5.本机网络

跨机网络通信

数据发送过程

- 用户数据被拷贝到内核态；然后通过协议栈处理后进入RingBuffer；然后网卡驱动真正将数据发送出去；发送后通过硬中断告知CPU，触发软中断清理RingBuffer

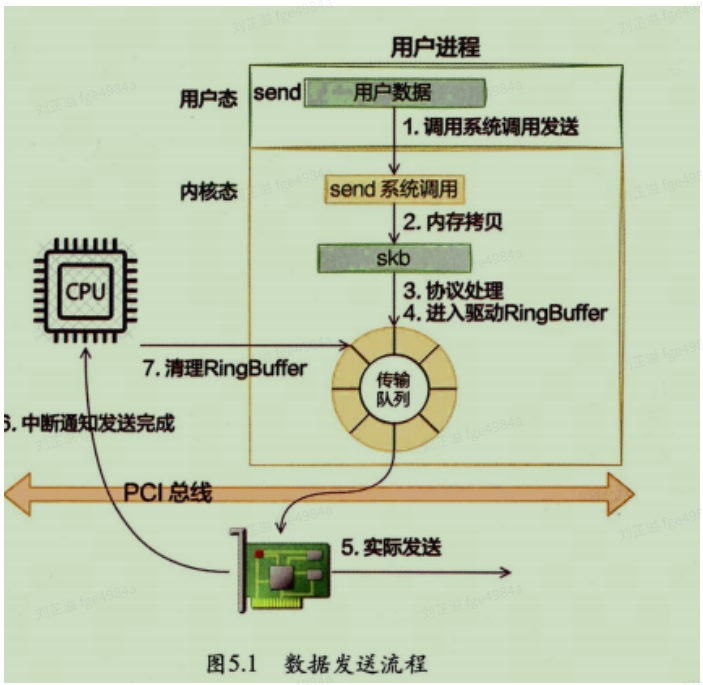
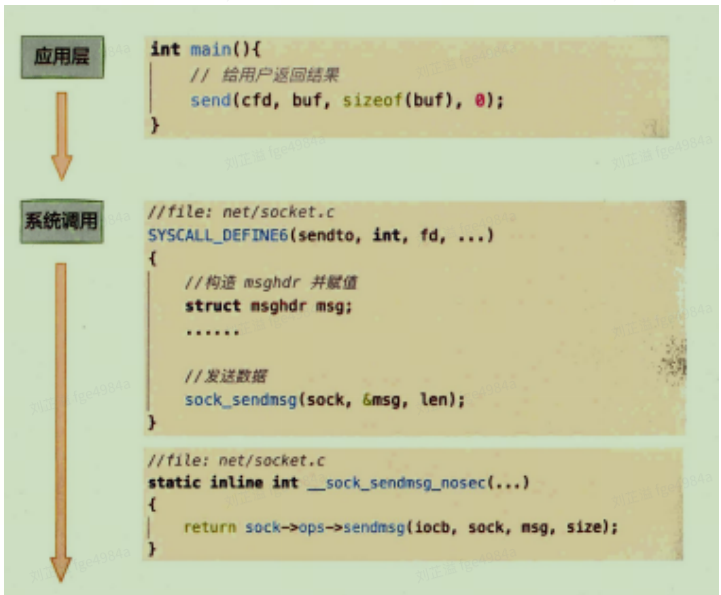


图5.1 数据发送流程



协议栈

```c
//file: net/ipv4/af_inet.c
int inet_sendmsg(......)
{
    return sk->sk_prot->sendmsg(iocb, sk, msg, size);
}
```

传输层

```c
//file: net/ipv4/tcp.c
int tcp_sendmsg(...)
{
    ...
}

//file: net/ipv4/tcp_output.c
static int tcp_transmit_skb(......)
{
    //封装TCP头
    th = tcp_hdr(skb);
    th->source      = inet->inet_sport;
    th->dest        = inet->inet_dport;

    //调用网络层发送接口
    err = icsk->icsk_af_ops->queue_xmit(skb);
}
```

网络层

```c
//file: net/ipv4/ip_output.c
int ip_queue_xmit(struct sk_buff *skb, struct flowi *fl)
{
    res = ip_local_out(skb);
}

//file: net/ipv4/ip_output.c
static inline int ip_finish_output2(struct sk_buff *skb)
{
    //继续向下层传递
    int res = dst_neigh_output(dst, neigh, skb);
}
```

邻居子系统

```c
//file: include/net/dst.h
static inline int dst_neigh_output(...)
{
    ......
    return neigh_hh_output(hh, skb);
}

//file: include/net/neighbour.h
static inline int neigh_hh_output(...)
{
    ......
    skb_push(skb, hh_len);
    return dev_queue_xmit(skb);
}
```

网络设备子系统

```c
//file: net/core/dev.c
int dev_queue_xmit(struct sk_buff *skb)
{
    //选择发送队列并获取 qdisc
    txq = netdev_pick_tx(dev, skb);
    q = rcu_dereference_bh(txq->qdisc);

    //则调用__dev_xmit_skb 继续发送
    rc = __dev_xmit_skb(skb, q, dev, txq);
}
```

```c
//file: net/core/dev.c
int dev_hard_start_xmit(...)
{
    //获取设备的回调函数集合 ops
    const struct net_device_ops *ops = dev->netdev_ops;

    //调用驱动里的发送回调函数 ndo_start_xmit 将数据包传给网卡设备
    skb_len = skb->len;
    rc = ops->ndo_start_xmit(skb, dev);
}
```

驱动程序

```c
//file: drivers/net/ethernet/intel/igb/igb_main.c
static netdev_tx_t igb_xmit_frame(...)
{
    return igb_xmit_frame_ring(skb, ...);
}

netdev_tx_t igb_xmit_frame_ring(...)
{
    //获取TX Queue 中下一个可用缓冲区信息
    first = &tx_ring->tx_buffer_info[tx_ring->next_to_use];
    first->skb = skb;
    first->bytecount = skb->len;

    //igb_tx_map 函数准备给设备发送的数据.
    igb_tx_map(tx_ring, first, hdr_len);
}
```

硬件

数据接收过程

- 网卡接收到消息，将数据通过DMA拷贝到内存，然后通过硬中断告知CPU；CPU调用网络驱动的中断处理函数触发软中断；ksoftirqd调用网卡的poll函数进行收包；交给协议栈进行处理；协议栈处理完后把数据放到接收队列之后，唤醒用户进程

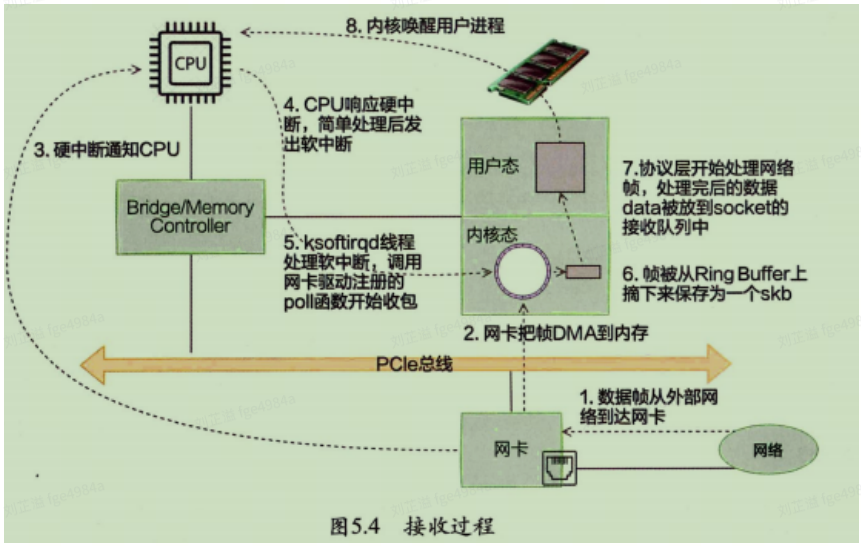
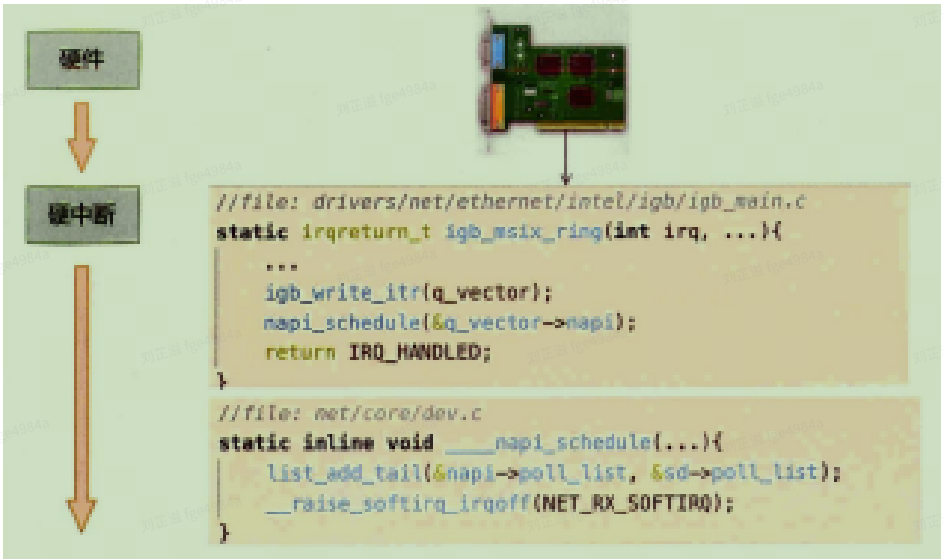
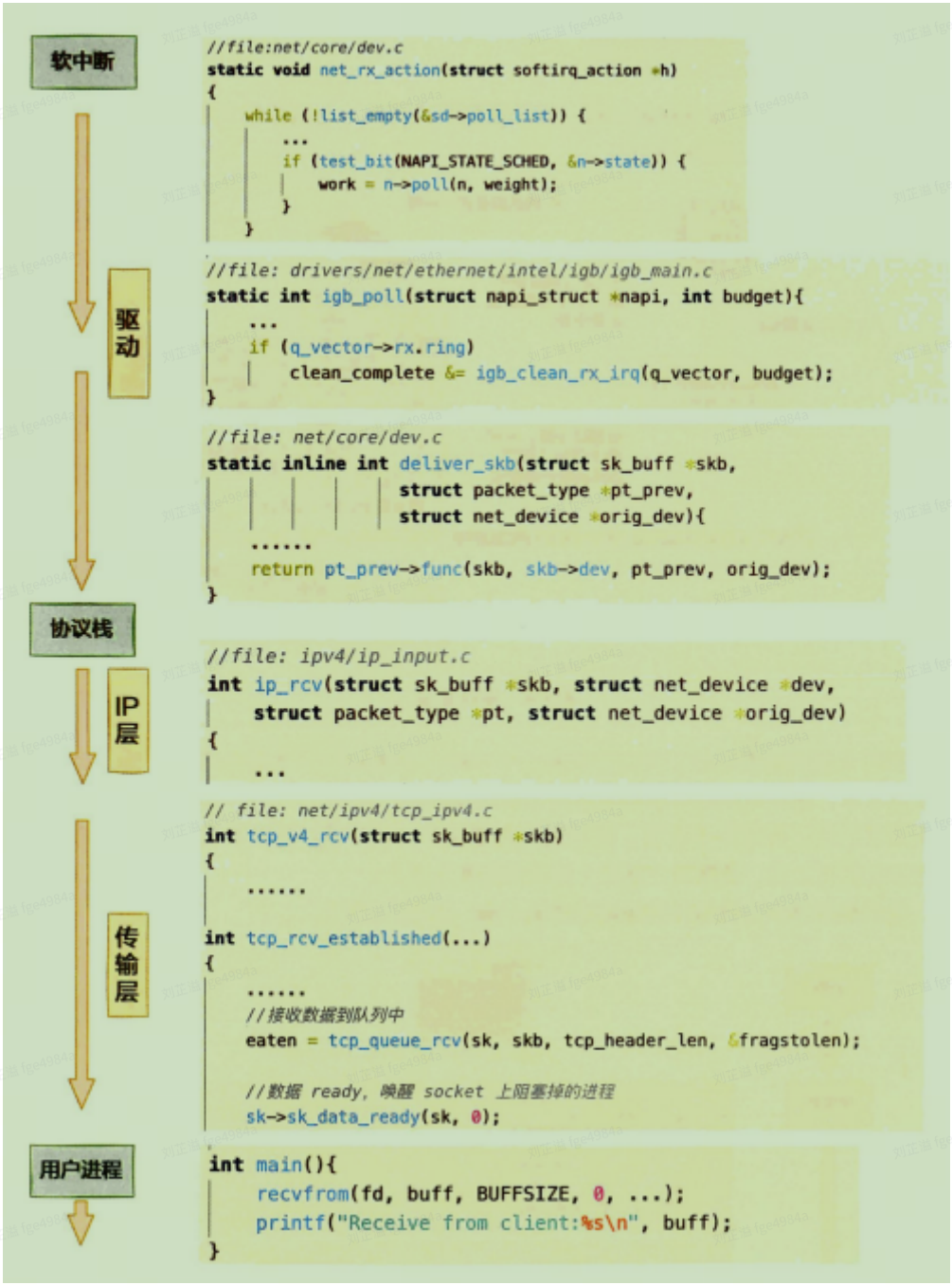


图5.4 接收过程




软中断
驱动
协议栈
IP层
传输层
用户进程
//file:net/core/dev.c
static void net_rx_action(struct softirq_action *h)
{
while (!list_empty(&sd->poll_list)) {
...
if (test_bit(NAPI_STATE_SCHED, &n->state)) {
work = n->poll(n, weight);
}
}
//file: drivers/net/ethernet/intel/igb/igb_main.c
static int igb_poll(struct napi_struct *napi, int budget){
...
if (q_vector->rx.ring)
clean_complete &= igb_clean_rx_irq(q_vector, budget);
}
//file: net/core/dev.c
static inline int deliver_skb(struct sk_buff *skb,
struct packet_type *pt_prev,
struct net_device *orig_dev){
.....
return pt_prev->func(skb, skb->dev, pt_prev, orig_dev);
}
//file: ipv4/ip_input.c
int ip_rcv(struct sk_buff *skb, struct net_device *dev,
struct packet_type *pt, struct net_device *orig_dev)
{
...
// file: net/ipv4/tcp_ipv4.c
int tcp_v4_rcv(struct sk_buff *skb)
{
.....
int tcp_rcv_established(...)
{
.....
//接收数据到队列中
eaten = tcp_queue_rcv(sk, skb, tcp_header_len, &fragstolen);
//数据 ready，唤醒 socket 上阻塞掉的进程
sk->sk_data_ready(sk, 0);
int main(){
recvfrom(fd, buff, BUFFSIZE, 0, ...);
printf("Receive from client:%s\n", buff);
}

网络通信汇总

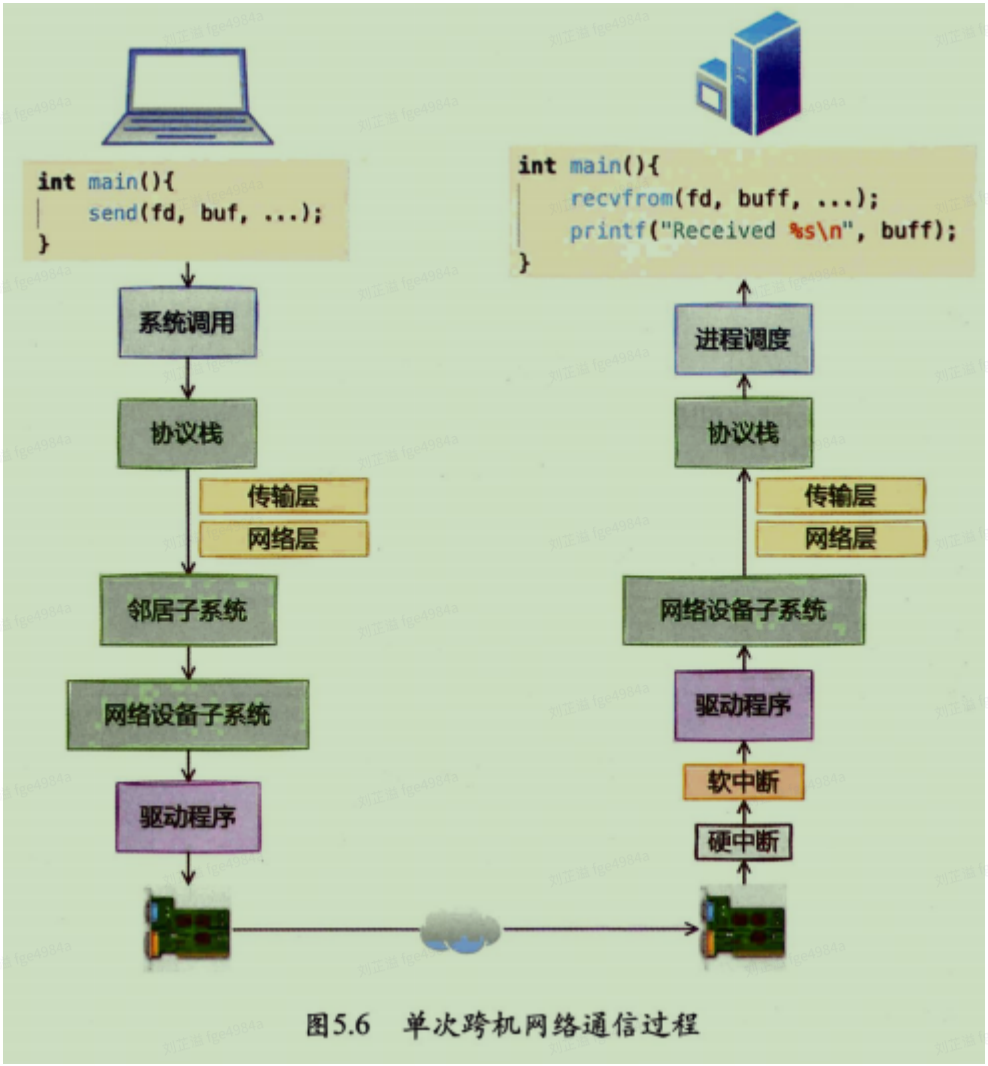


图5.6　单次跨机网络通信过程

本机发送过程

- 本机网络IO在local路由表中就能找到路由项；**对应的设备都将使用loopback网卡，即lo设备**

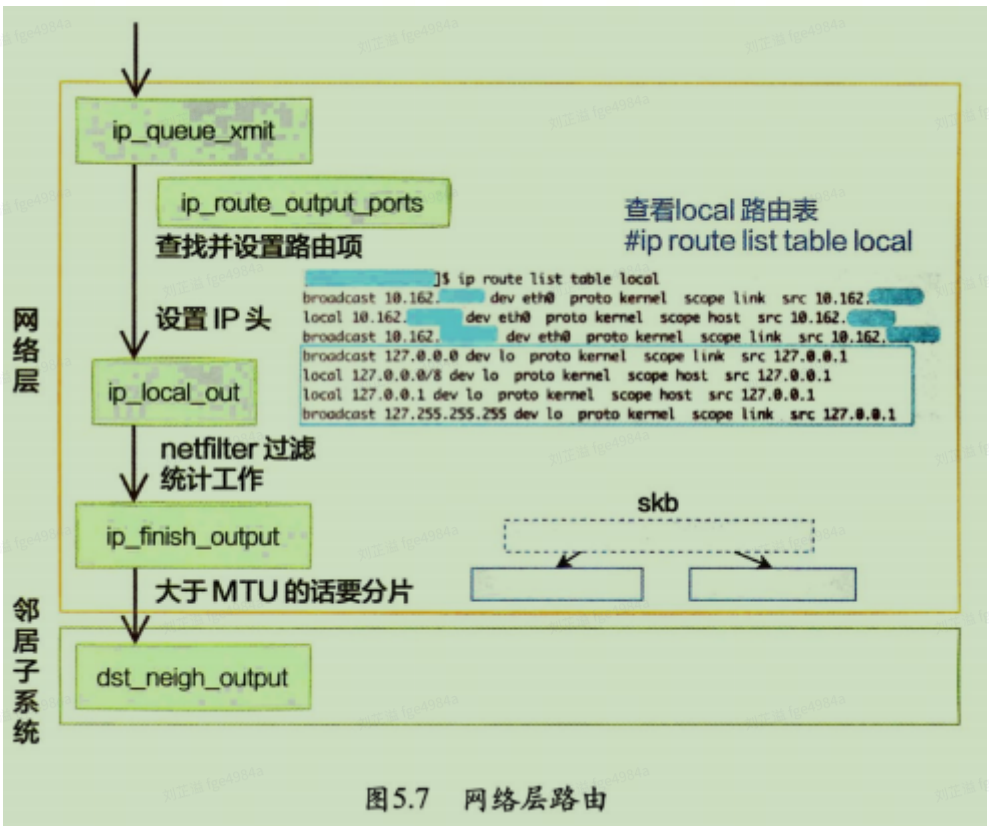


图5.7　网络层路由

```
//file: net/ipv4/ip_output.c
int ip_queue_xmit(struct sk_buff *skb, struct flowi *fl)
{
        //检查socket中是否有缓存的路由表
        rt = (struct rtable *)__sk_dst_check(sk, 0);
        if (rt == NULL) {
                //没有缓存则展开查找
                //查找路由项，并缓存到socket中
                rt = ip_route_output_ports(...);
                sk_setup_caps(sk, &rt->dst);
 }
```

查找路由项的函数是ip_route_output_ports，它又依次调用ip_route_output_flow、__ip_route_output_key、fib_lookup函数。调用过程略过，直接看fib_lookup的关键代码。

- 对local和main两个路由表进行查询

```
//file:include/net/ip_fib.h
static inline int fib_lookup(struct net *net, const struct flowi4 *flp,
                             struct fib_result *res)
{
```

```
        struct fib_table *table;

        table = fib_get_table(net, RT_TABLE_LOCAL);
        if (!fib_table_lookup(table, flp, res, FIB_LOOKUP_NOREF))
                return 0;

        table = fib_get_table(net, RT_TABLE_MAIN);
        if (!fib_table_lookup(table, flp, res, FIB_LOOKUP_NOREF))
                return 0;
        return -ENETUNREACH;
}
```

从上述结果可以看出，对于目的是127.0.0.1的路由在local路由表中就能够找到。fib_lookup的工作完成，返回__ip_route_output_key函数继续执行。

```
//file: net/ipv4/route.c
struct rtable *__ip_route_output_key(struct net *net, struct flowi4 *fl4)
{
        if (fib_lookup(net, fl4, &res)) {
        }
        if (res.type == RTN_LOCAL) {
                dev_out = net->loopback_dev;
                ......
        }

        rth = __mkroute_output(&res, fl4, orig_oif, dev_out, flags);
        return rth;
}
```

- 对于本机的网络请求，设备将全部使用net->loopback_dev
- 网络层会经过ip_finish_output，进入dst_neigh_output
- 在邻居子系统函数中经过处理后，进入网络设备子系统dev_queue_xmit

网络设备子系统

- 不涉及到网卡队列问题，直接进入dev_hard_start_xmit函数

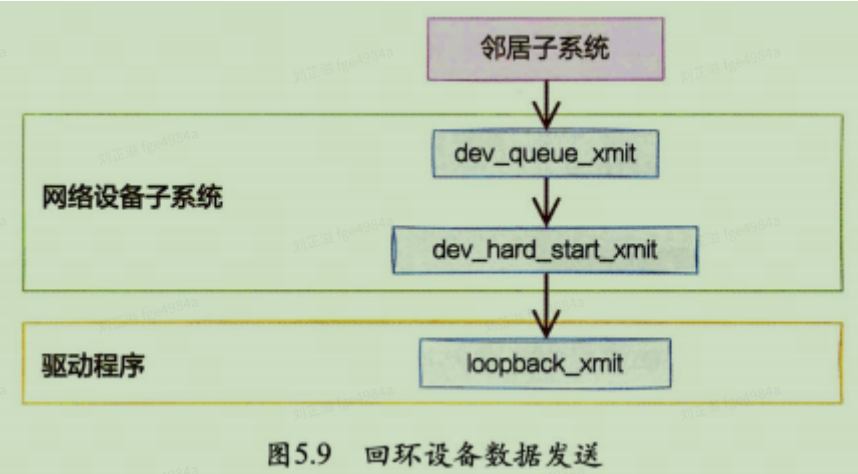


图5.9 回环设备数据发送

```
//file: net/core/dev.c
int dev_queue_xmit(struct sk_buff *skb)
{
        q = rcu_dereference_bh(txq->qdisc);
        if (q->enqueue) {//回环设备这里为false
                rc = __dev_xmit_skb(skb, q, dev, txq);
                goto out;
        }

        //开始回环设备处理
        if (dev->flags & IFF_UP) {
                dev_hard_start_xmit(skb, dev, txq, ...);
                ......
        }
}
```

在dev_hard_start_xmit函数中还将调用设备驱动的操作函数。

```
//file: net/core/dev.c
int dev_hard_start_xmit(struct sk_buff *skb, struct net_device *dev,
    struct netdev_queue *txq)
{
        //获取设备驱动的回调函数集合ops
        const struct net_device_ops *ops = dev->netdev_ops;

        //调用驱动的ndo_start_xmit进行发送
        rc = ops->ndo_start_xmit(skb, dev);
        ......
}
```

“驱动”程序

回环设备的“驱动”程序的工作流程如图5.10所示。

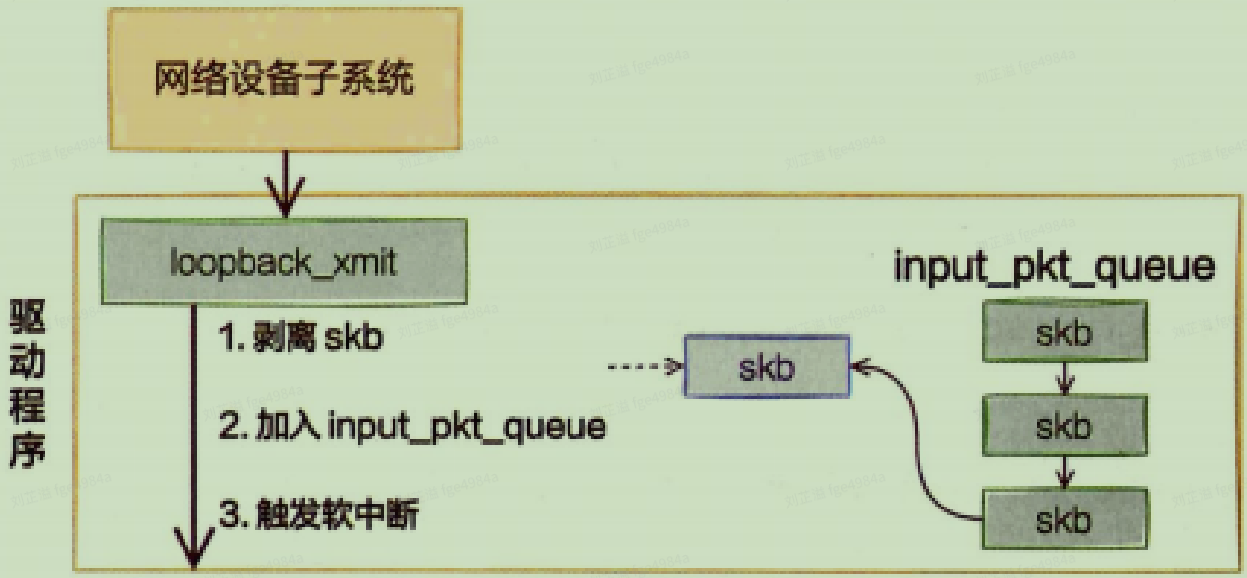


图5.10 回环设备的“驱动”程序的工作流程

- 在本机网络中，传输层下面的skb不需要释放，直接给接收方传送过去

```
//file:drivers/net/loopback.c
static const struct net_device_ops loopback_ops = {
        .ndo_init       = loopback_dev_init,
        .ndo_start_xmit= loopback_xmit,
        .ndo_get_stats64 = loopback_get_stats64,
};
```

所以对dev_hard_start_xmit调用实际上执行的是loopback“驱动”里的loopback_xmit。为什么我把“驱动”加个引号呢，因为loopback是一个纯软件性质的虚拟接口，并没有真正意义上对物理设备的驱动。

```
//file:drivers/net/loopback.c
static netdev_tx_t loopback_xmit(struct sk_buff *skb,
                                 struct net_device *dev)
{
        //剥离掉和原 socket的联系
        skb_orphan(skb);

        //调用netif_rx
        if (likely(netif_rx(skb) == NET_RX_SUCCESS)) {
        }
}
```

接着调用 netif_rx，在该方法中最终会执行到 enqueue_to_backlog（netif_rx -> netif_rx_internal -> enqueue_to_backlog）。

```
//file: net/core/dev.c
static int enqueue_to_backlog(struct sk_buff *skb, int cpu,
                              unsigned int *qtail)
{
        sd = &per_cpu(softnet_data, cpu);

        ......
        __skb_queue_tail(&sd->input_pkt_queue, skb);

        ......
        ____napi_schedule(sd, &sd->backlog);
```

在enqueue_to_backlog函数中，把要发送的skb插入softnet_data->input_pkt_queue队列中并调用napi_schedule来触发软中断。

```
//file:net/core/dev.c
static inline void ____napi_schedule(struct softnet_data *sd,
                                     struct napi_struct *napi)
{
        list_add_tail(&napi->poll_list, &sd->poll_list);
        __raise_softirq_irqoff(NET_RX_SOFTIRQ);
}
```

只有触发完软中断，发送过程才算完成了。

本机接收过程

发送过程触发软中断后，会进入软中断处理函数net_rx_action，如图5.11所示。

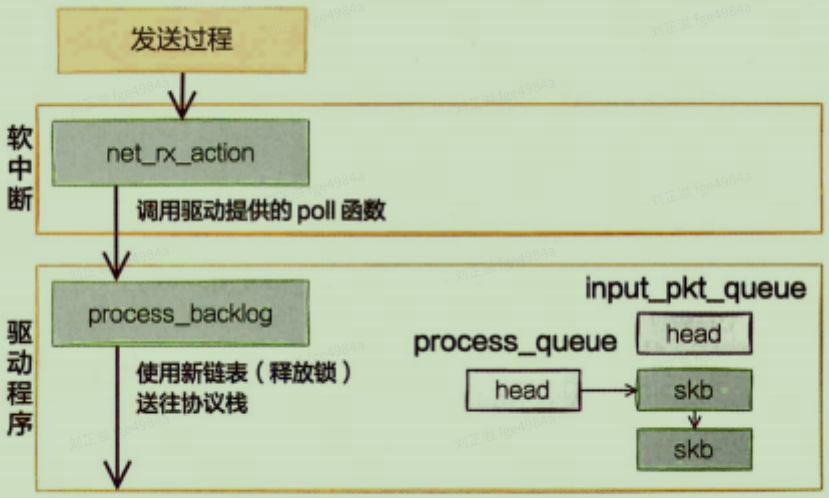


图5.11 数据接收

```
//file: net/core/dev.c
static void net_rx_action(struct softirq_action *h){
        while (!list_empty(&sd->poll_list)) {
                work = n->poll(n, weight);
        }
}
```

前面介绍过，对于igb网卡来说，poll实际调用的是igb_poll函数。那么loopback网卡的poll函数是哪个呢？由于poll_list里面是struct softnet_data 对象，我们在net_dev_init 中找到了蛛丝马迹。

```
//file:net/core/dev.c
static int __init net_dev_init(void)
{
        for_each_possible_cpu(i) {
                sd->backlog.poll = process_backlog;
        }
}
```

原来struct softnet_data 默认的poll 在初始化的时候设置成了 process_backlog 函数，来看看它都干了什么。

```
static int process_backlog(struct napi_struct *napi, int quota)
{
        while(){
                while ((skb = __skb_dequeue(&sd->process_queue))) {
                        __netif_receive_skb(skb);
                }

                //skb_queue_splice_tail_init()函数用于将链表a连接到链表b上，
                //形成一个新的链表b，并将原来a的头变成空链表。
                qlen = skb_queue_len(&sd->input_pkt_queue);
                if (qlen)
                        skb_queue_splice_tail_init(&sd->input_pkt_queue,
                                                   &sd->process_queue);
        }
}
```

这次先看对 skb_queue_splice_tail_init的调用。源码就不看了，直接说它的作用，是把 sd->input_pkt_queue 里的skb 链到 sd->process_queue 链表上去。

然后再看 __skb_dequeue， __skb_dequeue 是从 sd->process_queue取下来包进行处理。这样和前面发送过程的结尾处就对上了，发送过程是把包放到了input_pkt_queue队列里，如图5.12所示。

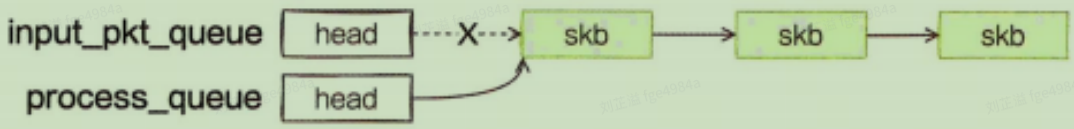


图5.12 队列的执行

- 最后调用__netif_receive_skb数据送往协议栈；__netif_receive_skb->__netif_receive_skb_core->deliver_skb；然后送入网络层ip_rcv，然后是传输层；最后唤醒用户进程

本机网络通信汇总

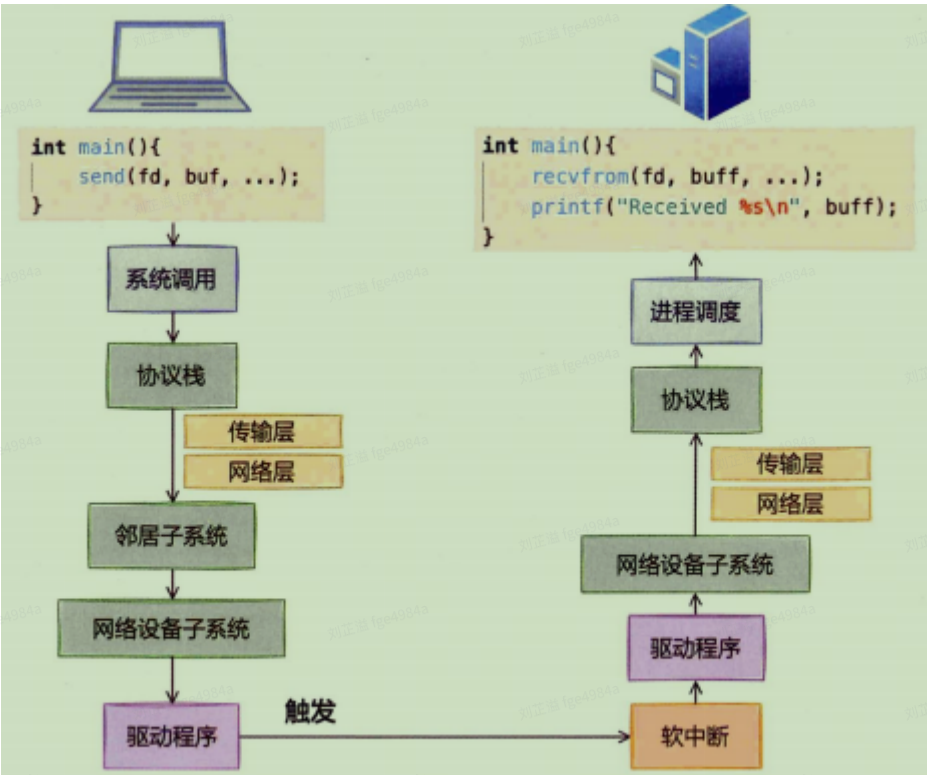


图5.13　本机网络IO过程

本机IP路由

很多人在看到这个路由表的时候就被它迷惑了，以为上面的10.162.*.*真的会被路由到eth0（其中10.162.*.*是我的本机局域网IP，我把后面两段用*号隐藏起来了）。

但其实内核在初始化local路由表的时候，把local路由表里所有的路由项都设置成了RTN_LOCAL，不只是127.0.0.1。这个过程是在设置本机IP的时候，调用fib_inetaddr_event函数完成设置的。

```
static int fib_inetaddr_event(struct notifier_block *this,
        unsigned long event, void *ptr)
{
        switch (event) {
        case NETDEV_UP:
                fib_add_ifaddr(ifa);
                break;
```

```
        case NETDEV_DOWN:
                fib_del_ifaddr(ifa, NULL);
//file:ipv4/fib_frontend.c
void fib_add_ifaddr(struct in_ifaddr *ifa)
{
        fib_magic(RTM_NEWROUTE, RTN_LOCAL, addr, 32, prim);
}
```

所以即使本机IP不用127.0.0.1，内核在路由项查找的时候判断类型是RTN_LOCAL，仍然会使用net->loopback_dev，也就是lo虚拟网卡。

总结

127.0.0.1本机网络IO需要经过网卡吗

- 不需要经过网卡

数据包在本机网络和外网流程上有何区别

> 总的来说，本机网络IO和跨机网络IO比较起来，确实是节约了驱动上的一些开销。发送数据不需要进 RingBuffer的驱动队列，直接把 skb 传给接收协议栈（经过软中断）。但是在内核其他组件上，可是一点儿都没少，系统调用、协议栈（传输层、网络层等）、设备子系统整个走了一遍。连“驱动”程序都走了（虽然对于回环设备来说只是一个纯软件的虚拟出来的东西）。所以即使是本机网络IO，切忌误以为没啥开销就滥用。

> 3）访问本机服务时，使用127.0.0.1能比使用本机IP（例如192.168.x.x）更快吗？
>
> 很多人的直觉是用本机IP会走网卡，但正确结论是和127.0.0.1没有差别，都是走虚拟的环回设备lo。这是因为内核在设置IP的时候，把所有的本机IP都初始化到local路由表里了，类型写死了是RTN_LOCAL。在后面的路由项选择的时候发现类型是RTN_LOCAL就会选择lo设备了。还不信的话你也动手抓包试试！