第 16 章 网络设备驱动程序

本章导读

在前面的章节中，我们依次涉及到了字符设备和块设备，本章我们将会接触到 Linux 下的另一种类型的设备即网络设备。网络设备是 Linux 下三大标准设备类型之一，现实中又通常被称为“网卡”或者 NIC(Network Interface Card)，用来完成高层网络协议(如 TCP/UDP 等)的底层数据传输以及一些设备控制等功能。

本章我们要讨论 Linux 内核中与网络设备驱动程序相关的内容，主要的文字将会围绕网络数据包的发送和接收来展开，这也是本章的主线。除此之外，我们还会探讨诸如 Qdisc、收发队列与拥塞控制、IP 路由过程中网络设备的选择、rps/xps、NAPI 以及多队列网络设备等话题，这些内容的某些部分或许跟网络设备驱动程序并没有直接的关系，但我还是觉得有必要看看。通常，对网络设备驱动程序而言，完成数据分组的接收任务要较之发送更具有挑战性，这是因为网络数据包的接收完全是个异步事件，而数据包发送的整个流程则可以完全控制在内核层面。对于那些更急于搞清楚网卡驱动是怎么回事的读者，建议先围绕着网卡数据包的接收及发送等内容展开，等略有所成心有大定之后再去看其他的内容，似乎要更加正统。在正式进入本章的主题之前，我们先啰嗦点互联网世界的协议以及 Linux 内核中 tcp/ip 协议栈开发历史等话题，其实就是扯淡或者花絮 :-)，权且做为开场白吧。

- **互联协议**

就数据传输这点上，网络设备类似于我们前面讨论过的块设备，通常情况下两者都被用来与系统进行大量的数据交互，根据上层模块的需求进行数据的发送和接收，但是网络设备如下的一些特性使得它与块设备区别开来：

首先网络设备在/dev 目录下没有入口点，换句话说，网络设备在系统中并不象块设备那样以一个设备文件的形式而存在，在应用层，用户通过套接口 API 的 socket 函数来使用网络设备。

> *socket API 的原型函数为：*
> *#include <sys/socket.h>*
> *int socket(int family, int type, int protocol);*
> *其中，参数 family 用来表示套接口所使用的协议族，包括 AF_INET(IPv4 协议族)和 AF_INET6(IPv6 协议族)等。参数 type 用来表示套接口的类型，有 SOCK_STREAM(字节流套接口)、SOCK_DGRAM(数据报套接口)和 SOCK_RAW(原始套接口)。一般来说，函数的第三个参数在实际使用中常设置为 0，除非用在原始套接口上。*

其次，网络设备除了响应来自内核的请求外，还需要异步地处理来自外部世界的数据包，而对于块设备而言，它只需要响应来自内核的请求，这使得网络设备驱动程序的设计模式无法等同于块设备驱动程序。除了处理数据之外，网络设备驱动程序还需要完成诸如地址设置、配置网络传输参数及流量统计等一些管理类的任务。

在继续下面的话题之前，我们不得不提一下经典的网络协议分层模型。虽然我们不打算在本书过多涉及这方面的内容，因为这不是本书的主题，但是作为互联网世界的幕后推手，以及从书籍逻辑完整性方面的考虑，这里还是有简介一下的必要。互联网发展的历史上曾出现过两种协议的分层模型，分别是国际标准化组织 ISO(International Organization for Standardization)的开放系统互联 OSI(Open Systems Interconnection)模型和 TCP/IP 参考模型。前者将组成网络的协议分为 7 层，分别是应用层、表示层、会话层、传输层、网络层、数据链路层和物理层，后者则将网络分为 4 层，分别是应用层、传输层、网际互联层和网络访问层。由于 ISO 的 OSI 模型只是一个理论上的模型，并没有成熟的产品，所以当今互联网事实上的国际标准其实是基于 TCP/IP 模型的，下图 16-1 显示了两种模型之间的区别：

在这两个参考模型中，各层只能与它相邻的层进行通信。按照 TCP/IP 的参考模型，网络设备及其驱动程序实际上完成的是最底层的网络访问层，该层直接面向实际承担数据传输的物理媒体，为数据通信的介质提供规范和定义，主要关心的是在通信线路上传输比特流的问题(信号与接口等)。

图 16-1 ISO/OSI 模型与 TCP/IP 参考模型

由于协议数据的封装性，各主机通过网卡与外部交换数据时最终是用一种称作网卡的硬件地址来标识各个主机，对于著名的以太网卡而言也称为 MAC 地址，是由 48 位长的二进制数组成。因为 MAC 地址作为一种网络世界的 ID 号，它必须具有全球唯一性，网卡生产商通常把分配到的 MAC 地址烧写进硬件中。

在 Linux 网络部分相关的内核源代码中，经常会出现"octet"这样的字眼，这是一个在网络世界中使用的术语，指代一个 8-bit 的数据位，跟字节是一个意思，它是网络设备和协议所能理解的最小单位。但是本书不会刻意去强调这种提法，因为从网络设备驱动程序员的角度，字节的叫法要更通用。

在实际使用的 TCP/IP 参考模型下，对于网络系统的数据传输而言，也相应地使用了数据封装的方式，它是分层模型的具体体现，下图 16-2 显示了网络数据包[1]在各协议层之间传递时的数据封装情况：

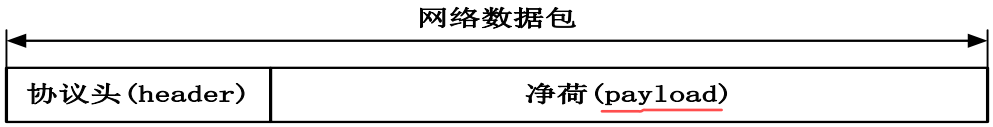

图 16-2 网络数据包的封装格式

[1] 在本书中，“网络数据包”的叫法通常与“skb”、“分组”或者“数据分组”混用，指的都是同一个东西。

在上图中，当一个网络数据包从上层往下层传递时，下层协议会将上层传下来的数据包视为一个净荷，然后再加上本层的协议头以完成该层协议所实现的功能控制信息，这个过程也就是平常所谓的数据包的封装，俗称打包，如果数据包从下层往上层传递时，过程与之相反，俗称解包。

广义地讲，网络设备种类繁多，比如网卡 NIC、交换机(switch)以及路由器(router)等等，然而在实际的工作中，读者接触最多的网络设备还是以太网设备 NIC，所以本书将以该设备作为讨论的主体对象，在后续的描述中，有时我会将 NIC、网络设备和网卡或以太网设备等称谓混用，除非有特别的说明，它们均指代同一个意思。这里我们给出一个经以太网设备传递的数据包的具体协议封装形式，以方便读者在本章后续部分的阅读过程中进行参考，下图 16-3 显示了某一以太网协议数据包的封装形式[2]：

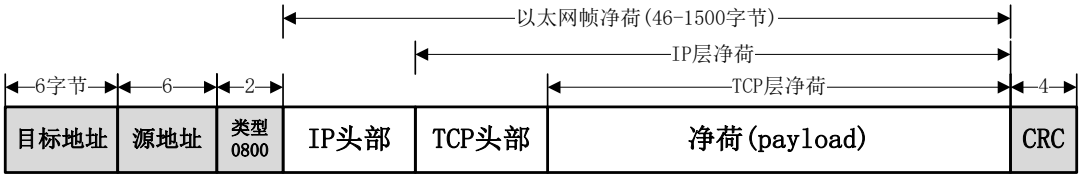

图 16-3 以太网帧封装格式

当某一网络数据包经高层一层一层往下传递，最终到达以太网卡时，将会被冠以以太网的头部，图中的目标地址和源地址即是通常所说 MAC 地址(6 字节)，是网络世界中设备实体唯一的身份号码，更多关于以太网协议的细节读者可以参考网络协议类的书籍。

- **Linux kernel network 开发历史**

此处 Linux kernel network 主要是指 Linux 内核中的 TCP/IP 协议栈等高层代码。Linux 内核中的网络协议栈代码并非移植某一个已有的实现，而是一个全新的设计。当时做这种决定主要是基于如下两个因素：
1. 担心已有实现的版权问题可能会对 Linux 将来发展形成的潜在困扰
2. Linux 的内核开发者们渴望实现一个全新的 TCP/IP 协议栈，他们希望能有机会接受这样一种挑战：一个全新的设计也许比现有的实现会更好。

接下来就是一些大牛们相继出场，这些已经载入 Linux 内核协议栈开发史册的早期开发者包括但不限于：Ross Biro, Orest Zborowski, Laurence Culhane, Alan Cox 以及 Donald Becker 等等。感兴趣的读者可以去自行翻看 Linux 内核开发历史，此处不再赘言。

现在，源起既明，下面我们开始进入正题。

[2] RFC894 的封装格式，也即通常所说的 Ethernet II 以太网帧，类型编码为 0x0800，另外还有两种类型的编码分别为 0x0806(用于 ARP 协议)和 0x8035(用于 RARP 协议)。

16.1 网络设备的软件抽象

一个顺理成章的逻辑是：在 Linux 内核中需要定义某种类型的数据结构来表示现实世界的网络设备，该数据结构的一个对象在软件层面便代表着现实世界的一种网络设备。Linux 内核不出意外的遵从了这个简单而直白的逻辑。

16.1.1　设备对象的数据结构

在 Linux 内核中，网络设备由数据结构 net_device 来表示，它存储着特定网络设备的所有信息，这是一个极其庞大的数据结构[3]，也是网络设备驱动程序开发者要面对的第一个核心数据。为了防止有胡乱粘贴代码的嫌疑，同时我也相信读者手边一定会很容易地得到一个最新的内核源码包，所以此处我只是将最精简的该数据结构定义从源码中摘录下来，一些在后文中经常被提及到的成员我会在再次谈及它们时再让其出场。需要注意的是，网络设备驱动程序并不会使用到该结构体中的所有成员，有些数据成员仅是提供给内核中的网络子系统所使用。在该结构体定义的后面，我们会将网络设备驱动程序可能用到的一些常见的成员变量先做简单的介绍：

```
<include/linux/netdevice.h>
struct net_device {
        char            name[IFNAMSIZ];

        struct hlist_node   name_hlist;
        unsigned long       state;
        struct list_head    dev_list;
        struct hlist_node   index_hlist;

        struct list_head    napi_list;
        int                 ifindex;
        const struct net_device_ops *netdev_ops;
        const struct ethtool_ops *ethtool_ops;
        unsigned int            flags;
        unsigned int            mtu;
        unsigned short          type;
        unsigned short          hard_header_len;
        unsigned char       *dev_addr;
        struct in_device __rcu  *ip_ptr;
        struct inet6_dev __rcu  *ip6_ptr;
```

[3] 源码中的注释认为，定义这么庞大的一个结构是个很大的错误，因为结构中各成员间的逻辑关系显得比较混乱。"Actually, this whole structure is a big mistake. It mixes I/O data with strictly "high-level" data, and it has to know about almost every data structure used in the INET module."。这个所谓的"big mistake"在我写本书第一版时所参考的 2.6 的代码中就存在了，到了现在我开始写第二版时，最新的 4.x 的内核版本依然没有消除掉这个"big mistake"。

```
    ...
#ifdef CONFIG_SYSFS
    struct netdev_rx_queue *_rx;

    unsigned int        num_rx_queues;
    unsigned int        real_num_rx_queues;
#endif
    struct netdev_queue __rcu *ingress_queue;
    struct netdev_queue     *_tx ____cacheline_aligned_in_smp;
    unsigned int        num_tx_queues;
    unsigned int        real_num_tx_queues;
    struct Qdisc        *qdisc;
    unsigned long       tx_queue_len;
#ifdef CONFIG_NET_NS
    struct net      *nd_net;
#endif
    struct device   dev;
    ...
}
```

- 部分成员变量的说明

char　　name[IFNAMSIZ]

网络设备的名称，当前内核为其指定的长度 IFNAMSIZ 为 16。在 Linux 内核中，设备名称字符串的末尾数字用来表示同一种网络设备类型的多个实例(比如系统中有两块以太网卡，则名称分别为 eth0 和 eth1)，下表是常见的一些网络设备的命名规则:

表 16-1 网络设备命名规则

名 称	设备类型
ethX	以太网设备
wifiX	无线网络设备
irdaX	红外设备
fddiX	光纤分布式数据接口设备
wlanX	WLAN 设备

unsigned long　　state

表示抽象设备的状态，内核中为此定义了一个枚举型变量:

```
<include/linux/netdevice.h>
enum netdev_state_t {
    __LINK_STATE_START,
    __LINK_STATE_PRESENT,
    __LINK_STATE_NOCARRIER,
    __LINK_STATE_LINKWATCH_PENDING,
    __LINK_STATE_DORMANT,
```

};

其中在驱动程序中最常见的是前三个，当用 register_netdev 向系统成功注册一个设备对象 dev 时，dev->state 的__LINK_STATE_PRESENT 位被置 1，表示设备已经进驻到了系统。当通过__dev_open 来打开该设备对象接口时，如果成功，dev->state 的__LINK_STATE_START 位将被置 1，表示设备将处于 running 状态。__LINK_STATE_NOCARRIER 和__LINK_STATE_LINKWATCH_PENDING 主要用于设备的链路监测部分，关于此点，我将在本章后续内容上继续讨论这个话题。

struct hlist_node　name_hlist
struct list_head　dev_list
struct hlist_node　index_hlist

用于系统中网络设备的列表管理，成功注册进系统的网络设备都将被加入到上述的三个链表中，其中 name_hlist 和 index_hlist 分别用于网络设备名称与接口索引的哈希表，dev_list 则将当前设备加入到所属命名空间(network namespace)中的 dev_base_head 所管理的全局链表。

struct list_head　napi_list

用于支持 NAPI 特性的网络设备，它将 struct napi_struct 对象的 dev_list 加入到 napi_list 链表中，本书后续的 NAPI 相关内容会有涉及。

const struct net_device_ops *netdev_ops;

网络设备方法操作集。该数据结构定义了针对当前设备的一组操作集合，比如 ndo_open、ndo_stop、ndo_start_xmit 等。稍早一点的内核将这些设备方法直接定义在 net_device 结构下面，当前内核已经将这些设备方法放到了一个内嵌在 net_device 中的 struct net_device_ops 数据结构中，设备驱动程序需要实现该操作集中的部分或者全部方法。

const struct header_ops *header_ops

针对网络访问层数据帧头部的一组操作集，比如根据上层协议(skb->protocol)的类型构造以太网帧的头部等等。对于以太网设备，这个操作集定义在全局变量 eth_header_ops 中。

int　ifindex

当前网络设备在其所属命名空间内的接口索引，用来唯一标示设备所提供的接口，内核利用它计算对应的 hash 值，然后根据此值将设备加入到 index_hlish 变量所指定的哈希表中，　便于内核其他模块能在指定的命名空间中快速查找到指定的网络设备。

unsigned int　flags

用于标记设备的使能状态：如果 IFF_UP 位被置 1，表明当前设备已经处于 enabled 状态，否则 disabled.

unsigned int　mtu

网络访问层的最大传输单元 mtu(maximum transfer unit)，是针对上一层的 payload。对于以太网设备，该值为 1500。

unsigned short　　hard_header_len

当前网络设备所对应的网络访问层协议头部的长度。对于以太网设备，该值为 14 个字节， 参见图 16-3。

unsigned char　　addr_len

网络访问层硬件地址长度。对于以太网设备，该值为 6 个字节(MAC 地址长度)。

unsigned char　　*dev_addr

NIC 设备的 MAC 地址。设备的 MAC 地址通常存放在设备的 EEPROM 中，由驱动程序负责读出，然后保存在该变量中。

struct in_device __rcu　*ip_ptr

struct inet6_dev __rcu　*ip6_ptr

当前 NIC 设备中与 IP 相关的数据结构，例如设备 IP、broadcast 以及 subnet mask 地址等，可用于数据包发送时在路由阶段查找到正确的承载数据包发送的设备。ip_ptr 用于 IPv4，在网络设备注册阶段通过通知链 netdev_chain 向系统通告一个新的网络设备加入时，由 inetdev_event()负责初始化该成员，ip6_ptr 则用于 IPv6，由 ipv6_add_dev()负责初始化。

struct netdev_rx_queue *_rx;

struct netdev_queue　　*_tx

网络设备用于接收和发送数据分组的队列。

unsigned int　　num_tx_queues

由 alloc_netdev_mq 函数分配的隶属当前网络设备的发送队列数量。

unsigned int　　real_num_tx_queues

网络设备中当前活动的发送队列的数量。

struct net　　*nd_net

网络设备所在的命名空间。当前内核已经开始对网络设备引入了命名空间的概念，相对于之前的单一的全区命名空间，struct net 的引入使得内核可以命名空间的方式在系统中建立多个独立的虚拟视图，各个视图之间彼此分隔。不过这种机制更多地被内核使用，对于设备驱动程序员而言，并不会直接和这些概念打交道。

int　　watchdog_timeo

用于设定网络设备在传输数据包时，传输超时的到期时间。

struct timer_list　　watchdog_timer;

发送分组超时定时器。当网络子系统发送队列中某一数据包在指定的时间窗口之后依然没有成功发送出去，那么该定时器将到期，最终导致设备驱动程序的传输超时函数被触发。

unsigned long　　tx_queue_len

该值用来决定当前的设备是否需要一个 qdisc 对象。如果此值为 0，那么在激活该设备

时便不会为该设备分配一个新的 qdisc 对象。

struct device　　dev

内嵌的内核对象所在的数据结构，该成员将会把网络设备纳入到设备驱动模型的体系当中。

16.1.2 设备对象的分配

对于网络设备驱动程序而言，在内核已经定义了用于抽象网络设备的数据结构(struct net_device)之后，其首要的任务自然是分配一个该类型的对象来代表所管理的设备，这是件颇为简单的任务。内核为了方便设备驱动程序开发者，特意代劳了一个专门用于分配 net_device 对象的宏 alloc_netdev，所以设备驱动程序员便无需自己再去调用 kmalloc 或者 vmalloc 函数然后再做些成员变量初始化等等琐事，因为 alloc_netdev 除了分配内存还执行必要的初始化任务。alloc_netdev 宏的具体定义如下：

```
<include/linux/netdevice.h>
#define alloc_netdev(sizeof_priv, name, name_assign_type, setup) \
    alloc_netdev_mqs(sizeof_priv, name, name_assign_type, setup, 1, 1)
```

由上可见 alloc_netdev 宏的本质其实是 alloc_netdev_mqs 函数，该函数要实现的主要功能为：

```
<net/core/dev.c>
struct net_device *alloc_netdev_mqs(int sizeof_priv, const char *name,
        unsigned char name_assign_type,
        void (*setup)(struct net_device *),
        unsigned int txqs, unsigned int rxqs)
{
    struct net_device *dev;
    size_t alloc_size;
    struct net_device *p;
    …
    p = kzalloc(alloc_size, GFP_KERNEL | __GFP_NOWARN | __GFP_REPEAT);
    if (!p)
        p = vzalloc(alloc_size);
    if (!p)
        return NULL;
    dev = PTR_ALIGN(p, NETDEV_ALIGN);
    dev->padded = (char *)dev - (char *)p;
    ...
    dev_net_set(dev, &init_net);
    ...
    setup(dev);
    ...
```

```
    dev->num_tx_queues = txqs;
    dev->real_num_tx_queues = txqs;
    if (netif_alloc_netdev_queues(dev))
        goto free_all;

#ifdef CONFIG_SYSFS
    dev->num_rx_queues = rxqs;
    dev->real_num_rx_queues = rxqs;
    if (netif_alloc_rx_queues(dev))
        goto free_all;
#endif

    strcpy(dev->name, name);
    dev->name_assign_type = name_assign_type;
    ...
    return dev;
}
```

关于这个函数，将主要围绕以下 4 个方面进行讨论：

- 函数参数

第一个参数是个整型的 sizeof_priv，它用来表示驱动程序的“私有数据”的大小。驱动程序根据自身需要决定是否需要“私有数据”，如果不需要，该参数置 0，否则函数将在 net_device 对象内存区之后另行分配一个 sizeof_priv 字节大小的内存区。若驱动程序使用了“私有数据”，那么它可以通过调用 netdev_priv 函数来得到该“私有数据”区的指针。

第二个参数 name，用来表示该网络接口的名称，比如"eth0"，此接口名称字符串将保存在 net_device 对象的 name 成员中，所以接口名称所在字符串的长度不能超过 16 字节。

第三个参数 name_assign_type 是新近(2014 年)才加入到内核的，主要用于表明 net_device 对象中接口名称 name 的来源方式，与驱动程序关系不大。

第四个参数是个函数指针，指针的类型为 void (*setup)(struct net_device *)，程序员需要在其设备驱动程序中负责实现一个该类型的函数，在调用 alloc_netdev 时作为实参传入，alloc_netdev_mqs 将会调用该函数用来初始化 net_device 对象中余下的一部分成员变量。

第五和第六个参数用来表示该 net_device 对象将拥有的收发队列的数量。

- 设备对象内存的分配

net_device 对象内存的分配是 alloc_netdev_mqs 函数要完成的首要功能，不过其原理甚为简单，只不过是利用了 kzalloc 或者 vzalloc。除了分配 net_device 对象之外，视设备驱动程序的具体需求，该函数还可能会在 net_device 对象内存区之后为驱动程序分配一块“私有数据”区。

- 对象成员变量初始化

主要用来初始化 net_device 对象的一些成员变量，比如 dev_addr_init(dev)用来初始化 dev 对象中的硬件地址链表 dev_addrs，dev_mc_init 和 dev_uc_init 函数分别用来初始化当前设备对象的组播 multicast 和单播 unicast MAC 地址列表，dev_net_set()则将该 net_device 对象的命名空间(namespace)设定为 init_net 等等。除了这些工作之外，该函数还会调用实参传递过来的 setup()函数继续初始化 net_device 对象的其他成员变量。

- 设备队列的分配

alloc_netdev_mqs()函数接下来还做了两件非常值得关注的事情，即分配发送和接收队列，之所以说它值得关注，是因为这两件事与网络分组的发送和接收密切相关，前文我们已经提到过，数据包的发送和接收是文章的主线。参数 txqs 和 rxqs 分别指明了该 net_device 对象 dev 所拥有的发送和接收队列数量。不过虽然从名称上看起来，dev->_tx 和 dev->_rx 的功能应该非常相似，其实不然，dev->_tx 的类型是 struct netdev_queue，而 dev->_rx 的类型则为 struct netdev_tx_queue，这二者在网络设备对数据包的处理上所起的作用迥异：dev->_tx 可以真正称为发送队列，而 dev->_rx 则主要用于 RPS，用来将接收到的分组分流到不同的 cpu 上处理，所以它其实是一个伪队列。而真正的接收队列实际上是由内核提供的一对{sd->input_pkt_queue， sd->process_queue}，对于 SMP 系统而言，系统中每个 cpu 都拥有一个此队列对(rx queue pair)。关于队列的问题，这里暂且讲这么多，因为后文我们还将用大量的篇幅讨论这些主角们。

现在，我们已经知道 alloc_netdev_mqs()用来产生一个 net_device 对象，这是网络设备驱动程序所要做的第一件事。内核已经通过 EXPORT_SYMBOL 宏将该函数导了出去，所以驱动程序中当然可以直接调用该函数。不过，为了方便驱动程序的编写，内核通过封装 alloc_netdev_mqs 的形式提供了各种常见网络设备对象的分配函数，如下表 16-2 所示[4]：

表 16-2 常见网络设备对象分配函数

设备类型	对象分配函数	setup 函数
以太网卡	alloc_etherdev()	ether_setup
光纤分布式数据接口 FDDI	alloc_fddidev()	fddi_setup
红外设备	alloc_irdadev()	irda_device_setup
光纤通道设备 FC	alloc_fcdev()	fc_setup
高性能并行接口 HIPPI	alloc_hippi_dev()	hippi_setup
Apple LocalTalk 设备	alloc_ltalkdev()	ltalk_setup

因为现实中我们遇到的网络设备绝大多数属于以太网卡设备，所以此处我们特别讨论一下上面提到的 alloc_etherdev 函数，它其实是一个调用 alloc_netdev_mqs 函数的宏，专门用来分配并初始化一个 NIC 设备对象。alloc_etherdev 宏的定义如下：

<include/linux/etherdevice.h>

```
#define alloc_etherdev(sizeof_priv) alloc_etherdev_mq(sizeof_priv, 1)
#define alloc_etherdev_mq(sizeof_priv, count) alloc_etherdev_mqs(sizeof_priv, count, count)
```

[4] 所有通过这些 alloc_*函数所分配的 net_device 对象都只有 1 个接收队列 1 个发送队列。

宏 alloc_etherdev_mqs 在其内部调用 alloc_netdev_mqs():

```
<net/ethernet/eth.c>
struct net_device *alloc_etherdev_mqs(int sizeof_priv, unsigned int txqs,
                                      unsigned int rxqs)
{
    return alloc_netdev_mqs(sizeof_priv, "eth%d", NET_NAME_UNKNOWN,
                            ether_setup, txqs, rxqs);
}
```

根据上面宏的定义可以看出，通过 alloc_etherdev 产生的 NIC 设备对象，它的_tx 和_rx 数量都是 1，所以如果你手头的设备需要多个收发队列，则不应使用 alloc_etherdev。

此处一个有趣的地方是调用 alloc_netdev_mqs 时传递给 net_device 对象的设备名称为"eth%d"，在后续讨论设备注册时我们将会看到此处的"eth%d"是如何转换成我们平时在用户空间看到的"eth0"及"eth1"这样的接口名称。另一个值得注意的地方是，调用 alloc_netdev_mqs()时将 ether_setup 函数作为实参传入，后者专门用于将 net_device 对象初始化成一个以太网卡设备，因此不妨看看内核中 ehter_setup 函数的实现：

```
<net/ethernet/eth.c>
void ether_setup(struct net_device *dev)
void ether_setup(struct net_device *dev)
{
    dev->header_ops         =   &eth_header_ops;
    dev->type               =   ARPHRD_ETHER;
    dev->hard_header_len    =   ETH_HLEN; //hard_header_len = 6 + 6 + 2 = 14
    dev->mtu                =   ETH_DATA_LEN; //mtu = 1500
    dev->addr_len           =   ETH_ALEN; // mac 地址长度：6 字节
    dev->tx_queue_len       =   1000;     /* Ethernet wants good queues */
    dev->flags              =   IFF_BROADCAST|IFF_MULTICAST;
    dev->priv_flags         |=  IFF_TX_SKB_SHARING;

    memset(dev->broadcast, 0xFF, ETH_ALEN);
}
```

所以，对于以太网设备的驱动程序而言，如果设备只需要使用 1 个接收队列和 1 个发送队列，那么应该使用 alloc_etherdev 函数来分配 net_device 对象，否则可以直接使用 alloc_etherdev_mqs 函数，该函数已经被内核用 EXPORT_SYMBOL 导出。
所以，在这一节的末尾，我们大体上得到了如下的一个 net_device 对象的内存分布图：

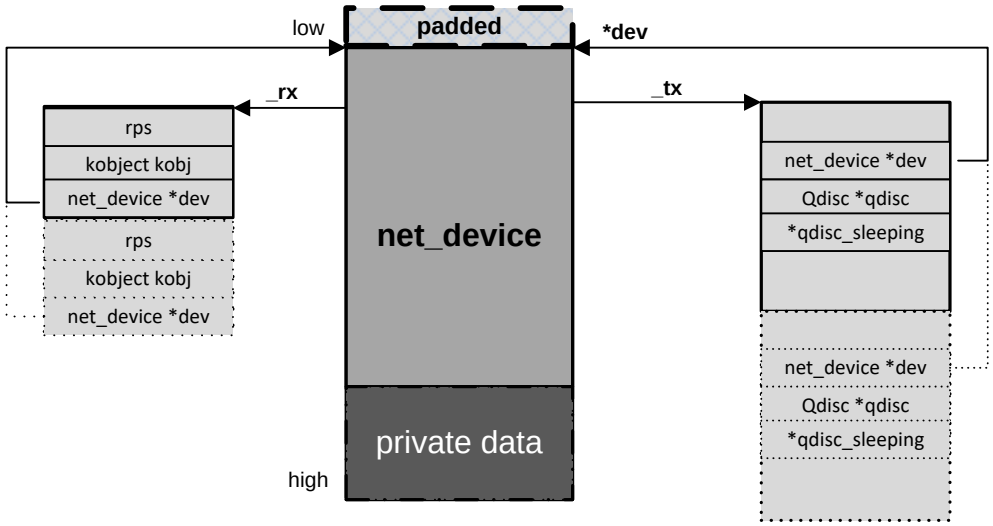

图 16-3 net_device 对象内存布局

16.1.3 设备对象的销毁

作为通用的规则，当驱动所在的模块从系统中移除时，设备驱动程序应该负责释放早前所产生的 net_device 对象，相对于生成对象的 alloc_netdev 函数，内核为对象释放所提供的函数叫 free_netdev。按照正常的逻辑顺序，设备驱动程序调用 free_netdev 的时机发生在将一个网络设备从系统中 unregister 之后，在这样的上下文中，让我们看看 free_netdev 在内核中的实现主体：

```
<net/core/dev.c>
void free_netdev(struct net_device *dev)
{
    ...
    netif_free_tx_queues(dev);
#ifdef CONFIG_SYSFS
    kvfree(dev->_rx);
#endif
    ...

    /*    Compatibility with error handling in drivers */
    if (dev->reg_state == NETREG_UNINITIALIZED) {
        netdev_freemem(dev);
        return;
    }

    BUG_ON(dev->reg_state != NETREG_UNREGISTERED);
    dev->reg_state = NETREG_RELEASED;
```

```
    /* will free via device release */
    put_device(&dev->dev);
}
```

函数的主要功能在于释放 alloc_netdev 分配的资源，同时更新对应的设备状态 reg_state，当前设备所对应的内核对象 device 的引用计数被减 1，如果当前函数的调用是对该设备对象的最后一个引用，那么设备所在的空间将最终被释放。

16.2 网络设备的注册

到目前为止，我们只是讨论了如何分配并初始化一个新的 net_device 对象，它只是对当前网络设备驱动程序所控制的网络设备的一个软件抽象，如果没有进一步的动作的话，系统并不会意识到有这样一个网络设备的存在，更具体地，当 Linux 内核的网络子系统需要发送或者接收一个网络数据包时，它不会调用到该 net_device 对象提供的功能函数。所以理所当然地，当驱动程序分配出一个新的 net_device 对象并将其初始化之后，接下来就需要将它向系统进行登记。将 net_device 对象注册进系统的任务由内核提供的 register_netdev 函数来完成，该函数的核心是 register_netdevice[5]：

```
<net/core/dev.c>
int register_netdevice(struct net_device *dev)
{
    struct net *net = dev_net(dev);
    /* When net_device's are persistent, this will be fatal. */
    BUG_ON(dev->reg_state != NETREG_UNINITIALIZED);
    BUG_ON(!net);
    ...
    ret = dev_get_valid_name(net, dev, dev->name);
    /* Init, if this function is available */
    if (dev->netdev_ops->ndo_init) {
        ret = dev->netdev_ops->ndo_init(dev);
    }
    ...
    if (!dev->ifindex)
        dev->ifindex = dev_new_index(net);
    else if (__dev_get_by_index(net, dev->ifindex))
        goto err_uninit;
    ...
    ret = netdev_register_kobject(dev);
    ...
    dev_init_scheduler(dev);
```

[5] 通常驱动程序需要调用 register_netdev 而不是 register_netdevice，除非驱动程序自己打算使用 rtnl_lock/unlock 对可能的并发进行保护。

```
    ...
    list_netdevice(dev);
    ...
    /* Notify protocols, that a new device appeared. */
    ret = call_netdevice_notifiers(NETDEV_REGISTER, dev);
    ...
}
```

上述从内核中摘录的代码显示了函数所要实现的几个主要功能点：

- sanity check

register_netdevice 函数被调用前，当前的网络设备对象 dev 在系统中注册的状态应该是 NETREG_UNINITIALIZED，后者是系统定义的枚举型变量之一：

```
    <include/linux/netdevice.h>
    enum { NETREG_UNINITIALIZED=0,
           NETREG_REGISTERED,     /* completed register_netdevice */
           NETREG_UNREGISTERING,      /* called unregister_netdevice */
           NETREG_UNREGISTERED, /* completed unregister todo */
           NETREG_RELEASED,       /* called free_netdev */
           NETREG_DUMMY,          /* dummy device for NAPI poll */
    } reg_state:8;
```

一个通过 kzalloc() 分配出的新的 net_device 对象 dev，其 dev->reg_state = NETREG_UNINITIALIZED，此外，dev->nd_net =& init_net;

- dev_get_valid_name()

当前的设备对象需要获得一个合乎规则的名称，此处的规则由内核函数 dev_valid_name()定义：它首先不能是个空字符串，然后它的长度不应超过 16，再然后它不能是"."或者"..", 最后它不应包含'/'、':'以及空格这些字符。对于由 alloc_etherdev()分配出的设备对象，其名称字符串为"eth%d"，所以 register_netdevice()会进入 dev_alloc_name_ns()路径中，后者负责为当前设备对象分配一个合法的名称。此处分配的原理为：函数首先分配一个初始值全为 0 的页面作为 bit map，然后在当前的 namespace 中扫描所有已注册的设备[6]名称(sscanf(d->name, "eth%d", &i)，对以太网设备而言，它关注所有名称为"ethX"的对象，比如，如果系统里已经有了一个"eth0"设备，那么 sscanf("eth0", "eth%d", &i)将返回 1，i = 0，这种情形下，它将此前分配的 bit map 第 i 位置 1，表示已经有设备占用了该位。如此循环，将 bit map 中所有已经被注册的设备位置 1。当退出这种置 1 循环后，它调用 find_first_zero_bit()在上述的 bit map 中找到第一个值是 0 的位并将该位的索引值作为"eth%d"的填充值，然后释放 bit map 所在的空间。如此，若系统中已经有了"eth0"、"eth2"和"eth3"等对象，那么当再次向系统注册一个以太网设备时，新设备将获得名称"eth1"。

- ndo_init()

[6] 所有成功注册进系统的网络设备都会被加入到设备所在 namespace 所指向的 dev_base_head 链表中，在当前上下文环境中，即为&init_net.dev_base_head，这也是 for_each_netdev_rcu 等宏定义背后的原理。

如果驱动程序为新分配的设备对象 dev 提供了 dev->netdev_ops->ndo_init 的实现，那么在将设备向系统注册的过程中，该设备方法集中的 ndo_init()将有机会被调用。设备驱动程序可以在该函数中做一些设备初始化等操作，不过目前只有很少的网络设备驱动会实现这一设备方法。

- dev_new_index()

dev_new_index 函数用来在当前设备所在的命名空间 net 为设备所提供的接口寻找一个唯一的接口索引值。当前 net->ifindex 初始值为 0，在 net 中每加入一个设备，系统都会将++net->ifindex 作为新设备的接口索引值。

- netdev_register_kobject()

一个在 register_netdevice 这个函数中需要重点关注的函数。它通过 Linux 设备驱动模型来向系统添加当前的设备，同时通过 sysfs 的方式将当前网络设备的一些属性等以文件或者目录的方式传递到用户空间。netdev_register_kobject 函数首先获取当前网络设备对象的 dev 成员，通过前面我们对网络设备数据结构 net_device 的解读，dev 成员是一个 struct device 类型的指针变量，内核通过它将网络设备变成一个内核对象(kobject)，继而通过 Linux 的设备模型来操控当前的网络设备，所以在 netdev_register_kobject 函数中我们可以看到如下代码：

```
<net/core/net-sysfs.c>
int netdev_register_kobject(struct net_device *ndev)
{
    struct device *dev = &( ndev->dev);
    …
    device_initialize(dev);
    dev->class = &net_class;
    ...
    dev_set_name(dev, "%s", ndev->name);
    …
    device_add(dev);
    ...
}
```

本书第九章“Linux 设备驱动模型”中已经详细讨论了有关内核对象 device 的操作，熟悉设备驱动模型的读者想必对这里的概念不会陌生，总之，网络设备内嵌的 dev 成员作为一个内核对象被加入到了系统中，并通过 sysfs 文件系统向用户空间暴露了它的存在，这正是用户空间中/sys/class/net 目录的来源。如果该网卡设备是一个 pci 设备，那么在系统启动阶段，内核会扫描 pci 总线，当发现该网卡设备时，会以 pci_device_add()调用的形式将该网卡的 pci_dev 对象加入到系统中，这里的 device_add()要发生在 pci_device_add()之后了，用于向系统添加 net_device 对象 ndev，而之前加入系统的 pci_dev 对象则成为 ndev 对象的父设备，这种关系最终被反映在用户空间的 sysfs 目录层次结构中。

- dev_init_scheduler()

该函数主要用来初始化设备对象 dev 所定义的发送队列的 Qdisc，在设备注册阶段，每个发送队列都使用同一个缺省的 Qdisc 对象 noop_qdisc：

```
struct netdev_queue *dev_queue;
struct Qdisc *qdisc = &noop_qdisc;
dev->qdisc = qdisc;
for (int i = 0; i < dev->num_tx_queues; i++) {
    dev_queue = &dev->_tx[i];
    rcu_assign_pointer(dev_queue->qdisc, qdisc);
    dev_queue->qdisc_sleeping = qdisc;
}
```

此外，该函数还为设备对象安装了一个 watchdog timer:

```
setup_timer(&dev->watchdog_timer, dev_watchdog, (unsigned long)dev);
```

当这个 watchdog timer 到期时，dev_watchdog 函数会被调用。记住这里只是安装一个 timer，至于该 timer 的触发时机以及 dev_watchdog 的具体用途，在本章后面会进一步讨论。

- list_netdevice()

该函数主要是将当前正在注册的网络设备对象加入到系统维护的几个链表中：

```
<net/core/dev.c>
static void list_netdevice(struct net_device *dev)
{
    struct net *net = dev_net(dev);

    ASSERT_RTNL();

    write_lock_bh(&dev_base_lock);
    list_add_tail_rcu(&dev->dev_list, &net->dev_base_head);
    hlist_add_head_rcu(&dev->name_hlist, dev_name_hash(net, dev->name));
    hlist_add_head_rcu(&dev->index_hlist,
                dev_index_hash(net, dev->ifindex));
    write_unlock_bh(&dev_base_lock);

    dev_base_seq_inc(net);
}
```

从函数的上述实现可以看到，它将当前设备对象加入到特定命名空间的几个哈希链表中，这样当网络子系统高层代码需要发送数据包时，它将能通过这些哈希链表找到特定的网络设备，关于这点，我们将会在本章后续讨论驱动程序中接收和发送数据包的实现部分时再次看到该哈希链表的用途。

- call_netdevice_notifiers(NETDEV_REGISTER, dev)

通过网络设备的通知链 netdev_chain 通告内核网络子系统一个新的网络设备 dev 已经加入到系统中，这导致内核的 inetdev_event 函数被调用，后者会分配并初始化一个 in_device 对象，然后把它的地址赋予 dev->ip_ptr，用来完成 IPv4 相关功能，比如路由等。

当 call_netdevice_notifiers 调用完毕，基本上 register_netdevice 的主要操作都已经结束。现在可以大体总结一下 register_netdevice 函数所要实现的核心功能：在设备对象 dev 所属的命名空间 net 中，要获得合法的 dev->name 和 dev->ifindex，这二者在所有注册进 net 的网络设备中必须具有唯一性。当设备被成功注册进系统后，它会被记录到当前 net 所指向的三个链表中。函数还会调用 netdev_register_kobject 函数将当前的 dev 对象加入到设备模型的框架体系中，这将在用户空间生成 sysfs 的文件或者目录。同时它将设备所拥有的发送队列的 Qdisc 对象初始化为 noop_disc，之后设备的状态处于__LINK_STATE_PRESENT。

一旦当前的设备被成功注册进系统，就意味着设备所提供的功能已经可由驱动模块所暴露的接口为外部其他模块所调用(比如内核网络协议栈代码或者是 ifconfig 这样的用户空间配置工具)，因此合理的逻辑顺序应该是只当设备所要完成的功能接口函数全部就绪后，设备模块才最终向系统注册该设备。至于内核的网络子系统高层在发送一个数据包时，如何确定由哪一个 NIC 设备(比如当前系统中拥有不止一个激活的 NIC 设备)来发送，那其实是“路由”相关的话题。

在 Linux 环境下用 route 命令可以查看到当前系统的路由信息，比如它的输出可能看起来向下面这样：

root@dennis-ws:/# route

Kernel IP routing table

Destination	*Gateway*	*Genmask*	*Flags*	*Metric*	*Ref*	*Use*	*Iface*
default	*192.16.10.1*	*0.0.0.0*	*UG*	*100*	*0*	*0*	*eth0*
link-local	*	*255.255.0.0*	*U*	*1000*	*0*	*0*	*eth0*
192.16.10.0	*	*255.255.255.0*	*U*	*0*	*0*	*0*	*eth0*

当网络子系统高层要发送一个数据包时，通过上述路由表得到接口信息，比如上面的的 eth0，然后再到设备列表中查找对应的设备。所以当一个 NIC 设备向系统成功注册后，它将被纳入到系统的网络设备列表管理体系中，这意味着从那以后它将“暴露”在网络子系统的高层代码之中，因此其携带的设备方法随时可能被高层代码所“征用”。

16.3 网络设备的注销

与设备注册过程相反，当驱动所在的模块要从系统中移除时，需要调用相应的设备注销函数 unregister_netdev，该函数的核心是 unregister_netdevice[7]：

<net/core/dev.c>
void unregister_netdev(struct net_device *dev)

[7] 如同 register_netdevice 一样，驱动程序应使用 unregister_netdev 而不是 unregister_netdevice.

```
{
    rtnl_lock();
    unregister_netdevice(dev);
    rtnl_unlock();
}
```

在 unregister_netdevice 函数发起的调用链中，rollback_registered_many 函数承载着设备注销的实质性任务，基本上 unregister_netdev 完成与 register_netdev 相反的功能。当一个网络设备从系统中注销后，它将不会再被网络子系统的高层所使用。

16.4 链路监测与设备激活

本节将讨论网络设备的链路监测(Link Watch)与设备激活相关话题。当一个 NIC 设备注册进系统时，内核并非马上就可以使用它们，至少我们需要打开该设备，在此情形下只有设备驱动程序监测到了设备的链路已经就绪并且被激活的情形下，设备才可以真正去接发数据分组。一个没有插入或者是虽然插入了网线但是对端设备没激活(比如对端是个未上电的路由设备等)的 NIC 设备可以被打开，但未必是被激活，所以需要链路检测。

16.4.1　链路监测

在本书中，链路就绪和 link carrier is ok 的称谓是一个意思，按照字面的意思，是链路载波。内核网络栈高层代码能使用一个网络设备(这里只针对实际物理设备而非一个纯软件定义的虚拟设备)的基本前提是：首先该设备要已经被 up 起来(dev->flags & IFF_UP)，其次该设备链路要就绪。第一点是__dev_open 要完成的任务，我们后面会讨论到它。第二点则跟 NIC 设备的链路状态监控有关，这是本节的主题。

设备驱动程序需要知道 NIC 设备是否已经插入了网线或者在插入了网线的情形下对端设备是否已经就绪(按照内核的说法就是 if (netif_carrier_ok() == true))，这是链路监测的任务。链路监测需要内核和设备驱动程序一起来完成，对内核而言，它叫 Link Watch，对设备驱动程序而言，它也许应该叫 link carrier detection. 这里权且笼统称之为 Link Watch。

通常，设备在链路状态发生变化时会产生一个中断，设备驱动程序可以采用中断与轮询相结合的方式来确定链路是否已经就绪，总之，这是个跟具体设备相关的任务，是设备 datasheet 要描述的内容。

当驱动程序监测到设备的链路已经就绪时(内核中的说法是："Device has detected that carrier")，它需要调用 netif_carrier_on()来通知内核：

```
<net/sched/sch_generic.c>
/*Device has detected that carrier.*/
void netif_carrier_on(struct net_device *dev)
{
```

```
    if (test_and_clear_bit(__LINK_STATE_NOCARRIER, &dev->state)) {
        if (dev->reg_state == NETREG_UNINITIALIZED)
            return;
        atomic_inc(&dev->carrier_changes);
        linkwatch_fire_event(dev);
        if (netif_running(dev))
            __netdev_watchdog_up(dev);
    }
}
```

netif_carrier_on()主要完成三件事：第一，就是操作 dev->state 的__LINK_STATE_NOCARRIER 这个比特位，这是个纯软件层面的工作，目的是让驱动程序的其他执行路径可以判断当前设备的链路情况，关于这一点可以看看内核中另一个函数 netif_carrier_ok()，它的实现代码清晰地展现了__LINK_STATE_NOCARRIER 位的作用，此处不再赘述。第二，是由 linkwatch_fire_event()的调用引发的，它实际上会导致一个 work 的 schedule，进而触发 linkwatch_do_dev()的调用，后者根据当前链路上的 carrier 是否 ok 来决定激活设备与否(dev_activate OR dev_deactivate)，考虑到一个没接入网线的 NIC 设备可以先被 up 起来，那么当用户在稍后的一个时间点再插入网线时，设备驱动程序监测到 NIC 设备的链路已经就绪并调用 netif_carrier_on()，此时 netif_carrier_on()的第二个工作就是告诉内核去 active 这个设备，这正是由 linkwatch_fire_event()完成。反之亦然。第三，就是在当前设备已经被成功 up 起来的情形下启动设备的 watchdog 定时器，关于此点，在本章后面会有专门的叙述。

如果设备驱动程序不使用 netif_carrier_on()来通知内核，那么在内核看来，这个设备便是处于待激活状态，因此内核不会使用该网络设备接口做网络数据包收发工作。

驱动程序在做链路状态监测时另一个可能用到的函数是 netif_carrier_off，当程序监测到当前 NIC 设备的链路没有就绪时(比如某人从你的电脑前经过时，不小心用脚把你网卡上的网线踢掉了)，那么就应该及时 deactive 掉当前的 NIC 设备。

16.4.2 设备的激活

在上述“链路监测”小节中，我们已多次提到了设备的激活，这个在内核中真正的“称谓”是 dev_activate()，相对应的，让一个设备由活动状态变为非活动状态则为 dev_deactivate().

一个网络设备只有当真正被激活之后，内核才能使用它收发分组。因此，内核中 dev_activate()主要做两件事，一是为设备的发送队列设置一个缺省的 Qdisc(queueing discipline)对象"pfifo_fast"，另一个是启动设备的 watch dog，这之后该定时器函数 dev_watchdog 将按照设备对象 dev 中 watchdog_timeo 成员所设定的频率运行，例如，如果 dev->watchdog_timeo = 6 * HZ，那么 dev_watchdog()将每隔 6 秒被调用一次，如果驱动程序没有为 dev->watchdog_timeo 准备一个确定的值，那么内核采用 5*HZ 作为其缺省值。无论 Qdisc 还是 dev watchdog 都是本章后文要详细讨论的内容，所以此处不再赘述。

16.4.3 设备的打开

打开一个网络设备的行为在内核代码中可以追溯到__dev_open 这个函数，__dev_open 与驱动程序监测到链路上出现载波时调用 netif_carrier_on()所做的事情大体有点类似，不过网络设备打开一般可视为一个同步事件，比如在用户空间使用 ifconfig eth0 up 命令时，就会涉及到打开一个设备的问题，而链路监测则基本上是一个异步事件，因为你不知道网线何时会插到网卡上，你也不知道对端设备何时会加电。说两者所做的事情大体类似，是因为它们最终都会试图去激活设备(调用 dev_active)，但是设备的打开会设置设备的 flags 变量：dev->flags |= IFF_UP，而 netif_carrior_on()所引起的 linkwatch_do_dev()只试图去检测这个 flags: if (dev->flags & IFF_UP). 除此之外，在设备打开的过程中，如果驱动程序中定义了 net_device_ops 中的 ndo_open 方法，那么在设备打开的过程中，ndo_open 例程有机会被调用到。

16.4.4 设备的状态

有了上文的铺垫，现在可以总结一下一个网络设备对象在注册、打开以及被激活等过程中的状态变迁。struct net_device 数据结构中为此定义了 flags 和 state 两个成员，state 成员的值是一个枚举量 netdev_state_t:

```
<include/linux/netdevice.h>
enum netdev_state_t {
    __LINK_STATE_START,
    __LINK_STATE_PRESENT,
    __LINK_STATE_NOCARRIER,
    __LINK_STATE_LINKWATCH_PENDING,
    __LINK_STATE_DORMANT,
};
```

从名称上看，dev->state 与 link 有关系，flags 上我们只关注一个标志位 IFF_UP. 此处我们依然以以太网卡为说明对象，当一个以太网卡对象 dev 通过 alloc_etherdev()分配出来是，dev->state=0, dev->flags = IFF_BROADCAST|IFF_MULTICAST;
紧接着我们调用 register_netdevice 来注册该设备对象 dev，注册动作完成后 dev->state 的__LINK_STATE_PRESENT 位被置上，表示设备在系统已经存在了(present)。接下来的操作逻辑自然是打开该 dev 设备，这导致 dev->state 的__LINK_STATE_PRESENT 和__LINK_STATE_START 位均被置 1，如此意味着该设备接口卡已经被 bring up 起来了，但是其 link 上有无 carrier 还需要驱动程序进一步确定，这种确定由 netif_carrier_on 和 netif_carrier_off 来告知内核。这些状态变迁过程见下图 16-4：

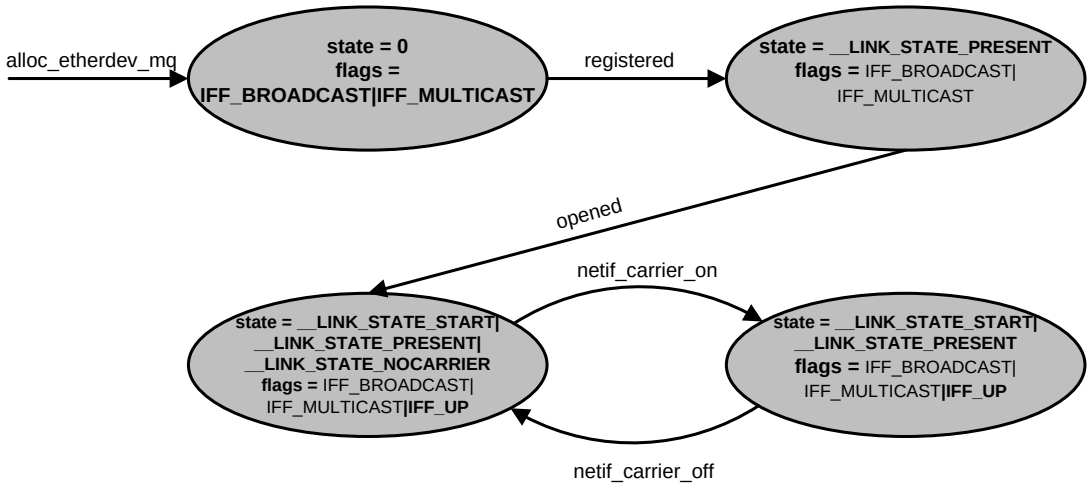

图 16-4 net_device 状态变迁

我画这张图，主要目的一是方便读者在阅读源码时很清楚地知道设备在当前处于一个什么样的状态，二是在你们实际写设备驱动程序时，对每行关键代码背后所实现的功能能了然于胸。

16.5 设备方法

如果类比面向对象的编程语言 C++，这里的设备方法如同一个 class 中的成员函数，用来描述 net_device 对象的动态行为逻辑。Linux 内核为 net_device 对象定义了若干个方法函数，这些函数统一定义在一个名为 net_device_ops 的结构体中。我在这里用单独的一小节来讨论设备方法，其目的只是想尽早引入设备方法这一概念，因为后续的讨论中很多地方都会出现对这里所出现的函数指针的引用。为此，当前我只会摘录一些设备方法中很常见的一些函数，然后再给与一个简明的描述，让读者大体了解这个方法所要达成的功能。至于这个方法具体的调用时机以及上下文环境，我会在本章的后续部分将它们单独拿出来讨论。

现在看看内核中这个设备方法集到底长什么样：

```
<inclue/linux/netdevice.h>
struct net_device_ops {
        int             (*ndo_init)(struct net_device *dev);
        ...
        int             (*ndo_open)(struct net_device *dev);
        int             (*ndo_stop)(struct net_device *dev);
        netdev_tx_t     (*ndo_start_xmit) (struct sk_buff *skb, struct net_device *dev);
        ...
};
```

上面我只摘录了 net_device_ops 中定义的有限几个函数指针，源码中这个结构体完整版包含了大约四五十个方法。需要说明的是，对于设备驱动程序而言，一个网络设备对象操作集中的函数是可选的，这意味着驱动程序需要根据自己所管理设备的实际功能决定实现哪些函

数，如果某一功能在驱动程序中没有实现，那么对应的函数指针应该是个空指针。

16.5.1 设备初始化

```
int   (*ndo_init)(struct net_device *dev)
```

这个函数我们在前面讨论网络设备的注册时，在 register_netdev 函数中已经看到过它的身影，正如名称所提示的那样，该函数用来对当前正在向系统注册的网络设备对象做一些晚期阶段的初始化工作。这里之所以说晚期阶段的初始化工作，是因为我们在前面讨论用于分配一个 net_device 对象的 alloc_netdev 函数的时候，已经看到 alloc_netdev 会对分配出的 net_device 对象的部分成员进行过初始化的工作，所以当设备驱动程序调用 register_netdev 函数向系统注册一个网络设备对象时，如果该设备对象的 net_device_ops 操作集中定义了 ndo_init 函数，它将有机会被 register_netdev 函数所调用。因为 alloc_netdev 中的初始化更多的是从内核角度产生的一个通用的过程，而如果你的设备需要一些特定的与众不同的初始化工作，那么就不应该忽略掉此处的 ndo_init 函数。如果 ndo_init 函数的执行过程失败，它应该返回一个错误码，后者将由 register_netdev 函数返回，这也意味着整个设备注册过程的失败。驱动程序是否需要实现一个 ndo_init 例程，由驱动程序根据自身需要自行决定，不过目前在内核源码中实现 ndo_init 的网络设备驱动程序并不多。

在 net_device_ops 操作集中与 ndo_init 函数对应的是 ndo_uninit，很显然这是个 ndo_init 的逆向过程，所以它在驱动程序中主要在一些收拾残局的场景中被使用，比如在 register_netdev 函数中，如果在 ndo_init 函数调用之后的一些流程中出现非正常的情况，ndo_uninit 将有机会被调用以恢复 ndo_init 所做的一些工作，再或者比如当设备所在的模块从系统中移除时所调用的 unregister_netdev 函数，ndo_uninit 也有机会被执行到。

16.5.2 设备的打开与停止

当设备所在的网络接口被激活时，比如使用 ifconfig ethX up 激活某一网络接口时，接口将被打开。此时如果对应的驱动程序提供了 ndo_open 函数，那么它将被调用。不同设备的 ndo_open 函数所完成的工作也是不一样的，这里并没有一个通用的准则，一些常见的操作包括分配接收和发送网络数据包所需要的资源，初始化设备的硬件中断并向系统注册中断，启动 watchdog 定时器，通知上层网络子系统当前接口已经就绪等等，不一而论。与之相反的过程则发生在 ndo_stop 函数中，它在当前的网络设备接口被关闭(shutdown, deactive…)时被调用，通常完成的工作与 ndo_open 相反。

16.5.3 数据包的发送

这是本章我们要重点讨论的内容之一，因为网络设备的核心功能就是用来发送和接收数据包。数据包的发送函数实现在 net_device_ops 的 ndo_start_xmit 成员中，该成员的原型是：

```
netdev_tx_t          (*ndo_start_xmit) (struct sk_buff *skb, struct net_device *dev);
```

关于数据包的发送，本章后文会有专门文字详细讨论之，这一节只是泛泛而谈，目的是让读者对 Linux 下网络设备驱动程序如何发送一个数据包有个整体的了解。

参数 skb 的类型 struct sk_buff 是网络设备驱动程序中另一个重要的数据结构，它的名字通常叫做套接字缓冲区，本章接下来会有专门一节讨论该数据结构及内核提供的操作该结构对象的相关接口函数，目前我们只要知道待发送的数据包的数据就包含在 skb 参数中就够了，如果再具体点，skb->data 指向要发送的数据包在内存中的位置，而 skb->len 则是以字节为单位的该数据包的长度。第二个参数 dev 自然就是本次用来发送网络数据包的设备对象了。

不同于网络数据包的接收，发送过程算得上是个同步的过程，当 Linux 内核中的网络子系统上层部分有数据包需要发送的时候，它将通过网络设备的 ndo_start_xmit 函数来发送该数据包，网络设备驱动程序的核心任务之一便是实现该函数。很显然，这是个跟具体的网络设备硬件相关的一个函数，作为一般的规则，驱动程序通常需要使用 DMA 的方式将套接字缓冲区中的数据传输到网络设备的存储空间中，然后由网络设备的硬件逻辑负责把设备存储空间中刚接收到的数据发送出去，当数据成功发送之后，设备会产生一个硬件中断以通知驱动程序进行相应的处理，内核将这种中断的处理分为 hardirq 和 softirq 两部分，在 hardirq 部分，通常的做法是检查中断状态寄存器[8]以确定中断原因。如果是分组成功发送的中断，那么驱动程序通常只需要调用 dev_kfree_skb_any 或者是 dev_kfree_skb_irq 等函数来释放分组所占用的内存空间[9]。这些函数并不做真正释放内存的工作，它们只是将 skb 放在一个名为 sd[cpu].completion_queue 的队列中，然后 raise 一个 NET_RX_SOFTIRQ 就返回了。真正释放分组所占空间的动作要一直延迟到 softirq 阶段，因为后续我们会详细讨论这里的各种技术细节，此处不再赘述。为了读者阅读与理解的方便，我们抽象出下面的一个具体模型来描述设备驱动程序实现发送函数的一般性流程，见下图 16-5：

[8] 此处中断状态控制器是一种统称，不同厂家的 NIC 有其各自的名称，但是作用都大同小异。
[9] 原则上，只要驱动程序能确定不再需要继续使用某一个 skb 所占用的内存空间，就可以释放之。

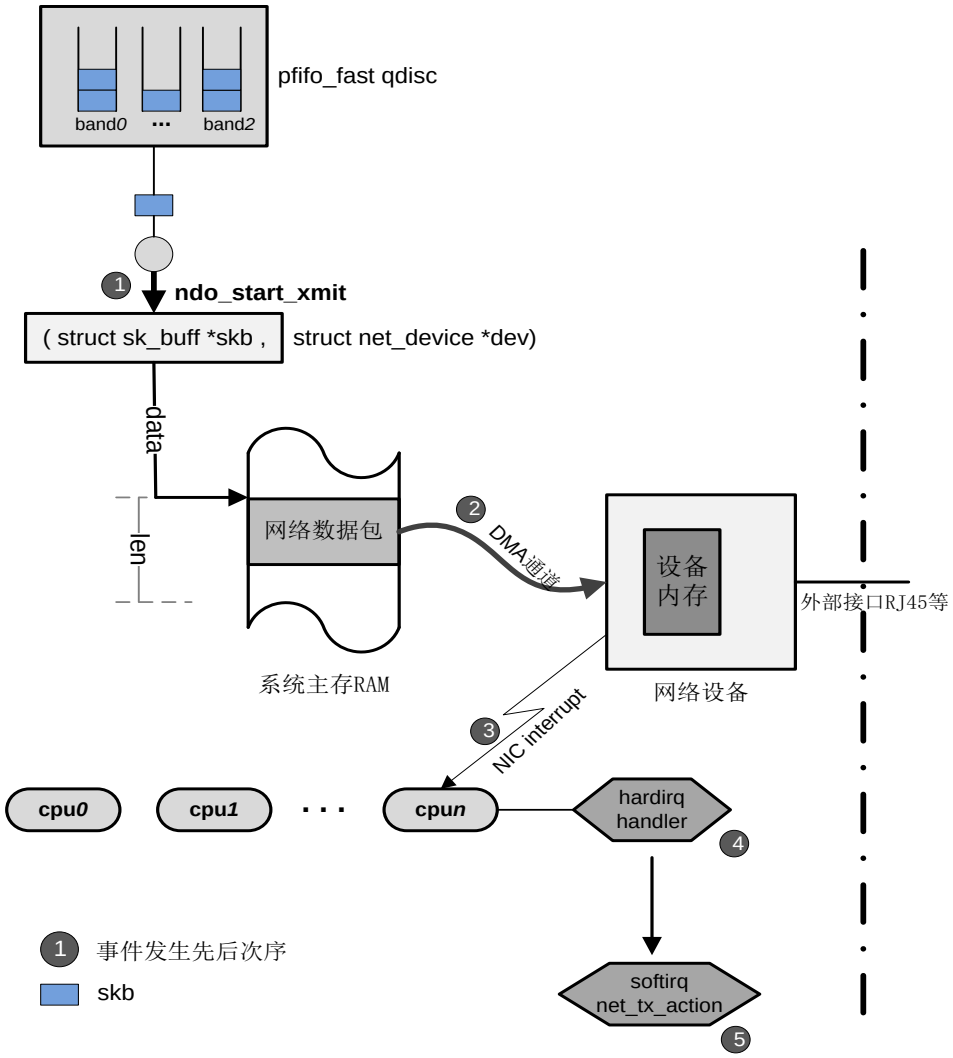

图 16-5 网络设备驱动程序数据包发送模型

针对图 16-5，我们稍微展开一点。当网络子系统上层有数据包要发送时，它首先会将分组放入到 qdisc 队列中(我们假设设备是使用 qdisc 队列的)，然后 qdisc 队列中的分组会被取出，最终通过调用网络设备驱动程中实现的 ndo_start_xmit 函数，将分组 skb 发送出去。在驱动程序的发送数据分组函数的具体实现中，通常的做法是，它首先在 skb 所在的主存中的数据空间和网络设备内部的设备内存间建立 DMA 通道，然后启动该 DMA 通道将数据包由系统主存传输到设备内存中，这之后便是由网络设备的硬件逻辑来负责将设备内存中新接收的数据发送出去。当设备内存中的数据成功发送完毕后，将会给处理器发出一个硬件中断信号，如此驱动程序的中断处理例程将会参与进来做一些数据包发送后的善后处理工作。

在上面的过程中，我们看到由于 DMA 通道的源端数据所在的缓冲区 skb->data 来自于内核的网络系统上层，换言之它不属于我们的驱动程序所能控制的范围之内，所以现实中为了建立对应的 DMA 映射，一般多是采用流式 DMA 映射，当然原理上采用一致性映射也是可行的，不过因为需要在 skb->data 与一致性缓冲区之间进行拷贝操作因而可能会付出性能上的代价。

理想的情况下，当新建的 DMA 通道成功地将套接字缓冲区 skb 中的数据传输到设备内存之后，网络设备的硬件发送逻辑会圆满完成本次发送任务。然而事实并非总是如此，软件层面的高速性与实际硬件的发送速度几乎总是会存在矛盾，此时作为网络设备驱动程序设计者，必须提供相应的处理机制以尽可能确保在数据包的传输过程中不会出现丢包的现象，这个话题也是本章后续小节要讨论的内容。

需要注意的是，网络子系统高层传下来的套接字缓冲区需要由设备驱动程序负责释放，设备驱动程序一般在中断处理例程中完成这个任务[10]，关于释放 skb 的话题，前面简单提及了一下，我们将在后续的“网络设备的中断处理”和“套接字缓冲区 skb”小节中予以讨论。

如果此处我们对数据包的发送过程做个简单小结的话，那就是：一个数据包的发送过程可以分成两个独立的逻辑部分，按照时间顺序，分别是网络子系统部分(Qdisc 模块)和设备驱动程序部分。网络子系统这块在整个 Linux 网络部分源码中是独立于底层的网络硬件的，出于性能及可靠性等因素的考虑，网络子系统这块实现有传输队列[11]，每个要发送的数据包都会先放到 qdisc 传输队列中。真正的发送过程发生在网络设备驱动程序所实现的 ndo_start_xmit 函数中，后者的实现依赖于具体的硬件设备，通常硬件在当前帧传输结束时会以中断的方式通知驱动程序。

16.6 套接字内核缓冲区 SKB

在我们打算进行后续话题的讨论之前，先来探讨一下所谓的套接字内核缓冲区 SKB(Socket Kernel Buffers)，这是个 struct sk_buff 类型的变量，为行文方便，以下将 SKB 对象记作 skb。说实在的，这个话题相比于本章其他话题而言，并不算太有趣。但是因为这个 skb 在整个内核网络子系统中是如此重要，网络设备驱动程序相关的文字自然不可能对它选择性的无视。

从软件层面的角度，skb 携带着网络分组数据在网络协议栈各层之间穿梭流动，当然也包括设备驱动这一层面。各层之间因为 skb 的存在而产生交互行为。我们在此处讨论它，则更多是从设备驱动程序的角度出发，了解其一些重要成员的作用以及内核为操作该数据对象所提供的一些接口函数，如此读者才能在实际的网络设备驱动程序的编写中娴熟地使用对应的各种函数来操作 skb，另外如果读者对 Linux 内核中网络子系统的高层代码感兴趣，也需要此处讨论的内容，如前所述，skb 的使用几乎频繁出现在网络组件的几乎所有关键场合。

下面是经过适当精简后的的 struct sk_buff 数据结构在内核源码中的定义：

```
<include/linux/skbuff.h>
struct sk_buff {
    union {
```

[10] 如前所述，成功发送出去的数据包所属 skb 真正的释放工作发生在 softirq 的 net_tx_action()阶段，它是 NET_TX_SOFTIRQ 对应的 softirq 处理例程。内核认为释放 skb 缓冲区是件比较耗时的操作，而驱动程序中的 hardirq 阶段只应该完成最关键的操作。

[11] 当然也可能某些设备没有使用传输队列，比如回环设备(loopback)，对于这种没有使用传输队列的设备而言，如果传输失败，因为没有队列可供使用，内核会直接丢弃该帧，此时传输的可靠性要依赖于更高层的协议，比如 TCP 协议可以通过超时重传机制让高层重新发送被丢弃的帧，否则该帧将永久丢失。

```
        struct {
            /* These two members must be first. */
            struct sk_buff      *next;
            struct sk_buff      *prev;
            …
        };
        …
    };
    struct sock         *sk;
    struct net_device   *dev;
    ...
    unsigned int        len,
                data_len;
    __u16               mac_len,
                hdr_len;
    __u16               queue_mapping;
    ...
    __u32               priority;
    __u32               hash;
    …
    __be16              protocol;
    __u16               transport_header;
    __u16               network_header;
    __u16               mac_header;
    …
    /* These elements must be at the end, see alloc_skb() for details.  */
    sk_buff_data_t          tail;
    sk_buff_data_t          end;
    unsigned char       *head,
                *data;
    ...
};
```

即便经过精简，从以上的定义中还是能感觉到这是个比较复杂的数据结构。读者没有必要对其每个成员都要非常清楚其用途及内涵，读者只需关注少量的和设备驱动程序相关的变量就足够了。不过仔细研读一下内核中的这个数据结构对于读者提升对复杂数据结构的定义能力应该还是蛮有帮助的 ☺

网络设备驱动程序中经常要使用到的一些成员及简单说明：

struct net_device　*dev

当前用于发送和接收该套接字缓冲区的网络设备对象。

```
__u32               priority;
```

标明 skb 分组的优先级，由 IP 头部的 ToS(Type of Service)字段决定，可用于 QoS.

__be16　　protocol

当前网络分组所对应的协议 ID，常见的有 IP(0x0800)、IPv6(0x86DD)以及 ARP(0x0806)等。

sk_buff_data_t　　transport_header

对应网络传输层协议头部数据的地址。

sk_buff_data_t　　network_header

对应网络层协议头部数据的地址。

sk_buff_data_t　　mac_header

对应网络 MAC 层协议头部数据的地址。

sk_buff_data_t　　tail
sk_buff_data_t　　end;
unsigned char　　*head, *data

指向套接字缓冲区中数据的指针。其中，head 指向一个已分配空间的头部，end 则指向该空间的尾部。data 指向上述分配的空间中有效数据的头部，tail 则指向该有效数据的尾部。当一个套接字缓冲区在网络各协议层间交互流动时，head 和 end 这两个值都是不变的，而 data 和 tail 则会在各层中由相应的模块根据需要进行修改，以容纳或者剥离对应的有效数据。因此，上述四个值其实是指向同一内存块的不同位置，后者所在的内存区域由__alloc_skb 函数负责分配。通过改变指针位置而不是数据拷贝或者移动的方式来管理套接字缓冲区中的数据，可以提升操作效率。

unsigned int　　len, data_len

len 是该套接字缓冲区中全部数据的长度，包括上述 data 指向的数据和 end 后面分片数据的总长，而 data_len 则只是分片数据段的长度。

以上我们简单介绍了 sk_buff 中一些常见成员变量的作用，接下来将讨论内核提供的一些操作 sk_buff 的函数(其实这样的函数在内核中有很多，这里我们只能挑一些比较常见的出来)：

- alloc_skb

函数用来分配一个套接字缓冲区 skb 以及 skb 所对应的数据区 data，其原型为：

```
<include/linux/skbuff.h>
static inline struct sk_buff *alloc_skb(unsigned int size, gfp_t priority)
```

参数 size 表示当前要分配的套接字缓冲区所对应的数据区的大小。函数的内部通过调用__alloc_skb 来做实际的内存分配，__alloc_skb 函数的核心实现为：

```
<net/core/skbuff.c>
struct sk_buff *__alloc_skb(unsigned int size, gfp_t gfp_mask,
                            int flags, int node)
```

```
{
    struct kmem_cache *cache;
    struct skb_shared_info *shinfo;
    struct sk_buff *skb;
    u8 *data;

    cache = (flags & SKB_ALLOC_FCLONE)
        ? skbuff_fclone_cache : skbuff_head_cache;
    ...
    /* Get the HEAD */
    skb = kmem_cache_alloc_node(cache, gfp_mask & ~__GFP_DMA, node);
    …
    size = SKB_DATA_ALIGN(size);
    size += SKB_DATA_ALIGN(sizeof(struct skb_shared_info));
    data = kmalloc_reserve(size, gfp_mask, node, &pfmemalloc);
    …

    /*
     * Only clear those fields we need to clear, not those that we will
     * actually initialise below. Hence, don't put any more fields after
     * the tail pointer in struct sk_buff!
     */
    memset(skb, 0, offsetof(struct sk_buff, tail));
    skb->truesize = SKB_TRUESIZE(size);
    atomic_set(&skb->users, 1);
    skb->head = data;
    skb->data = data;
    skb_reset_tail_pointer(skb);
    skb->end = skb->tail + size;
    …
    return skb;
}
```

因为 sk_buff 在 Linux 网络子系统中被分配和释放的频率非常高，所以内核采用 kmem_cache 的内存分配方式来分配 sk_buff 的空间，了解这种内存分配方式的读者一定会猜测到在系统初始化期间会有对 kmem_cache_create 的调用来产生这里的 skbuff_fclone_cache 和 skbuff_head_cache 两个全局变量，事实的确如此，在 Linux 系统初始化期间，通过调用链 sock_init()->skb_init()来产生这两个变量：

```
<net/core/skbuff.c>
void __init skb_init(void)
{
    skbuff_head_cache = kmem_cache_create("skbuff_head_cache",
                                          sizeof(struct sk_buff),
```

```
                                        0,
                                        SLAB_HWCACHE_ALIGN|SLAB_PANIC,
                                        NULL);
    skbuff_fclone_cache = kmem_cache_create("skbuff_fclone_cache",
                                        sizeof(struct sk_buff_fclones),
                                        0,
                                        SLAB_HWCACHE_ALIGN|SLAB_PANIC,
                                        NULL);
}
```

skb 所在的内存空间通过 kmem_cache_alloc_node 予以分配，本质上是以 slab 的内存分配方式来获得 skb 的地址空间(__do_cache_alloc)。

__alloc_skb 函数中接下来一个重要的步骤是调用 kmalloc_reserve 来分配 sk_buff 中的数据空间 data，kmalloc_node_track_caller 函数最终是调用__do_kmalloc 来为套接字缓冲区 sk_buff 的数据区分配空间，所以它们在物理地址空间是连续的，了解这点对网络设备驱动程序中正确使用 DMA 操作很重要。当然整个分配过程中充满了各种小技巧，各种对齐，各种性能优化意识等，但是这些细节对读者来说并不重要。函数的最后是对分配后的 sk_buff 对象进行必要的初始化，此处为了便于读者的理解，我们将 alloc_skb 分配出来的 skb 和 data 空间的关联用下图 12-7 来表示[12]：

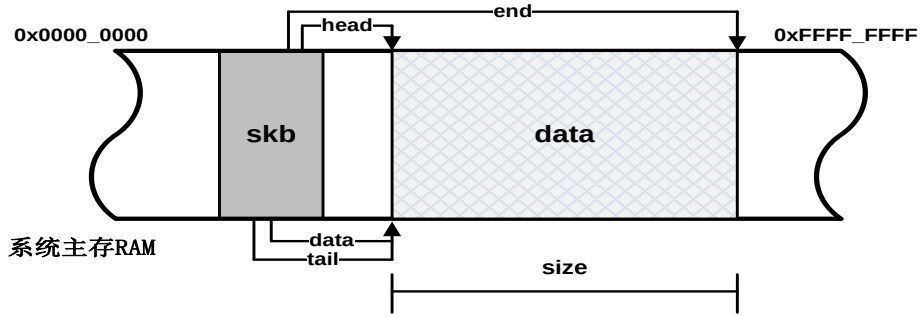

图 16-x alloc_skb 分配出的 skb 与 data 空间示意图

当一个套接字缓冲区对象 skb 在网络子系统的各协议层之间流动时，各层通过改变 skb->data 和 skb->tail 的值来获得当前层对应的协议数据首地址，而无需显式地进行内存复制等操作，下图 12-8 展示了一个 skb 所对应的数据包从 TCP 层传递到 MAC 层时，各协议层所对应的 skb->data 值的变化，如果是接收数据包，这个过程则正好相反：

为了便于各协议层的操作，在 skb 中的 transport_header、network_header 和 mac_header 成员则分别等同于网络数据包在 TCP 层、IP 层和 MAC 层时的 skb->data 值。

[12] alloc_skb 实际分配出的 data 空间比图中的 size 要大一些，因为在 data 空间底部会有一部分空间用来容纳 struct skb_shared_info 对象，但对驱动程序而言，可以不必理会这一部分的额外空间。

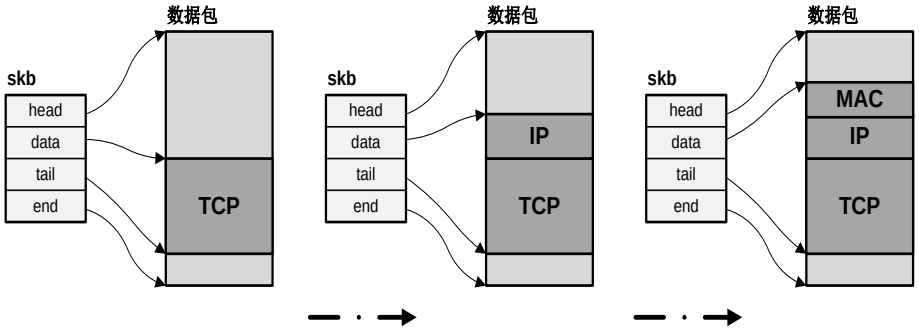

图 12-8 skb 所在的数据包依次通过 TCP、IP 和 MAC 层时 data 指针的变化

- netdev_alloc_skb

```
struct sk_buff *netdev_alloc_skb(struct net_device *dev, unsigned int length)
```

该函数主要用于在接收路径上为进接收到的数据包分配内存空间，其带有一个 struct net_device 类型的参数，意味着函数可以为某一个指定的网络设备对象分配 skb 及数据空间，另一个需要注意的是该函数在调用__netdev_alloc_skb 时指定 gfp_mask 为 GFP_ATOMIC，这意味着 netdev_alloc_skb 函数可以在中断上下文中被调用。其底层调用的是__alloc_rx_skb()。

- dev_alloc_skb

```
struct sk_buff *dev_alloc_skb(unsigned int length)
```

该函数是对上述 netdev_alloc_skb 的一个封装，只不过去掉了 dev 参数。

- napi_alloc_skb

```
struct sk_buff *__napi_alloc_skb(struct napi_struct *napi,
                                 unsigned int length, gfp_t gfp_mask)
```

该函数也是用于在接收路径上为某个指定的 napi 对象分配 skb，其下面也是调用__alloc_rx_skb()做实际的分配动作，至于参数 napi 的作用，代码里写得很清楚：

```
skb->dev = napi->dev;
```

所以实际上这个函数与 netdev_alloc_skb 本质上是一样的。

- kfree_skb 与 dev_kfree_skb

这两个函数的代码实现完全是一样的，都是通过__kfree_skb 来释放 skb 以及其对应的数据区所占有的内存空间，__kfree_skb 的主要操作分为两部分：一是调用 kfree 函数释放 skb 所对应的数据空间，二是通过 kmem_cache_free 来释放 skb 对象所占据的空间。之所以将同一个代码实现写成两个独立的函数，按照内核代码注释中的说法：非正常情形导致当前分组 skb 必须被丢弃时，使用 kfree_skb。而 dev_kfree_skb 实际上是一个宏，真正调用的是 consume_skb，按照函数名字面上的意义，表示这个分组 skb 被正常消费掉了。

- dev_kfree_skb_irq、dev_consume_skb_irq、dev_kfree_skb_any 以及 dev_consume_skb_any

这四个函数最终都会使用__kfree_skb()来释放分组所占用的内存空间（当然是在该分组没有任何引用的情形下才会发生），区别只是在于调用它们时所处的上下文环境，从这个角度可将上述四个函数归结为两大类：1 和 2 本质是一样的，这里不妨称为 A 组，3 和 4 本质上也是一样的，称为 B 组。

A 组的函数用在中断上下文环境中，其目的是尽快退出所谓硬中断的处理流程，此处不妨看看代码中的实现：

```
<net/core/dev.c>
void __dev_kfree_skb_irq(struct sk_buff *skb, enum skb_free_reason reason)
{
    unsigned long flags;

    if (likely(atomic_read(&skb->users) == 1)) {
        smp_rmb();
        atomic_set(&skb->users, 0);
    } else if (likely(!atomic_dec_and_test(&skb->users))) {
        return;
    }
    get_kfree_skb_cb(skb)->reason = reason;
    local_irq_save(flags);
    skb->next = __this_cpu_read(softnet_data.completion_queue);
    __this_cpu_write(softnet_data.completion_queue, skb);
    raise_softirq_irqoff(NET_TX_SOFTIRQ);
    local_irq_restore(flags);
}
```

因为在当前的上下文中(中断 hardirq 部分)处理器无法接收到新的中断，所以它并不直接在其内部释放 skb，而是将当前要释放的分组 skb 放到 softnet_data.completion_queue 所主导的链表中，softnet_data.completion_queue 是个 per-cpu 型的变量，所以每个 cpu 都会有这样一个队列用以存放要释放的 skb，真正 skb 所对应内存空间的释放则要延后到软中断处理函数 net_tx_action()中，因此我们看到上面的代码里有对 raise_softirq_irqoff(NET_TX_SOFTIRQ)的调用。内核之所以推迟释放的操作，原因是释放一个分组会占用较长的 cpu 时间，而一个分组的释放操作显然也不能算是一个非要放到硬中断环境下去处理的关键操作。

B 组的函数就很智能了，它可以自行判断当前调用该函数时的上下文环境，如果是非中断上下文，则最终会调用__kfree_skb()释放分组空间。不妨看看这段代码，来看看其智能是如何实现的：

```
<net/core/dev.c>
void __dev_kfree_skb_any(struct sk_buff *skb, enum skb_free_reason reason)
{
    if (in_irq() || irqs_disabled())
```

```
        __dev_kfree_skb_irq(skb, reason);
    else
        dev_kfree_skb(skb);
}
```

恩，原来判断的秘诀就是使用 in_irq()和 irqs_disabled()，所以看到这里也就没多少神秘感了。

- skb_put

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

该函数通过向后移动 skb->tail 的值从而在原来的 tail 和新的 tail 之间开辟出了一个新的空间。下图 12-9 展示了调用 skb_put(skb, 64)前后 tail 指针的变化，可以看到 skb_put(skb, 64)在原来数据块的尾部拓展了一大小为 64 字节的空间：

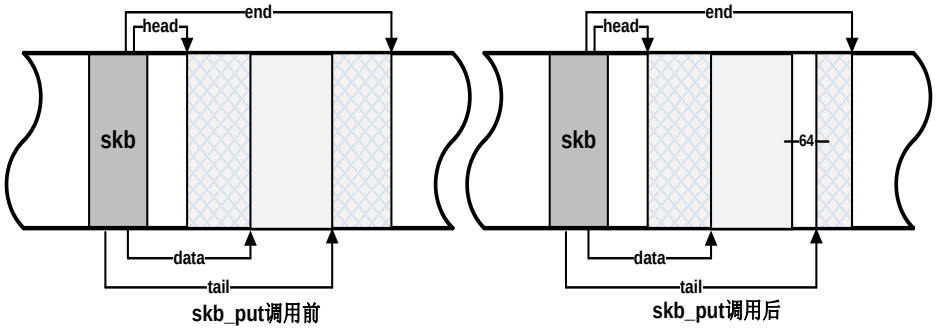

图 12-9 skb_put(skb, 64)调用前后对比

在将 skb->tail 移到新的位置后，老的 tail 值将作为返回值由函数 skb_put 返回。

- skb_push

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

与 skb_put 相反，该函数将向前移动 skb->data 值，skb->data -= len，这样将在原数据块的头部拓展出一大小为 len 的空间，函数将移动后的 skb->data 值返回。

- skb_headroom

```
unsigned int skb_headroom(const struct sk_buff *skb)
```

函数返回 skb->data 与 skb->head 之间的空闲空间大小，也即 skb->data - skb->head。

- skb_tailroom

```
int skb_tailroom(const struct sk_buff *skb)
```

函数返回 skb->tail 与 skb->end 之间的空闲空间大小，也即 skb->end - skb->tail。

- skb_reserve

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

该函数同时将 skb->data 和 skb->tail 增加参数 len 指定的字节数，这样将为 skb 的 head room 空间扩展了 len 个字节，该增加的空间从 tail room 里补充，相当于减少了 tail room 空间的大小。因此这种操作只允许针对空的 skb 对象，否则会导致 skb 数据区中的数据遭到破坏。

当然，内核提供的操作 skb 的函数远不止上面列出的这些，其他一些函数因为在网络设备驱动程序中用到的几率并不高，所以本书将不再一一列举这些函数。

16.7 数据包的发送

终于可以开始讨论数据包分组的发送这一有趣的话题了，也就是设备驱动程序中的 TX 路径。我们先从内核高层网络协议栈发送一个数据包所引发的调用链开始，然后逐级探讨这其中与设备驱动程序密切相关的每个重要模块的技术细节，包括原因、原理以及实现。

16.7.1 TX 路径中的调用链

下面这个内核发送网络分组的调用链[13]虽然并不完全，但对这里的讨论已经足够：

```
int dev_queue_xmit(struct sk_buff *skb)      -->
static int __dev_queue_xmit(struct sk_buff *skb, void *accel_priv)      -->
int sch_direct_xmit(struct sk_buff *skb, struct Qdisc *q,
                    struct net_device *dev, struct netdev_queue *txq,
                    spinlock_t *root_lock, bool validate)      -->
struct sk_buff *dev_hard_start_xmit(struct sk_buff *first, struct net_device *dev,
                                    struct netdev_queue *txq, int *ret)      -->
net_device_ops *ops->ndo_start_xmit(skb, dev)
```

总之经过层层调用，最终我们看到它进入到了设备驱动程序实现的 ndo_start_xmit 例程中。这里有几点提醒读者思考一下，我们看到在 dev_queue_xmit 的调用参数中，并没有 struct net_deivice 对象的身影，而到了 sch_direct_xmit()函数调用时，已经出现了用来发送当前分组的 dev 对象了，这期间的变化是如何实现的呢？换句话说，当上层代码有个 skb 要发送时，是如何与底层的 dev 设备关联起来的呢？

16.7.2 设备对象的发送队列

其实我们前面在讨论“网络设备的软件抽象”时，已经见过了这里所说的发送队列，至少当我们在这里讨论该话题时，你不应该感到太陌生。它就是 struct net_device 对象 dev 中的_tx 变量，我们不妨再回忆一下它在 struct net_device 中的定义：

---

[13] 此调用链是针对非 noqueue qdisc 的设备。

```
struct net_device {
    ...
    struct netdev_queue     *_tx ____cacheline_aligned_in_smp;
    ...
}
```

从该定义看，一个网络设备的发送队列可能不止一个，但是单个发送队列仍然是目前很多网络设备的主流标配，将来情况也许会慢慢改变。

按照正常的逻辑，此处我应该把 struct netdev_queue 的定义给摘录出来，但我真不想有大段粘贴代码的嫌疑，所以我假设大家手边都有最新版的 Linux 内核源码，因此我只把它最核心(至少在我看来，这些是与设备驱动程序以及下面的讨论密切相关的)部分摘录下来：

```
<include/linux/netdevice.h>
struct netdev_queue {
    struct net_device    *dev;
    struct Qdisc __rcu   *qdisc;
    struct Qdisc         *qdisc_sleeping;
    ...
    unsigned long        trans_start;
    unsigned long        trans_timeout;
    unsigned long        state;
    ...
}
```

通过这个 netdev_queue 定义可以发现，它的功能其实主要用作设备这一侧的队列管理的元数据，真正的队列操作是由其内部成员 qdisc 所指向的 Qdisc 模块来完成的，请注意此处的区别，netdev_queue 是驱动程序要关心的数据，而 qdisc 则属于内核的领域。我们后面马上就会讲到 qdisc 对象。只是出于行文方便，我将 netdev_queue 与 qdisc 混合到一起统称为发送队列，请务必仔细揣摩这里区别，一个是 qdisc 对象的链表式队列，另一个是 net_device 对象中的 netdev_queue，了解这两者之间差异非常有助于理解本章后续的很多内容。

netdev_queue 中其他的一些成员我们会在后续的文字中逐渐接触到，因此这里就不再赘述了。

现在读者也许会有一个疑问，为什么要引入这个发送队列呢？在回答这个问题之间，前读者先看下图 **x-xx,** 它展示了发送队列在内核高层网络协议栈与 NIC 设备间所处的位置：

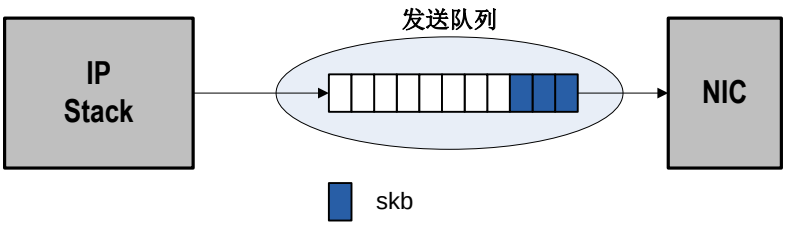

图 x-xx

图中椭圆所在的部分就是我们当前讨论的主角--发送队列，从图中看起来它像是一个数组，其实它在内核中的实现是一个链表结构。回到前面那个问题，为什么内核要引入发送队列这一功能模块呢？如果没有可以吗？ 答案是：当然可以。这个问题的本质是：内核协议栈和底层 NIC 设备的硬件发送逻辑是各自独立的任务，需要找到一种软件层面的实现方式，使得两个独立的任务可以最大程度地协调起来，达成性能的最优化。至少目前看来，发送队列的引入可以很好地解决这一问题，虽然它可能并不完美，比如由于队列的引入导致了数据包发送的延迟等。但是如果没有发送队列，对性能所带来的负面影响是显而易见的，比如考虑如下的场景：当 IP 协议栈有一个数据分组需要发送时，恰好此时 NIC 设备正忙，那么这个分组要么被丢弃，此时需要再重新生成一个分组，然后再穿越整个协议栈，要么协议栈等待 NIC 设备就绪，在 NIC 设备从忙到就绪这段时间内，协议栈只能等待，而无法执行新的任务。另外再比如，如果 NIC 设备成功发送了前一个分组，它现在有能力去发送下一个分组时，因为没有发送队列的存在，它不得不以某种方式向 IP 协议栈发出请求。如此无谓的周转下来无疑会影响网络子系统的性能。而引入队列就可以改观上述的这些问题：当 NIC 设备忙时，IP 协议栈可以把分组放到队列里，这样一旦 NIC 设备有能力再次发送分组时，就可以直接从发送队列里取得分组并在第一时间发送出去。在两个独立的非同步任务间建立些作为缓冲的仓储空间是一个非常自然而然的解决方案。

上图中只显示了一个发送队列_tx[0]，现实中可能存在着多个发送队列，比如所谓的多队列设备(multi-queue device)。

16.7.3 xps 与队列选择

首先声明，这一话题只对多队列设备有效，对只有一个发送队列的 NIC 设备而言，xps 与队列选择的问题就完全是浮云了。xps 是 Transmit Packet Steering 的简称，它其实跟设备驱动程序关系不大，更多地属于内核网络子系统的范畴。对于一个多队列设备而言，在硬件层面它可以同时独立地发送多个分组，因此设备驱动程序在生成该设备的对象时，也会为之产生相同数量的多个发送队列，dev->_tx[*N*]，其中 N 即为设备内部发送队列的数量。想象一下在 SMP 系统中，每个 cpu 上都可能产生需要发送的 skb，那么这个 skb 在有多个_tx 备选的情况下该如何进行选择呢？xps 的引入便是用来解决这一问题，它使用特定的方法将某个 cpu 与某一个_tx 进行 mapping，如此在 cpu 的 cache 使用上以及避免多个 cpu 同时访问某一_tx 时所带来的竞争问题上能获得额外的好处，这是内核引入 xps 主要目的。

我刚才提到，xps 与队列选择更多地属于内核范畴，那么对设备驱动程序而言意味着什么呢？意味着你基本不需要做太多的工作。不需要做太多的工作不意味没有工作，此处我们不妨总结一下：收发队列(_rx & _tx)是网络设备对象的基本属性之一，所以创建这些队列是你的驱动程序要做的工作之一，做这个工作之前你唯一需要确认的是你手头的 NIC 设备是否是个多队列设备，然后将这工作交给内核提供的 alloc_netdev_mq 这类的宏就可以了。接下来是此处我们谈到的 xps 和队列选择的问题，xps 是内核的机制，你可以忽略。对于队列选择的问题，内核提供了 netdev_pick_tx()函数来做这件事情，所以基本你也不用操心太多。只是 netdev_pick_tx()函数中有机会调用到与设备驱动相关的 ndo_select_queue 例程，幸运的是，大部分 NIC 设备的驱动程序都无需实现它，当前内核源码树中只有少量 FCoE 设备的驱动实

现了这一例程。再次明确，以上讨论的内容只限于多队列设备，因为单一队列设备根本就不存在这些问题，这个结论是显而易见的。

如果读者想了解 xps 如何将某一个 cpu 上产生的 skb 与某一队列进行 mapping，可以看看内核中的__netdev_pick_tx()函数，其本质是根据 skb 中 src 以及 dst 地址(所谓的 2-tuple)计算一个 hash 值(__skb_get_hash)，然后将此 skb 映射到某一发送队列之上(4-tuple 的话要在 2-tuple 的基础之上再加上 src 和 dst 的 port 值)。因此 xps 带来的结果是隶属于同一数据流的 skb 会被分配到同一个发送队列上。当然这里面没驱动程序什么事。

16.7.4 Qdisc-- Queueing discipline

Qdisc 部分其实与驱动的关系并不大，不过好歹也算是数据包发送流程中一个重要的节点，因此对那些想了解内核如何发送一个分组的过程的读者我还是强烈建议阅读，当然不感兴趣的读者跳过此节也不会有什么太大问题。

从字面的意思看，它为高层协议栈代码如何使用发送队列定义了某些规则。这样的描述只说出了 Qdisc 一部分的特性，因而并不全面，事实上除了队列使用策略之外，它还负责建立与维护这个发送队列 --- 一个真正的链表结构。Qdisc 在内核中的定义是：

```
<include/net/sch_generic.h>
struct Qdisc {
    int             (*enqueue)(struct sk_buff *skb, struct Qdisc *dev);
    struct sk_buff *    (*dequeue)(struct Qdisc *dev);
    unsigned int        flags;
    u32             limit;
    const struct Qdisc_ops  *ops;
    struct list_head    list;
    struct netdev_queue     *dev_queue;
    …
    unsigned long       state;
    struct sk_buff_head     q;
    …
}
```

这是一个典型的面向对象的 struct 定义，其队列相关的功能集合定义在 ops 成员中：

```
struct Qdisc_ops {
    struct Qdisc_ops    *next;
    const struct Qdisc_class_ops  *cl_ops;
    char            id[IFNAMSIZ];
    int             priv_size;

    int             (*enqueue)(struct sk_buff *, struct Qdisc *);
    struct sk_buff *    (*dequeue)(struct Qdisc *);
```

```
    struct sk_buff *    (*peek)(struct Qdisc *);
    unsigned int        (*drop)(struct Qdisc *);

    int             (*init)(struct Qdisc *, struct nlattr *arg);
    void            (*reset)(struct Qdisc *);
    void            (*destroy)(struct Qdisc *);
    int             (*change)(struct Qdisc *, struct nlattr *arg);
    void            (*attach)(struct Qdisc *);

    int             (*dump)(struct Qdisc *, struct sk_buff *);
    int             (*dump_stats)(struct Qdisc *, struct gnet_dump *);

    struct module       *owner;
};
```

这其中最常见的当属 enqueue 和 dequeue 两个方法，它们分别被用来向队列中添加一个 skb 和取出一个 skb。另一个成员 init 用来初始化当前的 qdisc 对象。

Qdisc 中的 limit 表示队列的容量，如果超出了队列容量，上层协议栈下发的 skb 通常将会被丢弃掉 (表现为直接 kfree 掉这个 skb)。Qdisc 中的 q 用来实现一个双向链表式的发送队列，state 成员自然是表示 qdisc 对象状态。dev_queue 成员指向驱动程序的 netdev_queue 对象，当然后者也有指向 qdisc 对象的成员，换言之，通过 netdev_queue 对象可以找到其所对应的 qdisc 对象，反之亦然。

我在本章早前的 16.4.2 小节“设备的激活”中曾提到当内核打算激活一个设备时，会给它分配 qdisc 对象[14]，此过程发生在 dev_activate()函数调用中。为一个设备分配 qdisc 对象可分为两个阶段：第一阶段是 attach_default_qdiscs，该阶段要完成的任务是调用 qdisc_alloc 分配一个新的 qdisc 对象，然后将该新对象赋值给 net_device 对象 dev 的_tx。如果 dev 的_tx 数量大于 1，那么内核为每个_tx 都分配一个新的 qdisc 对象:
netdev_for_each_tx_queue　i　{ dev->_tx[i]->qdisc_sleeping = qdisc;}
之所以是 sleeping qdisc，是因为由__dev_open 引发的设备激活过程中，设备的链路未必就绪: netif_carrier_ok(dev) != true. 这种情况下，先用 qdisc_sleeping 来记录新分配的 qdisc 对象，等将来设备的 netif_carrier_ok(dev)时，才真正将 sleeping qdisc 对象赋予 dev->_tx[]，也就是所谓的第二阶段：netdev_for_each_tx_queue　i　{ transition_one_qdisc();}
transition_one_qdisc()的主要工作就是将 sleeping qdisc 对象赋值给 dev->_tx[]:
rcu_assign_pointer(dev->_tx[]->qdisc, dev->_tx[]->qdisc_sleeping);

讨论到这里，我们在图 **x-xx** 的基础上进一步扩展出下图 x-xx：

[14] 当然，有些设备可能并没有发送队列，比如那个著名的 loopback 设备”lo”，它的 qdisc 对象是”noqueue”，qdisc->enqueue = NULL.

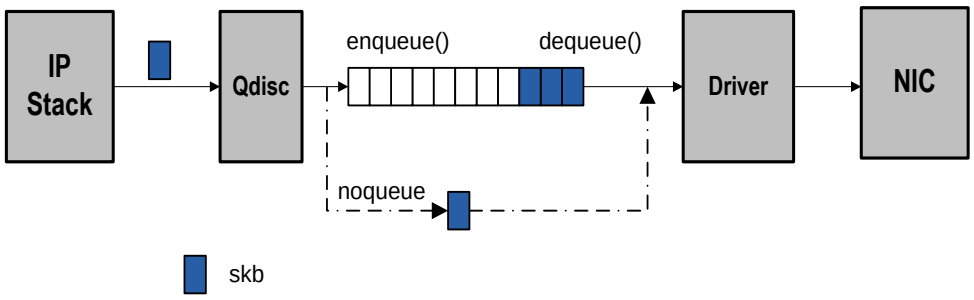

图 x-xx

上图基本概况了 Linux 内核发送一个网络分组的大体流程：当上层的网络协议栈有一个分组 skb 要发送时，它首先交给 Qdisc 模块，绝大数情形下，qdisc 对象建立并维护着一个双向链表，分组 skb 被放入到该链表构成的队列中(enqueue)，随后在内核 Qdisc 模块的调度下，分组被从队列中取出(dequeue)，交给驱动程序发送出去:dev->ops->ndo_start_xmit(skb…). 对于没有队列的设备，比如 loop back，那么流程就如同上图中虚线所代表的 noqueue 那样，协议栈下发的 skb 将直接交给设备驱动程序发送出去。

既然 Qdisc 的功能之一是对队列的使用规则进行定义，那么必然会有多种的选择，Linux 内核中就至少定义了以下几种 qdisc: pfifo_fast、pfifo 以及 bfifo 等，其中 pfifo_fast 是内核使用的缺省 qdisc。

作为一个纯软件层面的实现，在无需改动内核协议栈和设备驱动程序的前提下，Qdisc 定义的那些队列运营规则可以很轻易地被重新配置。Linux 下的 tc 工具可以用来为一个网络设备选择不同的 qdisc 对象。比如：

```
# tc -s qdisc ls dev eth1
qdisc pfifo_fast 0: root refcnt 2 bands 3 priomap  1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
 Sent 217699 bytes 962 pkt (dropped 0, overlimits 0 requeues 0)
 backlog 0b 0p requeues 0
# tc qdisc add dev eth1 root bfifo
# tc -s qdisc ls dev eth1
qdisc bfifo 8001: root refcnt 2 limit 1514000b
 Sent 0 bytes 0 pkt (dropped 0, overlimits 0 requeues 0)
 backlog 0b 0p requeues 0
```

16.7.5 pfifo_fast

pfifo_fast 是内核缺省使用的 qdisc 对象，在激活一个网络设备时，内核分配出 pfifo_fast 对象，并调用其初始化函数 pfifo_fast_init。从 pfifo_fast 初始化的代码可以看到，pfifo_fast 在其内部实现了三个独立的发送通道，用内核中的术语，分别是 band0、band1 和 band2。band0 具有最高优先级，band1 次之，band2 的优先级最低。当高层有一个分组需要发送出去时，pfifo_fast 根据 IP 头部的 ToS(Type of Service)字段决定该分组进入哪一个通道，在代码中则体现在 skb->priority 成员中。

放置在 band0 中的分组总是优先被分发给底层的硬件设备，band1 通道中的分组只有在 band0 中已经没有分组时才能获得分发的机会，band2 以此类推。

pfifo_fast 在将分组 skb 放进队列时(enqueue)，会根据 skb->priority 来将分组的优先级映射到对应的 band 上，该 band 记录到一个 bitmap 中(priv->bitmap)，然后将分组加入到该 band 所对应 list 的尾部。band 与 bitmap 的关系是：

band {0} --> bitmap = 1　　表示当前队列只有 band0 的分组
band {0, 1} --> bitmap = 3　　表示当前队列同时有 band0 和 band1 的分组
band {0,1,2} --> bitmap = 7　　表示当前队列同时有 band0、band1 和 band2 的分组
band {1, 2} -->bitmap = 6　　表示当前队列同时有 band1 和 band2 的分组
...

内核中定义有一个 bitmap2band 数组，用于将 bitmap 值映射到对应的 band 上：

```
static const int bitmap2band[] = {-1, 0, 1, 0, 2, 0, 1, 0};
```

在取出分组(dequeue)时，根据 priv->bitmap 的值以及 bitmap2band 来获得 band 值：

bitmap = 1 时，bitmap2band[bitmap] = 0， 所以 band = 0，表示当前队列只有 band0 分组时，因此自然只能从 band0 对应的 list 上取分组。

bitmap = 3 时，bitmap2band[bitmap] = 0，所以 band =0，表示当前队列同时有 band0 和 band1 的分组时，先从 band0 的队列中取分组。

bitmap = 7 时，bitmap2band[bitmap] = 0，所以 band =0，表示当前队列同时有 band0、band1 和 band2 的分组时，先从 band0 的队列中取分组。

bitmap = 6 时，bitmap2band[bitmap] = 0，所以 band =1，表示当前队列同时有 band1 和 band2 的分组时，先从 band1 的队列中取分组。

所以 qdisc 对象 pfifo_fast 可以将优先级高的分组优先发送出去。 在队列已满的情形下，后续的分组将直接被 pfifo_fast 丢弃：qdisc_drop(skb, ...)，对以太网设备而言，pfifo_fast 缺省的队列长度是 1500 个分组。

16.7.6 分组的发送

经过前面讨论的内容作为铺垫，现在可以仔细讨论一下内核中分组发送的具体技术细节。因为本书的主旨是讲设备驱动程序，所以我们不会过多讨论内核中协议栈部分的内容，而将重点放在与设备驱动程序关联密切的节点上，换句话说，我们讨论以下这些话题，是希望读者能了解设备驱动程序在发送一个分组时所处的上下文环境。

在此前之前，我想简单描述的一个问题是，skb 与网络设备的关联性。想象一下如下的场景，你的系统中有两个 NIC 设备：eth0 与 eth1，当应有层有一个分组要发送时，该选择哪个设备？这是协议栈中路由模块要解决的问题。总之，一个 skb 在穿越协议栈的过程时，经过路由模块，skb->dev 最终指向了用来发送该分组的网络设备对象。

分组继续下行，此时它需要选择一个发送队列_tx。因为它已经获得了最终发送它的网络设备对象：skb->dev，所以问题演变成在 skb->dev->_tx[]中选择其中的一个 netdev_queue 实例。对单队列设备而言(dev->real_num_tx_queues == 1)，该实例是 dev->_tx[0]，否则 xps 机制介

入[15]这种选择过程，参考本章稍早前“xps 与队列选择”小节中的描述。总之它有了一个 netdev_queue 的实例_tx，由此也获得了_tx 所对应的 qdisc 对象_tx->qdisc。

于是，分组来到了 Qdisc 模块前。此时内核有两种选择：如果 qdisc 对象是 noqueue，它将直接进入 dev_hard_start_xmit()，也就是将分组交给网络设备驱动程序处理。否则，内核将分组交给 qdisc。因为绝大部分设备是第二种情况，所以接下来讨论的就是这种情形。

Qdisc 原理虽然简单，但在实现的具体细节上还是有很多需仔细考量的地方，比如各种可能的并发、性能以及底层 NIC 设备发送能力等等。下图 x-xx 显示了分组发送时一个大体的流程图：

首先，如果当前 qdisc 对象的状态是 deactivated 时，分组将直接被 kfree 掉。在网络设备被 close 或者是 deactive 时，qdisc 对象将会处于这种状态，因此此种情形下分组将不会继续下行，否则就是在做一件毫无意义的事情。

其次，如果同时满足以下三个条件，那么分组将不会经过 qdisc 队列的 enqueue 和 dequeue 流程而直接交由设备驱动程序处理(sch_direct_xmit)，也就是越过了 Qdisc：
条件 a，qdisc 对象可以被 by-pass：q->flags & TCQ_F_CAN_BYPASS(内核缺省使用的 pfifo_fast qdisc 在其初始化函数中会设置该标志)
条件 b，qdisc 当前的队列长度为 0：q->q.qlen = 0.
条件 c，qdisc 当前还没有处于 RUNNING 状态：q->__state & __QDISC___STATE_RUNNING = 0;

三个条件同时成立表明，当前下行的分组正在进入一个空闲的 qdisc 对象中，既然队列里并无其他分组等待处理，那么就没有理由让分组进入 enqueue 和 dequeue 流程去浪费时间。

[15] 前提是配置内核时设置了 CONFIG_XPS，否则有另外的解决方案，参考__netdev_pick_tx()函数。

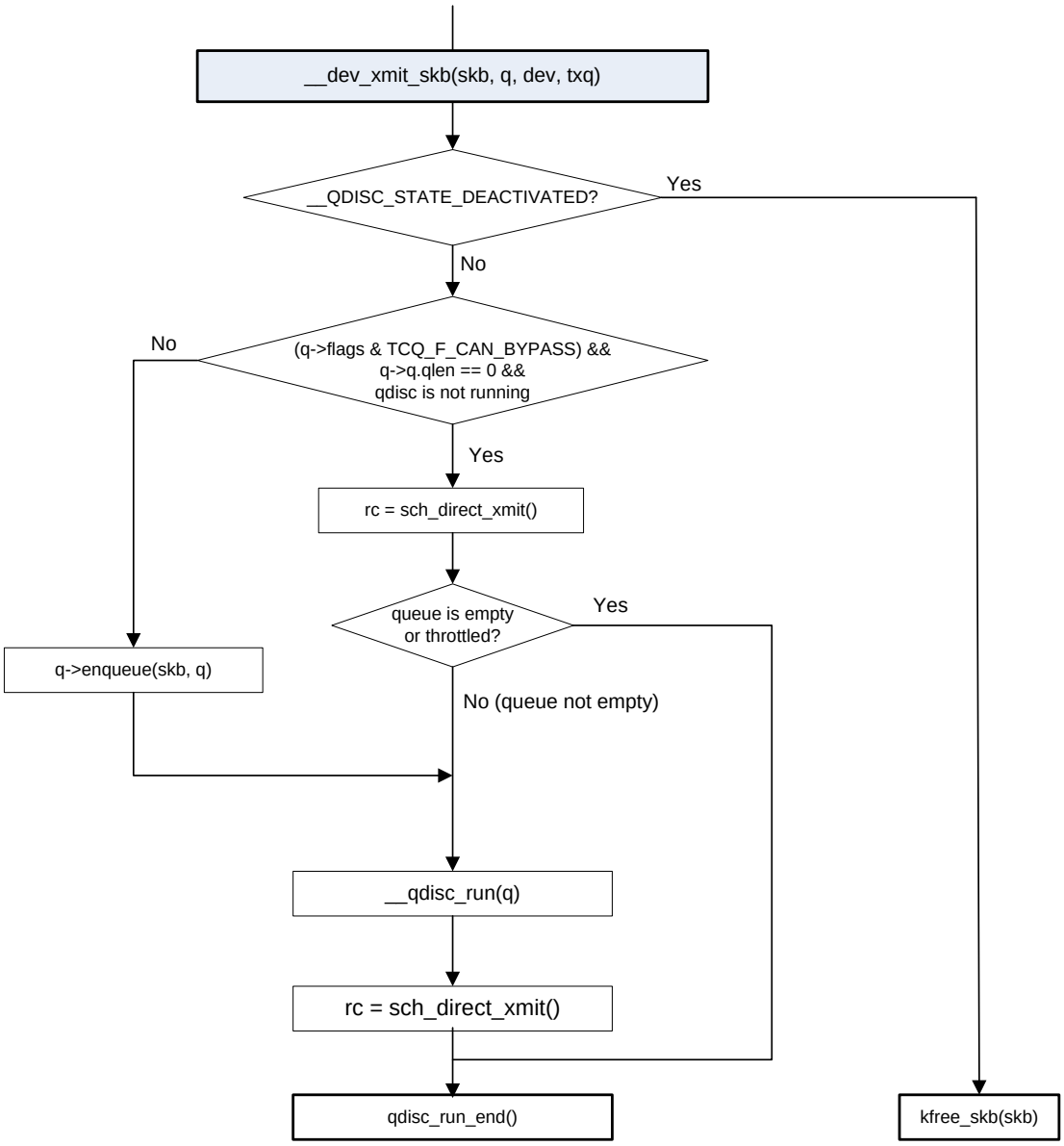

图 **x-xx**

sch_direct_xmit()被执行时的上下文环境是：qdisc 处于 RUNNING 状态且拥有 root_lock 锁。因为 qdisc 拥有队列，所以 sch_direct_xmit 不应该一直拥有 root_lock 锁，否则在其执行过程中，后续到来的分组将无法进入队列。因此 sch_direct_xmit 在调用网络设备对象中的 ndo_start_xmit 之前会释放 root_lock，这在 qdisc 对象与设备侧之间的并发做了切分：在设备侧发送分组的同时允许 qdisc 对象接收高层协议栈的分组，而设备侧的并发保护则由 dev->_tx[]._xmit_lock[16]承担。因为内核在调用设备驱程序的 ndo_start_xmit 例程之前已经拥有了 dev->_tx[]._xmit_lock 锁，因此驱动程序在实现这一例程时就无需再考虑并发的问题。

sch_direct_xmit()函数在上图 **x-xx** 中出现了两次，这是一个很重要值得仔细研究的函数。事实上它是 qdisc 对象对设备驱动程序发起 ndo_start_xmit 例程的最后入口点，也就是说 qdisc

[16] 早期的内核允许设备驱动程序在 dev->features 上设置 NETIF_F_LLTX 位来做免锁操作，但是现在的内核已经不再建议设备驱动程序使用该标志位，换言之，NETIF_F_LLTX 过时了。

对象中所有的分组最终都将通过它交给驱动程序处理。它的返回值如果为 0，有两种可能性，一是 qdisc 队列已经被排空，表明队列中所有分组都已经交给了驱动程序，另一个可能是，设备侧队列已经被 throttled，对于这种情况，我们在稍后的队列流控机制小节再详细讨论。如果 sch_direct_xmit()返回值大于 0，表明当前 qdisc 队列不为空。

正常情况下，网络设备会成功发送出当前的 skb，然后返回 NETDEV_TX_OK(0x00)这个值。当然也会有意外情况发生，比如网卡很忙，暂时没功夫搭理你，那么设备驱动的 ops->ndo_start_xmit()可能会返回一个 NETDEV_TX_BUSY(0x10)回来，这种情形说明 skb 并没有被网络设备成功发送出去，此时内核不会丢弃该分组，而是调用 dev_requeue_skb()将 skb 放入到当前 qdisc 对象的 gso_skb 成员中，并更新当前发送队列的一些统计值，等待下一次被调度发送的机会。这也是引入队列机制的好处之一，否则高层协议栈需要重新产生一个分组并要再次穿越整个协议栈，这种开销显然很大。

sch_direct_xmit()在将分组交给驱动程序处理过程中，可能会有后续的分组进入 qdisc，此时分组将通过 enqueue 进入队列，这之后前面的 sch_direct_xmit 的处理或者已经结束(队列已经不再处于 RUNNING 状态)或者没有，无论怎样，__dev_xmit_skb 进入到__qdisc_run()所在的路径中。当然，如果 sch_direct_xmit()执行过程中没有新的分组进入到 qdisc 队列，那么 sch_direct_xmit()返回后 qdisc 对象将不再处于 RUNNING 状态，也就意味着本次的 __dev_xmit_skb 结束了。

__qdisc_run()的主要功能是将队列中的分组排空，也就是试图将队列中所有的 skb 都交给驱动程序发送出去，这是后文要提到的 qdisc 队列的调度函数。__qdisc_run 在排空队列时可能会将取出的多个独立的 skb 构造一个 skb 链表，也就是内核中所谓的 try_bulk_dequeue_skb()[17]，此 skb 链表将交给 sch_direct_xmit()去处理，后者会再取出 skb 链表中的每个 skb 元素，依次交给 xmit_one 来发送。__qdisc_run()被调用时的上下文环境是：qdisc 对象是 RUNNING 状态，同时它拥有 root_lock 锁。理解此处的上下文环境非常重要：在 qdisc 队列被某处理器调度期间，其他处理器将无法获得该 qdisc 对象访问权，自然也就无法向其队列中发送分组。为了保证在 SMP 环境下其他处理器能获得公平的发送数据包的机会，内核引入了一个发送配额：weight_p。该配额的缺省值是 64，其他的值可以通过用户空间/proc/sys/net/core/dev_weight 文件予以调整。如果当前处理器使用 qdisc 发出的数据包已经超出了配额，内核便会将其发送过程推迟到 NET_TX_SOFTIRQ 阶段，这样做的目的是及时将当前处理器拥有的 qdisc 锁释放，否则其他处理器上的进程将会迟迟得不到机会向 qdisc 对象发送分组。关于此点，在本章稍后的“net_tx_action”一节中我们会继续讨论。

__qdisc_run 对理解 Qdisc 模块的工作方式非常重要，所以此处再简单总结一下它的设计思想：当__qdisc_run 运行时，其所属的 qdisc 对象处于 RUNNING 状态并且拥有 root_lock 锁，__qdisc_run 会从它的 qdisc 队列中取出分组(skb = dequeue_skb(q, ...))，然后将 skb 交由 sch_direct_xmit 来处理。如果队列不为空并且取出的分组没有超出配额，那么它将一直重复 dequeue_skb，再 sch_direct_xmit 这一过程。

[17] 需要内核在配置时，设定 CONFIG_BQL。BQL(Byte Queue Limit)是内核版本 3.3 之后引入的一个新的特性，感兴趣的读者可自行研究该特性。https://lwn.net/Articles/469652/

16.7.7 发送队列的流控机制

上一小节我们讲到了 sch_direct_xmit 函数，qdisc 对象将分组 skb 交给它，然后由驱动程序实现的 ndo_start_xmit 例程将分组交给底层的硬件设备发送出去。

理想情况下网络数据包的发送也许很简单，在软件的控制下由建立好的 DMA 通道去传输数据就可以了，但是现实情况往往比较复杂，这意味着当 ndo_start_xmit 函数返回时，底层的硬件设备未必已经将驱动程序交给它的分组成功发送了出去。这里除了硬件设备的实际行为，还设计到驱动程序自身的设计逻辑。先说此处的第二个层面，也就是所谓驱动自身的逻辑，用通俗点话说就是：它的地盘它做主。因为 ndo_start_xmit 是驱动程序的地盘，所以它可以做得很“任性”：在它觉得“不爽”的时候，它可以直接 free 掉这个 skb，然后返回 NETDEV_TX_OK 告诉 qdisc，我已经发送好了哦。至于什么事会令驱动感到“不爽”，这个就属于家家都有本难念的经范畴了。有读者也许会问，它明明擅自 free 掉了那个 skb，还说它已经 NETDEV_TX_OK 了，这不是欺骗 qdisc 的感情吗，应该后果很严重吧？其实不然，无非丢了个数据包而已了，更高层的协议(比如 TCP)会帮助让重新发送的。应该是这样吧？！我 TCP/IP 的书读得不是很多，此处暂时这么说吧。

下面我们开始讲此处的第一个层面，硬件设备的发送层面。这个层面是真开始做事了，但是做事就可能意味着会做错，原因或者是 NIC 设备压力太大了，或者是硬件逻辑偶尔吃错药了，等等。对于硬件偶尔吃错药了这种情况，一般都有硬件中断处理例程来解决，这个稍后我们有专门一小节来讲这玩意。

所以我们先讨论 NIC 设备压力太大了这种情况 (其实除了压力大以外，还有一种情况是 NETDEV_TX_LOCKED，但是这种情况对驱动程序来说太罕见了，属于非主流，此处就不浪费笔墨了)，压力大就是 NETDEV_TX_BUSY，NIC 说我很忙啊，有限的带宽都被占用了，没法继续干了，怎么办？有读者可能会问，这是怎么个情况呢。好吧，我们将这个问题稍稍具体化一点：内核子系统上层组件在很短的时间里调用了大量的 ndo_start_xmit 函数，CPU 执行这个过程总是很快，但是网络设备将其内在存储区中的数据包发送出去就未必如此神速了，于是此种情况导致的一个显而易见的后果是，网络设备内部的存储空间会很快被消耗光，它再无能力去接收网络子系统高层所发过来的数据包，直到有分组被成功发送出去才能稍稍缓解一下这一尴尬的局面。

很显然，针对这种情况，网络设备驱动程序的框架中需要设计这样一种机制，即当设备驱动程序发现网络设备的内部存储空间暂时无法使用(NETDEV_TX_BUSY)时，可以通知网络子系统的高层不要再将数据包交给自己。显然这是一种软件层面的流控机制，使用它可以避免对 CPU 资源的无谓的浪费：既然内核已经提前得知下层的网络设备当前已经不具备将一个数据包成功发送出去的能力，那么它就没必要再去调用驱动程序中实现的 ndo_start_xmit 函数，否则它就真得太任性了。

一个更智能的内核行为也许可以通过对发送队列的仔细观察来洞悉底层硬件的工作状态（是不是感觉有点机器学习的感念在里面了呢），从而决定是否调用驱动程序的 ndo_start_xmit 函数来发送当前的分组，但是无论从逻辑上还是实现的复杂度上，让驱动程序主动告知内核要来得更加自然，因为没有谁比设备驱动程序自身更了解它所控制的硬件的行为及能力。驱动程序所要完成的这种流控机制显然需要与内核中网络子系统其他模块协同工作，内核为此

专门给设备驱动程序提供了这样一组函数 netif_stop_queue()和 netif_tx_stop_all_queues()。netif_stop_queue()是基础，该函数的主要作用是让设备驱动程序告诉内核的网络子系统高层(其实主要就是 Qdisc 模块了)：当前底层的网络设备硬件无法继续传输数据包，高层代码需要停止将数据包传递给它。这个函数的实现非常简单，我把它稍作改写：：

```
<include/linux/netdevice.h>
static inline void netif_stop_queue(struct net_device *dev)
{
    struct netdev_queue *dev_queue = &dev->_tx[0];
    set_bit(__QUEUE_STATE_XOFF, &dev_queue->state);
}
```

所以我们看到 netif_stop_queue 其实就是将 net_device 对象 dev 中的发送队列_tx[0]的 state 变量的__QUEUE_STATE_XOFF 位置 1，后者是一 netdev_queue_state_t 类型的枚举变量，用来表示 net_device 对象中队列的状态，netdev_queue_state_t 在内核中的定义为：

```
<include/linux/netdevice.h>
enum netdev_queue_state_t {
    __QUEUE_STATE_DRV_XOFF,
    __QUEUE_STATE_STACK_XOFF,
    __QUEUE_STATE_FROZEN,
};
```

netif_tx_stop_all_queues 函数则用于多队列设备，它对于设备对象中的每个_tx 成员都调用 netif_stop_queue。这里我们再次看到网络设备对象中_tx 的具体作用。

至于 netif_stop_queue 为什么可以让高层网络代码停止调用 ndo_start_xmit，其实只需要看看此前我们提到的 sch_direct_xmit 函数代码便可一目了然：

```
<net/sched/sch_generic.c>
int sch_direct_xmit(struct sk_buff *skb, struct Qdisc *q,
                    struct net_device *dev, struct netdev_queue *txq,
                    spinlock_t *root_lock, bool validate)
{
    …
    if (skb) {
        HARD_TX_LOCK(dev, txq, smp_processor_id());
        if (!netif_xmit_frozen_or_stopped(txq))
            skb = dev_hard_start_xmit(skb, dev, txq, &ret);

        HARD_TX_UNLOCK(dev, txq);
    }
    …
    if (dev_xmit_complete(ret)) {
```

```
            /* Driver sent out skb successfully or skb was consumed */
            ret = qdisc_qlen(q);
        } else {
            /* Driver returned NETDEV_TX_BUSY - requeue skb */
            if (unlikely(ret != NETDEV_TX_BUSY))
                net_warn_ratelimited("BUG %s code %d qlen %d\n",
                                     dev->name, ret, q->q.qlen);

            ret = dev_requeue_skb(skb, q);
        }
        …
        return ret;
}
```

在上述的函数中，与流控相关的代码出现在那个 if (!netif_xmit_frozen_or_stopped(txq))中，如果网络设备驱动程序调用了 netif_stop_queue 函数，那么当前网络设备对象 dev 中的 dev->_tx[0].state 上的__QUEUE_STATE_XOFF 位将被置 1，如此 netif_xmit_frozen_or_stopped 将返回真，因此 dev_hard_start_xmit()将没有机会被调用，这种情况下，当前要发送的 skb 会被 requeue 到队列中-- dev_requeue_skb(skb, q)，以便后续再蓄时待发。

与{netif_stop_queue，netif_tx_stop_all_queues}对应的另一组流控函数是{netif_start_queue，netif_tx_start_all_queues}，其作用已经被其名字清晰地标示了出来，不再赘言。

现实中的网络设备驱动程序需要根据实际情况决定何时调用 netif_stop_queue 和 netif_start_queue 函数，一个典型的情形是，当打开一个网络设备的接口时(此时设备驱动程序中的 ndo_open 函数被调用)，驱动程序需要在 ndo_open 的实现中通过 netif_start_queue 告诉内核，网络子系统高层代码可以调用 dev_hard_start_xmit 进行数据包的发送，与此相反，当一个网络设备接口被关闭时(此时设备驱动程序中的 ndo_close 函数被调用)，对应的驱动程序应该调用 netif_stop_queue 以通知上层代码。

流控中的另一组重要的函数是{netif_wake_queue，netif_tx_wake_all_queues}，读者如果理解了 qdisc 对象中队列与网络设备对象中_tx 成员在整个网络子系统中的功能划分，就很容易理解这个函数与 netif_start_queue 之间本质的差异，而不仅仅是代码中那几个明显的标志位设置方面的不同。

先从内核代码实现的角度，netif_wake_queue 不仅需要考察当前设备设备对象 dev->_tx[].state 的__QUEUE_STATE_DRV_XOFF 状态位，还需要考察当前设备所对应的 qdisc 对象的状态__QDISC_STATE_SCHED。netif_wake_queue 对这两个状态的操作逻辑是：如果当前设备发送队列的__QUEUE_STATE_DRV_XOFF 位被置 1，那么将清除之，紧接着，它考察当前设备所对应的 qdisc 对象的状态，也即__QDISC_STATE_SCHED 位有没有被置 1，如果没有则表明当前 qdisc 对象尚未加入到正运行它的 CPU 的 qdisc 对象调度队列 output_queue[18]中，函数于是把它加到该调度队列的尾部，同时调用 raise_softirq_irqoff(NET_TX_SOFTIRQ)来标志网络设备中断处理流程的 softirq。下图 **x-xx** 描述了 netif_wake_queue 的处理流程：

[18] 由一个 per-cpu 型的 struct softnet_data 变量来管理

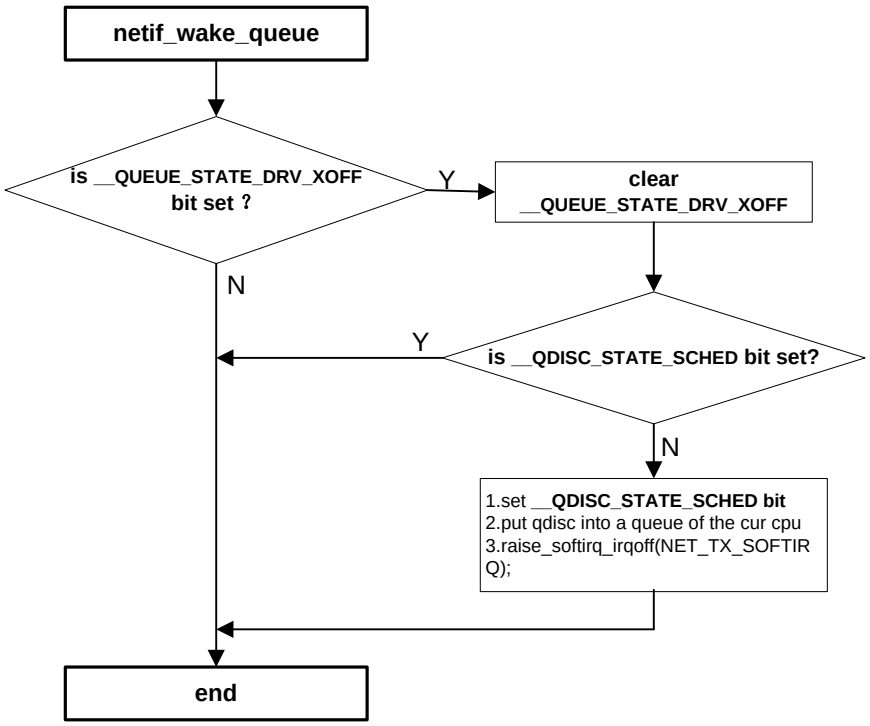

图 x-xx

弄懂这段内核代码的字面意义并不困难，可是这样设计的目的是什么呢？因为 netif_wake_queue 用来唤醒一个已经被驱动关闭了的 qdisc 队列，所以如果当前 qdisc 队列没有被关闭，那么自然就无需做额外的动作了，此其一。其二，高层协议栈代码将分组发送到 Qdisc 模块与驱动层面的分组发送是两个独立进行的任务，这意味着即便底层的 NIC 设备因各种原因不能发送分组时，协议栈的代码依然可以将分组交送给 qdisc 对象，也就是进入到它的队列中。如此，在 NIC 处于“黑障”[19]状态期间，qdisc 队列依然可以接收到分组。一旦 NIC 脱离这种“黑障”期，队列中的分组需要尽快发送出去，换言之，qdisc 队列需要在条件就绪时第一时间被排空。前面我们刚讨论过__dev_xmit_skb，这是由协议栈主动发起的能排空队列的操作，而 netif_wake_queue 则是另一种由驱动程序主导的排空队列的行为。相对于__dev_xmit_skb，netif_wake_queue 属于一种异步事件。

因此，内核中才说，netif_start_queue 是"allow transmit"，而 netif_wake_queue 则是"restart transmit"，注意此处的区别。至于 netif_wake_queue 如何利用 net_tx_action 这个 softirq 的 handler 来实现这个任务，本章稍后会有专门的小节来讨论这个问题。

现在一个值得关注的问题是，驱动程序需要在哪些情形下调用 netif_wake_queue 来排空队列呢？此处并无统一的标准，需具体问题具体分析。以下是两种比较典型的使用场景:

- 看门狗定时器(watchdog timer)超时，此种情形下驱动程序的 ndo_tx_timeout 函数会被内核调用以重新配置 NIC，使得因某种原因挂起的 NIC 可以重新开始工作。在网络设备

[19] 此处“黑障”的说法并不太准确，如果网卡在发生故障时连中断都无法向 cpu 报告，那么内核基本上就真的是回天无力了。

挂起的时间段内，在当前设备的 qdisc 队列上可能存在其他的传输尝试，因此当看门狗定时器超时重置 NIC 之后，驱动程序需要调用 netif_wake_queue 先开启队列，然后调度设备对应的 qdisc 对象以重新发送在设备挂起期间进入到发送队列的数据包。此种情况可以引申到因网络设备的硬件错误所导致的中断处理例程中，在那里驱动程序同样需要调用 netif_wake_queue 来重启 qdisc 队列中数据包的发送，因为在设备出错期间，可能有新的分组进入 Qdisc 模块。

- 当设备通过中断通知驱动程序可以进行数据包的传输时(而在此之前，驱动程序通常已经完成了：因设备无法进行数据包的传输而使用 netif_stop_queue 关闭了传输队列)，在中断处理例程中，qdisc 队列应该被唤醒，因为同样存在着设备队列被关闭期间 Qdisc 模块收到分组的可能性，所以此种情况下设备驱动程序也需要调用 netif_wake_queue 来重启 qdisc 队列的传输。

16.7.8 传输超时(watchdog timeout)

在本章前面 16.4.2 小节“设备的激活”中曾提到，当一个网络设备被激活时，内核会为之启动一个看门狗定时器，该定时器的触发间隔由 dev->watchdog_timeo 指定，如果驱动程序没有为该变量赋值，那么内核使用 5 * HZ 作为它的缺省值，也就是每隔 5 秒钟定时器函数被调用一次，内核实现的定时器函数为 dev_watchdog()。以上是我们继续下面内容讨论前的上下文环境。

将 watchdog 翻译成“看门狗”的确很形象，当然 watchdog 也不是网络设备驱动特有的概念，理论上凡是需要适时监测是否有不速之客来临的场景下，都可以祭出看门狗这一法宝来。作为一种故障监测手段，Linux 内核在网络数据包的发送路径上使用了看门狗。既然是看门狗，那么就必然需要定义什么样的异常情况才能让你的 watchdog 叫起来，以及叫起来之后你应该如何应对。

Linux 内核对触发看门狗的异常情况以及应对措施出现在 dev_watchdog()中：

```
<net/sched/sch_generic.c>
static void dev_watchdog(unsigned long arg)
{
    struct net_device *dev = (struct net_device *)arg;
    …
    if (!qdisc_tx_is_noop(dev)) {
        if (netif_device_present(dev) && netif_running(dev) && netif_carrier_ok(dev)) {
                …
                if (netif_xmit_stopped(txq) &&
                    time_after(jiffies, (trans_start + dev->watchdog_timeo))) {
                        WARN_ONCE(1, KERN_INFO "NETDEV WATCHDOG: %s (%s)…");
                        dev->netdev_ops->ndo_tx_timeout(dev);
                }
        }
}
```

由以上代码可见，看门狗的异常又叫传输超时(transmit timeout)，应对措施也很清楚，就是调用驱动程序里的 ndo_tx_timeout 例程。

在 dev_watchdog()中，如果要认定传输超时，则必须要同时满足如下两个条件[20]：

1. 当前网络设备的队列处于停止 (stopped) 状态。这主要是因为当设备驱动程序侦测到硬件在传输分组时资源不足(如设备已经用完了其内部所有的发送缓冲区时)时，会调用 netif_stop_queue()来停止发送队列。
2. 当前时间 (指调用 dev_watchdog 时的时间戳) 已经在最近一次发送分组的时间加上 dev->watchdog_timeo 的时间之后了。在 Linux 内核中，上述所谓的最近一次发送分组的时间的更新发生在 dev_hard_start_xmit()中，每次调用设备驱动程序的 ops->ndo_start_xmit()之后，如果一个分组被成功发送出去(rc == NETDEV_TX_OK)，txq->trans_start 将通过对 txq_trans_update(txq)函数的调用得到更新。

为什么要使用上面这两个条件呢，原因是设备驱动对 netif_stop_queue 的调用意味着底层硬件设备暂时没有足够的资源来发送分组，因此不得不通过驱动程序告知 Qdisc 模块暂缓将分组交给它处理。这种情形下如果硬件没有发生故障的话，在设定 dev->watchdog_timeo 时间窗口内(一般为 5 秒左右)，硬件设备有足够的时间去完成分组的发送工作。一旦网卡内部的这种工作成功完成，也即意味着硬件设备内部的资源耗尽的问题得到了解决，设备将以某种方式，比如中断，通知其驱动程序通过调用 netif_wake_queue 来重启传输队列，分组的传输将正常进行。此种情形下因 watchdog 无法满足 timeout 的条件所以不会执行设备驱动实现的 ndo_tx_timeout。但是如果当一直以固定频率被调用的 dev_watchdog()检测到距离上一次分组发送的时间已经过去了 dev->watchdog_timeo 设定的时间窗口值，系统就有理由认为底层的网络设备发生了故障，于是它将会调用 ops->ndo_tx_timeout()来处理这一状况，通常 ndo_tx_timeout()函数需要重启硬件设备以及 qdisc 传输队列等。所以 watchdog 只对非 noop qdisc 对象的网络设备有效。

总结一下就是：当网卡硬件资源耗尽不得不用 netif_stop_queue 通知高层暂缓分组发送时，内核会假设在 watchdog 设定的时间窗口内硬件资源耗尽的问题将会得以解决。所以，当在 watchdog 设定的时间窗口之外硬件设备仍然无法发送分组，那么就可断定发生了硬件故障，此时必须通知设备驱动程序来处理这种情况。

关于驱动程序中 ndo_tx_timeout 函数要完成的任务，此处并没有一个通行的规则，从逻辑上看它应该跟驱动程序中传输部分的代码联系比较紧密，常见的操作包括在接口的统计信息中记录本次的传输错误，调用 netif_wake_queue 函数以重启传输队列，有时甚至需要重新 reset 当前网络设备等等。

16.7.9 net_tx_action

net_tx_action 是网络设备驱动程序中 NIC 发送完分组产生的硬件中断的 softirq 部分的

[20] 此处我们假定当前设备的驱动程序没有使用 noop_qdisc 等一些通常都会被满足的条件，也即代码开始处的!qdisc_tx_is_noop(dev)以及 netif_device_present(dev) && netif_running(dev) && netif_carrier_ok(dev)

handler。其功能总体上分为两大块：一是当 NIC 设备将分组正常发送出去时，在 softirq 部分将这些 skb 对象所在内存空间释放出去。二是当一个 qdisc 对象用完其配额时，对于 SMP 环境而言，为了保证系统中其他处理器能获得公平的分组发送机会，内核会将正在当前 cpu 上调度执行的 qdisc 对象挂起，以便在后续合适的时间点上再次去调度它。内核挂起当前 qdisc 对象的做法是调用__netif_schedule。关于这点，我们在前面的 16.7.6“分组的发送”一小节中已经有过讨论。现在我们打算讨论__netif_schedule 的设计原理，这正好是 net_tx_action 要完成的第二大功能。

刚刚我们提到 qdisc 对象被挂起，然后再被调度等等，听起来似乎不是那么直白。qdisc 对象还可以被调度？没错。在目前单队列设备还是内核中的主流时，想象一下多处理器系统中分组的发送场景：假设每个 cpu 上都有一个网络应用程序在运行并都试图发送大量数据包，但是内存中的 qdisc 对象只有一个。这样竞争将不可避免：系统中每个处理器都试图获得 qdisc 对象的访问权以便可以将分组交给它。对这种并发的保护，qdisc 对象采用了__QDISC___STATE_RUNNING 以及 root_lock 两重措施。若某一个 cpu 幸运地获得了 qdisc 对象的访问权，理论上它可以一直占有它直到把 qdisc 队列里的分组全部发送完。但是如此一来，由于 qdisc 队列的调度函数__qdisc_run 执行时的串行化特点，其他处理器上的分组将不得不进行等待，等待越久，分组的延迟现象就会越严重。因此内核使用了配额(quota)这一做法试图缓解这种明显不太公平的做法。当一个 qdisc 对象用完了当前的配额，内核就必须__netif_schedule ()它。__netif_schedule ()之所以可以缓解其他处理器分组延迟的问题，是因为__netif_schedule 只是标示当前的 qdisc 对象的 state 为__QDISC_STATE_SCHED，然后再 raise 一个 NET_TX_SOFTIRQ 就可以快速结束了。结束了就意味着该 qdisc 对象将不再处于 RUNNING 状态，root_lock 也会被释放，因此其他处理器才有可能获得 qdisc 对象的访问权，将分组 enqueue 到 qdisc 队列中。

现在我们可以直观感受一下内核中__netif_schedule()的实现：

```
<net/core/dev.c>
void __netif_schedule(struct Qdisc *q)
{
    if (!test_and_set_bit(__QDISC_STATE_SCHED, &q->state))
        __netif_reschedule(q);
}
```

真正的功能被放到了__netif_reschedule 里，不过从上面的代码我们可以看到一个 qdisc 对象一旦被设置了__QDISC_STATE_SCHED 标志，就不会可能被再次 schedule 了，除非它已经被调度执行完，我们等下会看到 net_tx_action 里是如何清除掉该标志的。

__netif_reschedule()要做的事情也很简单，就是将当前的 qdisc 对象放到正运行它的 cpu 的 sd[cpu]->output_queue 队列里，然后 raise 一个 NET_TX_SOFTIRQ 就结束了。

当 NET_TX_SOFTIRQ 所标识的 net_tx_action 被运行时，它会试着去再次调度之前被挂起的 qdisc 对象：

```
<net/core/dev.c>
```

```
static void net_tx_action(struct softirq_action *h)
{
    struct softnet_data *sd = this_cpu_ptr(&softnet_data);
    ...
    if (sd->output_queue) {
        struct Qdisc *head;

        local_irq_disable();
        head = sd->output_queue;
        sd->output_queue = NULL;
        sd->output_queue_tailp = &sd->output_queue;
        local_irq_enable();

        while (head) {
            struct Qdisc *q = head;
            spinlock_t *root_lock;

            head = head->next_sched;

            root_lock = qdisc_lock(q);
            if (spin_trylock(root_lock)) {
                smp_mb__before_atomic();
                clear_bit(__QDISC_STATE_SCHED, &q->state);
                qdisc_run(q);
                spin_unlock(root_lock);
            } else {
                if (!test_bit(__QDISC_STATE_DEACTIVATED, &q->state)) {
                    __netif_reschedule(q);
                } else {
                    smp_mb__before_atomic();
                    clear_bit(__QDISC_STATE_SCHED, &q->state);
                }
            }
        }
    }
}
```

以上是 net_tx_action 的第二个功能，即再次调度 qdisc 队列。

net_tx_action 的另一个功能是释放分组 skb 所在的空间。如果驱动程序能确定分组已经成功发送了出去，那么该分组所占的内存空间就应该被释放掉。通常的使用场景是，分组 skb 被底层的 NIC 设备成功发送了出去，然后产生一个硬件中断通知处理器(这依赖于具体的硬件，此处并没有一个放之四海而皆准的规则，灵活应用即可)，该中断的 hardirq 部分的 handler 里调用 dev_kfree_skb_irq 或者 dev_kfree_skb_any 来释放分组，这些释放函数并没有真正做

__kfree_skb()，它只是将要释放的分组 skb 放到当前 cpu 的一个叫 completion_queue 队列中：
__this_cpu_write(softnet_data.completion_queue, skb)
然后在 raise 一个 NET_TX_SOFTIRQ，将分组的真正释放工作延迟到 softirq 阶段：
raise_softirq_irqoff(NET_TX_SOFTIRQ);

所以才有了 net_tx_action 的如下代码：
<net/core/dev.c>

```
static void net_tx_action(struct softirq_action *h)
{
    struct softnet_data *sd = this_cpu_ptr(&softnet_data);

    if (sd->completion_queue) {
        struct sk_buff *clist;
        local_irq_disable();
        clist = sd->completion_queue;
        sd->completion_queue = NULL;
        local_irq_enable();

        while (clist) {
            struct sk_buff *skb = clist;
            clist = clist->next;
            ...
            __kfree_skb(skb);
        }
    }
    ...
}
```

16.8 数据包的接收

相对于网络数据包的发送而言，内核在处理分组的接收工作时则要复杂许多。主要是因为接收路径上数据包的到达是一个随机的异步过程，因此不可能像数据包的发送过程那样，内核对分组接收往往做不到完全的控制。不过幸运的是，对于网络设备驱动程序而言，因为内核做了大量的工作，驱动部分的工作相对比较轻松了。

通常当网络设备成功接收到一个数据包时，需要通过中断的方式以引起驱动程序的干预。不同的网络设备在硬件逻辑设计上并不相同，在嵌入式领域，一些以太网控制器只需要驱动程序提供好 DMA 的描述符，在其他相关的功能寄存器都配置好的情况下，当网络设备成功地接收到一个数据包时，该数据包往往已经由硬件设备自动地启动 DMA 传输到了描述符所指向的系统主存空间中，同时硬件设备也会自动更新描述符中的某些成员，比如本次接收的数据包的长度等。

如同网络数据包的发送一样，驱动程序中接收数据包的实现方法依然依赖于具体的硬件设

备，但是通常驱动程序需要负责分配一个 skb 对象来容纳所收到的数据包，然后将 skb 传递到网络子系统的上层代码中，由后者负责释放该 skb 所占用的内存。DMA 的操作在这个过程中常常作为一个关键的步骤而存在，它将网络设备接收到的外部数据包从设备内存传输到系统内存中，现在所见到的几乎所有网卡设备都支持 DMA 操作，能够自发地将接收到的数据包传输到系统主存中。

我们的主题是要讲述设备驱动，所以从这个角度，我们更加关注驱动程序在整个数据包的接收过程中所要承担的任务。这部分的内容核心是数据包到达时，如何将分组从硬件设备取回到系统主存中，然后是如何将已经在主存中的 skb 交给高层网络协议栈处理。第一部分与硬件关系密切，是 NIC 设备 data sheet 应该描述的内容，第二部分主要是跟中断处理有关。中断处理又分为 hardirq 部分和 softirq 部分，驱动程序主要是实现 hardirq 部分(利用内核提供的基础设施)，softirq 部分由内核统一处理，也即是 net_rx_action 函数。驱动程序在实现 hardirq 部分时，有两种方式将内存中的分组交个高层协议栈代码：netif_rx 和 NAPI. 接下来我们分别讨论这两种方式。

16.8.1 netif_rx 方式

早期的网络设备驱动很多采用 netif_rx 的方式将接收到的 skb 传递给高层协议栈，其实等下读者将会看到，netif_rx 本质上也是一种 NAPI，只不过它使用的是内核内建的一个名为 backlog 的 NAPI 对象。这个 backlog 对象的初始化发生在 net_dev_init 中：

```
<net/core/dev.c>
static int __init net_dev_init(void)
{
    …
    for_each_possible_cpu(i) {
        struct work_struct *flush = per_cpu_ptr(&flush_works, i);
        struct softnet_data *sd = &per_cpu(softnet_data, i);
        …
        sd->backlog.poll = process_backlog;
        sd->backlog.weight = weight_p;
    }
    …
}
```

系统中每个 cpu 都拥有一个 struct softnet_data 类型的实例 sd，每个 sd 都有一个 struct napi_struct 实例 backlog，该 backlog 对象的 poll 函数为 process_backlog.

netif_rx 基本的工作原理是，网卡每接到一个数据分组，都产生一个中断，在驱动程序实现的 hardirq 部分，会调用 netif_rx 将该分组放置到当前处理器(假设为 cpu1)的 input_pkt_queue 队列，也即：sd[cpu1]-> input_pkt_queue， 然后到了 softirq 阶段，由 net_rx_action 来处理 sd[cpu1]-> input_pkt_queue 上的分组，最终调用__netif_receive_skb 将从队列中取出的 skb 交给上层。

所以它的工作模式可以简单概括为一次中断处理一个分组，这对那些慢速 NIC 设备是比较合适的，现在随着高速以太网卡的出现，在极短的时间内网卡就可能接收到大量数据包，因此这种方式对性能的影响已经到了不能忽略的地步，于是演变出了我们下一小节要讨论的 NAPI 方式。

内核中 netif_rx 的原型定义是：
int netif_rx(struct sk_buff *skb)；

由此定义可以看出，在调用这个函数时，skb 实际上应该已经产生了。netif_rx 真正的工作由其内部调用的 netif_rx_internal 来完成：

```
<net/core/dev.c>
static int netif_rx_internal(struct sk_buff *skb)
{
    int ret;
    ...
    unsigned int qtail;
    ret = enqueue_to_backlog(skb, get_cpu(), &qtail);
    ...
    return ret;
}
```

上述代码片段中我删除了 RPS 部分，因为后面我会专门讨论这个话题。所以 netif_rx 的核心功能就是 enqueue_to_backlog. 关于这个函数的代码此处我就不再摘录了，只简单介绍一下 enqueue_to_backlog 要完成的主要任务。

enqueue_to_backlog 的一个功能是将分组 skb 放到 sd[cpu]-> input_pkt_queue 的链表中，另一个功能是调整 sd[cpu]->backlog.state。功能一很简单，无非链表操作而已，需要注意的是 sd[cpu] 上的 input_pkt_queue 有一个最大长度限制，内核缺省设置的最大长度是 netdev_max_backlog = 1000，当然该值可以通过/proc/sys/net/core/netdev_max_backlog 文件予以调整。当要向队列里新增加一个分组 skb 时，如果当前队列的已经超出了 netdev_max_backlog 所设定的值，那么这个分组将直接被丢弃：
sd[cpu]->dropped++;
kfree_skb(skb);

否则当前 skb 将加入到 sd[cpu]->input_pkt_queue 队列的尾部：
__skb_queue_tail(&sd->input_pkt_queue, skb);
如果该分组在尚未加入队列前，该队列为空且 sd[cpu]->backlog.state 上也没有设置 NAPI_STATE_SCHED 标志位，则需要把 sd[cpu]->backlog 这个 napi 对象加入到由参数 cpu 指定的处理器 poll_list 链表中：
____napi_schedule(sd, &sd->backlog)。____napi_schedule 会将 sd[cpu]->backlog 加入到 sd[cpu]->poll_list 链表尾部，同时 raise 了 softirq: __raise_softirq_irqoff(NET_RX_SOFTIRQ)。

下图展示了 netif_rx 调用链中 enqueue_to_backlog 完成后，某个 cpu0 上 poll_list 以及

input_pkt_queue 队列上所发生的变化[21]：

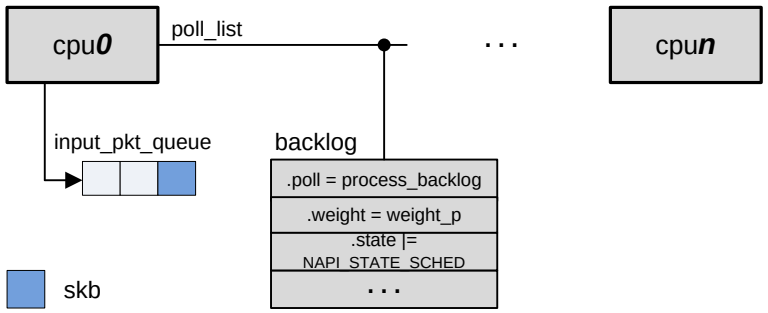

图 x-xx

截止到目前，从 NIC 设备接收到的分组 skb 只是放到了一个队列中，还没有真正开始向高层协议栈传递，这个工作要一直延迟到 softirq 部分。内核之所以这样设计，主要的原因是 hardirq 部分因为屏蔽了中断，所以应该尽快结束。softirq 部分在处理过程中本来就是开中断的，所以不会对系统的性能造成多大影响。这个我在本书前面“设备的中断与处理”一章中有过详细的论述。

完整地了解 netif_rx 的工作原理需要结合 softirq 部分的 handler 函数 net_rx_action 一起分析，不过因为接下来的 NAPI 方式也需要和 net_rx_action 一起协同工作，所以我也学着内核把 softirq 部分延迟到后面的小节中予以讨论。最后我用下面这张图来结束本节的讨论，该图

[21] 事实上对 SMP 系统而言，每个处理器上都有一个 softnet_data 变量，因此每个处理器上也都有一个 poll_list 和 input_pkt_queue。本书将某个 cpu 上的 sd 简单记为 sd[cpu]，这纯粹是为了行文的方便。

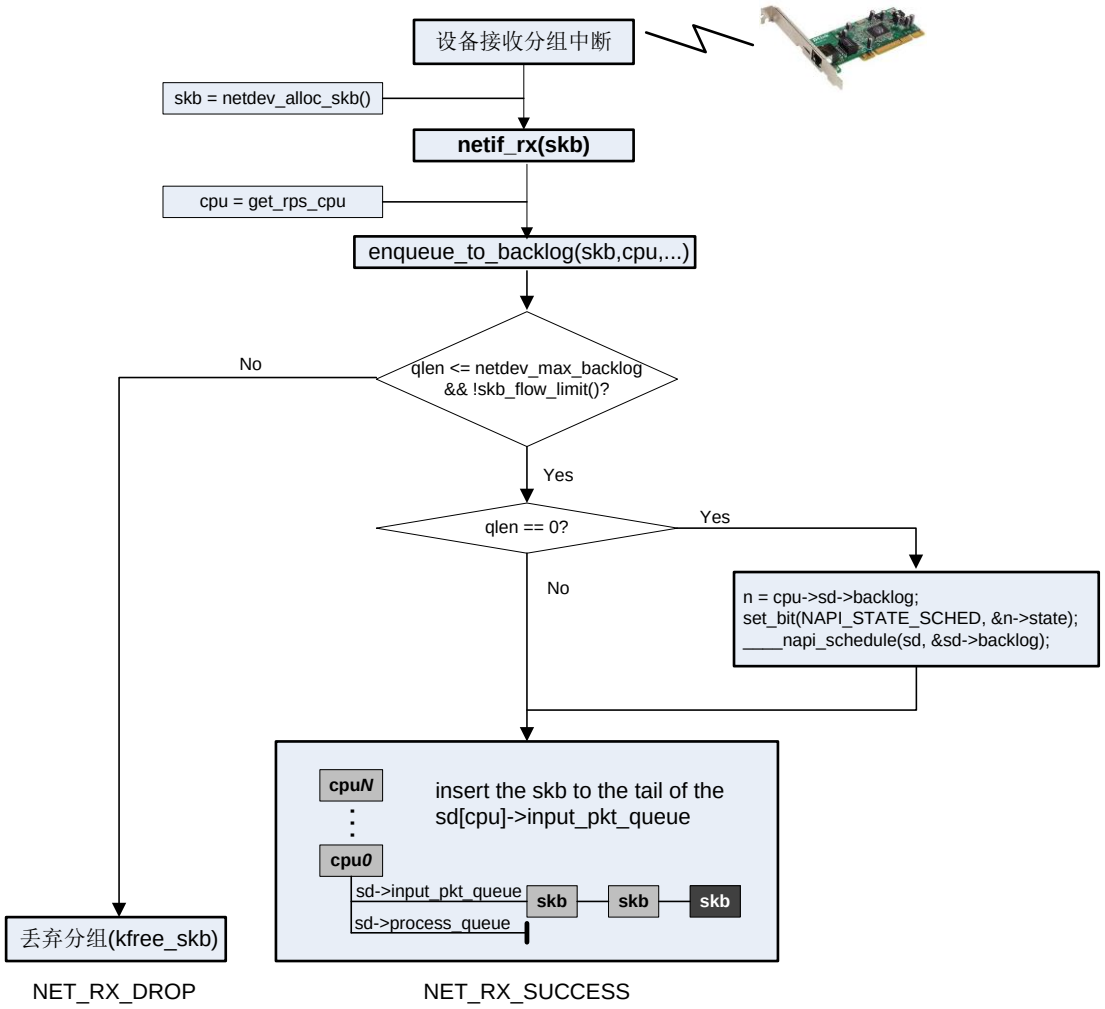

图 x-xx

揭示了 netif_rx 方式的工作流程。至于图中 sd->input_pkt_queue 与 sd->process_queue 之间的关系，将在 net_rx_action 一节中予以讨论。

16.8.2 NAPI 方式

如果仍然采用 netif_rx 方式，那么对于高速网络设备而言，系统中的处理器可能在短时间内接收到大量密集的网络数据包，如果每一个进入的数据包都向处理器产生一次中断请求，对这些请求的单独处理无疑会造成 CPU 资源的浪费甚至极端情形下导致系统的瘫痪。于是在这种情况下，一种称之为 NAPI(New API)的处理模式被引入到了内核中。现在的网络设备驱动程序已经大量使用 NAPI 这种方式。

NAPI 的设计思想其实是结合了中断与轮询的各自优势，虽然在设备驱动程序中，轮询的名声不大好，但是并非一无是处，比如在 NAPI 的机制中。NAPI 简单地说，就是当有数据包到达时将会触发硬件中断，在中断处理阶段关闭中断，系统对硬件的掌控将进入轮询模式，直

到所有的数据包[22]接收完毕，再重新开启中断，进入下一个中断轮询周期。显然在系统对硬件进行轮询期间，硬件可能会接收到大量进入的数据包，但是它们不会产生中断。

内核为支持 NAPI 机制定义了一个数据结构 struct napi_struct：

```
<include/linux/netdevice.h>
struct napi_struct {
        struct list_head        poll_list;
        unsigned long           state;
        int                     weight;
        …
        int                     (*poll)(struct napi_struct *, int);
        …
        struct net_device       *dev;
        struct list_head        dev_list;
        struct sk_buff          *gro_list;
        struct sk_buff          *skb;
        …
};
```

以上数据结构中驱动程序最常使用的成员是 poll_list、weight 和 poll，poll_list 用来将当前 napi 对象放置到 sd[cpu]->poll_list 链表中，weight 则表明了当前设备的权重(如果某一设备上长时间连续接收到分组，不采用配额的情况下将使得系统中其他设备迟迟得不到被轮训的机会)，poll 是一个函数指针，驱动程序需要实现这个函数，它将在内核对当前设备发起轮询时被调用。

支持 NAPI 的网卡设备驱动程序需要定义一个 napi_struct 类型的对象 napi，然后对该对象的成员进行初始化并把它加入到 net_device 对象的 napi_list 链表[23]中。内核提供了一个 netif_napi_add()函数供设备驱动程序调用以完成对 napi 对象的初始化工作：

```
<net/core/dev.c>
void netif_napi_add(struct net_device *dev, struct napi_struct *napi,
                    int (*poll)(struct napi_struct *, int), int weight)
{
        INIT_LIST_HEAD(&napi->poll_list);
        hrtimer_init(&napi->timer, CLOCK_MONOTONIC, HRTIMER_MODE_REL_PINNED);
        napi->timer.function = napi_watchdog;
        napi->gro_count = 0;
        napi->gro_list = NULL;
        napi->skb = NULL;
        napi->poll = poll;
```

[22] 其实是最多不超过配额所指定数量的数据包

[23] 这个链表主要是用来 for_each_netdev 时找到系统中所有网络设备上附加的 napi 对象。下文的 sd[cpu]->poll_list 才应是我们关注的重点。

```
    if (weight > NAPI_POLL_WEIGHT)
        pr_err_once("netif_napi_add() called with weight %d on device %s\n",
                    weight, dev->name);
    napi->weight = weight;
    list_add(&napi->dev_list, &dev->napi_list);
    napi->dev = dev;
#ifdef CONFIG_NETPOLL
    napi->poll_owner = -1;
#endif
    set_bit(NAPI_STATE_SCHED, &napi->state);
    napi_hash_add(napi);
}
```

就驱动程序而言，对 netif_napi_add()函数它最关心的是 poll 函数和 weight，这也是 netif_napi_add 函数中的参数。另外，weight 值不应该超过系统默认的 NAPI_POLL_WEIGHT=64 以及 netif_napi_add()将 napi->state 的 NAPI_STATE_SCHED 位置 1 了，也值得关注一下，因为一个处于 NAPI_STATE_SCHED 的 napi 对象是不能被再次成功 napi_schedule 的，所以驱动程序在用 netif_napi_add 初始化完了 napi 对象，通常会在其 ndo_open 函数中通过调用 napi_enable()来将 napi->state 上的 NAPI_STATE_SCHED 位清除掉。

接下来设备驱动程序需要调用 napi_schedule()将上述的 napi 对象加入到内核的 NAPI 框架体系中。在这个体系架构的实现里面，内核为系统中的每个 CPU 都分配了一个 struct softnet_data 对象 sd(per-cpu 型的变量）。napi_schedule()主要工作是将 napi->poll_list 节点加入到当前正运行这段代码 cpu 的 sd[cpu]->poll_list 链表的尾部，然后标志一个分组接收的软中断:__raise_softirq_irqoff(NET_RX_SOFTIRQ). 驱动程序通常在中断处理的 hardirq 部分调用 napi_schedule 函数，鉴于 napi_schedule()在内核中的代码实现，每个 cpu 的 sd->poll_list 链表中只允许一个 scheduled napi 对象的存在。既然这种链表的操作是建立在 per-cpu 型的 sd 对象上，所以只需要保证单一处理器上的互斥就足够了，因此内核在 __raise_softirq_irqoff(NET_RX_SOFTIRQ) 前后只使用了 local_irq_save(flags) 和 local_irq_restore(flags)，同样的处理也出现在 net_rx_action()中。

余下的主要工作则留给了 NET_RX_SOFTIRQ 标识的软中断处理程序：net_rx_action()。如果不了解这个函数的工作原理，你很难理解如何在驱动程序中去设定上述 napi 对象中的 weight 值以及如何去实现那个 poll 函数，因为这是 napi poll 函数运行的上下文背景。所以我们接下来会有专门的一个小节来讨论接收流程的这个 softirq handler.

现在继续讨论 NAPI 的 hardirq 部分，从上面的讨论看，驱动程序在通过 NAPI 这种方式接收数据包时，要做的工作其实并不多。如果对比一下 netif_rx(skb,...)这种方式，读者现在也许会有点小困惑：在 NAPI 方式下，谁来负责给接收到的分组分配一个 skb 空间呢？答案当然是设备驱动程序了，一般的流程是，NIC 设备先把分组放到它的内部存储空间中，然后在主存中通过 napi_alloc_skb()分配 skb 对象空间，然后再启动 DMA 将分组从设备空间传输到主存空间。这是个跟具体硬件相关的过程。

16.8.3 net_rx_action

在前面两节关于分组接收的 netif_rx 方式和 NAPI 方式，我都把接收到的分组最后如何处理推迟到了 softirq 部分。现在可以仔细探讨一下该函数到底做了些什么？

net_rx_action 的核心功能其实就是处理 sd[cpu]->poll_list 这个链表，这个链表我相信经过前面两节不断地铺陈渲染，读者对此早已烂熟于心。虽然实现细节上有些小的 trick，但是主体架构还是非常容易理解的。net_rx_action 用一个 for 循环来遍历当前 cpu 上的 sd[cpu]->poll_list 链表，每找到一个 napi 对象节点，则调用该对象上的 poll 函数：work = napi->poll(napi, weight)。

我们关注的重点当然是这个 poll 函数，因为驱动程序需要实现它。我并不很在意它的具体实现如何，因为轮询这种事嘛，都大抵不过如此，看看 datasheet 就足够了。所以这里我想谈一个比较有意思的东西：该函数的返回值。

如果驱动程序中的 poll 函数返回的 work (系指本次轮询中成功从网卡中接收到的数据包数量) < weight，表示该 NIC 并没有用完给定的配额，这背后暗含的意思是要么这是个"能量"不是很大的网卡，内部一次能容纳的分组数量比较小(比给定的配额还小)，要么这个 NIC 在接收方向的数据流量并不很大。基于此，网卡的驱动程序在这种情况下通常会调用 napi_complete() 将其对应的 napi 对象从 sd->poll_list 中摘除，同时清除 npai->state 上的 NAPI_STATE_SCHED 比特位，也就是让驱动退出 polling 模式。当然不退出该模式也是可以的，事实上通过在 napi->state 中设置 NAPI_STATE_NPSVC 位就可以达到此目的：

```
void napi_complete_done(struct napi_struct *n, int work_done)
{
    ...
    if (unlikely(test_bit(NAPI_STATE_NPSVC, &n->state)))
        return;
    ...
}
```

不过对于一个流量并不很大或者"小功率"的网卡而言，继续赖在 sd->poll_list 上并非明智之举，考虑一下如下的情形，假设系统中除了该"小功率"网卡外还有另外一张"大功率"网卡，两者在接收分组时都采用 NAPI 方式，此处为简单起见，我假设大小功率网卡的中断信号都 route 到了相同的 cpu 上进行处理，因而 sd[cpu]->poll_list 上就有对应这两块网卡的两个不同的 napi 对象节点，如此当 net_rx_action()被调用时，由于“小功率”网卡接收完全部分组后还赖在 sd[cpu]->poll_list 链表上，这将导致即便该"小功率"网卡并没有外来数据需要接收，在每次 softirq 阶段，它的 poll 函数依然会被调用，于是就出现了典型的"不干活还浪费纳税人的钱"的情况。所以大部分驱动程序都不会这么不识趣，它们会：通过 napi_complete()退出 polling 模式，并再次打开设备中断。这就为它下次真正接收到数据分组出现 RX 中断时再次调用 napi_schedule 创造了条件。

那么如果 work == weight 如何呢？两者相等，意味着分配的配额在一次 poll 中就用光了，也就意味着这很可能是块“能量”很大的设备或者当前数据流量很大，这也就意味着下次继续对该设备 poll 一下而有所收获的可能性极大，所以内核不希望该设备退出 polling 模式，于

是它采取的应对是将该设备所对应的 napi 对象移到 sd->poll_list 链表的尾部：

```
napi_poll()
{
    ...
    list_add_tail(&n->poll_list, repoll);
    ...
}
```

内核对这种状况的处理概况起来就六个字：不退出，排后面。

读者也许会问了，有无可能 work > weight，答案是可能。因为一旦进入设备驱动程序的 poll 函数中，那就是设备驱动程序的地盘了，配额是死的，程序是活的，网卡超过配额多收若干数据包并非大逆不道的事情。如果是设备驱动程序故意这么做，那就是一种比较流氓的做法了：你超配额做事，必然会挤掉别人的资源。所以除非写这个驱动程序的伙计本身就是个流氓，否则撑死了 work <= weight. 如果 work > weight，内核会产生一个 warnning:

```
napi_poll()
{
    ...
    WARN_ON_ONCE(work > weight);
    ...
}
```

我在内核源码中大体查了几个网卡的设备驱动，还没发现这种故意去当流氓的做派，希望读者也不会。内核在很多情形下需要依靠设备驱动程序编写者的自觉来保证系统运行的可靠性，所以还是素质问题，素质真得很重要。

所以，从设备驱动程序的角度出发，可以简单总结以下三原则：
1. 当 work < weight 时，调用 napi_complete()退出 polling 模式。
2. 当 work = weight 时，只需简单返回 work 即可，不会试图退出 polling 模式
3. 杜绝在 poll 函数中处理多于 weight 数量的数据包

上面我们主要讨论了 poll 函数，没有特别的针对某个 napi 对象。但接下来我们要特别讨论一个特殊的 napi 对象，那就是 sd[cpu]->backlog. 与 NAPI 方式不同，backlog 对象的 poll 函数为 process_backlog，这是内核实现的而不是驱动程序。我们前面在讨论 netif_rx 的时候，貌似卖了一些关子，现在一切都应该明朗才对了。

再回忆一下 netif_rx 的流程，接收分组中断的 hardirq 部分，驱动程序调用 netif_rx(skb, ...)，它会把该 skb 放到 sd[cpu]->input_pkt_queue 队列中，然后将 sd[cpu]->backlog 对象节点加入到 sd[cpu]->poll_list 链表中。接下来的 softirq 阶段，net_rx_action 将会调用到 backlog 对象中的 poll 函数：process_backlog. 后者主要从 sd[cpu]->process_queue 中取 skb，读者应记得 netif_rx 是把 skb 放到 sd[cpu]->input_pkt_queue 中，那 process_backlog 怎么会去处理 process_queue 呢？其实又是一个小小的 trick：当 process_queue 为空时，process_backlog

会把 input_pkt_queue 队列加到 process_queue 队列的尾部：
skb_queue_splice_tail_init(&sd->input_pkt_queue, &sd->process_queue);
所以你现在应该明白了两者的关系了吧，为什么用两个队列呢？那是要提高并发性。softirq 部分使用 process_queue 不就解放了 input_pkt_queue 了吗，这样 hardirq 部分可以继续将分组放入到 input_pkt_queue 里，而 softirq 部分则可同时处理 process_queue.

好吧，讲完这些之后，让我们再次把目光投向 net_rx_action，看看还能不能挖出点有趣的东西出来，别说，还真有。

在继续讨论之前，我们需要看看 napi 对象中的 state 成员，因为后续的叙述中需要用到此处的知识点。在内核中为 state 成员定义了如下几个状态：

```
enum {
    NAPI_STATE_SCHED,     /* Poll is scheduled */
    NAPI_STATE_DISABLE,   /* Disable pending */
    NAPI_STATE_NPSVC,     /* Netpoll - don't dequeue from poll_list */
    NAPI_STATE_HASHED,    /* In NAPI hash */
};
```

其中最常见的是前面两个，NAPI_STATE_SCHED 表示这是一个处于 pending 状态的 napi，所谓 pending 即指该 napi 对象已经被调度但尚未被执行，一个处于 pending 状态的 napi 对象是无法再次被 schedule 的，而一个无法被 schedule 的 napi 其实就是内核里面所谓的 disable，这可能会与 NAPI_STATE_DISABLE 字面上的意义相混淆，但事实上 NAPI_STATE_DISABLE 在内核中只是用来协助实现 NAPI_STATE_SCHED 这一状态的变迁。它可以阻止一个 napi 正在软中断中被处理时被强行从 sd->poll_list 链表中摘除。

听起来有点乱，此处不妨简单做个总结：如果一个 napi 对象的 state 上的 NAPI_STATE_SCHED 比特位被置 1，那么它就是一个 disabled 的 napi，这是内核中 napi_disable()和 napi_enable()的基础。现在可以回想一下我们之前刚刚提到的 netif_napi_add()，在那里 napi->state 上的 NAPI_STATE_SCHED 位被置 1，意味着如果就此罢手的话那么后续对此 napi 对象调用 napi_schedule()将不会把它加入到 sd[cpu]->poll_list 链表中。此前我们曾说过 netif_napi_add()是作为 napi 对象初始化之用，因此这里故事的通用发展情节将是：设备驱动的初始化部分(probe 阶段)调用 netif_napi_add()来初始化 napi 对象，在设备激活阶段(__dev_open)调用 napi_enable()来清除 napi->state 的 NAPI_STATE_SCHED 位，于是 napi 可被调度。在设备中断到来的处理函数(hardirq 部分)中调用 napi_schedule()将 napi 对象加入到 sd[cpu]->poll_list 链表，然后是软中断部分的开始，也即 net_rx_action()被调用，在那里 sd[cpu]->poll_list 中的每个 napi 对象节点都会被单独拎出来处理。虽然不同的网卡驱动程序都因自己各自独特的考虑而使得与这里总结的故事情节可能略有不同，但情况大抵如此。那么问题来了，一个已经被提交到 sd->poll_list 链表中但尚未被处理的 napi 对象可否被重复 napi_schedule()？经过前面的分析，答案显然是否定的，为什么要这么做，想想 NAPI 被引入的初衷，想想读者可以想通内核为何如此设计。

Linux 内核中对此的记载如下：

```
static inline void napi_schedule(struct napi_struct *n)
{
    if (napi_schedule_prep(n))
        __napi_schedule(n);
}
```

因为 if 语句不满足，所以__napi_schedule(n)不会被调用。

好吧，截止到目前，我们已经讲完了 netif_rx 和 NAPI 方式，现在可以总结一下这两者有何异同：

1. netif_rx 方式不会在中断处理的 hardirq handler 中屏蔽外部设备中断，而新的 NAPI 则会这么做。
2. 在为给接收到的分组分配 skb 空间方面，netif_rx 通过 netdev_alloc_skb()分配，NAPI 则一般在轮询函数里通过 napi_alloc_skb()的方式分配。(两者有何区别吗？)
3. netif_rx 将 backlog 这个 napi 对象加入到 sd[cpu]->poll_list 上之后，就不再从上面摘除（不过虽然 backlog 没有从 sd->poll_list 链表中摘除，但是因为 backlog->state 上的 NAPI_STATE_SCHED 在 processs_backlog 时如果 input_pkt_queue 队列为空时会被清除，所以其虽然在 poll_list 链表中，但在这种情况下其 poll 函数并不会被调用），而对 NAPI 而言，驱动程序可通过 napi_complete 将 napi 对象节点从 sd[cpu]->poll_list 链表中摘除。

16.8.4 RPS

RPS 是 Receive Packet Steering 首字母的缩写，这个词语怎么翻译呢，steering 有转向，轮船转向舵等意思。所以我们大体上可以将 RPS 翻译成“接收包的定向分流”，虽然这个翻译有点蹩脚，但是名称并不重要，重要的是了解它背后的含义。这里所谓分流，就是说系统从 NIC 接收到的数据包可以经 RPS 模块分发到不同的处理器上进行处理，所以其引入的主要意义在于提高数据包处理的并发性，因而可提升网络系统的性能。因此，RPS 主要是 SMP 系统中的一种机制，对于单处理器而言，RPS 特性的存在没有意义了。

RPS 这种技术其实跟设备驱动几乎没啥关系，这一节讲讲 RPS 纯粹是出于技术补充的目的。所以，对于那些不感兴趣的小伙伴们，可以直接无视这里的内容。不过我本人还是建议诸位有空时去琢磨一下，毕竟下面要讲的这些东西如果能静下心来仔细钻研进去，内容还是非常精彩有趣的，错过了真是种遗憾。我的叙述基本是蜻蜓点水式的，因为如果深挖下去本书的篇幅就要暴增了，而且其他小伙伴很可能又会说我离题万里。

在讨论 RPS 背后的原理时，我想先讲一下怎么使用 RPS 这种特性。如果要启用 RPS 机制，那么在配置内核时必须设置 CONFIG_RPS，对 SMP 系统而言，在最新版的内核中 CONFIG_RPS 缺省是打开的。打开了内核配置选项 CONFIG_RPS，只表明内核中加入了支持 RPS 的代码，但要要真正让 RPS 起作用，还需要通过用户空间的 sysfs 文件系统进行额外的配置，以使得多处理系统中接收到的包有处理器愿意处理它们，这个 sysfs 的入口点在我自己使用的系统中是：

```
root@dennis-ws:/sys/class/net/eth0/queues/rx-0/rps_cpus
```

缺省情形下 RPS 是被关闭的(rps_cpu=0)，在这种情形下哪个处理器接到分组到达的中断就由那个处理器来处理该分组。可以通过改变 rps_cpus 的值来指定系统中哪些 cpu 可以被用来处理接收到的网络分组。

现在可以探讨一下 RPS 背后的设计原理。还记得我们前面提到过的 process_backlog 函数吗？就是那个内核内建的 backlog 对象的 poll 函数，被用来处理 sd[cpu]->process_queue 中的分组。在它的实现里有这样一段代码片段：

```
static int process_backlog(struct napi_struct *napi, int quota)
{
    int work = 0;
    struct softnet_data *sd = container_of(napi, struct softnet_data, backlog);

    /* Check if we have pending ipi, its better to send them now,
     * not waiting net_rx_action() end.
     */
    if (sd_has_rps_ipi_waiting(sd)) {
        local_irq_disable();
        net_rps_action_and_irq_enable(sd);
    }
    …
}
```

而在 net_dev_init()中，有下面的代码片段：

```
for_each_possible_cpu(i) {
    …
#ifdef CONFIG_RPS
    sd->csd.func = rps_trigger_softirq;
    sd->csd.info = sd;
    sd->cpu = i;
#endif
    …
}
```

可以简单认为，在 RPS 机制下，每个进入系统的 skb 都试图通过 hash 方式与系统中某一 cpu 相关联。用个具体的例子，假设系统中有 4 个 cpu，分别是 cpu[0,1,2,3]，假设当前进入的 skb 按照 hash 算法应该关联到 cpu1 上，但由于设备中断的路由独立于 skb 的 hash 关联机制，这就有可能导致当前这个 skb 所引起的设备中断被送给了 cpu0，于是就出现了在 netif_rx_internal() 中通过 get_rps_cpu() 获得的 cpu(此例中为 cpu1) 与当前正运行 enqueue_to_backlog 代码的 cpu(此例中为 cpu0) 不一致的情形，这种情况下 enqueue_to_backlog()所操作的 sd 对象是隶属于 cpu1 的，也就意味着这个 skb 会被加入到 cpu1 的 input_pkt_queue 中，所以理应由 cpu1 上的 process_backlog()来处理之，因此为了达成此目的，内核在当前 cpu0 上运行的 enqueue_to_backlog()里就有了个 rps_ipi_queued()的调用：

```
static int rps_ipi_queued(struct softnet_data *sd)
{
#ifdef CONFIG_RPS
        struct softnet_data *mysd = this_cpu_ptr(&softnet_data);

        if (sd != mysd) {
                sd->rps_ipi_next = mysd->rps_ipi_list;
                mysd->rps_ipi_list = sd;

                __raise_softirq_irqoff(NET_RX_SOFTIRQ);
                return 1;
        }
#endif /* CONFIG_RPS */
        return 0;
}
```

针对上面的例子，该函数传递进来的 sd 应该属于 cpu1，而 mysd 则属于 cpu0,为方便叙述，分别称之为 sd1 和 sd0.于是得到下面的结果：

[cpu0] sd0->rps_ipi_list = sd1;

[cpu1] sd1->rps_ipi_next = sd0->rps_ipi_list = sd1;

这个调用与当前 cpu0 上执行的 process_backlog()中的 if (sd_has_rps_ipi_waiting(sd))引入的代码共同完成将后续对数据包的处理分流到 cpu1 上去处理(通过处理器间的 IPI 机制)。此处稍微多解释一下，当在 cpu0 上运行 process_backlog()时，后者计算出的的 sd 应为 sd0：

struct softnet_data *sd = container_of(napi, struct softnet_data, backlog);

所以当它执行 sd_has_rps_ipi_waiting(sd)时，因为前面刚提到过 sd0->rps_ipi_list = sd1，所以 sd->rps_ipi_list != NULL 成立，if 中的条件返回 true，所以调用 net_rps_action_and_irq_enable()，这是个非常有意思的函数，它利用 IPI 机制导致 cpu1 上产生一个中断，IPI 的中断在软件处理层面跟普通的设备中断并无本质的不同，也分为 hardirq 和 softirq 两部分。总之呢，经过 IPI 使得 cpu1 去执行 sd1->csd.func，在 net_dev_init()中我们已经看到这个 sd1->csd.func 实际上就是 rps_trigger_softirq()，于是触发了 cpu1 上的 softirq 部分，导致 cpu1 去执行 process_backlog()函数，于是刚才接收到的分组将在 cpu1 上被处理。所以当 NIC 接收到一个数据包 skb 时，先要计算出该 skb 的 hash 值，由该值决定此 skb 将放入到哪个 cpu 接收队列中等待处理。

所以 RPS 以及类似的相关技术用在多处理器系统中旨在提升数据包处理的并发度，并进而提升网络子系统的性能。读者也可以反过来思考该问题，假设没有 RPS 这种机制，那么同一时间将只有一个处理器用于处理数据包。(跟中断的 affinity 的关系是什么呢？)

skb 的 hash 值的计算根据协议的不同，可以是基于数据包中源和目的 IP 地址，也就是所谓的 2-tuple，或者是数据包中源和目的 IP 再结合上端口号，也就是所谓的 4-tuple。

感兴趣的读者可以参考内核中的 skb_get_hash()函数。skb 为什么要做 hash 呢？这主要是基于以下的一个基本常识，如果两个 skb 的 2-tuple 或者是 4-tuple 相等，那么意味着它们隶属于同一条数据流，因此将这两个 skb 绑定到同一个处理器可以充分利用到处理器的 local cache，进而提高系统性能。

16.9 中断处理

现在几乎所有网卡都支持中断操作模式，通过中断的方式，网卡可以在一个网络分组成功发送出去、成功接收到一个网络分组或者是硬件内部出现错误状态时通知 CPU 进行处理。但是网络设备驱动程序相对于一般其他设备有其自身的特殊性，比如在高负载的情况下，网络设备可能需要接收大量外部进来的数据包，这种密集型的数据包接收对于网络设备驱动程序中的中断处理程序而言是个极大的考验。设备驱动程序实现的中断处理例程必须完成最关键的操作而迅速返回以为下一次的中断到来做好准备，它应该将一些耗时的工作延迟到 softirq 中来完成。高速网络设备的中断处理是件很具有挑战性的工作，内核为此做了很多的工作。

其实我们在本章前面的内容中已经多次涉及到了网卡设备的中断处理，尤其是 softirq 部分。现在我们用一小节的篇幅来梳理一下设备驱动程序应该如何设计自己的中断处理逻辑。

大家都知道，设备驱动程序要处理来自设备的中断，首要的是通过 request_irq 来安装一个中断处理的 handler，至于 request_irq 函数的使用，本书前面的章节中已经讲了很多。这里我想讲一下网络设备驱动程序安装中断处理 handler 的时机，从理论上讲，只要在我们的设备驱动开始能接收中断前安装好中断处理的 handler 都是可以的，不过这里有个问题是，request_irq 是会占用系统资源的，所以过早地(比如在设备刚注册完成后)给网络设备安装中断处理函数并不是一个好主意，因为设备要一直等到激活后才可能被系统用来接收分组。因此，对网络驱动程序而言，通常的做法是在其 ndo_open 函数里来安装中断处理函数，在 ndo_close 函数里 free_irq 掉这个 handler，这主要是出于优化使用系统资源的目的。

在本小节最后，我想给出一个内核中具体的网络设备驱动程序中实现的中断处理的 hardirq 部分的 handler，让读者有个直观印象。因为我的 Linux 系统使用的是 realtek r8169 的 NIC，所以我就摘录这个设备的中断处理函数：

```
<drivers/net/ethernet/realtek/r8169.c>
static irqreturn_t rtl8169_interrupt(int irq, void *dev_instance)
{
    struct net_device *dev = dev_instance;
    struct rtl8169_private *tp = netdev_priv(dev);
    int handled = 0;
    u16 status;

    status = rtl_get_events(tp);
    if (status && status != 0xffff) {
        status &= RTL_EVENT_NAPI | tp->event_slow;
        if (status) {
```

```
                handled = 1;

                rtl_irq_disable(tp);
                napi_schedule(&tp->napi);
            }
        }
        return IRQ_RETVAL(handled);
}
```

在中断发生时，r8169 首先读它的中断状态寄存器(实现在 rtl_get_events 函数中)以确定中断发生的原因，如果是发送(TxOK | TxErr)或者接收(RxOK | RxErr)相关的中断，则处理之。可以看到它在使用 napi_schedule 调度一个 napi 对象前，调用了 rtl_irq_disable 来屏蔽掉 r8169 网卡的中断，这是与 netif_rx 方式是不同的。

16.10 本章小结

本章我们讨论了 Linux 设备驱动世界的第三大类设备---网络设备，因为内核代码中的网络子系统是个极其复杂的结构，所以这些技术细节不是本书要讨论的主题，但是作为讲述网络设备驱动程序内核机制的书籍，在讨论网络设备驱动程序相关主题时将不可避免地要进入到网络子系统的某些细节内幕。正是因为网络系统涉及到的范围极其宽广，所以本章主要从网络设备的数据结构抽象，网络设备注册以及网络设备实现的方法入手，试图让读者对网络设备驱动程序台前幕后有个比较深入的理解，在脑海中形成一个全画幅的网络设备的运作机制。

在本章中，我们将大量的篇幅集中在网络数据包的发送和接收上，因为这两种功能是一个 NIC 设备驱动程序要实现的核心功能。就驱动程序本身而言，发送与接收数据包的功能实现与硬件紧密相关，但是一般的流程是，对于发送路径，高层网络代码负责分配 skb 以存放网络数据包，然后该 skb 传到驱动程序的发送函数中，驱动程序的发送函数一般会使用 DMA 来将位于主存中的 skb 传输到网络设备内存中然后再由硬件逻辑发送出去，当硬件成功发送完一个数据帧时，通常以中断的方式通知处理器，相对应的驱动程序的中断处理函数被调用来出来这种情况。对于成功发送分组产生的中断，在其中断处理函数中会释放该分组所在的 skb。对于接收路径而言，当底层网络设备成功将一个分组传输到主存中时，它也会用中断的方式通知驱动程序，后者负责分配一个新的 skb 来容纳该新入的分组，然后通过 netif_rx 方式或者是 NAPI 方式将接收到数据包传递搞高层的网络协议栈代码。

内核做了很大的努力来优化分组发送与接收路径上的性能，比如在发送路径上的 Qdisc 模块，针对多队列设备的 xps 机制，GSO(Generic Segmentation Offload)等等，接收路径上的 rps，NAPI，GRO(Generic Receive Offload)等，这其中大部分跟网络设备驱动程序的关系不大，但是了解这些技术有助于加深了解网络设备在整个网络子系统框架中所处的位置。

本章同时也讨论了分组发送过程中的的流控机制，它们的目的是尽量在早期对一个可能不会成功发送出去的分组不使用驱动程序中的发送函数，这样可以避免系统资源的浪费，内核为此提供了 netif_stop_queue、netif_start_queue 以及 netif_wake_queue 等流控函数供驱动程序使用。

在本章的最后还讨论了对于高速设备而引入的 NAPI 机制，这种机制混合了中断与轮询操作模式的优点，避免了对于密集而至的每个分组都产生硬件中断的问题，因为这可能会导致 CPU 资源的大量消耗甚至极端情形下导致系统的崩溃。当前 NAPI 已经成了主流的用于接收网卡数据包的方式。但是如果驱动程序要使用这种机制，需要在其内部按照内核中 NAPI 机制的要求实现对应的函数。