI

dev->flags 中的这个标志被网络软件设置以告诉驱动程序从网络中抽取所有播送包。这在 multicast-routing 被使用时发生。如果这个标志被置位，dev->mc_list 将不再使用去过滤选播包。

IFF_PROMISC

当接口被置为杂乱模式时，dev->flags 中的这个标志被设置。所有的包都被接口抽取，不考虑 dev->mc_list。

驱动程序开发者需要的最后一点信息是结构 dev_mc_list 的定义，它居于<linux/netdevice.h>中。

（代码 337）

由于选播和硬件地址与包的实际发送无关，这个结构可在不同的网络实现上移植，每个地址由一串八元组和一个长度确定，就象 dev->dev_addr。

一个典型的实现

描述 *set_multicast_list* 设计的最号办法是给出一些伪代码。

下面的函数是这个函数在一个全特征(ff)驱动程序中的一个典型实现。说这个驱动程序是全特征的是因为它控制的接口有一个复杂的硬件包过滤器，它可以存放一个由本机接收的选播地址表。这个表的最大尺寸是 FF_TABLE_SIZE。

所由带有前缀 ff_的函数是放置硬件特定操作的地方。

（代码 338）

如果这个接口不能在硬件过滤器中存储到来包的选播表，那么这个实现还可以简化。在这种情况下，FF_TABLE_SIZE 减为 0，代码的最后四行也不需要了。

现在，接口板一般不能存储选播表。不过，这并不是一个大问题，因为网络代码的高层会负责将不需要的包丢弃。
如我前面建议的，即使不能处理选播包的接口也需要实现 *set_multicast_list* 方法，这样当 dev->flags 发生改变时可以被通知。我称这个为“无特征”(nf)的实现。它非常简单，如下面所示：
（代码 339）
处理 IFF_PROMISC 是很重要的，因为不然的话，用户将无法运行 *tcpdump* 或其它一些网络分析工具。另一方面，如果接口运行一个点到点的连接，则没有任何实现 *set_multicast_list* 的必要，因为它们接收所有的包。

快速参考

本节提供在本章中介绍的概念的参考。它也解释了驱动程序应包含的每个头文件的作用。不过，device 和 sk_buff 的每个域的列表，就不再重复了。

#include <linux/netdevice.h>

这个头文件含有 struct device 的定义，还包含了网络驱动程序需要的几个其它头文件。

void netif_rx(struct sk_buff *skb);

这个函数在中断时可以被调用通知核心一个包被收到了，并且封装在一个套接字缓冲区中。

#include <linux/if.h>

被 *netdevice.h* 包含，这个文件声明接口标志(IFF_macros)和结构 ifmap，它在网络驱动程序的 *ioctl* 实现中用重要的作用。

#include <linux/if_ether.h>

ETH_ALEN

ETH_P_IP

struct ethhdr;

struct enet_statistics;

被 *netdevice.h* 包含，*if_ether.h* 定义所有的 ETH_macros，用来表示八元组长度（象地址长度）和网络协议（象 IP）。它也定义了结构 ethhdr 和 enet_statistics。注意，不要看 enet_statistics 的名字和包含它的头文件，事实上，所有的接口都要用到它，不仅仅是以太网。

#include <linux/skbuff.h>

结构 sk_buff 和一些相关结构的定义，以及在缓冲区上操作的线入函数。这个头文件包含在 *netdevice.h* 中。

#include <linux/etherdevice.h>

void ether_setup(struct device *dev);

这个函数为以太网驱动程序设置大多数设备方法为通用目的的实现。它同样设置 dev->flags，并且在名字中的第一个字符是空格或空字符时，将下一个可用的 ethx 名赋给 dev->name。

unsigned short eth_type_trans(struct sk_buff *skb, struct device *dev)

当以太网接口收到一个包，这个函数将被调用来设置 skb->pkt_type。返回值是一个协议号，通常被存在 skb->protocol 中。

#include <linux/sockios.h>

SIOCDEVPRIVATE

这是 16 个 *ioctl* 命令的第一个，可以被每个驱动程序实现以供私用。所有的网络 *ioctl* 命令都在 *sockios.h* 中定义。