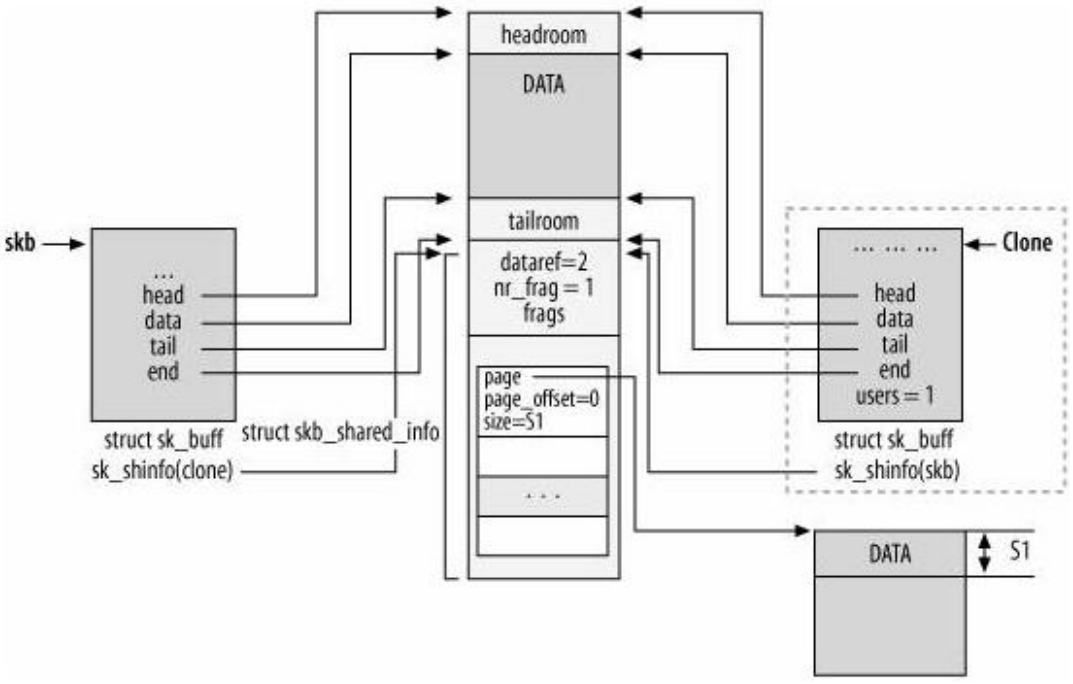
网络子系统在链路层的收发过程剖析

R.wen (rwen2012@126.com)

1），Skb_buff

```
/* To allow 64K frame to be packed as single skb without frag_list */
#define MAX_SKB_FRAGS (65536/PAGE_SIZE + 2)

typedef struct skb_frag_struct skb_frag_t;

struct skb_frag_struct {
	struct page *page;
	__u16 page_offset;
	__u16 size;
};

/* This data is invariant across clones and lives at
 * the end of the header data, ie. at skb->end.
 */
struct skb_shared_info {
	atomic_t	dataref;
	unsigned short	nr_frags;
	unsigned short	gso_size;
	/* Warning: this field is not always filled in (UFO)! */
	unsigned short	gso_segs;
	unsigned short  gso_type;
	unsigned int	ip6_frag_id;
	struct sk_buff	*frag_list;
	skb_frag_t	frags[MAX_SKB_FRAGS];
};
```

Skb 比较复杂的部分在于 skb_shared_info 部分，alloc_skb()在为数据分配空间的时候，会在这个数据的末尾加上一个 skb_shared_info 结构，这个结构就是用于 scatter/gather IO 的实现的。它主要用于提高性能，避免数据的多次拷贝。例如，当用户用 sendmsg 分送一个数组结构的数据时，这些数据在物理可能是不连续的（大多数情况），在不支持 scatter/gather IO 的网卡上，它只能通过重新拷贝，将它重装成连续的 skb（skb_linearize）,才可以进行 DMA 操作。而在支持 S/G IO 上，它就省去了这次拷贝。



2），网卡（PCI 设备的注册）

系统启动的时候，pci 会扫描所有的 PCI 设备然后根据注册驱动的 id_table，找到相匹配的驱动，实现关联。当找到匹配的驱动时，它会执行相关驱动程序中的 probe 函数，而网卡的 net_device 就是在这个函数里面初始化的并注册到内核的。

3），网卡链路状态检测

当网卡链路状态变化时（如断开或连上），网卡会通知驱动程序或者由驱动程序去查询网卡的相关寄存器位（例如在 timeout 时去查询这些位），然后由 netif_carrier_on/off 去通知内核这个变化。

```
void netif_carrier_on(struct net_device *dev)
{     // test_and_clear_bit - Clear a bit and return its old value
      if (test_and_clear_bit(__LINK_STATE_NOCARRIER, &dev->state))
            linkwatch_fire_event(dev);
      if (netif_running(dev))
            __netdev_watchdog_up(dev);
}

static inline netif_carrier_off(struct net_device *dev)
{
      //test_and_set_bit - Set a bit and return its old value
       if (!test_and_set_bit(_ _LINK_STATE_NOCARRIER, &dev->state))
            linkwatch_fire_event(dev);
```

}

这样，netif_carrier_on 会调用 linkwatch_fire_event，它会创建一个 lw_event 结构：

```
    struct lw_event {
    struct list_head list;
    struct net_device *dev;
};
```

并将这个结构初始化后（event->dev = dev;）加入到事件队列中：

```
        spin_lock_irqsave(&lweventlist_lock, flags);
        list_add_tail(&event->list, &lweventlist);
        spin_unlock_irqrestore(&lweventlist_lock, flags);
```

然后它调用 schedule_work(&linkwatch_work)由内核线程去处理这些事件。它最终由 linkwatch_run_queue(void)去完成这些处理工作：

```
list_for_each_safe(n, next, &head) {
        struct lw_event *event = list_entry(n, struct lw_event, list);
        struct net_device *dev = event->dev;
    …
        if (dev->flags & IFF_UP) {
            if (netif_carrier_ok(dev)) {
                dev_activate(dev);
            } else
                dev_deactivate(dev);
            netdev_state_change(dev);
        }
}
```

可以看到，它的最主要工作之一就是 netdev_state_change(dev)：

```
void netdev_state_change(struct net_device *dev)
{
    if (dev->flags & IFF_UP) {
        raw_notifier_call_chain(&netdev_chain,
                NETDEV_CHANGE, dev);
        rtmsg_ifinfo(RTM_NEWLINK, dev, 0);
    }
}
```

这个函数通知注册到 netdev_chain 链表的所有子系统，这个网卡的链路状态有了变化。就是说，如果某个子系统对网卡的链路状态变化感兴趣，它就可以注册到进这个链表，在变化产生时，内核便会通知这些子系统。

注意：a. 它只会在网卡状态为 UP 时，才会发出通知，因为，如果状态为 DOWN，网卡链

路的状态改变也没什么意义。

b. 每个见网卡的这些状态变化的事件 lw_event 是不会队列的，即每个网卡只有一个事件的实例在队列中。还有由上面看到的 lw_event 结构，它只是包含发生状态变化的网卡设备，而没有包含它是链上或是断开的状状参数。

4），数据包的接收

```
* Incoming packets are placed on per-cpu queues so that
 * no locking is needed.
 */
struct softnet_data
{
	struct net_device	*output_queue;
	struct sk_buff_head	input_pkt_queue;
	struct list_head	poll_list;
	struct sk_buff		*completion_queue;

	struct net_device	backlog_dev;  /* Sorry. 8) */
#ifdef CONFIG_NET_DMA
	struct dma_chan		*net_dma;
#endif
};
```

这个数据结构同时用于接收与发送数据包，它为 per_CPU 结构，这样每个 CPU 有自己独立的信息，这样在 SMP 之间就避免了加锁操作，从而大大提高了信息处理的并行性。

```
struct net_device	*output_queue;
struct sk_buff		*completion_queue;
```

这两个域用于发送数据，将在下一节中描述。

```
struct sk_buff_head	input_pkt_queue;
struct list_head	poll_list;
struct net_device	backlog_dev;
```

这三个域用于接收数据，其中 input_pkt_queue 与 backlog_dev 仅用于 non-NAPI 的 NIC，input_pkt_queue 是接收到的数据队列头，它用于 netif_rx()中，并最终由虚拟的 poll 函数 process_backlog()处理这个 SKB 队列。

poll_list 则是有数据包等待处理的 NIC 设备队列。对于 non-NAPI 驱动来说，它始终是 backlog_dev。

接收过程：

当一个数据包到来时，NIC 会产生一个中断，这时，它会执行中断处理全程。

(1), NON-NAPI 方式：

如 3c59x 中的 vortex_interrupt(),它会判断寄存器的值作出相应的动作：

```
        if (status & RxComplete)
            vortex_rx(dev);
```

如上，当中断指示，有数据包在等待接收，这时，中断例程会调用接收函数 vortex_rx(dev)接收新到来的包（如下，只保留核心部分）：

```
            int pkt_len = rx_status & 0x1fff;
            struct sk_buff *skb;

            skb = dev_alloc_skb(pkt_len + 5);

            if (skb != NULL) {
                skb->dev = dev;
                skb_reserve(skb, 2);        /* Align IP on 16 byte boundaries */
                /* 'skb_put()' points to the start of sk_buff data area. */
                if (vp->bus_master &&
                    ! (ioread16(ioaddr + Wn7_MasterStatus) & 0x8000)) {
                    dma_addr_t dma = pci_map_single(VORTEX_PCI(vp), skb_put(skb,
pkt_len),
                                                    pkt_len, PCI_DMA_FROMDEVICE);
                    iowrite32(dma, ioaddr + Wn7_MasterAddr);
                    iowrite16((skb->len + 3) & ~3, ioaddr + Wn7_MasterLen);
                    iowrite16(StartDMAUp, ioaddr + EL3_CMD);
                    while (ioread16(ioaddr + Wn7_MasterStatus) & 0x8000)
                        ;
                    pci_unmap_single(VORTEX_PCI(vp), dma, pkt_len,
PCI_DMA_FROMDEVICE);

                }
                iowrite16(RxDiscard, ioaddr + EL3_CMD); /* Pop top Rx packet. */
                skb->protocol = eth_type_trans(skb, dev);
                netif_rx(skb);
```

它首先为新到来的数据包分配一个 skb 结构及 pkt_len+5 大小的数据长度，然后便将接收到的数据从网卡复制到（DMA）这个 SKB 的数据部分中。最后，调用 netif_rx(skb)进一步处理数据：

```
int netif_rx(struct sk_buff *skb)
{
    struct softnet_data *queue;
    unsigned long flags;

    /*
```

```
         * The code is rearranged so that the path is the most
         * short when CPU is congested, but is still operating.
         */
        local_irq_save(flags);
        queue = &__get_cpu_var(softnet_data);

        if (queue->input_pkt_queue.qlen <= netdev_max_backlog) {
            if (queue->input_pkt_queue.qlen) {
enqueue:
                dev_hold(skb->dev);
                __skb_queue_tail(&queue->input_pkt_queue, skb);
                local_irq_restore(flags);
                return NET_RX_SUCCESS;
            }

            netif_rx_schedule(&queue->backlog_dev);
            goto enqueue;
        }
}
```

这段代码关键是，将这个 SKB 加入到相应的 input_pkt_queue 队列中，并调用 netif_rx_schedule()，而对于 NAPI 方式，它没有使用 input_pkt_queue 队列，而是使用私有的队列，所以它没有这一个步骤。至此，中断的上半部已经完成，以下的工作则交由中断的下半部来实现。

```
void __netif_rx_schedule(struct net_device *dev)
{
        unsigned long flags;

        local_irq_save(flags);
        dev_hold(dev);
        list_add_tail(&dev->poll_list, &__get_cpu_var(softnet_data).poll_list);
        if (dev->quota < 0)
            dev->quota += dev->weight;
        else
            dev->quota = dev->weight;
        __raise_softirq_irqoff(NET_RX_SOFTIRQ);
        local_irq_restore(flags);
}
```

netif_rx_schedule()就是将有等待接收数据包的 NIC 链入 softnet_data 的 poll_list 队列，然后触发软中断，让后半部去完成数据的处理工作。

注意：这里是否调用 netif_rx_schedule()是有条件的，即当 queue->input_pkt_queue.qlen==0

时才会调用，否则由于这个队列的长度不为 0，这个中断下半部的执行已由先前的中断触发，它会断续处理余下来的数据包的接收，所以，这里就不必要再次触发它的执行了。

总之，NON－NAPI 的中断上半部接收过程可以简单的描述为，它首先为新到来的数据帧分配合适长度的 SKB，再将接收到的数据从 NIC 中拷贝过来，然后将这个 SKB 链入当前 CPU 的 softnet_data 中的链表中，最后进一步触发中断下半部发继续处理。

(2), NAPI 方式：

```
static irqreturn_t e100_intr(int irq, void *dev_id)
{

    if(likely(netif_rx_schedule_prep(netdev))) {
        e100_disable_irq(nic);
        __netif_rx_schedule(netdev);
    }

    return IRQ_HANDLED;
}
```

可以看到，两种方式的不同之处在于，NAPI 方式直接调用__netif_rx_schedule()，而非 NAPI 方式则要通过辅助函数 netif_rx()设置好接收队列再调用 netif_rx_schedule()，再者，在非 NAPI 方式中，提交的是 netif_rx_schedule(&queue->backlog_dev)，而 NAPI 中，提交的是__netif_rx_schedule(netdev)，即是设备驱动的 net_device 结构，而不是 queue 中的 backlog_dev。

（3），net_rx_action()

netif_rx_schedule()触发中断下半部的执行，这个下半部将执行 net_rx_action()：

```
static void net_rx_action(struct softirq_action *h)
{
    struct softnet_data *queue = &__get_cpu_var(softnet_data);
    unsigned long start_time = jiffies;

    local_irq_disable();

    while (!list_empty(&queue->poll_list)) {
        struct net_device *dev;

        local_irq_enable();

        dev = list_entry(queue->poll_list.next,
                    struct net_device, poll_list);
```

```
        if (dev->quota <= 0 || dev->poll(dev, &budget)) {
            … //出错处理
        } else {
            netpoll_poll_unlock(have);
            dev_put(dev);
            local_irq_disable();
        }
}
```

由上可以看到，下半部的主要工作是遍历有数据帧等待接收的设备链表，对于每个设备，执行它相应的 poll 函数。

（4），poll 函数

NON—NAPI 方式：

这种方式对应该的 poll 函数为 process_backlog：

```
struct softnet_data *queue = &__get_cpu_var(softnet_data);
for (;;) {

    local_irq_disable();
    skb = __skb_dequeue(&queue->input_pkt_queue);
    local_irq_enable();

    netif_receive_skb(skb);
}
```

它首先找到当前 CPU 的 softnet_data 结构，然后遍历其数据队 SKB，并将数据上交 netif_receive_skb(skb)处理。

NAPI 方式：

这种方式下，NIC 驱动程序会提供自己的 poll 函数和私有接收队列。

如 intel 8255x 系列网卡程序 e100,它有在初始化的时候首先分配一个接收队列，而不像以上那种方式在接收到数据帧的时候再为其分配数据空间。这样，NAPI 的 poll 函数在处理接收的时候，它遍历的是自己的私有队列：

```
static int e100_poll(struct net_device *netdev, int *budget)
{
    e100_rx_clean(nic, &work_done, work_to_do);
    ……
}

static void e100_rx_clean(struct nic *nic, unsigned int *work_done,
    unsigned int work_to_do)
```

```
{
        …….
        /* Indicate newly arrived packets */
        for(rx = nic->rx_to_clean; rx->skb; rx = nic->rx_to_clean = rx->next) {
                int err = e100_rx_indicate(nic, rx, work_done, work_to_do);
                if(-EAGAIN == err) {
                ……
        }
                ……
}

static int e100_rx_indicate(struct nic *nic, struct rx *rx,
        unsigned int *work_done, unsigned int work_to_do)
{
        struct sk_buff *skb = rx->skb;
        struct rfd *rfd = (struct rfd *)skb->data;
        rfd_status = le16_to_cpu(rfd->status);

        /* Get actual data size */
        actual_size = le16_to_cpu(rfd->actual_size) & 0x3FFF;

        /* Pull off the RFD and put the actual data (minus eth hdr) */
        skb_reserve(skb, sizeof(struct rfd));
        skb_put(skb, actual_size);
        skb->protocol = eth_type_trans(skb, nic->netdev);

        netif_receive_skb(skb);

        return 0;
}
```

主要工作在 e100_rx_indicate()中完成，这主要重设 SKB 的一些参数，然后跟 process_backlog(), 一样，最终调用 netif_receive_skb(skb)。

（5），netif_receive_skb(skb)

这是一个辅助函数，用于在 poll 中处理接收到的帧。它主要是向各个已注册的协议处理例程发送一个 SKB。

每个协议的类型由一个 packet_type 结构表示：

```
struct packet_type {
        __be16                  type;/* This is really htons(ether_type). */
        struct net_device       *dev;       /* NULL is wildcarded here            */
        int                     (*func) (struct sk_buff *,
```

```
                                struct net_device *,
                                struct packet_type *,
                                struct net_device *);
        struct sk_buff          *(*gso_segment)(struct sk_buff *skb,
                                        int features);
        int                     (*gso_send_check)(struct sk_buff *skb);
        void                    *af_packet_priv;
        struct list_head        list;
};
```

它的主要域为：type，为要处理的协议
func，为处理这个协议的例程

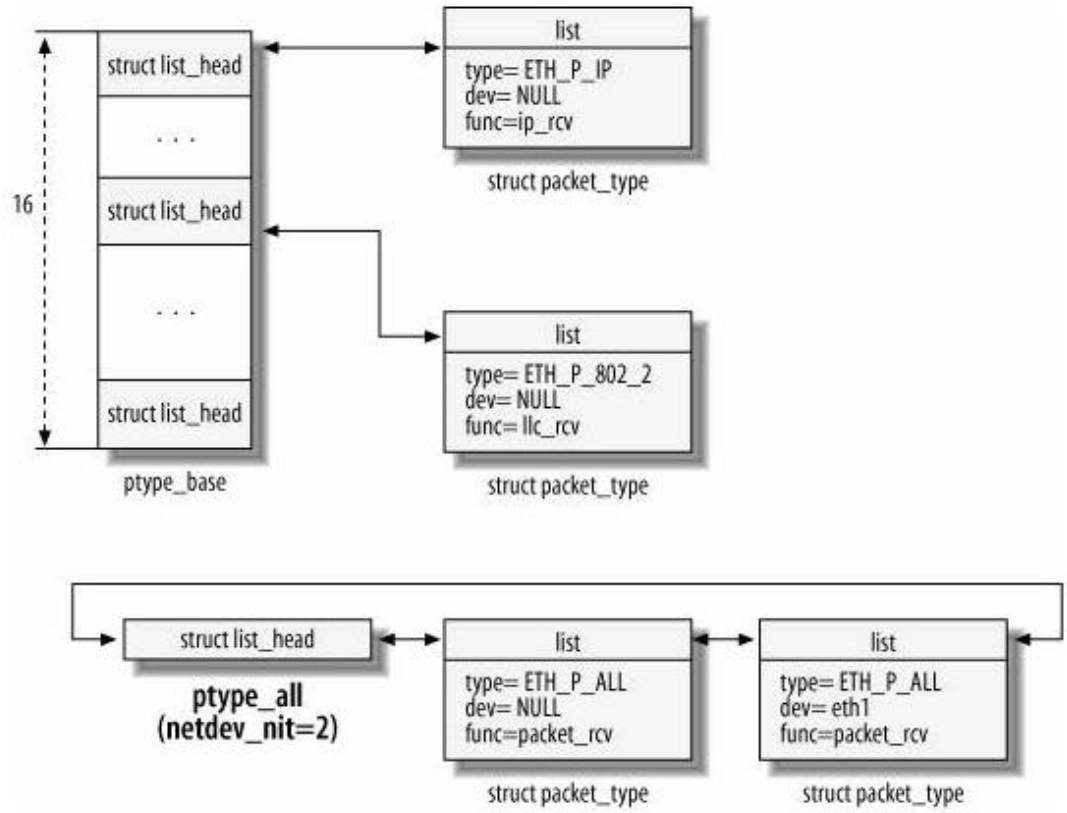
所用到的协议在系统或模块加载的时候初始化，如 IP 协议：

```
static struct packet_type ip_packet_type = {
        .type = __constant_htons(ETH_P_IP),
        .func = ip_rcv,
        .gso_send_check = inet_gso_send_check,
        .gso_segment = inet_gso_segment,
};

static int __init inet_init(void)
{
        ……
        dev_add_pack(&ip_packet_type);
        ……
}

void dev_add_pack(struct packet_type *pt)
{
        int hash;

        spin_lock_bh(&ptype_lock);
        if (pt->type == htons(ETH_P_ALL)) {
                netdev_nit++;
                list_add_rcu(&pt->list, &ptype_all);
        } else {
                hash = ntohs(pt->type) & 15;
                list_add_rcu(&pt->list, &ptype_base[hash]);
        }
        spin_unlock_bh(&ptype_lock);
}
```

可以看到，dev_add_pack()是将一个协议类型结构链入某一个链表，当协议类型为

ETH_P_ALL 时，它将被链入 ptype_all 链表，这个链表是用于 sniffer 这样一些程序的，它接收所有 NIC 收到的包。还有一个是 HASH 链表 ptype_base，用于各种协议，它是一个 16 个元素的数组，dev_add_pack()会根据协议类型将这个 packet_type 链入相应的 HASH 链表中。

而 ptype_base 与 ptype_all 的组织结构如下，一个为 HASH 链表，一个为双向链表：




```
int netif_receive_skb(struct sk_buff *skb)
{

        list_for_each_entry_rcu(ptype, &ptype_all, list) {
                if (!ptype->dev || ptype->dev == skb->dev) {
                        if (pt_prev)
                                ret = deliver_skb(skb, pt_prev, orig_dev);
                        pt_prev = ptype;
                }
        }

        type = skb->protocol;
        list_for_each_entry_rcu(ptype, &ptype_base[ntohs(type)&15], list) {
                if (ptype->type == type &&
                        (!ptype->dev || ptype->dev == skb->dev)) {
```

```
                if (pt_prev)
                    ret = deliver_skb(skb, pt_prev, orig_dev);
                pt_prev = ptype;
            }
        }

        if (pt_prev) {
            ret = pt_prev->func(skb, skb->dev, pt_prev, orig_dev);
        }
        return ret;
}
```

netif_receive_skb()的主要作用体现在两个遍历链表的操作中，其中之一为遍历 ptype_all 链，这些为注册到内核的一些 sniffer，将上传给这些 sniffer，另一个就是遍历 ptype_base，这个就是具体的协议类型。假高如上图如示，当 eth1 接收到一个 IP 数据包时，它首先分别发送一份副本给两个 ptype_all 链表中的 packet_type，它们都由 package_rcv 处理，然后再根据 HASH 值，在遍历另一个 HASH 表时，发送一份给类型为 ETH_P_IP 的类型，它由 ip_rcv 处理。如果这个链中还注册有其它 IP 层的协议，它也会同时发送一个副本给它。

其中，这个是由 deliver_skb(skb, pt_prev, orig_dev)去完成的：

```
static __inline__ int deliver_skb(struct sk_buff *skb,
                                  struct packet_type *pt_prev,
                                  struct net_device *orig_dev)
{
        atomic_inc(&skb->users);
        return pt_prev->func(skb, skb->dev, pt_prev, orig_dev);
}
```

可以看到，它只是一个包装函数，它只去执行相应 packet_type 里的 func 处理函数，如对于 ETH_P_IP 类型，由上面可以看到，它执行的就是 ip_rcv 了。

至此，一个以太网帧的链路层接收过程就全部完成，再下去就是网络层的处理了。

5），数据包的发送

数据包的发送为接收的反过程，发送过程较之接收过程的复杂性在于它有一个流量控制层（Trafficing Control Layer），用于实现ＱoS，但不是本文关注的目标。

(1)，__netif_schedule ()

当内核有数据包等待发送时，它会间接调用__netif_schedule ()去处理这些数据包：

```
void __netif_schedule(struct net_device *dev)
```

```
{
    if (!test_and_set_bit(__LINK_STATE_SCHED, &dev->state)) {
        unsigned long flags;
        struct softnet_data *sd;

        local_irq_save(flags);
        sd = &__get_cpu_var(softnet_data);
        dev->next_sched = sd->output_queue;
        sd->output_queue = dev;
        raise_softirq_irqoff(NET_TX_SOFTIRQ);
        local_irq_restore(flags);
    }
}
```

这个函数的功能很简单，就是将要有数据要发送的设备加 softnet_data 的 output_queue 队列的头部，这里要注意，一个设备加入是有条件的，如果一个设备的状态为 __LINK_STATE_SCHED 时，表示这个设备已经被 scheduled，就不必要再一次执行这个函数了。然后这个函数触发软中断，由软中断去执行 net_tx_action()。

(2)，net_tx_action()
这个函数的功能有两个，其一是释放 softirq_action 中完成队列 completion_queue 中的 skb。

我们知道，当系统运行在中断上下文中，它应该执行的时间应该越短越好，但如果我们需要在中断上下文中释放 SKB，这就需要比较长的时间了，所以在个时间段里处理内核的释放并不是一个好的选择。所以，网络子系统在 softirq_action 结构中设置了一个完成队列 completion_queue，当内核要在中断上下文中释放 skb 时，它将调 dev_kfree_skb_irq(skb)：

```
static inline void dev_kfree_skb_irq(struct sk_buff *skb)
{
    if (atomic_dec_and_test(&skb->users)) {
        struct softnet_data *sd;
        unsigned long flags;

        local_irq_save(flags);
        sd = &__get_cpu_var(softnet_data);
        skb->next = sd->completion_queue;
        sd->completion_queue = skb;
        raise_softirq_irqoff(NET_TX_SOFTIRQ);
        local_irq_restore(flags);
    }
}
```

可以看到，它并没有真正的释放 skb 空间，而只是将它链入完成队列 completion_queue 中，并触发软中断，由软中断来执行真正的释放操作，这就是上面提到的 net_tx_action()来完成

的，这是它的任务之一：

```
        clist = sd->completion_queue;
        sd->completion_queue = NULL;
        local_irq_enable();

        while (clist) {
            struct sk_buff *skb = clist;
            clist = clist->next;

            BUG_TRAP(!atomic_read(&skb->users));
            __kfree_skb(skb);
        }
```

net_tx_action()的另一个任务，也是根本的任务，当然是发送数据包了：

```
    if (sd->output_queue) {
        struct net_device *head;

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

        while (head) {
            struct net_device *dev = head;
            head = head->next_sched;

            smp_mb__before_clear_bit();
            clear_bit(__LINK_STATE_SCHED, &dev->state);

            if (spin_trylock(&dev->queue_lock)) {
                qdisc_run(dev);
                spin_unlock(&dev->queue_lock);
            } else {
                netif_schedule(dev);
            }
        }
    }
```

正常情况下，它会将 output_queue 队列中的有待发送的队列中的设备遍历一次，并对各个设备调用 qdisc_run(dev)发送数据包。在这里，qdisc_run(dev)是属于 QoS 的内容了。这里我们只需要知道，qdisc_run(dev)会选择“合适”的 skb 然后传递给 dev_hard_start_xmit(skb, dev)。

(3)，dev_hard_start_xmit(skb, dev)
这也只是一个包装函数，它首先看有没有注册的 sniffer，要是存在的话（netdev_nit 不等于 0），便将一个副本通过 dev_queue_xmit_nit(skb, dev)发送给它：

```
if (likely(!skb->next)) {
    if (netdev_nit)
        dev_queue_xmit_nit(skb, dev);
```

再之后，就是调用驱动程序的 hard_start_xmit 完成最后的发送工作了：

```
return dev->hard_start_xmit(skb, dev);
```

hard_start_xmit()只要是跟硬件打交道，一般是通知 D M A 完成数据的发送工作。这里还有一个问题是，如果驱动或是硬件本身不支持 scatter/gather IO，在上面传送过来的数据又是存在分片的（fragments，即 skb_shinfo(skb)->nr_frags 不等于 0），它只能通过 skb_linearize(skb)将原来的 skb 重新组装成一个没有分片的 skb 再进行 D M A 。

07.01.30