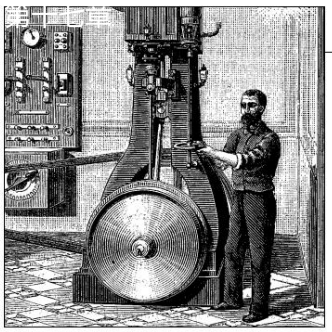

第十七章

网络驱动程序

讨论了字符和块驱动程序之后,我们现在准备进入网络世界。网络接口是 Linux 设备的第三类标准,本章描述了它们如何与内核的其余部分交互。

系统中网络接口的作用类似于挂载的块设备。块设备向内核注册其磁盘和方法,然后通过其请求函数根据请求“传输”和“接收”块。

类似地,网络接口必须在特定的内核数据结构中注册自己,以便在与外界交换数据包时调用。

挂载的磁盘和数据包传送接口之间有一些重要的区别。首先,磁盘作为/dev目录中的特殊文件存在,而网络接口没有这样的入口点。正常的文件操作（读、写等)在应用于网络接口时没有意义,因此不可能对它们应用 Unix“一切都是文件”的方法。因此,网络接口存在于它们自己的命名空间中并导出一组不同的操作。

尽管您可能会反对应用程序在使用套接字时使用read和write系统调用,但这些调用作用于与接口不同的软件对象。
数百个套接字可以在同一个物理接口上复用。

但两者之间最重要的区别是块驱动程序仅在响应来自内核的请求时运行,而网络驱动程序从外部异步接收数据包。因此,当块驱动程序被要求向内核发送缓冲区时,网络设备要求将传入的数据包推向内核。网络驱动程序的内核接口是为这种不同的操作模式而设计的。

网络驱动程序还必须准备好支持许多管理任务,例如设置地址、修改传输参数以及维护流量和错误统计信息。网络驱动程序的 API 反映了这种需求,因此看起来与我们目前看到的接口有些不同。

Linux 内核的网络子系统被设计成完全独立于协议。这适用于网络协议（互联网协议 [IP] 与 IPX 或其他协议)和硬件协议（以太网与令牌环等）。

网络驱动程序和内核之间的交互一次正确地处理一个网络数据包;这允许从驱动程序中巧妙地隐藏协议问题,并从协议中隐藏物理传输。

本章描述了网络接口如何与 Linux 内核的其余部分相匹配,并以基于内存的模块化网络接口的形式提供示例,称为（您猜对了） snull。为简化讨论,接口使用以太网硬件协议并传输 IP 数据包。您通过检查snull获得的知识可以很容易地应用于 IP 以外的协议,并且编写非以太网驱动程序仅在与实际网络协议相关的微小细节上有所不同。

本章不讨论 IP 编号方案、网络协议或其他一般网络概念。驱动程序编写者（通常)不关心这些主题,并且不可能在不到几百页的时间内提供令人满意的网络技术概述。建议感兴趣的读者参考其他描述网络问题的书籍。

在进入网络设备之前,需要对术语进行注释。网络世界使用术语八位字节来指代一组八位，通常是网络设备和协议所理解的最小单位。在这种情况下几乎从未遇到过术语字节。为了与标准用法保持一致,我们在谈论网络设备时将使用八位字节。

“标题”一词也值得一提。标头是在数据包通过网络子系统的各个层时附加到数据包的一组字节（错误,八位字节）。当应用程序通过 TCP 套接字发送数据块时,网络子系统会将数据分解为数据包,并在开头放置一个 TCP 标头,描述每个数据包在流中的位置。然后,较低级别的 TCP 标头前面放置一个 IP 标头,用于将数据包路由到其目的地。如果数据包在类似以太网的介质上移动,则由硬件解释的以太网报头位于其余部分之前。网络驱动程序（通常)不需要关心更高级别的标头,但它们通常必须参与硬件级别标头的创建。

snull 是如何设计的

本节讨论导致snull网络接口的设计概念。
尽管此信息可能看起来没什么用,但如果不理解它可能会在您使用示例代码时导致问题。

第一个也是最重要的设计决定是示例接口应该保持独立于真实硬件,就像本文中使用的大多数示例代码一样

书。这种约束导致了类似于环回接口的东西。 snull不是环回接口;但是,它模拟了与真实远程主机的对话,以便更好地演示编写网络驱动程序的任务。 Linux 环回驱动程序实际上非常简单;它可以在drivers/net/loopback.c 中找到。

snull的另一个特点是它只支持 IP 流量。这是接口内部工作的结果 snull必须查看内部并解释数据包以正确模拟一对硬件接口。真正的接口不依赖于传输的协议， snull的这种限制不会影响本章中显示的代码片段。

分配 IP 编号snull模块创建两

个接口。这些接口不同于简单的环回,因为您通过其中一个接口传输的任何内容都会环回另一个接口,而不是环回自身。看起来您有两个外部链接,但实际上您的计算机正在回复自己。

不幸的是,这种效果不能单独通过 IP 号分配来实现,因为内核不会通过接口 A 发送一个指向它自己的接口 B 的数据包。相反,它会使用环回通道而不通过snull。为了能够通过snull接口建立通信,需要在数据传输过程中修改源地址和目的地址。换句话说,通过其中一个接口发送的数据包应该被另一个接口接收,但传出数据包的接收者不应该被识别为本地主机。这同样适用于接收数据包的源地址。

为了实现这种“隐藏环回”， snull接口切换源地址和目标地址的第三个八位字节的最低有效位;即改变C类IP号的网络号和主机号。最终结果是发送到网络 A（连接到sn0,第一个接口)的数据包在sn1接口上显示为属于网络 B 的数据包。

为避免处理太多数字,让我们为所涉及的 IP 号码分配符号名称：

- snullnet0是连接到 sn0 接口的网络。同样， snullnet1是连接到sn1 的网络。这些网络的地址应该仅在第三个八位字节的最低有效位上有所不同。这些网络必须具有 24 位网络掩码。

- local0是分配给sn0接口的IP 地址;它属于snullnet0。
  与sn1关联的地址是local1。 local0和local1的第三个八位字节和第四个八位字节的最低有效位必须不同。 · remote0是snullnet0 中的一个主机,它的第四个八位字节与local1 的相同。
  发送到remote0的任何数据包在其网络地址被删除后到达local1

由接口代码修改。主机remote1属于snullnet1,其第四个八位字节与local0相同。

snull接口的操作如图 17-1 所示,其中与每个接口关联的主机名打印在接口名称附近。

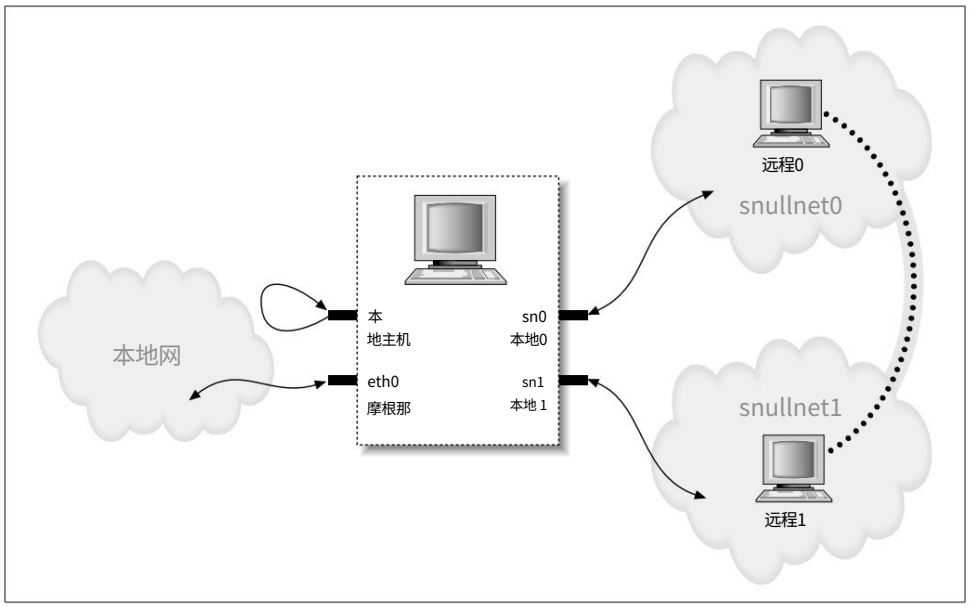

图 17-1。主机如何查看其接口

以下是网络编号的可能值。将这些行放入/etc/networks 后,您可以按名称调用您的网络。这些值是从保留供私人使用的数字范围中选择的。

```
snullnet0       192.168.0.0
snullnet1       192.168.1.0
```

以下是可以放入/etc/hosts 的主机号:

```
192.168.0.1 本地 0 192.168.0.2
远程 0 192.168.1.2 本地 1
192.168.1.1 远程 1
```

这些数字的重要特征是local0的主机部分与remote1的主机部分相同， local1的主机部分与remote0的主机部分相同。只要这种关系适用,您就可以使用完全不同的数字。

但是,如果您的计算机已经连接到网络,请小心。您选择的号码可能是真实的 Internet 或 Intranet 号码,将它们分配给您的接口会阻止与真实主机的通信。例如,虽然

刚刚显示的号码不是可路由的 Internet 号码,它们可能已经被您的专用网络使用。

无论您选择什么数字,您都可以通过发出以下命令正确设置操作界面:

```
ifconfig sn0 local0
ifconfig sn1 local1
```

如果选择的地址范围不是 C 类范围,您可能需要添加网络掩码 255.255.255.0参数。

此时,可以到达界面的“远程”端。以下屏幕转储显示主机如何通过snull接口到达remote0和remote1:

```
morgana% ping -c 2 remote0来
自 192.168.0.99 的 64 个字节:icmp_seq=0 ttl=64 time=1.6 ms 来自
192.168.0.99 的 64 个字节:icmp_seq=1 ttl=64 time=0.9 ms 发送 2 个数
据包,接收 2 个数据包,0 % 数据包丢失

morgana% ping -c 2 remote1来
自 192.168.1.88 的 64 个字节:icmp_seq=0 ttl=64 time=1.8 ms 来自
192.168.1.88 的 64 个字节:icmp_seq=1 ttl=64 time=0.9 ms 2 个数据包
传输,2 个数据包接收,0 % 数据包丢失
```

请注意,您将无法访问属于这两个网络的任何其他“主机”,因为在修改地址并收到数据包后,您的计算机会丢弃数据包。例如,针对 192.168.0.32 的数据包将通过 sn0 离开并重新出现在 sn1 ,其目标地址为 192.168.1.32,这不是主机的本地地址。

数据包的物理传输就数据传输而言, snull接

口属于以太网类。 snull模拟以太网,因为绝大多数现有网络(至少是工作站连接的网段)都基

于以太网技术,无论是 10base-T、100base-T 还是千兆位。此外,内核为以太网设备提供了一些通用支持,没有理由不使用它。作为以太网设备的优势是如此强大,以至于即使是plip接口(使用打印机端口的接口)也将自己声明为以太网设备。

为snull使用以太网设置的最后一个优点是您可以在接口上运行tcpdump以查看数据包经过。使用tcpdump观察接口可能是了解这两个接口如何工作的有用方法。

如前所述, snull仅适用于 IP 数据包。此限制是由于snull窥探数据包甚至修改数据包以使代码正常工作的结果。该代码修改了每个数据包的 IP 标头中的源、目标和校验和,而不检查它是否真正传达了 IP 信息。

这种快速而肮脏的数据修改会破坏非 IP 数据包。如果你想通过snull传递其他协议,你必须修改模块的源代码。

连接到内核

我们通过剖析snull源代码开始研究网络驱动程序的结构。
将几个驱动程序的源代码放在手边可能会帮助您关注讨论并了解真实世界的 Linux 网络驱动程序是如何运行的。作为开始,我们建议使用loopback.c、 plip.c和e100.c,以增加复杂性。所有这些文件都存在于内核源代码树中的驱动程序/网络中。

设备注册当一个驱动模块被

加载到一个正在运行的内核中时,它会请求资源并提供设施;没有什么新鲜的。请求资源的方式也没有什么新东西。驱动程序应该探测它的设备和它的硬件位置（I/O 端口和 IRQ 线） 但不注册它们 如第 10 章“安装中断处理程序”中所述。网络驱动程序通过其 mod 注册的方式ule 初始化函数不同于字符和块驱动程序。由于网络接口没有等效的主要和次要编号,因此网络驱动程序不需要这样的编号。相反,驱动程序将每个新检测到的接口的数据结构插入到网络设备的全局列表中。

每个接口都由一个结构 net_device项目描述,该项目在<linux/netdevice.h> 中定义。 snull驱动程序将指向其中两个结构（用于sn0和sn1)的指针保存在一个简单的数组中：

```
结构 net_device *snull_devs[2];
```

与许多其他内核结构一样， net_device结构包含一个 kobject,因此通过 sysfs 进行引用计数和导出。与其他此类结构一样,它必须动态分配。提供用于执行此分配的内核函数是alloc_netdev,它具有以下原型： struct net_device *alloc_netdev(int sizeof_priv, const char *name, void (*setup)(struct net_device *));这里， sizeof_priv是驱动程序“私有数据”区域的大小;

对于网络设备,该区域与net_device结构一起分配。事实上,这两者被分配在一大块内存中,但驱动程序作者应该假装他们不知道这一点。 name是这个接口的名字,就像用户空间看到的那样;此名称中可以

包含printf 样式的%d 。内核将%d替换为下一个可用的接口号。最后， setup是一个指向初始化函数的指针,调用该函数来设置net_device结构的其余部分。我们进入初始化函数

很快,但现在,只要说snull以这种方式分配它的两个设备结构就足够了:

```
snull_devs[0] = alloc_netdev(sizeof(struct snull_priv),  sn%d  ,
        snull_init); snull_devs[1] = alloc_netdev(sizeof(struct snull_priv),
  sn%d  , snull_init); if (snull_devs[0] = = NULL || snull_devs[1] = = NULL)
        退出;
```

与往常一样,我们必须检查返回值以确保分配成功。

网络子系统为各种类型的接口提供了许多围绕alloc_netdev的辅助函数。最常见的是alloc_etherdev,在<linux/etherdevice.h>中定义:

```
结构 net_device *alloc_etherdev(int sizeof_priv);
```

此函数使用eth%d作为 name 参数分配网络设备。它提供了自己的初始化函数（ether_setup），它为以太网设备设置了几个带有适当值的net_device字段。因此， alloc_etherdev没有驱动程序提供的初始化函数;驱动程序应该在成功分配后直接进行所需的初始化。其他类型设备的驱动程序编写者可能希望利用其他辅助函数之一,例如用于光纤通道设备的 alloc_fcdev（在 <linux/fcdevice.h> 中定义）、 alloc_fddidev （< linux /fddidevice.h>)用于 FDDI 设备,或alloc_trdev (<linux/trdevice.h>)用于令牌环设备。

snull可以毫无问题地使用alloc_etherdev ;我们选择使用alloc_netdev来代替,作为演示较低级别接口的一种方式,并让我们能够控制分配给接口的名称。

一旦net_device结构被初始化,完成这个过程只需将结构传递给register_netdev。在snull 中,调用如下所示:

```
对于 (i = 0; i < 2; i++)
    如果 ((结果 = register_netdev(snull_devs[i])))
        printk(  snull: error %i 注册设备\  %s\  \n  ,
                结果,snull_devs[i]->name);
```

通常的注意事项在这里适用:一旦您调用register_netdev,您的驱动程序可能会被调用以对设备进行操作。因此,在所有东西都完全初始化之前,您不应该注册设备。

初始化每个设备我们已经查看了

net_device结构的分配和注册,但是我们忽略了完全初始化该结构的中间步骤。注意struct net_device总是在运行时放在一起;它不能在编译时以与file_operations或block_device_operations结构相同的方式设置。

此初始化必须在调用register_netdev 之前完成。网络设备

结构大而复杂;幸运的是,内核通过ether_setup函数（由alloc_etherdev 调用)处理了一些以太网范围的默认值。

由于snull使用alloc_netdev,它有一个单独的初始化函数。这个的核心函数（snull_init)如下：

```
以太设置（开发）； /* 分配一些字段 */

开发->打开= snull_open；
开发->停止 = snull_release;
开发->set_config = snull_config;
开发->hard_start_xmit = snull_tx;
开发->do_ioctl = snull_ioctl;
开发->get_stats = snull_stats；
dev->rebuild_header = snull_rebuild_header;
开发->hard_header = snull_header；
开发-> tx_timeout = snull_tx_timeout；
dev->watchdog_timeo = 超时；
/* 保留默认标志,只需添加 NOARP */
开发->标志                |= IFF_NOARP;
开发->功能                |= NETIF_F_NO_CSUM；
dev->hard_header_cache = NULL；          /* 禁用缓存 */
```

上面的代码是net_device结构的一个相当常规的初始化;这是主要是存储指向我们各种驱动程序函数的指针。单人代码的不寻常功能是在标志中设置IFF_NOARP 。这指定了接口不能使用地址解析协议 (ARP)。 ARP是低级的以太网协议;它的工作是将 IP 地址转换为以太网介质访问控制 (MAC) 地址。由于snull模拟的“远程”系统并不真正存在,没有人可以为他们回答 ARP 请求。我们没有通过添加 ARP 实现使snull 变得复杂,而是选择将接口无法处理该协议。分配给hard_header_缓存出于类似的原因:它禁用了（不存在的)ARP的缓存在这个界面上回复。该主题将在“MAC”一节中详细讨论地址解析”在本章后面。

初始化代码还设置了几个字段（tx_timeout和watchdog_timeo）这与处理传输超时有关。我们彻底涵盖了这个主题在“传输超时”部分。

我们现在再看一个struct net_device字段priv。它的作用类似于我们用于 char 驱动程序的private_data指针。与fops->private_data 不同，此priv指针与net_device结构一起分配。直接访问出于性能和灵活性的原因,也不鼓励使用priv字段。当一个司机需要访问私有数据指针,它应该使用netdev_priv函数。因此， snull驱动程序充满了如下声明：

```
结构 snull_priv *priv = netdev_priv(dev);
```

snull模块声明了一个用于priv的snull_priv数据结构：

```
结构 snull_priv { 结构
    net_device_stats 统计;整数状态;结
    构 snull_packet *ppool;结构
    snull_packet *rx_queue; /* 传入数
    据包列表 */ int rx_int_enabled; int tx_packetlen; u8 *tx_packetdata;结
    构 sk_buff *skb; spinlock_t 锁；




};
```

除其他外，该结构包括一个struct net_device_stats 实例，它是保存接口统计信息的标准位置。snull_init中的以下行分配和初始化dev->priv：

```
priv = netdev_priv(dev);
memset(priv, 0, sizeof(struct snull_priv));
spin_lock_init(&priv->lock); snull_rx_ints（开发，
1）； /* 启用接收中断 */
```

模块卸载 卸载模块时没有

什么特别的事情发生。模块清理函数简单地取消注册接口，执行任何需要的内部清理，并将net_device结构释放回系统：

```
无效snull_cleanup（无效）{

    诠释我;

    for (i = 0; i < 2; i++) { if
        (snull_devs[i])
            { unregister_netdev(snull_devs[i]);
            snull_teardown_pool(snull_devs[i]);
            free_netdev(snull_devs[i]);
        }

    } 返回;
}
```

对unregister_netdev的调用将接口从系统中移除； free_netdev将net_device结构返回给内核。如果对该结构的引用存在于某处，它可能会继续存在，但您的驱动程序不需要关心这一点。一旦你注销了接口，内核就不再调用它的方法。

请注意，在取消注册设备之前，我们的内部清理（在snull_teardown_pool 中完成）不会发生。然而，它必须在我们将net_device结构返回系统之前发生；一旦我们调用了 free_netdev，我们就不能进一步引用该设备或我们的私人区域。

详细的 net_device 结构

net_device结构是网络驱动层的核心,值得完整描述。此列表描述了所有字段,但更多的是提供参考而不是记忆。本章的其余部分将在示例代码中使用每个字段时简要介绍每个字段,因此您无需继续参考本节。

全球资讯

struct net_device的第一部分由以下字段组成:

字符名称[IFNAMSIZ];
: 设备的名称。如果驱动设置的名称中包含%d格式字符串,则register_netdev将其替换为一个数字,使其成为唯一的名称;分配的编号从0 开始。

无符号长状态;设备状
: 态。该字段包括几个标志。驱动程序通常不会直接处理这些标志;相反,提供了一组实用功能。

当我们进入驱动程序操作时,将很快讨论这些函数。

结构网络设备*下一个;指向
: 全局链表中下一个设备的指针。驱动程序不应触及该字段。 int (*init)(struct net_device *dev);

一个初始化函数。如果设置了该指针,则由register_net dev调用该函数来完成net_device结构的初始化。大多数现代网络驱动程序不再使用此功能;相反,在注册接口之前执行初始化。

硬件信息

以下字段包含相对简单设备的低级硬件信息。它们是早期 Linux 网络的遗留物。大多数现代驱动程序都使用它们(if_port 可能除外)。为了完整起见,我们在此处列出它们。

无符号长rmem_end;无符
号长rmem_start;无符号长
mem_end;无符号长
mem_start;设备内存信息。这
: 些字段保存设备使用的共享内存的开始和结束地址。如果设备具有不同的接收和发送存储器,则mem字段用于发送存储器,而rmem字段用于接收存储器。 rmem字段从不在外部引用

驱动程序本身。按照惯例,设置结束字段以便结束 - 开始是可用的板载内存量。

无符号长基地址;网络接口的 I/
O 基地址。该字段与前面的字段一样,由驱动程序在设备探测期间分配。 ifconfig命令可用于显示或修改当前值。 base_addr可以在系统启动时（通过netdev=参数)或模块加载时在内核命令行上显式分配。与上面描述的内存字段一样,该字段不被内核使用。无符号字符中断;分配的中断号。当列出接口时,ifconfig会打印dev->irq的值。该值通常可以在引导或加载时设置,稍后使用ifconfig 进行修改。

无符号字符 if_port;多端口
设备上使用的端口。例如,此字段用于同时支持同轴(IF_PORT_10BASE2)和双绞线(IF_PORT_100BASET)的设备
以太网连接。完整的已知端口类型集在<linux/netdevice.h> 中定义。

无符号字符 DMA;设备
分配的 DMA 通道。该字段仅对某些外围总线有意义,例如 ISA。它不在设备驱动程序本身之外使用,而是用于信息目的（在ifconfig 中）。

接口信息

关于接口的大部分信息都由ether_setup函数正确设置（或任何其他适合给定硬件类型的设置函数）。以太网卡的大部分字段都可以依赖此通用功能,但flags和dev_addr字段是特定于设备的,必须在初始化时显式分配。

一些非以太网接口可以使用类似于ether_setup 的辅助函数。 drivers/net/net_init.c导出了许多这样的函数,包括:

无效 ltalk_setup(struct net_device *dev);
设置 LocalTalk 设备的字段

无效 fc_setup(struct net_device *dev);
初始化光纤通道设备的字段

无效 fddi_setup(struct net_device *dev);
为光纤分布式数据接口 (FDDI) 网络配置接口

无效 hippi_setup(struct net_device *dev);
为高性能并行接口 (HIPPI) 高速互连驱动程序准备字段

无效 tr_setup(struct net_device *dev);
处理令牌环网络接口的设置

大多数设备都包含在这些类之一中。但是,如果您的内容是全新的和不同的,则需要手动分配以下字段:

无符号短 hard_header_len;硬件报
头长度,即在 IP 报头或其他协议信息之前引导传输的数据包的八位字节数。对于以太网接口,hard_header_len的值为14 (ETH_HLEN) 。

未签名的mtu;
最大传输单位 (MTU)。网络层使用该字段来驱动数据包传输。以太网的 MTU 为 1500 个八位字节(ETH_DATA_LEN)。
可以使用ifconfig 更改此值。无符号长
tx_queue_len;
可以在设备的传输队列中排队的最大帧数。该值由ether_setup 设置为 1000,但您可以更改它。例如， plip使用 10 来避免浪费系统内存（plip的吞吐量低于真正的以太网接口）。

无符号短类型;接口的硬
件类型。 ARP使用类型字段来确定接口支持的硬件地址类型。以太网接口的正确值是ARPHRD_ETHER,也就是 ether_setup 设置的值。可识别的类型在<linux/if_arp.h> 中定义。

无符号字符 addr_len;无符
号字符广播[MAX_ADDR_LEN];无符号字符
dev_addr[MAX_ADDR_LEN];
硬件 (MAC) 地址长度和设备硬件地址。以太网地址长度为六个八位字节（我们指的是接口板的硬件ID）,广播地址由六个0xff八位字节组成； ether_setup安排这些值是正确的。另一方面,设备地址必须以设备特定的方式从接口板上读取,驱动程序应将其复制到dev_addr。在将数据包移交给驱动程序进行传输之前,硬件地址用于生成正确的以太网标头。

snull设备不使用物理接口,它发明了自己的硬件地址。

无符号短标志;整数特征;
接口标志（详见下文）。

flags字段是一个位掩码,包括以下位值。 IFF_前缀代表“接口标志”。一些标志由内核管理,一些由接口在初始化时设置,以声明接口的各种功能和其他特性。在<linux/if.h>中定义的有效标志是:

IFF_UP
该标志对驱动程序是只读的。当接口处于活动状态并准备好传输数据包时,内核将其打开。

IFF_BROADCAST
这个标志(由网络代码维护)表明接口允许广播。以太网板可以。

IFF_DEBUG
这标志着调试模式。该标志可用于控制printk调用的详细程度或用于其他调试目的。虽然目前没有 in-tree 驱动程序使用此标志,但它可以由用户程序通过ioctl 设置和重置,并且您的驱动程序可以使用它。 misc-progs/netifdebug程序可用于打开和关闭标志。

IFF_LOOPBACK
该标志只能在环回接口中设置。内核检查IFF_LOOPBACK而不是将lo名称硬连线为特殊接口。

IFF_POINTOPOINT
该标志表示接口连接到点对点链路。它由驱动程序设置,有时由ifconfig 设置。例如, plip和PPP 驱动程序对其进行了设置。

IFF_NOARP
这意味着接口不能执行ARP。例如,点对点接口不需要运行 ARP,这只会增加额外的流量而不会检索有用的信息。 snull在没有 ARP 功能的情况下运行,因此它设置了标志。

IFF_PROMISC
该标志设置(由网络代码)以激活混杂操作。默认情况下,以太网接口使用硬件过滤器来确保它们仅接收广播数据包和定向到该接口硬件地址的数据包。数据包嗅探器(例如tcpdump)在接口上设置混杂模式,以便检索在接口传输介质上传输的所有数据包。

IFF_MULTICAST
此标志由驱动程序设置以标记能够进行多播传输的接口。 ether_setup默认设置IFF_MULTICAST ,因此如果您的驱动程序不支持多播,则必须在初始化时清除该标志。

IFF_ALLMULTI
该标志告诉接口接收所有多播数据包。只有在设置了IFF_MULTICAST时,内核才会在主机执行多播路由时设置它。 IFF_ALLMULTI是

驱动程序只读。本章后面的“组播”部分使用了组播标志。

IFF_MASTER IFF_SLAVE这些标志由负载均衡代码使用。接口驱动程序不需要知道它们。

IFF_PORTSEL
IFF_AUTOMEDIA
这些标志表明设备能够在多种媒体类型之间切换;例如,非屏蔽双绞线 (UTP) 与同轴以太网电缆。
如果设置了IFF_AUTOMEDIA ,设备会自动选择正确的介质。实际上,内核不使用任何一个标志。

IFF_DYNAMIC
这个标志,由驱动设置,表示这个接口的地址可以改变。内核当前不使用它。

IFF_RUNNING
此标志指示接口已启动并正在运行。它主要用于 BSD 兼容性;内核很少使用它。大多数网络驱动程序不必担心IFF_RUNNING。

IFF_NOTRAILERS
该标志在 Linux 中未使用,但它的存在是为了与 BSD 兼容。

当程序更改IFF_UP 时,将调用打开或停止设备方法。此外,当IFF_UP或任何其他标志被修改时,调用set_multicast_list方法。如果驱动程序需要执行某些操作以响应标志的修改,它必须在set_multicast_list 中执行该操作。例如,当IFF_PROMISC设置或重置时,set_multicast_list必须通知板载硬件过滤器。此设备方法的职责在“多播”部分中进行了概述。

net_device结构的features字段由驱动程序设置,以告诉内核此接口具有的任何特殊硬件功能。我们将讨论其中的一些功能;其他的超出了本书的范围。全套是:

NETIF_F_SG
NETIF_F_FRAGLIST
这两个标志都控制分散/收集 I/O 的使用。如果您的接口可以传输已拆分为多个不同内存段的数据包,则应设置NETIF_F_SG。当然,您必须实际实现分散/聚集 I/O(我们将在“分散/聚集 I/O”一节中描述如何完成)。 NETIF_F_FRAGLIST表明您的接口可以处理已被分段的数据包;只有环回驱动程序在 2.6 中执行此操作。

请注意,如果内核也不提供某种形式的校验和,则内核不会对您的设备执行分散/收集 I/O。原因是,如果

内核必须通过一个分段的（“非线性”）数据包来计算校验和,它可能会同时复制数据并合并数据包。

NETIF_F_IP_CSUM
NETIF_F_NO_CSUM
NETIF_F_HW_CSUM
这些标志都是告诉内核它不需要对通过这个接口离开系统的部分或所有数据包应用校验和的方式。如果您的接口可以校验和 IP 数据包但不能校验其他数据包,请设置NETIF_F_IP_CSUM 。如果此接口不需要校验和,请设置NETIF_F_NO_CSUM。环回驱动设置了这个标志， snull也设置了;由于数据包仅通过系统内存传输,因此（希望！）它们没有机会被破坏,也无需检查它们。如果您的硬件自己进行校验和,请设置NETIF_F_HW_CSUM。

NETIF_F_HIGHDMA如果您的设备可以对高端内存执行 DMA,则设置此标志。在没有这个标志的情况下,提供给驱动程序的所有数据包缓冲区都分配在低内存中。

NETIF_F_HW_VLAN_TX
NETIF_F_HW_VLAN_RX
NETIF_F_HW_VLAN_FILTER
NETIF_F_VLAN_CHALLENGED
这些选项描述了您的硬件对 802.1q VLAN 数据包的支持。 VLAN 支持超出了本章的范围。如果 VLAN 数据包混淆了您的设备（它们确实不应该）,请设置NETIF_F_VLAN_CHALLENGED标志。

NETIF_F_TSO
如果您的设备可以执行 TCP 分段卸载,请设置此标志。 TSO 是一项高级功能,我们无法在此介绍。

设备方法

与 char 和 block 驱动程序一样,每个网络设备都声明了作用于它的函数。本节列出了可以在网络接口上执行的操作。某些操作可以保留为NULL,而其他操作通常保持不变,因为ether_setup为它们分配了合适的方法。

网络接口的设备方法可以分为两组:基本的和可选的。基本方法包括能够使用接口所需的方法;可选方法实现了并非严格要求的更高级功能。以下是基本方法： int (*open)(struct net_device *dev);

打开界面。每当ifconfig激活该界面时,该界面就会打开。 open方法应该注册它需要的任何系统资源（I/O 端口、IRQ、

DMA 等），打开硬件，然后执行设备所需的任何其他设置。 int (*stop)(struct net_device *dev);

停止界面。接口关闭时停止。此功能应反转在打开时执行的操作。 int (*hard_start_xmit) (struct sk_buff *skb, struct net_device *dev);

启动数据包传输的方法。完整的数据包（协议头和所有）包含在套接字缓冲区（sk_buff）结构中。套接字缓冲区将在本章后面介绍。

int (*hard_header) (struct sk_buff *skb, struct net_device *dev, unsigned short type, void *daddr, void *saddr, unsigned len);函数（在hard_start_xmit 之前调用）从先前检索到的源和目标硬件地址构建硬件头；它的工作是将作为参数传递给它的信息组织到适当的、特定于设备的硬件头中。 eth_header是类以太网接口的默认函数，ether_setup 相应地分配该字段。 int (*rebuild_header)(struct sk_buff *skb);用于在 ARP 解析完成后但在数据包传输之前重建硬件头的函数。以太网设备使用的默认功能使用 ARP 支持代码来填充数据包的缺失信息。

void (*tx_timeout)(struct net_device *dev);当数据包传输未能在合理的时间内完成时，网络代码调用的方法，假设中断已丢失或接口已锁定。它应该处理问题并恢复数据包传输。

struct net_device_stats *(*get_stats)(struct net_device *dev);每当应用程序需要获取接口的统计信息时，都会调用此方法。例如，在运行ifconfig或netstat -i 时会发生这种情况。 “统计信息”部分介绍了snull的示例实现。

int (*set_config)(struct net_device *dev, struct ifmap *map);更改接口配置。此方法是配置驱动程序的入口点。可以在运行时使用set_config更改设备的 I/O 地址及其中断号。如果无法探测接口，系统管理员可以使用此功能。现代硬件的驱动程序通常不需要实现这种方法。

其余设备操作是可选的：

重量； int
(*poll)(struct net_device *dev; int *quota);由符合
NAPI 的驱动程序提供的方法，用于在轮询模式下操作接口，并禁用中断。 NAPI（和权重字段）在“接收中断缓解”部分中介绍。

void (*poll_controller)(struct net_device *dev);在中断
被禁用的情况下要求驱动程序检查接口上的事件的函数。它用于特定的内核内网络任务，例如远程控制台和通过网络进行内核调试。 int (*do_ioctl)(struct net_device *dev, struct ifreq *ifr, int cmd);

执行特定于接口的ioctl命令。 （这些命令的实现在“自定义 ioctl 命令”一节中进行了描述。）如果接口不需要任何特定于接口的命令，则struct net_device中的相应字段可以保留为NULL 。

void (*set_multicast_list)(struct net_device *dev);
当设备的多播列表更改和标志更改时调用的方法。有关更多详细信息和示例实现，请参阅“多播”部分。 int (*set_mac_address)(struct net_device *dev, void *addr);如果接口支持更改其硬件地址的能力，则可以实现的功能。许多接口根本不支持这种能力。其他人使用默认的eth_mac_addr实现（来自drivers/net/net_init.c）。 eth_mac_addr仅将新地址复制到dev->dev_addr 中，并且仅在接口未运行时才会这样做。使用eth_mac_addr的驱动程序应该在他们的open方法中从dev->dev_addr设置硬件 MAC 地址。 int (*change_mtu)(struct net_device *dev, int new_mtu);如果接口的最大传输单元 (MTU) 发生变化，该函数会采取行动。如果驱动程序在用户更改 MTU 时需要做任何特定的事情，它应该声明自己的函数；否则，默认值会做正确的事情。如果您有兴趣， snull有该函数的模板。

int (*header_cache) (结构邻居 *neigh, 结构 hh_cache *hh);
调用header_cache以使用 ARP 查询的结果填充hh_cache结构。几乎所有类似以太网的驱动程序都可以使用默认的eth_header_cache实现。

int (*header_cache_update) (struct hh_cache *hh, struct net_device *dev,
unsigned char *haddr);更新hh_cache结构中的目标地址以响应更改的方法。以太网
设备使用eth_header_cache_update。 int (*hard_header_parse) (struct sk_buff
*skb, unsigned char *haddr);

hard_header_parse方法从包含在skb中的数据包中提取源地址,并将其复制到haddr 的缓冲区中。函数的返回值是该地址的长度。以太网设备通常使用eth_header_parse。

实用程序字段接口使

用剩余的struct net_device数据字段来保存有用的状态信息。 ifconfig和netstat使用某些字段向用户提供有关当前配置的信息。因此,接口应该为这些字段赋值:

无符号长 trans_start;无符号 long
last_rx;包含 jiffies 值的字段。驱动
程序负责分别在传输开始和接收到数据包时更新这些值。网络子系统使用trans_start值来检测发射机锁定。 last_rx当前未使用,但驱动程序无论如何都应维护此字段以备将来使用。 int watchdog_timeo;
在网络层决定发生传输超时并调用驱动程序的tx_time out函数之前应该经过的最短时间(以 jiffies 为单位)。

无效*私人;相
当于filp->private_data。在现代驱动程序中,该字段由alloc_netdev设置,不应直接访问;改用
netdev_priv 。结构 dev_mc_list *mc_list;诠释 mc_count;处理多播传输的字段。 mc_count是
mc_list 中的项目数。

有关详细信息,请参阅“组播”部分。

spinlock_t xmit_lock;诠
释 xmit_lock_owner;
xmit_lock用于避免多次同时调用驱动程序的hard_start_xmit函数。 xmit_lock_owner是获得
xmit_lock 的 CPU 的编号。驱动程序不应更改这些字段。

struct net_device中还有其他字段,但它们不被网络驱动程序使用。

打开和关闭

我们的驱动程序可以在模块加载时或内核启动时探测接口。然而,在接口可以传送数据包之前,内核必须打开它并为其分配一个地址。内核响应ifconfig命令打开或关闭接口。

当ifconfig用于为接口分配地址时,它执行两个任务。
首先,它通过ioctl(SIOCSIFADDR) （Socket I/O Control Set Interface Address)分配地址。然后它通过ioctl(SIOCSIFFLAGS) (Socket I/O Control Set Interface Flags) 设置dev->flag中的IFF_UP位以打开接口。

就设备而言， ioctl(SIOCSIFADDR)什么也不做。不调用驱动程序函数　任务独立于设备,由内核执行。然而,后面的命令(ioctl(SIOCSIFFLAGS))调用设备的open方法。

同样,当接口关闭时， ifconfig使用ioctl(SIOCSIFFLAGS)清除IFF_UP,并调用stop方法。

两种设备方法在成功的情况下返回0 ,在出错的情况下返回通常的负值。

就实际代码而言,驱动程序必须执行许多与字符和块驱动程序相同的任务。 open请求它需要的任何系统资源并告诉接口出现； stop关闭接口并释放系统资源。然而,网络驱动程序必须在开放时执行一些额外的步骤。

首先,需要将硬件（MAC)地址从硬件设备复制到dev->dev_addr ,然后接口才能与外界通信。然后可以在打开时将硬件地址复制到设备。 snull软件接口从 open 内部分配它;它只是使用长度为ETH_ALEN的 ASCII 字符串（以太网硬件地址的长度)来伪造硬件编号。

一旦准备好开始发送数据,open 方法还应该启动接口的传输队列（允许它接受数据包进行传输） 。内核提供了一个函数来启动队列：

```
无效 netif_start_queue(struct net_device *dev);
```

snull的开放代码如下所示：

```
int snull_open(struct net_device *dev) {

    /* request_region( ), request_irq( ), .... (如 fops->open) */

    /*
     * 分配板子的硬件地址:使用“\0SNULx”,其中x为0或1。第一个字节为“\0”,避
       免成为多播
     * 地址（多播地址的第一个字节是奇数）。 */

    memcpy(dev->dev_addr,  \0SNUL0  ,
    ETH_ALEN); if (dev = = snull_devs[1])
```

```
        开发->dev_addr[ETH_ALEN-1]++； /* \0SNUL1 */
    netif_start_queue(dev);返回0；

}
```

如您所见,在没有真正的硬件的情况下， open方法几乎没有什么可做的。 stop方法也是如此;它只是颠倒了open 的操作。
出于这个原因,实现停止的函数通常被称为关闭或释放。

```
int snull_release(struct net_device *dev) {

    /* 释放端口、irq 等 -- 比如 fops->close */

    netif_stop_queue(dev); /* 不能再传输了 */ return 0;

}
```

功能：

```
无效 netif_stop_queue(struct net_device *dev);
```

与netif_start_queue 相反;它将设备标记为无法再传输任何数据包。该函数必须在接口关闭时调用（在stop方法中）,但也可用于暂时停止传输,如下一节所述。

包传输

网络接口执行的最重要任务是数据传输和接收。我们从传输开始,因为它更容易理解。

传输是指通过网络链接发送数据包的行为。每当内核需要传输数据包时,它就会调用驱动程序的hard_start_transmit方法将数据放入传出队列。内核处理的每个数据包都包含在一个套接字缓冲区结构（struct sk_buff)中,其定义可在<linux/skbuff.h> 中找到。该结构的名称来自用于表示网络连接的 Unix 抽象,即套接字。即使接口与套接字无关,每个网络数据包都属于更高网络层的一个套接字,并且任何套接字的输入/输出缓冲区都是struct sk_buff结构的列表。相同的sk_buff结构用于在所有 Linux 网络子系统中托管网络数据,但就接口而言,套接字缓冲区只是一个数据包。

指向sk_buff的指针通常称为skb,我们在示例代码和文本中都遵循这种做法。

套接字缓冲区是一个复杂的结构,内核提供了许多作用于它的函数。这些函数将在后面的“套接字缓冲区”部分中描述;目前,关于sk_buff的一些基本事实足以让我们编写一个工作驱动程序。

传递给hard_start_xmit的套接字缓冲区包含应该出现在媒体上的物理数据包,并带有传输级别的标头。接口不需要修改正在传输的数据。 skb->data指向正在传输的数据包,而skb->len是它的八位字节长度。如果您的驱动程序可以处理分散/收集 I/O,这种情况会变得更复杂一些;我们在“Scatter/Gather I/O”一节中讨论了这个问题。

snull包传输代码如下;物理传输机制已被隔离在另一个函数中,因为每个接口驱动程序都必须根据被驱动的特定硬件来实现它:

```
int snull_tx(struct sk_buff *skb, struct net_device *dev) {

    国际化;
    char *数据,shortpkt[ETH_ZLEN];结构
    snull_priv *priv = netdev_priv(dev);

    数据=skb->数据;
    len = skb->len; if
    (len < ETH_ZLEN)
        { memset(shortpkt, 0, ETH_ZLEN);
        memcpy(shortpkt, skb->data, skb->len); len
        = ETH_ZLEN;数据=短包;


    } dev->trans_start = jiffies; /* 保存时间戳 */

    /* 记住 skb,所以我们可以在中断时释放它 */ priv->skb = skb;


    /* 数据的实际传递是特定于设备的,这里没有显示 */ snull_hw_tx(data, len, dev);


    返回0;  /* 我们的简单设备不会失败 */
}
```

因此,传输函数只是对数据包执行一些完整性检查,并通过硬件相关函数传输数据。但是请注意,当要传输的数据包短于底层媒体支持的最小长度时要小心（对于snull,它是我们的虚拟“以太网”）。许多 Linux 网络驱动程序（以及其他操作系统的驱动程序)被发现在这种情况下会泄漏数据。我们没有创建那种安全漏洞,而是将短数据包复制到一个单独的数组中,我们可以显式地将其零填充到媒体所需的全长。 （我们可以安全地将数据放入堆栈,因为最小长度（60 字节)非常小）。

成功时hard_start_xmit的返回值应为0 ;此时,您的驱动程序已对数据包负责,应尽最大努力确保传输成功,并且必须在最后释放 skb。非零返回值表示此时无法传输数据包;内核将重试

之后。在这种情况下,您的驱动程序应该停止排队,直到导致故障的任何情况得到解决。

这里省略了“硬件相关”传输函数（snull_hw_tx）,因为它完全用于实现snull设备的诡计,包括操纵源地址和目标地址,并且对真实网络驱动程序的作者没有什么兴趣。当然,它存在于示例源中,供那些想要进去看看它是如何工作的人使用的。

控制传输并发hard_start_xmit函数通过net_device

结构中的自旋锁(xmit_lock)来防止并发调用。然而,一旦函数返回,它可能会再次被调用。当软件完成指示硬件进行数据包传输时,该函数返回,但硬件传输可能尚未完成。这不是snull 的问题,它使用 CPU 完成所有工作,因此数据包传输在传输函数返回之前完成。

另一方面,真正的硬件接口异步传输数据包,并且可用于存储传出数据包的内存量有限。当内存用尽时（对于某些硬件,这发生在要传输的单个未完成的数据包中）,驱动程序需要告诉网络系统不要再开始传输,直到硬件准备好接受新数据。

此通知是通过调用netif_stop_queue 来完成的,该函数是前面介绍的用于停止队列的函数。一旦您的驱动程序停止了它的队列,它必须安排在将来的某个时间重新启动队列,当它再次能够接受数据包进行传输时。为此,它应该调用：

```
无效 netif_wake_queue(struct net_device *dev);
```

这个函数和netif_start_queue 一样,只是它也会戳网络系统,让它重新开始传输数据包。

大多数现代网络硬件都维护一个内部队列,其中包含多个要传输的数据包;通过这种方式,它可以从网络中获得最佳性能。这些设备的网络驱动程序必须支持在任何给定时间有多个未完成的传输,但是无论硬件是否支持多个未完成的传输,设备内存都会填满。每当设备内存填充到没有空间容纳最大可能的数据包时,驱动程序应该停止队列,直到空间再次可用。

如果您必须从hard_start_xmit函数以外的任何地方禁用数据包传输（也许是为了响应重新配置请求）,您要使用的函数是：

```
无效 netif_tx_disable(struct net_device *dev);
```

这个函数的行为很像netif_stop_queue,但它也确保当它返回时,你的hard_start_xmit方法不在另一个CPU 上运行。可以像往常一样使用netif_wake_queue重新启动队列。

传输超时

大多数处理真实硬件的驱动程序必须为该硬件偶尔无法响应做好准备。接口可能会忘记它们在做什么,或者系统可能会丢失中断。这种问题在一些设计用于在个人计算机上运行的设备中很常见。

许多司机通过设置定时器来处理这个问题。如果在计时器到期时操作尚未完成,则说明有问题。事实上,网络系统本质上是由大量计时器控制的状态机的复杂组合。因此,网络代码可以很好地检测传输超时,作为其常规操作的一部分。

因此,网络驱动程序不必担心自己检测此类问题。
相反,他们只需要设置一个超时期限,该期限在net_device结构的watchdog_timeo字段中。这段时间很短,应该足够长,以解决正常的传输延迟（例如由网络媒体拥塞引起的冲突）。

如果当前系统时间超过设备的trans_start时间至少超时时间,网络层最终会调用驱动程序的tx_timeout方法。
该方法的工作是做任何必要的事情来解决问题并确保正确完成任何已经在进行中的传输。特别重要的是,驱动程序不要丢失网络代码委托给它的任何套接字缓冲区的跟踪。 snull能够模拟发射机锁定,这由两个加载时间参数控制:

```
静态int锁定= 0;模块参数
（锁定,int,0）；

静态 int 超时 = SNULL_TIMEOUT;模块参
数（超时,int,0）；
```

如果驱动程序加载了参数lockup=n,则每传输n个数据包模拟一次锁定,并且watchdog_timeo字段设置为给定的超时值。
在模拟锁定时， snull还会调用netif_stop_queue以防止发生其他传输尝试。

snull传输超时处理程序如下所示:

```
无效 snull_tx_timeout (struct net_device *dev) {

    结构 snull_priv *priv = netdev_priv(dev);
```

```
        PDEBUG(  传输超时 %ld, 延迟 %ld\n  , jiffies, jiffies - dev-
                >trans_start);
            /* 模拟传输中断以使事物移动 */
        priv->状态 = SNULL_TX_INTR;
        snull_interrupt(0, dev, NULL);
        priv->stats.tx_errors++;
        netif_wake_queue(dev);返回;

    }
```

当发生传输超时时,驱动程序必须在接口统计信息中标记错误并安排设备重置为正常状态,以便可以传输新的数据包。当snull 发生超时时,驱动程序调用snull_interrupt来填充“丢失”的中断,并使用netif_wake_queue 重新启动传输队列。

分散/收集 I/O

创建用于在网络上传输的数据包的过程涉及组装多个部分。数据包数据通常必须从用户空间复制进来,并且还必须添加不同级别的网络堆栈使用的标头。该程序集可能需要大量的数据复制。但是,如果注定要传输数据包的网络接口可以执行分散/聚集 I/O,则无需将数据包组装成单个块,并且可以避免大部分复制。 Scatter/gather I/O 还支持直接从用户空间缓冲区“零复制”传输网络数据。

除非在设备结构的features字段中设置了NETIF_F_SG位,否则内核不会将分散的数据包传递给您的hard_start_xmit方法。如果您设置了该标志,则需要查看 skb 中的特殊“共享信息”字段,以查看数据包是由单个片段还是由多个片段组成,并在需要时找到分散的片段。存在一个特殊的宏来访问此信息;它被称为skb_shinfo。传输可能分片的数据包的第一步通常如下所示:

```
    如果（skb_shinfo（skb）->nr_frags = = 0){
        /* 像往常一样使用 skb->data 和 skb->len */
    }
```

nr_frags字段告诉我们使用了多少个片段来构建数据包。如果为0,则数据包存在于单片中,并且可以像往常一样通过数据字段访问。
但是,如果它不为零,则您的驱动程序必须通过并安排传输每个单独的片段。 skb 结构的数据字段方便地指向第一个片段（与完整的数据包相比,如在未分片的情况下）。片段的长度必须通过从skb->len中减去skb->data_len来计算（仍然包含完整数据包的长度）。剩余的片段将在共享信息结构中称为frags的数组中找到； frags中的每个条目都是一个skb_frag_struct结构：

```
    结构 skb_frag_struct { 结构
        页 *页;
```

```
    __u16 page_offset;
    __u16 大小;
};
```

如您所见,我们再次处理的是页面结构,而不是内核虚拟地址。您的驱动程序应该遍历片段,映射每个片段以进行 DMA 传输,并且不要忘记 skb 直接指向的第一个片段。当然,您的硬件必须组装这些片段并将它们作为单个数据包传输。请注意,如果您设置了NETIF_F_HIGHDMA功能标志,则部分或全部片段可能位于高内存中。

数据包接收

从网络接收数据比传输数据更棘手,因为sk_buff必须在原子上下文中分配并传递给上层。

网络驱动程序可以实现两种数据包接收模式:中断驱动和轮询。大多数驱动程序都实现了中断驱动技术,这是我们首先介绍的技术。一些高带宽适配器的驱动程序也可以实现轮询技术;我们在“接收中断缓解”一节中介绍了这种方法。

snull的实现将“硬件”细节与设备无关的内务管理分开。因此,函数snull_rx在硬件接收到数据包后从snull “中断”处理程序中调用,并且它已经在计算机的内存中。 snull_rx接收指向数据的指针和数据包的长度;它的唯一职责是将数据包和一些附加信息发送到网络代码的上层。此代码与获取数据指针和长度的方式无关。

```
无效 snull_rx（结构 net_device *dev,结构 snull_packet *pkt){

    结构 sk_buff *skb;结构
    snull_priv *priv = netdev_priv(dev);

    /*
     * 数据包已从传输介质中检索到。围绕它构建一个 skb,以便上层可以
     处理它 */

    skb = dev_alloc_skb(pt->datalen + 2); if (!
    skb) { if (printk_ratelimit())

            printk(KERN_NOTICE    snull rx: 内存不足 - 丢包\n    ); priv-
        >stats.rx_dropped++;出去;


    } memcpy(skb_put(skb, pkt->datas), pkt->data, pkt->datas);

    /* 写入元数据,然后传递到接收层 */ skb->dev = dev;
```

```
        skb->protocol = eth_type_trans(skb, dev); skb-
        >ip_summed = CHECKSUM_UNNECESSARY； /* 不检查 */ priv-
        >stats.rx_packets++; priv->stats.rx_bytes += pkt->datalen； netif_rx(skb);


    出去：
        返回;
}
```

该函数足够通用,可以作为任何网络驱动程序的模板,但是在您可以放心地重用此代码片段之前,需要进行一些解释。

第一步是分配一个缓冲区来保存数据包。注意缓冲区分配函数（dev_alloc_skb)需要知道数据长度。函数使用该信息为缓冲区分配空间。 dev_alloc_skb以原子优先级调用kmalloc ,因此可以在中断时安全使用。内核提供了其他接口来分配套接字缓冲区,但这里不值得介绍;套接字缓冲区在“套接字缓冲区”部分中有详细说明。

当然,必须检查 dev_alloc_skb 的返回值,而snull就是这样做的。
然而,我们在抱怨失败之前调用printk_ratelimit 。每秒生成成百上千条控制台消息是完全阻塞系统并隐藏问题真正根源的好方法； printk_ratelimit通过在控制台输出过多时返回0来帮助防止该问题,并且需要放慢速度。

一旦有一个有效的skb指针,就通过调用memcpy将数据包数据复制到缓冲区中； skb_put函数更新缓冲区中的数据结束指针并返回一个指向新创建空间的指针。

如果您正在为可以进行完全总线主控 I/O 的接口编写高性能驱动程序,那么这里有一个值得考虑的可能优化。一些驱动程序在接收到传入的数据包之前为其分配套接字缓冲区,然后指示接口将数据包数据直接放入套接字缓冲区的空间。网络工作层通过在支持 DMA 的空间中分配所有套接字缓冲区来配合此策略（如果您的设备设置了NETIF_F_HIGHDMA功能标志,则可能在高内存中）。这样做可以避免需要单独的复制操作来填充套接字缓冲区,但需要注意缓冲区大小,因为您不会事先知道传入的数据包有多大。在这种情况下， change_mtu方法的实现也很重要,因为它允许驱动程序响应最大数据包大小的变化。

网络层需要先说明一些信息,然后才能理解数据包。为此,必须在缓冲区通过楼上之前分配dev和protocol字段。以太网支持代码导出一个帮助函数(eth_type_trans),它会找到一个合适的值来放入协议中。然后我们需要指定校验和将如何执行或已经在

数据包（snull不需要执行任何校验和）。 skb->ip_summed的可能策略是：

CHECKSUM_HW设备已经在硬件中执行了校验和。硬件校验和的一个示例是 SPARC HME 接口。

CHECKSUM_NONE校验和尚未验证，任务必须由系统软件完成。这是新分配的缓冲区中的默认值。

CHECKSUM_UNNECESSARY不要做任何校验和。这是snull和 loopback 接口中的策略。

您可能想知道为什么我们已经在net_device结构的features字段中设置了一个标志，为什么还要在这里指定校验和状态。答案是features标志告诉内核我们的设备如何处理传出的数据包。

它不用于传入的数据包，而是必须单独标记。

最后，驱动程序更新其统计计数器以记录已接收到数据包。统计结构由几个字段组成；最重要的是rx_packets、 rx_bytes、 tx_packets和tx_bytes，它们包含接收和发送的数据包数量以及传输的八位字节总数。所有字段都在“统计信息”部分进行了详尽的描述。

数据包接收的最后一步由netif_rx 执行，它将套接字缓冲区移交给上层。 netif_rx实际上返回一个整数值； NET_RX_SUCCESS (0)表示成功接收到数据包；任何其他值都表示有问题。

有三个返回值（NET_RX_CN_LOW、 NET_RX_CN_MOD和NET_RX_CN_HIGH）表示网络子系统中的拥塞程度不断增加； NET_RX_DROP表示数据包被丢弃。当拥塞变高时，驱动程序可以使用这些值来停止向内核提供数据包，但实际上，大多数驱动程序会忽略netif_rx 的返回值。如果您正在为高带宽设备编写驱动程序并希望做正确的事情来响应拥塞，那么最好的方法是实现NAPI，我们在快速讨论中断处理程序后得到它。

中断处理程序

大多数硬件接口都是通过中断处理程序控制的。硬件中断处理器以发出两种可能事件之一的信号：新数据包已到达或传出数据包的传输完成。网络接口还可以生成中断以发出错误信号、链接状态更改等。

通常的中断例程可以通过检查物理设备上的状态寄存器来区分新数据包到达中断和完成传输通知之间的区别。 snull接口的工作方式类似，但实现了它的状态字

在软件中并居住在dev->priv 中。网络接口的中断处理程序如下所示：

```
静态无效 snull_regular_interrupt(int irq, void *dev_id, struct pt_regs *regs) {

    诠释状态字;结构
    snull_priv *priv;结构
    snull_packet *pkt = NULL; /* * 像往常
    一样,检查“设备”指针以确保它是真正的
      中断。

      * 然后分配“struct device *dev” */

    结构 net_device *dev = (struct net_device *)dev_id; /* ... 并检查
    硬件是否真的是我们的 */

    /* 偏执 */ if (!dev)
    return;


    /* 锁定设备 */ priv =
    netdev_priv(dev);
    spin_lock(&priv->lock);

    /* 检索状态字:真实网络设备使用 I/O 指令 */ statusword = priv->status;隐私->状
    态 = 0； if (statusword & SNULL_RX_INTR) { /* 发送到 snull_rx 处理 */ pkt =
    priv->rx_queue; if (pkt) { priv->rx_queue = pkt->next; snull_rx（开发,pkt）；




          }

    } if (statusword & SNULL_TX_INTR) { /*
          传输结束:释放 skb */ priv->stats.tx_packets++; priv-
          >stats.tx_bytes += priv->tx_packetlen；
          dev_kfree_skb(priv->skb);

    }

    /* 解锁设备,我们就完成了 */ spin_unlock(&priv-
    >lock);如果（pkt)snull_release_buffer（pkt）； /
    * 在锁外执行此操作！ */ 返回;

}
```

处理程序的第一个任务是检索指向正确结构 net_device 的指针。该指针通常来自作为参数接收的dev_id指针。

这个处理程序的有趣部分是处理“传输完成”的情况。在这种情况下,更新统计信息,并调用dev_kfree_skb返回（没有

不再需要)到系统的套接字缓冲区。实际上,可以调用此函数的三个变体： dev_kfree_skb(struct sk_buff *skb);

当您知道您的代码不会在中断上下文中运行时,应该调用此版本。由于snull没有实际的硬件中断,因此这是我们使用的版本。

dev_kfree_skb_irq(struct sk_buff *skb);
如果您知道您将在中断处理程序中释放缓冲区,请使用此版本,该版本已针对这种情况进行了优化。

dev_kfree_skb_any(struct sk_buff *skb);
如果相关代码可以在中断或非中断上下文中运行,这是要使用的版本。

最后,如果您的驱动程序暂时停止了传输队列,这通常是使用netif_wake_queue 重新启动它的地方。

与发送相比,数据包接收不需要任何特殊的中断处理。只需调用snull_rx （我们已经看到)即可。

接收中断缓解

当我们如上所述编写网络驱动程序时,处理器会因接口接收到的每个数据包而中断。在许多情况下,这是所需的操作模式,这不是问题。然而,高带宽接口每秒可以接收数千个数据包。在这种中断负载下,系统的整体性能可能会受到影响。

作为提高 Linux 在高端系统上的性能的一种方式,网络子系统开发人员创建了一个基于轮询的替代接口（称为 NAPI)*。 “轮询”在驱动程序开发人员中可能是一个肮脏的词,他们经常认为轮询技术不优雅且效率低下。然而,轮询是低效的,只有在没有工作可做的情况下轮询接口。当系统具有处理大量流量的高速接口时,总会有更多的数据包需要处理。在这种情况下无需中断处理器;每隔一段时间从接口收集新数据包就足够了。

停止接收中断可以减轻处理器的大量负载。
如果这些数据包由于拥塞而被丢弃在网络代码中,也可以告诉符合 NAPI 的驱动程序不要将数据包输入内核,这也可以在最需要帮助时提高性能。由于各种原因,NAPI 驱动程序也不太可能重新排序数据包。

* NAPI 代表“新 API”;网络黑客更擅长创建接口而不是命名它们。

但是,并非所有设备都可以在 NAPI 模式下运行。支持 NAPI 的接口必须能够存储多个数据包（在卡本身上或在内存中的 DMA 环中）。该接口应该能够为接收到的数据包禁用中断,同时继续为成功的传输和其他事件中断。

还有其他一些微妙的问题会使编写符合 NAPI 的驱动程序更加困难;有关详细信息,请参阅内核源代码树中的Documentation/networking/NAPI_HOWTO.txt 。

相对较少的驱动程序实现了 NAPI 接口。但是,如果您正在为可能产生大量中断的接口编写驱动程序,那么花时间来实现 NAPI 可能是值得的。

当使用设置为非零值的use_napi参数加载时， snull驱动程序在 NAPI 模式下运行。在初始化时，我们必须设置几个额外的struct net_device字段： if (use_napi) { dev->poll dev->weight

```
                = snull_poll; =
                2;
}
```

poll字段必须设置为您的驱动程序的轮询功能;我们很快就会看到snull_poll 。权重字段描述了接口的相对重要性:当资源紧张时,应该从接口接受多少流量。对于如何设置权重参数没有严格的规定;按照惯例,10 MBps 以太网接口将权重设置为16,而更快的接口使用64。您不应将权重设置为大于接口可以存储的数据包数量的值。在snull 中,我们将权重设置为 2,作为演示延迟数据包接收的一种方式。

创建 NAPI 兼容驱动程序的下一步是更改中断处理程序。当您的接口（应从启用接收中断开始)发出数据包已到达的信号时,中断处理程序不应处理该数据包。相反,它应该禁用进一步的接收中断并告诉内核是时候开始轮询接口了。在snull “中断”处理程序中,响应数据包接收中断的代码已更改为以下内容：

```
if (statusword & SNULL_RX_INTR)
    { snull_rx_ints(dev, 0); /* 禁用更多中断 */ netif_rx_schedule(dev);

}
```

当接口告诉我们一个数据包可用时,中断处理程序将它留在接口中;此时需要做的就是调用netif_rx_schedule,这会导致我们的poll方法在未来某个时间点被调用。

poll方法有这个原型：

```
int (*poll)(struct net_device *dev, int *budget);
```

poll方法的snull实现如下所示：

```
静态 int snull_poll(struct net_device *dev, int *budget) {

    int npackets = 0, 配额 = min(dev->quota, *budget);结构
    sk_buff *skb;结构 snull_priv *priv = netdev_priv(dev);结构
    snull_packet *pkt;


    while (npackets < quota && priv->rx_queue) { pkt =
        snull_dequeue_buf(dev); skb =
        dev_alloc_skb(pkt->datalen + 2); if (!skb) { if
        (printk_ratelimit( )) printk(KERN_NOTICE
                snull: packet dropped\n    );

            priv->stats.rx_dropped++;
            snull_release_buffer(pkt);继
            续;

        } memcpy(skb_put(skb, pkt->datalen), pkt->data, pkt->datalen); skb-
        >开发=开发； skb->protocol = eth_type_trans(skb, dev); skb-
        >ip_summed = CHECKSUM_UNNECESSARY； /* 不检查 */
        netif_receive_skb(skb);


            /* 维护统计信息 */
        npackets+
        +； priv->stats.rx_packets+
        +； priv->stats.rx_bytes += pkt->datalen;
        snull_release_buffer(pkt);
    }
    /* 如果我们处理了所有的数据包,我们就完成了;告诉内核并重新启用整数 */ *budget -= npackets;开
    发->配额-= npackets； if (!priv->rx_queue) { netif_rx_complete(dev); snull_rx_ints（开发,1）；
    返回0;




    }
    /* 我们无法处理所有内容。 */ 返回 1;

}
```

该函数的核心部分是创建一个保存数据包的 skb;这段代码和我们之前在snull_rx中看到的一样。然而,许多事情是不同的：

- 预算参数提供了我们被允许传递到内核的最大数据包数。在设备结构中,配额字段给出了另一个最大值； poll方法必须遵守两个限制中的较低者。它还应该将dev->quota和*budget都减少实际收到的数据包数。预算值是当前CPU可以从所有接口接收的最大数据包数,而配额是每个接口的值

这通常从初始化时分配给接口的权重开始。

· 应使用netif_receive_skb而非netif_rx 将数据包馈送到内核。 · 如果poll方法能够在给定的限制范围内处理所有可用的数据包,它应该重新启用接收中断,调用netif_rx_complete关闭轮询,并返回0。返回值1表示有数据包剩余待处理。

网络子系统保证不会在多个处理器上同时调用任何给定设备的poll方法。但是,对poll的调用仍然可以与对其他设备方法的调用同时发生。

链接状态的变化

根据定义,网络连接与本地系统之外的世界打交道。
因此,它们经常受到外部事件的影响,并且可能是暂时的。
网络子系统需要知道网络链接何时启动或关闭,并且它提供了一些驱动程序可以用来传达该信息的功能。

大多数涉及实际物理连接的网络技术都提供了载体状态。载体的存在意味着硬件存在并且准备好运行。例如,以太网适配器可以感知线路上的载波信号;当用户绊倒电缆时,该载体消失,链路断开。默认情况下,假定网络设备存在载波信号。但是,驱动程序可以使用以下函数显式更改该状态:

```
无效 netif_carrier_off（结构 net_device *dev）;无效
netif_carrier_on(struct net_device *dev);
```

如果您的驱动程序检测到其中一个设备上缺少运营商,它应该调用netif_carrier_off以通知内核此更改。当运营商返回时,应该调用netif_carrier_on 。一些驱动程序在进行重大配置更改（例如媒体类型)时也会调用netif_carrier_off ;一旦适配器完成自我重置,就会检测到新的汽车载体,并且可以恢复流量。

还存在一个整数函数:

```
int netif_carrier_ok(struct net_device *dev);
```

这可用于测试当前的载流子状态（反映在器件结构中）。

套接字缓冲区

我们现在已经讨论了与网络接口相关的大部分问题。仍然缺少的是对sk_buff结构的一些更详细的讨论。结构是Linux内核网络子系统的核心,现在我们介绍结构的主要领域和作用于它的函数。

虽然没有严格要求了解sk_buff的内部结构,但是当您跟踪问题和尝试优化代码时,查看其内容的能力会很有帮助。例如,如果您查看loopback.c,您会发现基于sk_buff内部知识的优化。通常的警告在这里适用:如果您编写的代码利用了sk_buff结构的知识,您应该准备好看到它在未来的内核版本中中断。尽管如此,有时性能优势证明了额外的维护成本是合理的。

我们不会在这里描述整个结构,只是可能在驱动程序中使用的字段。如果您想了解更多,可以查看<linux/skbuff.h>,其中定义了结构和原型化功能。可以通过在内核源中使用 grepping 轻松检索有关如何使用字段和函数的其他详细信息。

重要字段这里介绍的字段是驱

动程序可能需要访问的字段。它们没有按特定顺序列出。

结构 net_device *dev;接
收或发送此缓冲区的设备。
联合 { /* ... */ } h;联合 { /
* ... */ } nh;联合 { /*... */}
mac;指向数据包中包含的
各个级别的标头的指针。联合的每个字段都是指向不同类型数据结构的指针。 h承载指向传输层标头的指针（例如， struct tcphdr *th）； nh包括网络层标头（例如struct iphdr *iph）； mac收集指向链路层标头的指针（例如struct ethdr *ethernet）。

如果您的驱动程序需要查看 TCP 数据包的源地址和目标地址,它可以在skb->h.th 中找到它们。有关可以通过这种方式访问的完整头类型集,请参见头文件。

请注意,网络驱动程序负责设置传入数据包的mac指针。此任务通常由eth_type_trans 处理,但非以太网驱动程序必须直接设置skb->mac.raw ,如“非以太网标头”部分所示。

无符号字符*头;无符号
字符*数据;无符号字符
*tail;无符号字符*结束;
用于寻址数据包中数据
的指针。 head指向分配空间的开头， data是有效八位字节的开头（通常略大于head），tail是有效八位字节的结尾， end指向

tail可以达到的最大地址。另一种看待它的方式是,可用的缓冲区空间是skb->end - skb->head,当前使用的数据空间是skb->tail - skb->data。

无符号整数;无符号
整数数据长度； len是数
据包中数据的全长,而data_len是存储在单独片段中的数据包部分的长度。除非正在使用分散/收集 I/O,否则data_len字段为0 。

无符号字符 ip_summed;此
数据包的校验和策略。该字段由驱动程序在传入数据包上设置,如“数据包接收”部分所述。无符号字符 pkt_type;交付时使用的数据包分类。驱动程序负责将其设置为PACKET_HOST （这个包是给我的）、 PACKET_OTHERHOST （不,这个包不是给我的）、 PACKET_BROADCAST或PACKET_MULTICAST。以太网驱动程序不会显式修改pkt_ 类型,因为eth_type_trans会为它们执行此操作。

shinfo(struct sk_buff *skb);无
符号整数 shinfo(skb)->nr_frags;
skb_frag_t shinfo(skb)->frags;出于性能
原因,一些 skb 信息存储在一个单独的结构中,该结构紧跟在内存中的 skb 之后。必须通过shinfo宏访问这个“共享信息”（之所以这么称呼是因为它可以在网络代码中的 skb 副本之间共享）。这个结构中有几个领域,但大部分都超出了本书的范围。我们在“Scatter/Gather I/O”部分看到了nr_frags和frags 。

结构中的其余字段并不是特别有趣。它们用于维护缓冲区列表,说明属于拥有缓冲区的套接字的内存,等等。

作用于套接字缓冲区的函数使用sk_buff结构的网

络设备通过官方接口函数作用于它。许多函数对套接字缓冲区进行操作;以下是最有趣的:

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函数分配一个缓冲区并将skb- >data和skb->tail初始化为skb->head。dev_alloc_skb函数是一个快捷方式,它以GFP_ATOMIC优先级调用alloc_skb并在skb->head和skb->data之间保留一些空间。该数据空间用于网络层内的优化,不应被驱动程序触及。

无效 kfree_skb（结构 sk_buff *skb）;无
效 dev_kfree_skb（结构 sk_buff *skb）；
void dev_kfree_skb_irq(struct sk_buff *skb);
void dev_kfree_skb_any(struct sk_buff *skb);释
放缓冲区。 kfree_skb调用由内核内部使用。驱动程序应该使用 dev_kfree_skb 的一种形式： dev_kfree_skb用于非中断上下文， dev_kfree_skb_irq用于中断上下文,或
dev_kfree_skb_any用于可以在任一上下文中运行的代码。

无符号字符 *skb_put(struct sk_buff *skb, int len);无符号字
符 *__skb_put(struct sk_buff *skb, int len);更新sk_buff结构
的tail和len字段;它们用于将数据添加到缓冲区的末尾。每个函数的返回值都是skb->tail的前一个值（也就是说,它指向刚刚创建的数据空间）。驱动程序可以通过调用
memcpy(skb_put(...), data, len)或等效函数来使用返回值来复制数据。这两个函数之间的区别在于skb_put检查以确保数据适合缓冲区,而__skb_put省略了检查。

无符号字符 *skb_push(struct sk_buff *skb, int len);无符号字
符 *__skb_push(struct sk_buff *skb, int len);递减skb->data和
递增skb->len 的函数。它们与skb_put 类似,只是将数据添加到数据包的开头而不是结尾。返回值指向刚刚创建的数据空间。这些函数用于在传输数据包之前添加硬件标头。同样，
__skb_push的不同之处在于它不检查是否有足够的可用空间。

int skb_tailroom(struct sk_buff *skb);
返回可用于将数据放入缓冲区的空间量。如果驱动程序将更多的数据放入缓冲区,超出了它的容量,系统就会出现混乱。尽管您可能会反对printk足以标记错误,但内存损坏对系统非常有害,因此开发人员决定采取明确的措施。实际上,如果缓冲区已正确分配,则不需要检查可用空间。由于驱动程序通常在分配缓冲区之前获取数据包大小,因此只有严重损坏的驱动程序才会将过多数据放入缓冲区,并且恐慌可能被视为应有的惩罚。

int skb_headroom(struct sk_buff *skb);
返回数据前面的可用空间量,即一个可以“推送”到缓冲区的字节数。 void
skb_reserve(struct sk_buff *skb, int len);增加数据和尾部。该功能可用于在填充缓冲区
之前保留空间。大多数以太网接口在数据包前面保留两个字节;因此,IP 报头在 14 字节以太网报头之后在 16 字节边界上对齐。 snull也这样做了,尽管在“数据包接收”中没有显示该指令，
以避免在这一点引入额外的概念。

无符号字符 *skb_pull(struct sk_buff *skb, int len);从数据包
的头部删除数据。驱动程序不需要使用此功能,但为了完整起见,将其包含在此处。它递减skb->len并递增skb->data;这就是从传入数据包的开头剥离硬件标头（以太网或等效)的方式。

int skb_is_nonlinear(struct sk_buff *skb);如果
此 skb 被分成多个片段以用于分散/聚集 I/O,则返回一个真值。 int skb_headlen(struct sk_buff *skb);

返回 skb 的第一段的长度（由skb-> data 指向的部分）。

无效 *kmap_skb_frag(skb_frag_t *frag);无
效 kunmap_skb_frag（无效 *vaddr）；
如果您必须从内核中直接访问非线性 skb 中的片段,这些函数会为您映射和取消映射它们。使用原子 kmap,因此一次不能映射多个片段。

内核定义了其他几个作用于套接字缓冲区的函数,但它们旨在用于更高层的网络代码,驱动程序不需要它们。

MAC 地址解析

以太网通信的一个有趣问题是如何将 MAC 地址（接口的唯一硬件 ID)与 IP 号相关联。大多数协议都有类似的问题,但我们在这里专注于类似以太网的情况。我们试图提供对该问题的完整描述,因此我们展示了三种情况:ARP、没有 ARP 的以太网标头（例如plip)和非以太网标头。

在以太网中使用 ARP处理地址解析

的常用方法是使用地址解析协议 (ARP)。幸运的是,ARP 是由内核管理的,以太网接口不需要做任何特殊的事情来支持 ARP。只要在打开时正确分配了dev->addr和dev->addr_len ,驱动程序就无需担心将IP号解析为MAC地址； ether_setup将正确的设备方法分配给dev->hard_header和dev->rebuild_header。

尽管内核通常处理地址解析的细节（和结果的缓存）,但它会调用接口驱动程序来帮助构建数据包。
毕竟,驱动程序知道物理层标头的细节,而网络代码的作者试图将内核的其余部分与这些知识隔离开来。为此,内核调用驱动程序的hard_header方法来布局

带有 ARP 查询结果的数据包。通常,以太网驱动程序编写者不需要知道这个过程 通用以太网代码会处理所有事情。

覆盖 ARP简单的点对

点网络接口,例如plip,可能会受益于使用以太网报头,同时避免来回发送 ARP 数据包的开销。snull中的示例代码也属于此类网络设备。 snull不能使用 ARP,因为驱动程序会更改正在传输的数据包中的 IP 地址,并且 ARP 数据包也会交换 IP 地址。虽然我们可以轻松地实现一个简单的 ARP 回复生成器,但展示如何直接处理物理层标头更能说明问题。

如果您的设备想要使用通常的硬件标头而不运行 ARP,则需要覆盖默认的dev->hard_header方法。这就是snull实现它的方式,作为一个非常短的函数:

```
int snull_header(struct sk_buff *skb, struct net_device *dev, unsigned
                 short 类型, void *daddr, void *saddr, unsigned int
                 len)
{
    结构 ethhdr *eth = (结构 ethhdr *)skb_push(skb,ETH_HLEN);

    eth->h_proto = htons(type);
    memcpy（eth->h_source,saddr?saddr:dev->dev_addr,dev->addr_len）;
    memcpy(eth->h_dest, daddr ? daddr : dev->dev_addr, dev->addr_len); eth-
    >h_dest[ETH_ALEN-1] ^= 0x01; /* dest 是我们 xor 1 */ return (dev-
    >hard_header_len);
}
```

该函数仅获取内核提供的信息并将其格式化为标准以太网标头。由于稍后描述的原因,它还在目标以太网地址中切换了位。

当接口接收到数据包时， eth_type_trans以多种方式使用硬件头。我们已经在snull_rx 中看到了这个调用:

```
skb->protocol = eth_type_trans(skb,
```

dev);该函数从以太网报头中提取协议标识符（在本例中为 ETH_P_IP）;它还分配skb->mac.raw,从数据包数据中删除硬件头（使用skb_pull）,并设置skb->pkt_type。这最后一项在skb分配时默认为PACKET_HOST （这表明数据包被定向到此主机）,并且eth_type_trans更改它以反映以太网目标地址:如果该地址与接收它的接口的地址不匹配,则pkt_type字段设置为PACKET_OTHERHOST。

随后,除非接口处于混杂模式或内核中启用了数据包转发,否则netif_rx会丢弃任何PACKET_OTHERHOST 类型的数据包。出于这个原因， snull_header会小心地使目标硬件地址与“接收”接口的地址相匹配。

如果您的接口是点对点链接,您将不希望收到意外的多播数据包。为避免此问题,请记住第一个八位字节的最低有效位 (LSB) 为0的目标地址被定向到单个主机（即,它是PACKET_HOST或PACKET_OTHERHOST）。 plip驱动程序使用0xfc作为其硬件地址的第一个八位字节,而snull使用0x00 。这两个地址都产生了一个像点对点链路一样工作的以太网。

非以太网标头

我们刚刚看到,硬件头中除了目的地址之外还包含一些信息,其中最重要的是通信协议。我们现在描述如何使用硬件头来封装相关信息。

如果您需要了解详细信息,可以从内核源代码或特定传输介质的技术文档中提取它们。大多数驱动程序编写者可以忽略这个讨论,而只使用以太网实现。

值得注意的是,并非每个协议都必须提供所有信息。诸如plip或snull 之类的点对点链路可以避免传输整个以太网报头而不失一般性。 hard_header设备方法,如前面所示,由 snull_header 实现,从内核接收传递信息　协议级别和硬件地址。它还在类型参数中接收 16 位协议号;例如,IP 由 ETH_P_IP 标识。驱动程序应正确地将数据包数据和协议号传送到接收主机。点对点链路可以从其硬件报头中省略地址,只传输协议号,因为保证交付与源地址和目标地址无关。仅 IP 链接甚至可以避免传输任何硬件标头。

当在链路的另一端拾取数据包时,驱动程序中的接收函数应正确设置字段skb->protocol、 skb->pkt_type和skb->mac.raw。 skb->mac.raw是一个字符指针,由在更高层网络代码（例如,net/ipv4/arp.c)中实现的地址解析机制使用。它必须指向与dev->type 匹配的机器地址。设备类型的可能值在<linux/if_arp.h> 中定义;以太网接口使用ARPHRD_ETHER。例如,这里是eth_type_trans如何处理接收数据包的以太网标头：

```
skb->mac.raw = skb->数
据； skb_pull(skb, dev-
```

>hard_header_len);在最简单的情况下（没有标头的点对点链接）， skb->mac.raw可以指向一个包含该接口硬件地址的静态缓冲区,协议可以设置为ETH_P_IP， packet_type可以保留其默认值为PACKET_HOST。

因为每种硬件类型都是独一无二的,所以很难给出比已经讨论过的更具体的建议。然而,内核充满了例子。例如,参见

AppleTalk 驱动程序(drivers/net/appletalk/cops.c)、红外驱动程序 (例如drivers/net/irda/smc_ircc.c)或 PPP 驱动程序(drivers/net/ppp_generic.c)。

自定义 ioctl 命令

我们已经看到ioctl系统调用是针对套接字实现的； SIOCSIFADDR和SIOCSIFMAP是“socket ioctl”的例子。现在让我们看看网络代码如何使用系统调用的第三个参数。

当在socket上调用ioctl系统调用时,命令号是<linux/sockios.h>中定义的符号之一， sock_ioctl函数直接调用了一个特定于协议的函数（其中“protocol”指的是主网使用的协议,例如 IP 或 AppleTalk）。

协议层无法识别的任何ioctl命令都会传递给设备层。这些与设备相关的ioctl命令接受来自用户空间的第三个参数,即struct ifreq*。这个结构在<linux/if.h> 中定义。 SIOCSIFADDR和SIOCSIFMAP命令实际上适用于ifreq结构。 SIOCSIFMAP的额外参数虽然定义为ifmap,但只是ifreq 的一个字段。

除了使用标准化调用外,每个接口还可以定义自己的ioctl命令。例如， plip接口允许接口通过ioctl修改其内部超时值。用于套接字的ioctl实现将 16 个命令识别为接口的私有命令：SIOCDEVPRIVATE到SIOCDEVPRIVATE+15。*当这些命令之一被识别时,在相关接口驱动程序中调用dev->do_ioctl 。该函数接收通用ioctl函数使用的相同struct ifreq*指针：

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

ifr指针指向一个内核空间地址,该地址保存用户传递的结构的副本。 do_ioctl返回后,将该结构体复制回用户空间;因此,驱动程序可以使用私有命令来接收和返回数据。

特定于设备的命令可以选择使用struct ifreq 中的字段,但它们已经传达了标准化的含义,驱动程序不太可能根据需要调整结构。 ifr_data字段是一个caddr_t项（指针）,旨在用于特定于设备的需求。用于调用其ioctl命令的驱动程序和程序应该就ifr_data 的使用达成一致。例如， ppp stats使用特定于设备的命令从ppp接口驱动程序中检索信息。

* 请注意,根据<linux/sockios.h>,不推荐使用SIOCDEVPRIVATE命令。然而,应该用什么替代它们尚不清楚,许多树内驱动程序仍在使用它们。

此处不值得展示do_ioctl的实现,但借助本章中的信息和内核示例,您应该可以在需要时编写一个。但是请注意， plip实现错误地使用了 ifr_data ,不应将其用作ioctl实现的示例。

统计资料

驱动程序需要的最后一个方法是get_stats。此方法返回指向设备统计信息的指针。它的实现非常简单;即使多个接口由同一个驱动程序管理,所显示的也可以工作,因为统计信息托管在设备数据结构中。

```
struct net_device_stats *snull_stats(struct net_device *dev) {

    结构 snull_priv *priv = netdev_priv(dev);返回
    &priv->stats;
}
```

返回有意义的统计数据所需的实际工作分布在整个驱动程序中,其中各种字段都被更新。以下列表显示了struct net_device_stats 中最有趣的字段：

无符号长 rx_packets;无符号长
tx_packets;
接口成功传输的传入和传出数据包总数。

无符号长 rx_bytes;无符号长
tx_bytes;
接口接收和发送的字节数。

无符号长 rx_errors;无符号
长 tx_errors;错误接收和传
输的数量。数据包传输可能出错的事情没有尽头， net_device_stats结构包括六个用于特定接收错误的计数器和五个用于传输错误的计数器。完整列表参见<linux/netdevice.h> 。如果可能,您的驱动程序应该保留详细的错误统计信息,因为它们对试图追踪问题的系统管理员最有帮助。无符号长 rx_dropped;无符号长 tx_dropped;

在接收和传输过程中丢弃的数据包数。当没有可用于数据包数据的内存时,数据包将被丢弃。tx_dropped很少使用。

无符号长碰撞;由于介质上的拥
塞而导致的冲突次数。无符号长多播;接收到的多播数据包数。

值得重申的是， get_stats方法可以随时调用 即使接口关闭 所以只要net_device结构存在，驱动程序就必须保留统计信息。

组播

多播数据包是一种网络数据包,旨在由多个主机接收,但不是由所有主机接收。通过为主机组分配特殊的硬件地址来获得此功能。该组中的所有主机都应接收定向到其中一个特殊地址的数据包。在以太网的情况下,多播地址在目标地址中设置了第一个地址八位字节的最低有效位,而每个设备板在其自己的硬件地址中都清除了该位。

处理主机组和硬件地址的棘手部分由应用程序和内核执行,接口驱动程序不需要处理这些问题。

多播数据包的传输是一个简单的问题,因为它们看起来与任何其他数据包完全一样。接口通过通信介质传输它们而不查看目标地址。内核必须分配正确的硬件目标地址;如果定义了hard_header设备方法,则不需要查看它排列的数据。

内核处理跟踪在任何给定时间感兴趣的多播地址的工作。该列表可以经常更改,因为它是在任何给定时间运行的应用程序和用户兴趣的函数。接受感兴趣的多播地址列表并将发送到这些地址的任何数据包传递给内核是驱动程序的工作。驱动程序如何实现多播列表在某种程度上取决于底层硬件的工作方式。通常,就多播而言,硬件属于以下三个类别之一：

- 不能处理多播的接口。这些接口要么接收专门针对其硬件地址的数据包（加上广播数据包），要么接收每个数据包。他们只能通过接收每个数据包来接收多播数据包,因此,可能会用大量“无趣”的数据包压倒操作系统。您通常不会将这些接口视为支持多播,并且驱动程序不会在dev->flags中设置IFF_MULTICAST 。

  点对点接口是一种特殊情况,因为它们总是接收每个数据包而不执行任何硬件过滤。

· 可以将多播数据包与其他数据包（主机到主机或广播)区分开来的接口。可以指示这些接口接收每个多播数据包,并让软件确定该地址是否对该主机感兴趣。在这种情况下引入的开销是可以接受的,因为典型网络上的多播数据包数量很少。 · 可以执行多播地址硬件检测的接口。可以向这些接口传递要接收数据包的多播地址列表,并忽略其他多播数据包。这是内核的最佳情况,因为它不会浪费处理器时间丢弃接口接收到的“无趣”数据包。

内核试图通过支持第三个设备类来利用高级接口的功能,这是最通用的,最好的。因此,每当有效多播地址列表发生变化时,内核都会通知驱动程序,并将新列表传递给驱动程序,以便它可以根据新信息更新硬件过滤器。

多播的内核支持

对多播数据包的支持由以下几项组成:设备方法、数据结构和设备标志:

void (*dev->set_multicast_list)(struct net_device *dev);每当与设备关联的机器地址列表发生更改时调用的设备方法。修改dev->flags时也会调用它,因为某些标志（例如IFF_PROMISC)可能还需要您重新编程硬件过滤器。该方法接收指向struct net_device的指针作为参数并返回void。对实现此方法不感兴趣的驱动程序可以保留字段集为空。

结构 dev_mc_list *dev->mc_list;
与设备关联的所有多播地址的链表。本节末尾介绍了结构的实际定义。

int dev->mc_count;
链表中的项目数。此信息有些多余,但检查mc_count是否为0是检查列表的有用快捷方式。

IFF_MULTICAST
除非驱动程序在dev->flags中设置此标志,否则不会要求接口处理多播数据包。尽管如此,当dev->flags更改时,内核调用驱动程序的set_multicast_list方法,因为多播列表可能在接口未激活时已更改。

IFF_ALLMULTI
网络软件在dev->flags中设置的标志,用于告诉驱动程序从网络中检索所有多播数据包。启用多播路由时会发生这种情况。如果设置了该标志,则不应使用dev->mc_list过滤多播数据包。

IFF_PROMISC
接口进入混杂模式时在dev->flags中设置的标志。每个数据包都应该由接口接收,独立于dev->mc_list。

驱动程序开发人员需要的最后一点信息是struct dev_mc_list 的定义,它位于<linux/netdevice.h> 中:

```
结构 dev_mc_list { 结构
    dev_mc_list *next;                          /* 列表中的下一个地址 */
    char dmi_addrlen; dmi_addr[MAX_ADDR_LEN];   /* 硬件地址 */ __u8 unsigned
                                                /* 地址长度 */ int
    dmi_users;                                  /* 用户数 */ int dmi_gusers;
                                                /* 组数 */
};
```

因为多播和硬件地址独立于数据包的实际传输,所以这个结构在网络实现中是可移植的,每个地址都由一个八位字节串和一个长度标识,就像dev->dev_addr 一样。

典型实现描述set_multicast_list

设计的最佳方式是向您展示一些伪代码。

以下函数是该函数在全功能(ff)驱动程序中的典型实现。该驱动程序功能齐全,它控制的接口具有复杂的硬件数据包过滤器,它可以保存该主机要接收的多播地址表。表的最大大小为FF_TABLE_SIZE。

所有以ff_为前缀的函数都是硬件特定操作的占位符:

```
void ff_set_multicast_list(struct net_device *dev) {

    结构 dev_mc_list *mcptr;

    if (dev->flags & IFF_PROMISC)
        { ff_get_all_packets( );返回;

    }
    /* 如果地址多于我们处理的地址,则获取所有多播数据包并在软件中对其进行排序。
    */ if (dev->flags & IFF_ALLMULTI || dev->mc_count > FF_TABLE_SIZE) {

        ff_get_all_multicast_packets();返回;

    }
    /* 没有多播?只需要我们自己的东西 */ if (dev->mc_count
    = = 0) {
```

```
            ff_get_only_own_packets();
            返回;
        }
        /* 将所有多播地址存储在硬件过滤器中 */ ff_clear_mc_list( );对于（mc_ptr =
        dev->mc_list;mc_ptr;mc_ptr = mc_ptr->next)ff_store_mc_address（mc_ptr-
        >dmi_addr）；

        ff_get_packets_in_multicast_list();
    }
```

如果接口不能在硬件过滤器中为传入数据包存储多播表,则可以简化此实现。在这种情况下，FF_TABLE_SIZE减少到0,并且不需要最后四行代码。

如前所述,即使是不能处理多播数据包的接口也需要实现set_multicast_list方法,以便在dev->标志的变化时得到通知。这种方法可以称为“非特色” （nf)实现。实现非常简单,如下代码所示： void nf_set_multicast_list(struct net_device *dev) {

```
        if (dev->flags & IFF_PROMISC)
            nf_get_all_packets();别的

            nf_get_only_own_packets();
    }
```

实现IFF_PROMISC很重要,否则用户将无法运行tcpdump或任何其他网络分析器。另一方面,如果接口运行点对点链接,则根本不需要实现set_multicast_list ,因为无论如何用户都会收到每个数据包。

其他一些细节

本节涵盖了网络驱动程序作者可能感兴趣的其他一些主题。在每种情况下,我们只是试图为您指明正确的方向。获得该主题的完整图片可能还需要花费一些时间来挖掘内核源代码。

媒体独立接口支持媒体独立接口（或 MII)是 IEEE

802.3 标准,描述了以太网收发器如何与网络控制器接口;市面上很多产品都符合这个接口。如果您正在为符合 MII 的控制器编写驱动程序,内核会导出一个通用的 MII 支持层,这可能会使您的生活更轻松。

要使用通用 MII 层,您应该包含<linux/mii.h>。您需要填写一个mii_if_info结构,其中包含有关收发器的物理 ID、全双工是否有效等信息。 mii_if_info结构还需要两个方法:

```
int (*mdio_read) (struct net_device *dev, int phy_id, int location); void
(*mdio_write) (struct net_device *dev, int phy_id, int location, int val);
```

如您所料,这些方法应该实现与您的特定 MII 接口的通信。

通用 MII 代码提供了一组用于查询和更改收发器工作模式的函数;其中许多是为与ethtool实用程序一起工作而设计的(在下一节中描述)。查看<linux/mii.h>和drivers/net/mii.c了解详细信息。

Ethtool 支持Ethtool

是一个实用程序,旨在为系统管理员提供对网络接口操作的大量控制。使用ethtool,可以控制各种接口参数,包括速度、媒体类型、双工操作、DMA 环设置、硬件校验和、LAN 唤醒操作等,但前提是驱动程序支持ethtool 。 Ethtool可以从http://sf.net/projects/gkernel/ 下载。

ethtool支持的相关声明可以在<linux/ethtool.h> 中找到。它的核心是一个ethtool_ops 类型的结构,它包含完整的 24 种不同的ethtool支持方法。这些方法中的大多数都相对简单。有关详细信息,请参见<linux/ethtool.h> 。如果您的驱动程序使用 MII 层,您可以使用mii_ethtool_gset和mii_ethtool_sset分别实现get_settings和set_settings方法。

要使ethtool与您的设备一起工作,您必须在net_device结构中放置指向ethtool_ops结构的指针。应使用宏SET_ETHTOOL_OPS (在<linux/netdevice.h> 中定义)来实现此目的。请注意,即使接口关闭,也可以调用您的ethtool方法。

Netpoll

“Netpoll” 是网络堆栈中相对较晚的(2.6.5)添加;其目的是使内核能够在整个网络和 I/O 子系统可能不可用的情况下发送和接收数据包。它用于远程网络控制台和远程内核调试等功能。在您的驱动程序中支持 netpoll 无论如何都不是必需的,但它可能会使您的设备在某些情况下更有用。

在大多数情况下,支持 netpoll 也相对容易。

实现 netpoll 的驱动程序应该实现poll_controller方法。它的工作是在没有设备中断的情况下跟上控制器上可能发生的任何事情。几乎所有poll_controller方法都采用以下形式：

```
无效 my_poll_controller(struct net_device *dev) {

    禁用设备中断（开发）；
    call_interrupt_handler(dev->irq, dev, NULL);重新
    启用设备中断（开发）；
}
```

poll_controller方法本质上只是模拟来自给定设备的中断。

快速参考

本节为本章介绍的概念提供参考。它还解释了驱动程序需要包含的每个头文件的作用。但是，net_device和sk_buff结构中的字段列表在此不再赘述。

#include <linux/netdevice.h>

包含struct net_device和struct net_device_ stats定义的标头，并包含网络驱动程序所需的一些其他标头。

struct net_device *alloc_netdev(int sizeof_priv, char *name, void (*setup)(struct net_device *); struct net_device *alloc_etherdev(int sizeof_priv); void free_netdev(struct net_device *dev);分配和释放net_device结构的函数。 register_netdev(struct net_device *dev); void unregister_netdev(struct net_device *dev);注册和注销一个网络设备。

无效*netdev_priv（结构网络设备*dev）；检索指向网络设备结构的驱动程序专用区域的指针的函数。

结构 net_device_stats;保存设备统计信息的结构。

netif_start_queue(struct net_device *dev);
netif_stop_queue(struct net_device *dev);
netif_wake_queue(struct net_device *dev);控制将数据包传递给驱动程序以进行传输的函数。在调用netif_start_queue之前不会传输任何数据包。 netif_stop_queue暂停传输， netif_wake_queue重新启动队列并戳网络层重新开始传输数据包。

skb_shinfo(struct sk_buff *skb);
提供对数据包缓冲区“共享信息”部分的访问的宏。

无效 netif_rx（结构 sk_buff *skb）；
可以调用的函数（包括在中断时)以通知内核已接收到数据包并将其封装到套接字缓冲区中。

无效 netif_rx_schedule(dev);
通知内核数据包可用并且应该在接口上启动轮询的功能;它仅由符合 NAPI 的驱动程序使用。 int netif_receive_skb(struct sk_buff *skb);无效 netif_rx_complete（结构 net_device *dev）;仅应由符合 NAPI 的驱动程序使用的函数。 netif_receive_skb是等效于 netif_rx 的 NAPI;它将一个数据包送入内核。当 NAPI 兼容的驱动程序用尽了接收数据包的供应时,它应该重新启用中断,并调用netif_rx_complete来停止轮询。

#include <linux/if.h>
该文件包含在netdevice.h 中,声明了接口标志（IFF_宏)和结构 ifmap,它在网络驱动程序的ioctl实现中起主要作用。

无效 netif_carrier_off（结构 net_device *dev）;无效 netif_carrier_on(struct net_device *dev); int netif_carrier_ok(struct net_device *dev);前两个函数
可用于告诉内核载波信号当前是否存在于给定接口上。 netif_carrier_ok测试设备结构中反映的载体状态。

#include <linux/if_ether.h>
ETH_ALEN ETH_P_IP 结构 ethhdr;包含在netdevice.h 中的if_ether.h定义了所有用于表示八位字节长度（例如地址长度)和网络协议（例如 IP)的ETH_宏。它还定义了ethhdr结构。

#include <linux/skbuff.h>
struct sk_buff和相关结构的定义,以及作用于缓冲区的几个内联函数。此标头包含在 netdevice.h 中。

struct sk_buff *alloc_skb(unsigned int len, int priority); struct
sk_buff *dev_alloc_skb(unsigned int len);无效 kfree_skb（结构
sk_buff *skb）;无效 dev_kfree_skb（结构 sk_buff *skb）； void
dev_kfree_skb_irq(struct sk_buff *skb); void
dev_kfree_skb_any(struct sk_buff *skb);处理套接字缓冲区分配和
释放的函数。驱动程序通常应使用专门用于此目的的dev_变体。无符号字
符 *skb_put(struct sk_buff *skb, int len);无符号字符 *__skb_put(struct sk_buff *skb,
int len);无符号字符 *skb_push(struct sk_buff *skb, int len);无符号字符
*__skb_push(struct sk_buff *skb, int len);向skb添加数据的函数； skb_put将数据放在skb
的末尾，而skb_push将其放在开头。常规版本执行检查以确保有足够的可用空间；双下划线版本将
这些测试排除在外。

int skb_headroom(struct sk_buff *skb);
int skb_tailroom(struct sk_buff *skb);
void skb_reserve(struct sk_buff *skb, int len);执行
skb 内空间管理的函数。 skb_headroom和skb_tailroom 分别告诉 skb 的开头和结尾有多
少可用空间。 skb_reserve可用于在skb的开头保留空间，该空间必须为空。

无符号字符 *skb_pull(struct sk_buff *skb, int len); skb_pull
通过调整内部指针从skb中“删除”数据。 int skb_is_nonlinear(struct
sk_buff *skb);如果此 skb 被分成多个片段以用于分散/收集 I/O，则返回真值的函
数。 int skb_headlen(struct sk_buff *skb);

返回 skb 的第一段的长度，即skb-> data 指向的部分。

无效 *kmap_skb_frag(skb_frag_t *frag);无
效 kunmap_skb_frag（无效 *vaddr）；
提供对非线性 skb 中片段的直接访问的函数。
#include <linux/etherdevice.h>
void ether_setup(struct net_device *dev);将
大多数设备方法设置为以太网驱动程序的通用实现的函数。如果名称中的第一个字符是空格或
NULL字符，它还会设置dev->flags并将下一个可用的ethx名称分配给dev->name 。

unsigned short eth_type_trans(struct sk_buff *skb, struct net_device *dev);

当以太网接口接收到数据包时,可以调用该函数设置skb->pkt_type。返回值是一个协议号,通常存储在skb->protocol 中。

#include <linux/sockios.h>

SIOCDEV私人

16 个ioctl命令中的第一个,可由每个驱动程序实现以供其私人使用。所有网络ioctl命令都在sockios.h 中定义。

#include <linux/mii.h> 结构

mii_if_info;支持实现 MII 标准的

设备驱动程序的声明和结构。

#include <linux/ethtool.h>

结构 ethtool_ops;允许设备使

用ethtool实用程序的声明和结构。