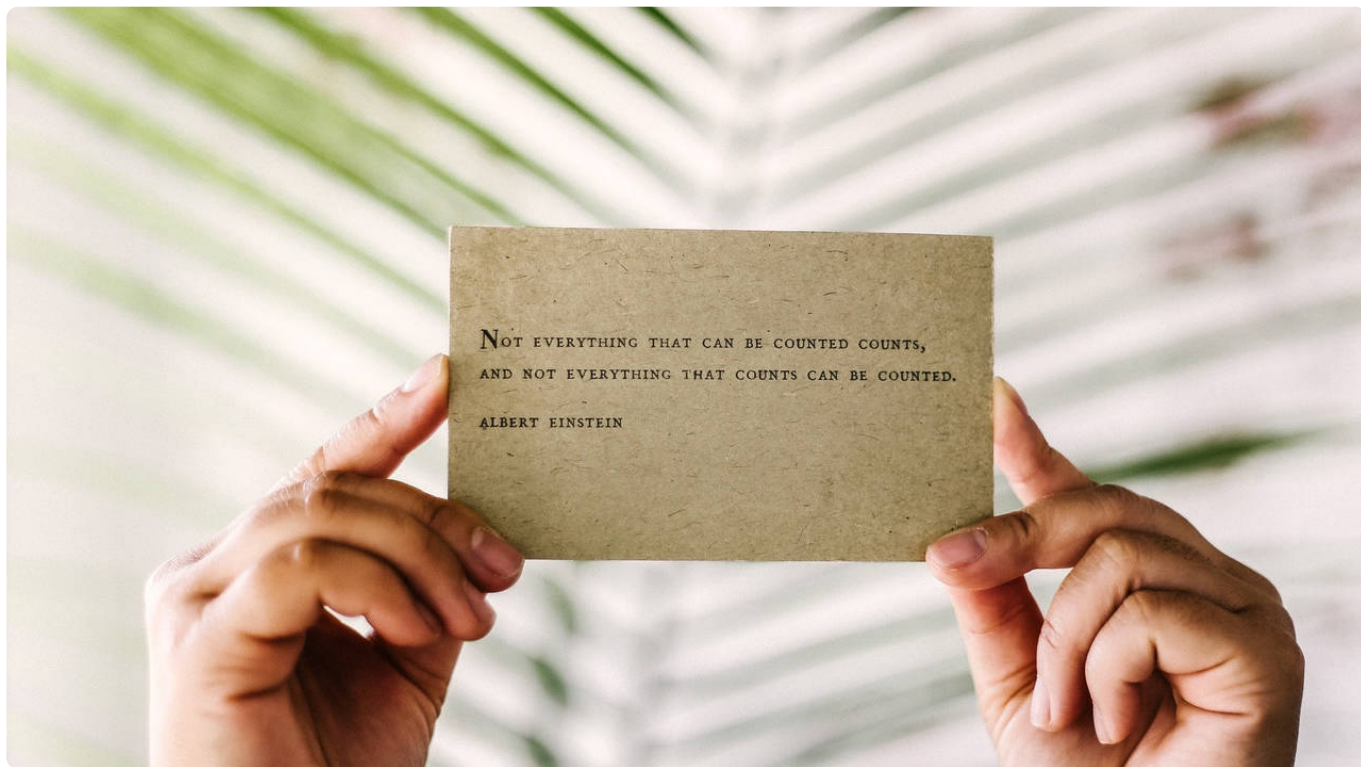


47 | 接收网络包（上）：如何搞明白合作伙伴让我们做什么？

2019-07-15 刘超

趣谈Linux操作系统

[进入课程 >](#)



讲述：刘超

时长 12:57 大小 11.87M



前面两节，我们分析了发送网络包的整个过程。这一节，我们来解析接收网络包的过程。

如果说网络包的发送是从应用层开始，层层调用，一直到网卡驱动程序的话，网络包的结束过程，就是一个反过来的过程，我们不能从应用层的读取开始，而应该从网卡接收到一个网络包开始。我们用两节来解析这个过程，这一节我们从硬件网卡解析到 IP 层，下一节，我们从 IP 层解析到 Socket 层。

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？咱们学习过输入输出设备和中断。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**。

为了帮你了解设备驱动层的工作机制，我们还是以上一节发送网络包时的网卡 `drivers/net/ethernet/intel/ixgb/ixgb_main.c` 为例子，来进行解析。

 复制代码


```
1 static struct pci_driver ixgb_driver = {
2     .name      = ixgb_driver_name,
3     .id_table  = ixgb_pci_tbl,
4     .probe     = ixgb_probe,
5     .remove    = ixgb_remove,
6     .err_handler = &ixgb_err_handler
7 };
8
9 MODULE_AUTHOR("Intel Corporation, <linux.nics@intel.com>");
10 MODULE_DESCRIPTION("Intel(R) PRO/10GbE Network Driver");
11 MODULE_LICENSE("GPL");
12 MODULE_VERSION(DRV_VERSION);
13
14 /**
15  * ixgb_init_module - Driver Registration Routine
16  *
17  * ixgb_init_module is the first routine called when the driver is
18  * loaded. All it does is register with the PCI subsystem.
19  */
20
21 static int __init
22 ixgb_init_module(void)
23 {
24     pr_info("%s - version %s\n", ixgb_driver_string, ixgb_driver_version);
25     pr_info("%s\n", ixgb_copyright);
```

```

26
27     return pci_register_driver(&ixgb_driver);
28 }
29
30 module_init(ixgb_init_module);

```

在网卡驱动程序初始化的时候，我们会调用 `ixgb_init_module`，注册一个驱动 `ixgb_driver`，并且调用它的 `probe` 函数 `ixgb_probe`。

 复制代码

```


1 static int
2 ixgb_probe(struct pci_dev *pdev, const struct pci_device_id *ent)
3 {
4     struct net_device *netdev = NULL;
5     struct ixgb_adapter *adapter;
6     .....
7     netdev = alloc_etherdev(sizeof(struct ixgb_adapter));
8     SET_NETDEV_DEV(netdev, &pdev->dev);
9
10    pci_set_drvdata(pdev, netdev);
11    adapter = netdev_priv(netdev);
12    adapter->netdev = netdev;
13    adapter->pdev = pdev;
14    adapter->hw.back = adapter;
15    adapter->msg_enable = netif_msg_init(debug, DEFAULT_MSG_ENABLE);
16
17    adapter->hw.hw_addr = pci_ioremap_bar(pdev, BAR_0);
18    .....
19    netdev->netdev_ops = &ixgb_netdev_ops;
20    ixgb_set_ethtool_ops(netdev);
21    netdev->watchdog_timeo = 5 * HZ;
22    netif_napi_add(netdev, &adapter->napi, ixgb_clean, 64);
23
24    strncpy(netdev->name, pci_name(pdev), sizeof(netdev->name) - 1);
25
26    adapter->bd_number = cards_found;
27    adapter->link_speed = 0;
28    adapter->link_duplex = 0;
29    .....
30 }

```

在 `ixgb_probe` 中，我们会创建一个 `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，就是要通过他来轮询了。

当一个网卡被激活的时候，我们会调用函数 `ixgb_open->ixgb_up`，在这里面注册一个硬件的中断处理函数。

 复制代码

```
1 int
2 ixgb_up(struct ixgb_adapter *adapter)
3 {
4     struct net_device *netdev = adapter->netdev;
5     .....
6     err = request_irq(adapter->pdev->irq, ixgb_intr, irq_flags,
7                       netdev->name, netdev);
8     .....
9 }
10
11 /**
12  * ixgb_intr - Interrupt Handler
13  * @irq: interrupt number
14  * @data: pointer to a network interface device structure
15  */
16
17 static irqreturn_t
18 ixgb_intr(int irq, void *data)
19 {
20     struct net_device *netdev = data;
21     struct ixgb_adapter *adapter = netdev_priv(netdev);
22     struct ixgb_hw *hw = &adapter->hw;
23     .....
24     if (napi_schedule_prep(&adapter->napi)) {
25         IXGB_WRITE_REG(&adapter->hw, IMC, ~0);
26         __napi_schedule(&adapter->napi);
27     }
28     return IRQ_HANDLED;
29 }
```

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`。

 复制代码

```
1 /**
2  * __napi_schedule - schedule for receive
3  * @n: entry to schedule
4  *
```

```


5  * The entry's receive function will be scheduled to run.
6  * Consider using __napi_schedule_irqoff() if hard irqs are masked.
7  */
8  void __napi_schedule(struct napi_struct *n)
9  {
10     unsigned long flags;
11
12     local_irq_save(flags);
13     ____napi_schedule(this_cpu_ptr(&softnet_data), n);
14     local_irq_restore(flags);
15 }
16
17 static inline void ____napi_schedule(struct softnet_data *sd,
18                                     struct napi_struct *napi)
19 {
20     list_add_tail(&napi->poll_list, &sd->poll_list);
21     __raise_softirq_irqoff(NET_RX_SOFTIRQ);
22 }

```

`__napi_schedule` 是处于中断处理的关键部分，在他被调用的时候，中断是暂时关闭的，但是处理网络包是个复杂的过程，需要到延迟处理部分，所以 `____napi_schedule` 将当前设备放到 `struct softnet_data` 结构的 `poll_list` 里面，说明在延迟处理部分可以接着处理这个 `poll_list` 里面的网络设备。

然后 `____napi_schedule` 触发一个软中断 `NET_RX_SOFTIRQ`，通过软中断触发中断处理的延迟处理部分，也是常用的手段。

上一节，我们知道，软中断 `NET_RX_SOFTIRQ` 对应的中断处理函数是 `net_rx_action`。


 复制代码

```

1  static __latent_entropy void net_rx_action(struct softirq_action *h)
2  {
3     struct softnet_data *sd = this_cpu_ptr(&softnet_data);
4     LIST_HEAD(list);
5     list_splice_init(&sd->poll_list, &list);
6     .....
7     for (;;) {
8         struct napi_struct *n;
9         .....
10         n = list_first_entry(&list, struct napi_struct, poll_list);
11         budget -= napi_poll(n, &repoll);
12     }
13     .....
14 }

```


在 `net_rx_action` 中，会得到 `struct softnet_data` 结构，这个结构在发送的时候我们也遇到过。当时它的 `output_queue` 用于网络包的发送，这里的 `poll_list` 用于网络包的接收。

 复制代码

```
1 struct softnet_data {
2     struct list_head      poll_list;
3     .....
4     struct Qdisc          *output_queue;
5     struct Qdisc          **output_queue_tailp;
6     .....
7 }
```

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，`napi_poll` 会调用最初设备初始化的时候，注册的 `poll` 函数，对于 `ixgb_driver`，对应的函数是 `ixgb_clean`。

`ixgb_clean` 会调用 `ixgb_clean_rx_irq`。

 复制代码

```
1 static bool
2 ixgb_clean_rx_irq(struct ixgb_adapter *adapter, int *work_done, int work_to_do)
3 {
4     struct ixgb_desc_ring *rx_ring = &adapter->rx_ring;
5     struct net_device *netdev = adapter->netdev;
6     struct pci_dev *pdev = adapter->pdev;
7     struct ixgb_rx_desc *rx_desc, *next_rxd;
8     struct ixgb_buffer *buffer_info, *next_buffer, *next2_buffer;
9     u32 length;
10    unsigned int i, j;
11    int cleaned_count = 0;
12    bool cleaned = false;
13
14    i = rx_ring->next_to_clean;
15    rx_desc = IXGB_RX_DESC(*rx_ring, i);
16    buffer_info = &rx_ring->buffer_info[i];
17
18    while (rx_desc->status & IXGB_RX_DESC_STATUS_DD) {
19        struct sk_buff *skb;
20        u8 status;
21
```

```

22         status = rx_desc->status;
23         skb = buffer_info->skb;
24         buffer_info->skb = NULL;
25
26         prefetch(skb->data - NET_IP_ALIGN);
27
28         if (++i == rx_ring->count)
29             i = 0;
30         next_rxd = IXGB_RX_DESC(*rx_ring, i);
31         prefetch(next_rxd);
32
33         j = i + 1;
34         if (j == rx_ring->count)
35             j = 0;
36         next2_buffer = &rx_ring->buffer_info[j];
37         prefetch(next2_buffer);
38
39         next_buffer = &rx_ring->buffer_info[i];
40         .....
41         length = le16_to_cpu(rx_desc->length);
42         rx_desc->length = 0;
43         .....
44         ixgb_check_copybreak(&adapter->napi, buffer_info, length, &skb);
45
46         /* Good Receive */
47         skb_put(skb, length);
48
49         /* Receive Checksum Offload */
50         ixgb_rx_checksum(adapter, rx_desc, skb);
51
52         skb->protocol = eth_type_trans(skb, netdev);
53
54         netif_receive_skb(skb);
55         .....
56         /* use prefetched values */
57         rx_desc = next_rxd;
58         buffer_info = next_buffer;
59     }
60
61     rx_ring->next_to_clean = i;
62     .....
63 }

```

在网络设备的驱动层，有一个用于接收网络包的 `rx_ring`。它是一个环，从网卡硬件接收的包会放在这个环里面。这个环里面的 `buffer_info[]` 是一个数组，存放的是网络包的内容。`i` 和 `j` 是这个数组的下标，在 `ixgb_clean_rx_irq` 里面的 `while` 循环中，依次处理环里面的数

据。在这里面，我们看到了 i 和 j 加一之后，如果超过了数组的大小，就跳回下标 0，就说明这是一个环。

ixgb_check_copybreak 函数将 buffer_info 里面的内容，拷贝到 struct sk_buff *skb，从而可以作为一个网络包进行后续的处理，然后调用 netif_receive_skb。

网络协议栈的二层逻辑

从 netif_receive_skb 函数开始，我们就进入了内核的网络协议栈。

接下来的调用链为：netif_receive_skb->netif_receive_skb_internal->__netif_receive_skb->__netif_receive_skb_core。

在 __netif_receive_skb_core 中，我们先是处理了二层的一些逻辑。例如，对于 VLAN 的处理，接下来要想办法交给第三层。

 复制代码

```
1 static int __netif_receive_skb_core(struct sk_buff *skb, bool pfmemalloc)
2 {
3     struct packet_type *ptype, *pt_prev;
4     .....
5     type = skb->protocol;
6     .....
7     deliver_ptype_list_skb(skb, &pt_prev, orig_dev, type,
8                             &orig_dev->ptype_specific);
9     if (pt_prev) {
10         ret = pt_prev->func(skb, skb->dev, pt_prev, orig_dev);
11     }
12     .....
13 }
14
15 static inline void deliver_ptype_list_skb(struct sk_buff *skb,
16                                           struct packet_type **pt,
17                                           struct net_device *orig_dev,
18                                           __be16 type,
19                                           struct list_head *ptype_list)
20 {
21     struct packet_type *ptype, *pt_prev = *pt;
22
23     list_for_each_entry_rcu(ptype, ptype_list, list) {
24         if (ptype->type != type)
25             continue;
26         if (pt_prev)
27             deliver_skb(skb, pt_prev, orig_dev);
```




```

28         pt_prev = ptype;
29     }
30     *pt = pt_prev;
31 }

```

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。


 复制代码

```

1 void dev_add_pack(struct packet_type *pt)
2 {
3     struct list_head *head = ptype_head(pt);
4     list_add_rcu(&pt->list, head);
5 }
6
7 static inline struct list_head *ptype_head(const struct packet_type *pt)
8 {
9     if (pt->type == htons(ETH_P_ALL))
10         return pt->dev ? &pt->dev->ptype_all : &ptype_all;
11     else
12         return pt->dev ? &pt->dev->ptype_specific : &ptype_base[ntohs(pt->type) & PTYPE_
13 }

```

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。

 复制代码

```


1 static struct packet_type ip_packet_type __read_mostly = {
2     .type = cpu_to_be16(ETH_P_IP),
3     .func = ip_rcv,
4 };

```

从上面的定义我们可以看出，接下来，ip_rcv 会被调用。


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。

 复制代码


```
1 int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt, struct net_device **pdev)
2 {
3     const struct iphdr *iph;
4     struct net *net;
5     u32 len;
6     .....
7     net = dev_net(dev);
8     .....
9     iph = ip_hdr(skb);
10    len = ntohs(iph->tot_len);
11    skb->transport_header = skb->network_header + iph->ihl*4;
12    .....
13    return NF_HOOK(NFPROTO_IPV4, NF_INET_PRE_ROUTING,
14                  net, NULL, skb, dev, NULL,
15                  ip_rcv_finish);
16    .....
17 }
```

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。

 复制代码


```
1 static int ip_rcv_finish(struct net *net, struct sock *sk, struct sk_buff *skb)
2 {
3     const struct iphdr *iph = ip_hdr(skb);
4     struct net_device *dev = skb->dev;
5     struct rtable *rt;
6     int err;
7     .....
8     rt = skb_rtable(skb);
9     .....
10    return dst_input(skb);
11 }
12
13 static inline int dst_input(struct sk_buff *skb)
14 {
15     return skb_dst(skb)->input(skb);
16 }
```

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。

 复制代码


```
1 int ip_local_deliver(struct sk_buff *skb)
2 {
3     /*
4      *      Reassemble IP fragments.
5      */
6     struct net *net = dev_net(skb->dev);
7
8     if (ip_is_fragment(ip_hdr(skb))) {
9         if (ip_defrag(net, skb, IP_DEFRAG_LOCAL_DELIVER))
10             return 0;
11     }
12
13     return NF_HOOK(NFPROTO_IPV4, NF_INET_LOCAL_IN,
14                   net, NULL, skb, skb->dev, NULL,
15                   ip_local_deliver_finish);
16 }
```

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。

 复制代码

```
1 static int ip_local_deliver_finish(struct net *net, struct sock *sk, struct sk_buff *skb)
2 {
3     __skb_pull(skb, skb_network_header_len(skb));
4
5     int protocol = ip_hdr(skb)->protocol;
6     const struct net_protocol *ipprot;
7
8     ipprot = rcu_dereference(inet_protos[protocol]);
9     if (ipprot) {
10         int ret;
11         ret = ipprot->handler(skb);
12         .....
13     }
14     .....
```

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`。

 复制代码

```

1 struct net_protocol __rcu *inet_protos[MAX_INET_PROTOS] __read_mostly;
2
3 int inet_add_protocol(const struct net_protocol *prot, unsigned char protocol)
4 {
5     .....
6     return !cmpxchg((const struct net_protocol **)&inet_protos[protocol],
7                     NULL, prot) ? 0 : -1;
8 }
9
10 static int __init inet_init(void)
11 {
12     .....
13     if (inet_add_protocol(&udp_protocol, IPPROTO_UDP) < 0)
14         pr_crit("%s: Cannot add UDP protocol\n", __func__);
15     if (inet_add_protocol(&tcp_protocol, IPPROTO_TCP) < 0)
16         pr_crit("%s: Cannot add TCP protocol\n", __func__);
17     .....
18 }
19
20 static struct net_protocol tcp_protocol = {
21     .early_demux      =      tcp_v4_early_demux,
22     .early_demux_handler = tcp_v4_early_demux,
23     .handler          =      tcp_v4_rcv,
24     .err_handler      =      tcp_v4_err,
25     .no_policy        =      1,
26     .netns_ok         =      1,
27     .icmp_strict_tag_validation = 1,
28 };
29
30 static struct net_protocol udp_protocol = {
31     .early_demux      =      udp_v4_early_demux,
32     .early_demux_handler = udp_v4_early_demux,
33     .handler          =      udp_rcv,
34     .err_handler      =      udp_err,
35     .no_policy        =      1,
36     .netns_ok         =      1,
37 };

```

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。

总结时刻

这一节我们讲了接收网络包的上半部分，分以下几个层次。

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`。

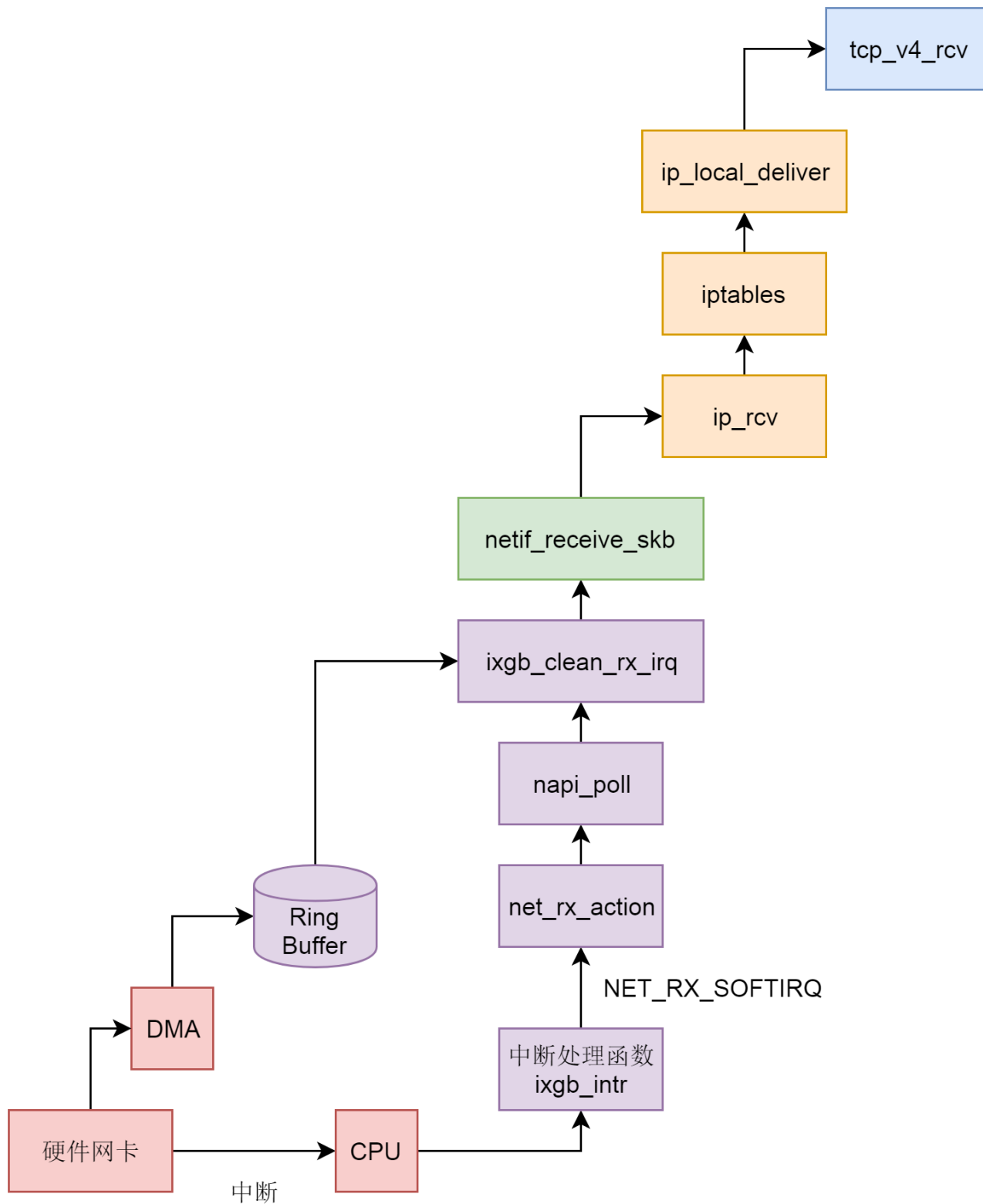
中断处理函数处理完需要暂时屏蔽中断的核心流程之后，通过软中断 `NET_RX_SOFTIRQ` 触发接下来的处理过程。

`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`。



课堂练习

我们没有仔细分析对于二层 VLAN 的处理，请你研究一下 VLAN 的原理，然后在代码中看一下对于 VLAN 的处理过程，这是一项重要的网络基础知识。

欢迎留言和我分享你的疑惑和见解，也欢迎可以收藏本节内容，反复研读。你也可以把今天的内容分享给你的朋友，和他一起学习和进步。



趣谈 Linux 操作系统

像故事一样的操作系统入门课

刘超

网易杭州研究院
云计算技术部首席架构师



新版升级：点击「👤请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 46 | 发送网络包（下）：如何表达我们想让合作伙伴做什么？

精选留言 (3)

写留言



许童童

2019-07-15

VLAN 的原理有些忘了，希望老师可以在答疑中给我们答疑一下。



Cyril

2019-07-15

老师能否详细写一点关于 smp 相关的知识，比如多 cpu 如何处理网卡过来的中断，多 cpu 如何进程调度，多 cpu 又是如何解决共享变量访问冲突的问题，对这一部分知识点一直比较模糊



安排

2019-07-15

牛

展开

