网络包接收过程

一、MAC层到IP层的接收过程

1、设备驱动层

网卡作为一个硬件，接收到网络包，应该怎么通知操作系统，这个网络包到达了呢？没错，我们可以触发一个中断。但是这里有个问题，就是网络包的到来，往往是很难预期的。网络吞吐量比较大的时候，网络包的到达会十分频繁。这个时候，如果非常频繁地去触发中断，想想就觉得是个灾难。

比如说，CPU正在做某个事情，一些网络包来了，触发了中断，CPU停下手里的事情，去处理这些网络包，处理完毕按照中断处理的逻辑，应该回去继续处理其他事情。这个时候，另一些网络包又来了，又触发了中断，CPU手里的事情还没捂热，又要停下来去处理网络包。能不能大家要来的一起来，把网络包好好处理一把，然后再回去集中处理其他事情呢？

网络包能不能一起来，这个我们没法儿控制，但是我们可以有一种机制，就是当一些网络包到来触发了中断，内核处理完这些网络包之后，我们可以先进入主动轮询poll网卡的方式，主动去接收到来的网络包。如果一直有，就一直处理，等处理告一段落，就返回干其他的事情。当再有下一批网络包到来的时候，再中断，再轮询poll。这样就会大大减少中断的数量，提升网络处理的效率，这种处理方式我们称为NAPI。

下面以drivers/net/ethernet/intel/ixgb/ixgb_main.c为例解释网卡驱动接收的工作机制。

```
static struct pci_driver ixgb_driver = {
    .name     = ixgb_driver_name,
    .id_table = ixgb_pci_tbl,
    .probe    = ixgb_probe,      /* 注册到总线匹配后会触发 */
    .remove   = ixgb_remove,
    .err_handler = &ixgb_err_handler
};

.....

static const struct net_device_ops ixgb_netdev_ops = {
    .ndo_open       = ixgb_open,    /*  */
    .ndo_stop       = ixgb_close,   /*  */
    .ndo_start_xmit     = ixgb_xmit_frame,
    .ndo_get_stats      = ixgb_get_stats,
    .ndo_set_rx_mode    = ixgb_set_multi,
    .ndo_validate_addr  = eth_validate_addr,
```

```
    .ndo_set_mac_address    = ixgb_set_mac,
    .ndo_change_mtu     = ixgb_change_mtu,
    .ndo_tx_timeout     = ixgb_tx_timeout,
    .ndo_vlan_rx_add_vid    = ixgb_vlan_rx_add_vid,
    .ndo_vlan_rx_kill_vid   = ixgb_vlan_rx_kill_vid,
#ifdef CONFIG_NET_POLL_CONTROLLER
    .ndo_poll_controller    = ixgb_netpoll,
#endif
    .ndo_fix_features       = ixgb_fix_features,
    .ndo_set_features       = ixgb_set_features,
};

.....

static int __init
ixgb_init_module(void)
{
    pr_info("%s - version %s\n", ixgb_driver_string, ixgb_driver_version);
    pr_info("%s\n", ixgb_copyright);

    return pci_register_driver(&ixgb_driver);
}

module_init(ixgb_init_module);
```

在网卡驱动程序初始化的时候会调用ixgb_init_module注册一个驱动ixgb_driver到pci总线上，然后触发ixgb_probe。

```
static int
ixgb_probe(struct pci_dev *pdev, const struct pci_device_id *ent)
{
    struct net_device *netdev = NULL;   /* 网卡 */
    struct ixgb_adapter *adapter;   /* 网卡控制器 */

    .....

     /* 创建网卡设备结构体并部分初始化 */
    netdev = alloc_etherdev(sizeof(struct ixgb_adapter));
    if (!netdev) {
        err = -ENOMEM;
        goto err_alloc_etherdev;
    }


    SET_NETDEV_DEV(netdev, &pdev->dev);
    pci_set_drvdata(pdev, netdev);  /* 设置netdev和pci dev关联 */

    adapter = netdev_priv(netdev);
    adapter->netdev = netdev;       /* 设置netdev和adapter关联 */
    adapter->pdev = pdev;
    adapter->hw.back = adapter;
    adapter->msg_enable = netif_msg_init(debug, DEFAULT_MSG_ENABLE);
    adapter->hw.hw_addr = pci_ioremap_bar(pdev, BAR_0);
```

```
    .....

    /* 设置netdev的netdev_ops */
    netdev->netdev_ops = &ixgb_netdev_ops;
    ixgb_set_ethtool_ops(netdev);
    netdev->watchdog_timeo = 5 * HZ;
    /* 注册netdev的napi poll函数为ixgb_clean */
    netif_napi_add(netdev, &adapter->napi, ixgb_clean, 64);
    strncpy(netdev->name, pci_name(pdev), sizeof(netdev->name) - 1);
    adapter->bd_number = cards_found;
    adapter->link_speed = 0;
    adapter->link_duplex = 0;

    /* setup the private structure */
    /* 其他的一些设置初始化 */
    err = ixgb_sw_init(adapter);
    .....
    init_timer(&adapter->watchdog_timer);
    .....
    err = register_netdev(netdev);
    .....
}
```

在ixgb_probe中，创建一个struct net_device表示这个网络设备，并且netif_napi_add函数为这个网络设备注册一个轮询poll函数ixgb_clean，将来一旦出现网络包的时候，就是要通过它来轮询了。

当网卡被激活的时候，会调用netdev->netdev_ops中的ndo_open，即ixgb_open，在ixgb_open中调用ixgb_up，里面注册了一个硬件中断处理函数ixgb_intr。

```
int
ixgb_up(struct ixgb_adapter *adapter)
{
    struct net_device *netdev = adapter->netdev;
    .....
     /* 注册中断处理函数ixgb_intr */
    err = request_irq(adapter->pdev->irq, ixgb_intr, irq_flags,
                      netdev->name, netdev);
    .....
}


/**
 * ixgb_intr - Interrupt Handler
 * @irq: interrupt number
 * @data: pointer to a network interface device structure
 **/
static irqreturn_t
ixgb_intr(int irq, void *data)
{
    struct net_device *netdev = data;
    struct ixgb_adapter *adapter = netdev_priv(netdev);
```

```
    struct ixgb_hw *hw = &adapter->hw;
    u32 icr = IXGB_READ_REG(hw, ICR);

    .....

    if (napi_schedule_prep(&adapter->napi)) {

        /* Disable interrupts and register for poll. The flush
          of the posted write is intentionally left out.
        */

        IXGB_WRITE_REG(&adapter->hw, IMC, ~0);
        __napi_schedule(&adapter->napi);    /* 触发软中断 */
    }
    return IRQ_HANDLED;
}
```

如果一个网络包到来，触发了硬件中断，就会调用ixgb_intr，这里面会调用__napi_schedule。

```
void __napi_schedule(struct napi_struct *n)
{
    unsigned long flags;

    local_irq_save(flags);
    ____napi_schedule(this_cpu_ptr(&softnet_data), n);  /* 触发软中断 */
    local_irq_restore(flags);
}

.....

static inline void ____napi_schedule(struct softnet_data *sd,
                     struct napi_struct *napi)
{
    /* 将当前设备的挂在softnet_data列表中 */
    list_add_tail(&napi->poll_list, &sd->poll_list);
    /* 触发网络包接收软中断NET_RX_SOFTIRQ */
    __raise_softirq_irqoff(NET_RX_SOFTIRQ);
}
```

软中断NET_RX_SOFTIRQ对应的中断处理函数是net_rx_action。

```
static void net_rx_action(struct softirq_action *h)
{
    struct softnet_data *sd = this_cpu_ptr(&softnet_data);  /* 获取struct softnet_data结构
*/
    unsigned long time_limit = jiffies + 2;
    int budget = netdev_budget;
    LIST_HEAD(list);
    LIST_HEAD(repoll);

    local_irq_disable();
    list_splice_init(&sd->poll_list, &list);
```

```
    local_irq_enable();

    for (;;) {
        struct napi_struct *n;
        .....
         /* 取出网络包到达的设备 */
        n = list_first_entry(&list, struct napi_struct, poll_list);
        /* 轮询网络包到达的设备 */
        budget -= napi_poll(n, &repoll);
        .....
    }

    .....
}
```

在net_rx_action中，接下来是一个循环，在poll_list里面取出网络包到达的设备，然后调用napi_poll来轮询这些设备。

```
static int napi_poll(struct napi_struct *n, struct list_head *repoll)
{
    .....
    if (test_bit(NAPI_STATE_SCHED, &n->state)) {
        work = n->poll(n, weight);  /* 调用ixgb_clean */
        trace_napi_poll(n);
    }

    .....
}
```

在napi_poll中调用ixgb_clean，ixgb_clean中调用ixgb_clean_rx_irq。

```
static bool
ixgb_clean_rx_irq(struct ixgb_adapter *adapter, int *work_done, int work_to_do)
{
    struct ixgb_desc_ring *rx_ring = &adapter->rx_ring;      /* 接收网络包的环形缓冲区描述符 */
    struct net_device *netdev = adapter->netdev;
    struct pci_dev *pdev = adapter->pdev;
    struct ixgb_rx_desc *rx_desc, *next_rxd;
    struct ixgb_buffer *buffer_info, *next_buffer, *next2_buffer;
    u32 length;
    unsigned int i, j;
    int cleaned_count = 0;
    bool cleaned = false;

    i = rx_ring->next_to_clean;
    rx_desc = IXGB_RX_DESC(*rx_ring, i);
    buffer_info = &rx_ring->buffer_info[i];      /* 环形缓冲区 */

    while (rx_desc->status & IXGB_RX_DESC_STATUS_DD) {
        struct sk_buff *skb;
        .....
        skb = buffer_info->skb;          /* 取缓冲区中未处理的第一个包 */
```

```
        buffer_info->skb = NULL;

        prefetch(skb->data - NET_IP_ALIGN);

        if (++i == rx_ring->count)
            i = 0;
        next_rxd = IXGB_RX_DESC(*rx_ring, i);
        prefetch(next_rxd);

        j = i + 1;
        if (j == rx_ring->count)
            j = 0;
        next2_buffer = &rx_ring->buffer_info[j];
        prefetch(next2_buffer);

        next_buffer = &rx_ring->buffer_info[i];     /* 取缓冲区中的下一个包 */

        .....

        /* 将缓冲区的第一个包拷贝到skb */
        ixgb_check_copybreak(&adapter->napi, buffer_info, length, &skb);

        .....

        /* 进入内核协议栈mac层处理skb */
        netif_receive_skb(skb);

rxdesc_done:
        /* clean up descriptor, might be written over by hw */
        rx_desc->status = 0;
        .....
        /* use prefetched values */
        rx_desc = next_rxd;
        buffer_info = next_buffer;      /* 缓冲区指针指向下一个包 */
    }

    .....
}
```

在网络设备的驱动层，有一个用于接收网络包的rx_ring。它是缓冲区描述符，rx_ring中的buffer_info是缓冲区，存放sk_buff。ixgb_check_copybreak函数将buffer_info里面的一个包拷贝到struct sk_buff *skb，然后通过netif_receive_skb进入协议栈处理网络包。

2、网络协议栈的MAC层逻辑

从netif_receive_skb函数开始就进入了内核的网络协议栈，调用链为:

> netif_receive_skb -> netif_receive_skb_internal -> __netif_receive_skb -> __netif_receive_skb_core

```
static int __netif_receive_skb_core(struct sk_buff *skb, bool pfmemalloc)
{
    struct packet_type *ptype, *pt_prev;
    .....
```

```
    type = skb->protocol;
    .....
    deliver_ptype_list_skb(skb, &pt_prev, orig_dev, type,
                  &orig_dev->ptype_specific);
    .....
    if (pt_prev) {
        .....
        ret = pt_prev->func(skb, skb->dev, pt_prev, orig_dev);
    }
    .....
}


static inline void deliver_ptype_list_skb(struct sk_buff *skb,
                      struct packet_type **pt,
                      struct net_device *dev, __be16 type,
                      struct list_head *ptype_list)
{
    struct packet_type *ptype, *pt_prev = *pt;

    list_for_each_entry_rcu(ptype, ptype_list, list) {
        if (ptype->type != type)
            continue;
        if (pt_prev)
            deliver_skb(skb, pt_prev, dev);
        pt_prev = ptype;
    }
    *pt = pt_prev;
}
```

在网络包struct sk_buff里面，二层的头里面有一个protocol，表示里面一层，也即三层是什么协议。deliver_ptype_list_skb在一个协议列表中逐个匹配。如果能够匹配到，就返回。这些协议的注册在网络协议栈初始化的时候， inet_init函数调用dev_add_pack(&ip_packet_type)，添加IP协议。协议被放在一个链表里面。

```
void dev_add_pack(struct packet_type *pt)
{
    struct list_head *head = ptype_head(pt);

    spin_lock(&ptype_lock);
    list_add_rcu(&pt->list, head);
    spin_unlock(&ptype_lock);
}

static inline struct list_head *ptype_head(const struct packet_type *pt)
{
    if (pt->type == htons(ETH_P_ALL))
        return pt->dev ? &pt->dev->ptype_all : &ptype_all;
    else
        return pt->dev ? &pt->dev->ptype_specific :
                 &ptype_base[ntohs(pt->type) & PTYPE_HASH_MASK];
}
```

假设这个时候的网络包是一个IP包，则在这个链表里面一定能够找到ip_packet_type，在__netif_receive_skb_core中会调用ip_packet_type的func函数。

```
static struct packet_type ip_packet_type __read_mostly = {
    .type = cpu_to_be16(ETH_P_IP),
    .func = ip_rcv,
};
```

从上面的定义我们可以看出，接下来，ip_rcv会被调用。

3、网络协议栈的IP层

从ip_rcv函数开始，我们的处理逻辑就从二层到了三层，IP层。

```
int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt, struct
net_device *orig_dev)
{
    const struct iphdr *iph;
    u32 len;

    .....

    iph = ip_hdr(skb);
    .....
    len = ntohs(iph->tot_len);
    .....

    skb->transport_header = skb->network_header + iph->ihl*4;

    .....

    return NF_HOOK(NFPROTO_IPV4, NF_INET_PRE_ROUTING, skb, dev, NULL,
               ip_rcv_finish);
    .....
}
```

在ip_rcv中，得到IP头，然后又遇到了我们见过多次的NF_HOOK，这次因为是接收网络包，第一个hook点是NF_INET_PRE_ROUTING，也就是iptables的PREROUTING链。如果里面有规则，则执行规则，然后调用ip_rcv_finish。

```
static int ip_rcv_finish(struct sk_buff *skb)
{
    const struct iphdr *iph = ip_hdr(skb);
    struct rtable *rt;
    .....
    rt = skb_rtable(skb);
    .....
    return dst_input(skb);
    .....
}
```

```
static inline int dst_input(struct sk_buff *skb)
{
    return skb_dst(skb)->input(skb);
}
```

ip_rcv_finish得到网络包对应的路由表，然后调用dst_input，在dst_input中，调用的是struct rtable的成员的dst的input函数。在rt_dst_alloc中，我们可以看到，input函数指向的是ip_local_deliver。

```
int ip_local_deliver(struct sk_buff *skb)
{
    /*
     *  Reassemble IP fragments.
     */

    if (ip_is_fragment(ip_hdr(skb))) {
        if (ip_defrag(skb, IP_DEFRAG_LOCAL_DELIVER))
            return 0;
    }

    return NF_HOOK(NFPROTO_IPV4, NF_INET_LOCAL_IN, skb, skb->dev, NULL,
               ip_local_deliver_finish);
}
```

在ip_local_deliver函数中，如果IP层进行了分段，则进行重新的组合。接下来就是我们熟悉的NF_HOOK。hook点在NF_INET_LOCAL_IN，对应iptables里面的INPUT链。在经过iptables规则处理完毕后，我们调用ip_local_deliver_finish。

```
static int ip_local_deliver_finish(struct sk_buff *skb)
{
    struct net *net = dev_net(skb->dev);

    __skb_pull(skb, skb_network_header_len(skb));

    rcu_read_lock();
    {
        int protocol = ip_hdr(skb)->protocol;
        const struct net_protocol *ipprot;
        .....
        ipprot = rcu_dereference(inet_protos[protocol]);
        if (ipprot != NULL) {
            int ret;
            .....
            ret = ipprot->handler(skb);
            .....
        }
    }
    .....
}
```

在IP头中，有一个字段protocol用于指定里面一层的协议，在这里应该是TCP协议。于是，从inet_protos数组中，找出TCP协议对应的处理函数。这个数组的定义如下，里面的内容是struct net_protocol。

```
const struct net_protocol __rcu *inet_protos[MAX_INET_PROTOS] __read_mostly;

int inet_add_protocol(const struct net_protocol *prot, unsigned char protocol)
{
    .....

    return !cmpxchg((const struct net_protocol **)&inet_protos[protocol],
            NULL, prot) ? 0 : -1;
}

static int __init inet_init(void)
{
    .....
    if (inet_add_protocol(&icmp_protocol, IPPROTO_ICMP) < 0)
        pr_crit("%s: Cannot add ICMP protocol\n", __func__);
    if (inet_add_protocol(&udp_protocol, IPPROTO_UDP) < 0)
        pr_crit("%s: Cannot add UDP protocol\n", __func__);
    if (inet_add_protocol(&tcp_protocol, IPPROTO_TCP) < 0)
        pr_crit("%s: Cannot add TCP protocol\n", __func__);
    .....

}

static const struct net_protocol tcp_protocol = {
    .early_demux    =   tcp_v4_early_demux,
    .handler    =   tcp_v4_rcv,
    .err_handler    =   tcp_v4_err,
    .no_policy  =   1,
    .netns_ok   =   1,
    .icmp_strict_tag_validation = 1,
};

static const struct net_protocol udp_protocol = {
    .early_demux =  udp_v4_early_demux,
    .handler =  udp_rcv,
    .err_handler =  udp_err,
    .no_policy =    1,
    .netns_ok = 1,
};
```

在系统初始化的时候，网络协议栈的初始化调用的是inet_init，它会调用inet_add_protocol，将TCP协议对应的处理函数tcp_protocol、UDP协议对应的处理函数udp_protocol，放到inet_protos数组中。在上面的网络包的接收过程中，会取出TCP协议对应的处理函数tcp_protocol，然后调用handler函数，也即tcp_v4_rcv函数。

二、IP层到SOCKET层的接收过程

三、总结

1、内核接收网络包

- 硬件网卡接收到网络包之后，通过DMA技术，将网络包放入Ring Buffer；
- 硬件网卡通过中断通知CPU新的网络包的到来；
- 网卡驱动程序会注册中断处理函数ixgb_intr；
- 中断处理函数处理完需要暂时屏蔽中断的核心流程之后，通过软中断NET_RX_SOFTIRQ触发接下来的处理过程；
- NET_RX_SOFTIRQ软中断处理函数net_rx_action，net_rx_action会调用napi_poll，进而调用ixgb_clean_rx_irq，从Ring Buffer中读取数据到内核struct sk_buff；
- 调用netif_receive_skb进入内核网络协议栈，进行一些关于VLAN的二层逻辑处理后，调用ip_rcv进入三层IP层；
- 在IP层，会处理iptables规则，然后调用ip_local_deliver交给更上层TCP层；
- 在TCP层调用tcp_v4_rcv，这里面有三个队列需要处理，如果当前的Socket不是正在被读取，则放入backlog队列，如果正在被读取，不需要很实时的话，则放入prequeue队列，其他情况调用tcp_v4_do_rcv；
- 在tcp_v4_do_rcv中，如果是处于TCP_ESTABLISHED状态，调用tcp_rcv_established，其他的状态，调用tcp_rcv_state_process；
- 在tcp_rcv_established中，调用tcp_data_queue，如果序列号能够接的上，则放入sk_receive_queue队列；如果序列号接不上，则暂时放入out_of_order_queue队列，等序列号能够接上的时候，再放入sk_receive_queue队列。

2、用户读取网络包

- **VFS层：**read系统调用找到struct file，根据里面的file_operations的定义，调用sock_read_iter函数，sock_read_iter函数调用sock_recvmsg函数。
- **Socket层：**从struct file里面的private_data得到struct socket，根据里面ops的定义，调用inet_recvmsg函数。
- **Sock层：**从struct socket里面的sk得到struct sock，根据里面sk_prot的定义，调用tcp_recvmsg函数。
- **TCP层：**tcp_recvmsg函数会依次读取receive_queue队列、prequeue队列和backlog队列。