

48 | 接收网络包（下）：如何搞明白合作伙伴让我们做什么？

2019-07-17 刘超

趣谈Linux操作系统

[进入课程 >](#)



讲述：刘超

时长 18:52 大小 17.29M



上一节，我们解析了网络包接收的上半部分，从硬件网卡到 IP 层。这一节，我们接着来解析 TCP 层和 Socket 层都做了哪些事情。

网络协议栈的 TCP 层

从 `tcp_v4_rcv` 函数开始，我们的处理逻辑就从 IP 层到了 TCP 层。

复制代码

```
1 int tcp_v4_rcv(struct sk_buff *skb)
2 {
3     struct net *net = dev_net(skb->dev);
4     const struct iphdr *iph;
5     const struct tcphdr *th;
6     bool refcounted;
```

```

7      struct sock *sk;
8      int ret;
9      .....
10     th = (const struct tcphdr *)skb->data;
11     iph = ip_hdr(skb);
12     .....
13     TCP_SKB_CB(skb)->seq = ntohl(th->seq);
14     TCP_SKB_CB(skb)->end_seq = (TCP_SKB_CB(skb)->seq + th->syn + th->fin + skb->len
15     TCP_SKB_CB(skb)->ack_seq = ntohl(th->ack_seq);
16     TCP_SKB_CB(skb)->tcp_flags = tcp_flag_byte(th);
17     TCP_SKB_CB(skb)->tcp_tw_isn = 0;
18     TCP_SKB_CB(skb)->ip_dsfield = ipv4_get_dsfield(iph);
19     TCP_SKB_CB(skb)->sacked = 0;
20
21 lookup:
22     sk = __inet_lookup_skb(&tcp_hashinfo, skb, __tcp_hdrlen(th), th->source, th->de:
23
24 process:
25     if (sk->sk_state == TCP_TIME_WAIT)
26         goto do_time_wait;
27
28     if (sk->sk_state == TCP_NEW_SYN_RECV) {
29         .....
30     }
31     .....
32     th = (const struct tcphdr *)skb->data;
33     iph = ip_hdr(skb);
34
35     skb->dev = NULL;
36
37     if (sk->sk_state == TCP_LISTEN) {
38         ret = tcp_v4_do_rcv(sk, skb);
39         goto put_and_return;
40     }
41     .....
42     if (!sock_owned_by_user(sk)) {
43         if (!tcp_prequeue(sk, skb))
44             ret = tcp_v4_do_rcv(sk, skb);
45     } else if (tcp_add_backlog(sk, skb)) {
46         goto discard_and_relse;
47     }
48     .....
49 }

```

在 tcp_v4_rcv 中，得到 TCP 的头之后，我们可以开始处理 TCP 层的事情。因为 TCP 层是分状态的，状态被维护在数据结构 struct sock 里面，因而我们要根据 IP 地址以及 TCP 头

里面的内容，在 tcp_hashinfo 中找到这个包对应的 struct sock，从而得到这个包对应的连接的状态。

接下来，我们就根据不同的状态做不同的处理，例如，上面代码中的 TCP_LISTEN、TCP_NEW_SYN_RECV 状态属于连接建立过程中。这个我们在讲三次握手的时候讲过了。再如，TCP_TIME_WAIT 状态是连接结束的时候的状态，这个我们暂时可以不用看。

接下来，我们来分析最主流的网络包的接收过程，这里面涉及三个队列：

backlog 队列

prequeue 队列

sk_receive_queue 队列

为什么接收网络包的过程，需要在这三个队列里面倒腾过来、倒腾过去呢？这是因为，同样一个网络包要在三个主体之间交接。

第一个主体是**软中断的处理过程**。如果你没忘记的话，我们在执行 tcp_v4_rcv 函数的时候，依然处于软中断的处理逻辑里，所以必然会占用这个软中断。

第二个主体就是**用户态进程**。如果用户态触发系统调用 read 读取网络包，也要从队列里面找。


第三个主体就是**内核协议栈**。哪怕用户进程没有调用 read，读取网络包，当网络包来的时候，也得有一个地方收着呀。

这时候，我们就能够了解上面代码中 sock_owned_by_user 的意思了，其实就是说，当前这个 sock 是不是正有一个用户态进程等着读数据呢，如果没有，内核协议栈也调用 tcp_add_backlog，暂存在 backlog 队列中，并且抓紧离开软中断的处理过程。

如果有一个用户态进程等待读取数据呢？我们先调用 tcp_prequeue，也即赶紧放入 prequeue 队列，并且离开软中断的处理过程。在这个函数里面，我们会看到对于 sysctl_tcp_low_latency 的判断，也即是不是要低时延地处理网络包。

如果把 sysctl_tcp_low_latency 设置为 0，那就要放在 prequeue 队列中暂存，这样不用等待网络包处理完毕，就可以离开软中断的处理过程，但是会造成比较长的时延。如果把

sysctl_tcp_low_latency 设置为 1，我们还是调用 tcp_v4_do_rcv。

 复制代码

```
1 int tcp_v4_do_rcv(struct sock *sk, struct sk_buff *skb)
2 {
3     struct sock *rsk;
4
5     if (sk->sk_state == TCP_ESTABLISHED) { /* Fast path */
6         struct dst_entry *dst = sk->sk_rx_dst;
7         .....
8         tcp_rcv_established(sk, skb, tcp_hdr(skb), skb->len);
9         return 0;
10    }
11    .....
12    if (tcp_rcv_state_process(sk, skb)) {
13        .....
14    }
15    return 0;
16    .....
17 }
```

在 tcp_v4_do_rcv 中，分两种情况，一种情况是连接已经建立，处于 TCP_ESTABLISHED 状态，调用 tcp_rcv_established。另一种情况，就是其他的状态，调用 tcp_rcv_state_process。

 复制代码

```
1 int tcp_rcv_state_process(struct sock *sk, struct sk_buff *skb)
2 {
3     struct tcp_sock *tp = tcp_sk(sk);
4     struct inet_connection_sock *icsk = inet_csk(sk);
5     const struct tcphdr *th = tcp_hdr(skb);
6     struct request_sock *req;
7     int queued = 0;
8     bool acceptable;
9
10    switch (sk->sk_state) {
11        case TCP_CLOSE:
12            .....
13        case TCP_LISTEN:
14            .....
15        case TCP_SYN_SENT:
16            .....
17    }
18    .....
19    switch (sk->sk_state) {
```

```
20         case TCP_SYN_RECV:
21         .....
22         case TCP_FIN_WAIT1:
23         .....
24         case TCP_CLOSING:
25         .....
26         case TCP_LAST_ACK:
27         .....
28     }
29
30     /* step 7: process the segment text */
31     switch (sk->sk_state) {
32     case TCP_CLOSE_WAIT:
33     case TCP_CLOSING:
34     case TCP_LAST_ACK:
35     .....
36     case TCP_FIN_WAIT1:
37     case TCP_FIN_WAIT2:
38     .....
39     case TCP_ESTABLISHED:
40     .....
41     }
42 }
```

在 tcp_rcv_state_process 中，如果我们对 TCP 的状态图进行比对，能看到，对于 TCP 所有状态的处理，其中和连接建立相关的状态，咱们已经分析过，所以我们重点关注连接状态下的工作模式。


```

1 static void tcp_data_queue(struct sock *sk, struct sk_buff *skb)
2 {
3     struct tcp_sock *tp = tcp_sk(sk);
4     bool fragstolen = false;
5     .....
6     if (TCP_SKB_CB(skb)->seq == tp->rcv_nxt) {
7         if (tcp_receive_window(tp) == 0)
8             goto out_of_window;
9
10        /* Ok. In sequence. In window. */
11        if (tp->ucopy.task == current &&
12            tp->copied_seq == tp->rcv_nxt && tp->ucopy.len &&
13            sock_owned_by_user(sk) && !tp->urg_data) {
14            int chunk = min_t(unsigned int, skb->len,
15                             tp->ucopy.len);
16
17            __set_current_state(TASK_RUNNING);
18
19            if (!skb_copy_datagram_msg(skb, 0, tp->ucopy.msg, chunk)) {
20                tp->ucopy.len -= chunk;
21                tp->copied_seq += chunk;
22                eaten = (chunk == skb->len);
23                tcp_rcv_space_adjust(sk);
24            }
25        }
26
27        if (eaten <= 0) {
28queue_and_out:
29        .....
30            eaten = tcp_queue_rcv(sk, skb, 0, &fragstolen);
31        }
32        tcp_rcv_nxt_update(tp, TCP_SKB_CB(skb)->end_seq);
33        .....
34        if (!RB_EMPTY_ROOT(&tp->out_of_order_queue)) {
35            tcp_ofo_queue(sk);
36        .....
37        }
38        .....
39        return;
40    }
41
42    if (!after(TCP_SKB_CB(skb)->end_seq, tp->rcv_nxt)) {
43        /* A retransmit, 2nd most common case. Force an immediate ack. */
44        tcp_dsack_set(sk, TCP_SKB_CB(skb)->seq, TCP_SKB_CB(skb)->end_seq);
45
46    out_of_window:
47        tcp_enter_quickack_mode(sk);
48        inet_csk_schedule_ack(sk);
49    drop:
50        tcp_drop(sk, skb);

```

```

51         return;
52     }
53
54     /* Out of window. F.e. zero window probe. */
55     if (!before(TCP_SKB_CB(skb)->seq, tp->rcv_nxt + tcp_receive_window(tp)))
56         goto out_of_window;
57
58     tcp_enter_quickack_mode(sk);
59
60     if (before(TCP_SKB_CB(skb)->seq, tp->rcv_nxt)) {
61         /* Partial packet, seq < rcv_next < end_seq */
62         tcp_dsack_set(sk, TCP_SKB_CB(skb)->seq, tp->rcv_nxt);
63         /* If window is closed, drop tail of packet. But after
64          * remembering D-SACK for its head made in previous line.
65          */
66         if (!tcp_receive_window(tp))
67             goto out_of_window;
68         goto queue_and_out;
69     }
70
71     tcp_data_queue_ofo(sk, skb);
72 }

```

在 `tcp_data_queue` 中，对于收到的网络包，我们要分情况进行处理。

第一种情况，`seq == tp->rcv_nxt`，说明来的网络包正是我服务端期望的下一个网络包。这个时候我们判断 `sock_owned_by_user`，也即用户进程也是正在等待读取，这种情况下，就直接 `skb_copy_datagram_msg`，将网络包拷贝给用户进程就可以了。

如果用户进程没有正在等待读取，或者因为内存原因没有能够拷贝成功，`tcp_queue_rcv` 里面还是将网络包放入 `sk_receive_queue` 队列。

接下来，`tcp_rcv_nxt_update` 将 `tp->rcv_nxt` 设置为 `end_seq`，也即当前的网络包接收成功后，更新下一个期待的网络包。

这个时候，我们还会判断一下另一个队列，`out_of_order_queue`，也看看乱序队列的情况，看看乱序队列里面的包，会不会因为这个新的网络包的到来，也能放入到 `sk_receive_queue` 队列中。

例如，客户端发送的网络包序号为 5、6、7、8、9。在 5 还没有到达的时候，服务端的 `rcv_nxt` 应该是 5，也即期望下一个网络包是 5。但是由于中间网络通路的问题，5、6 还没

到达服务端，7、8 已经到达了服务端了，这就出现了乱序。

乱序的包不能进入 `sk_receive_queue` 队列。因为一旦进入到这个队列，意味着可以发送给用户进程。然而，按照 TCP 的定义，用户进程应该是按顺序收到包的，没有排好序，就不能给用户进程。所以，7、8 不能进入 `sk_receive_queue` 队列，只能暂时放在 `out_of_order_queue` 乱序队列中。

当 5、6 到达的时候，5、6 先进入 `sk_receive_queue` 队列。这个时候我们再来看 `out_of_order_queue` 乱序队列中的 7、8，发现能够接上。于是，7、8 也能进入 `sk_receive_queue` 队列了。`tcp_ofo_queue` 函数就是做这个事情的。

至此第一种情况处理完毕。

第二种情况，`end_seq` 不大于 `rcv_nxt`，也即服务端期望网络包 5。但是，来了一个网络包 3，怎样才会出现这种情况呢？肯定是服务端早就收到了网络包 3，但是 ACK 没有到达客户端，中途丢了，那客户端就认为网络包 3 没有发送成功，于是又发送了一遍，这种情况下，要赶紧给客户端再发送一次 ACK，表示早就收到了。

第三种情况，`seq` 不小于 `rcv_nxt + tcp_receive_window`。这说明客户端发送得太猛了。本来 `seq` 肯定应该在接收窗口里面的，这样服务端才来得及处理，结果现在超出了接收窗口，说明客户端一下子把服务端给塞满了。

这种情况下，服务端不能再接收数据包了，只能发送 ACK 了，在 ACK 中会将接收窗口为 0 的情况告知客户端，客户端就知道不能再发送了。这个时候双方只能交互窗口探测数据包，直到服务端因为用户进程把数据读走了，空出接收窗口，才能在 ACK 里面再次告诉客户端，又有窗口了，又能发送数据包了。

第四种情况，`seq` 小于 `rcv_nxt`，但是 `end_seq` 大于 `rcv_nxt`，这说明从 `seq` 到 `rcv_nxt` 这部分网络包原来的 ACK 客户端没有收到，所以重新发送了一次，从 `rcv_nxt` 到 `end_seq` 时新发送的，可以放入 `sk_receive_queue` 队列。

当前四种情况都排除掉了，说明网络包一定是一个乱序包了。这里有点儿难理解，我们还是用上面那个乱序的例子仔细分析一下 `rcv_nxt=5`。

我们假设 `tcp_receive_window` 也是 5，也即超过 10 服务端就接收不了了。当前来的这个网络包既不在 `rcv_nxt` 之前（不是 3 这种），也不在 `rcv_nxt + tcp_receive_window` 之后（不是 11 这种），说明这正在我们期望的接收窗口里面，但是又不是 `rcv_nxt`（不是我们马上期望的网络包 5），这正是上面的例子中网络包 7、8 的情况。

对于网络包 7、8，我们只好调用 `tcp_data_queue_ofo` 进入 `out_of_order_queue` 乱序队列，但是没有关系，当网络包 5、6 到来的时候，我们会走第一种情况，把 7、8 拿出来放到 `sk_receive_queue` 队列中。

至此，网络协议栈的处理过程就结束了。


Socket 层

当接收的网络包进入各种队列之后，接下来我们就要等待用户进程去读取它们了。

读取一个 `socket`，就像读取一个文件一样，读取 `socket` 的文件描述符，通过 `read` 系统调用。


`read` 系统调用对于一个文件描述符的操作，大致过程都是类似的，在文件系统那一节，我们已经详细解析过。最终它会调用到用来表示一个打开文件的结构 `struct file` 指向的 `file_operations` 操作。

对于 `socket` 来讲，它的 `file_operations` 定义如下：

 复制代码


```
1 static const struct file_operations socket_file_ops = {
2     .owner =          THIS_MODULE,
3     .llseek =         no_llseek,
4     .read_iter =      sock_read_iter,
5     .write_iter =     sock_write_iter,
6     .poll =           sock_poll,
7     .unlocked_ioctl = sock_ioctl,
8     .mmap =           sock_mmap,
9     .release =        sock_close,
10    .fsync =           sock_fsync,
11    .sendpage =        sock_sendpage,
12    .splice_write =    generic_splice_sendpage,
13    .splice_read =     sock_splice_read,
14 };
```

按照文件系统的读取流程，调用的是 `sock_read_iter`。

 复制代码


```
1 static ssize_t sock_read_iter(struct kiocb *iocb, struct iov_iter *to)
2 {
3     struct file *file = iocb->ki_filp;
4     struct socket *sock = file->private_data;
5     struct msghdr msg = {.msg_iter = *to,
6                          .msg_iocb = iocb};
7     ssize_t res;
8
9     if (file->f_flags & O_NONBLOCK)
10         msg.msg_flags = MSG_DONTWAIT;
11     .....
12     res = sock_recvmsg(sock, &msg, msg.msg_flags);
13     *to = msg.msg_iter;
14     return res;
15 }
```

在 `sock_read_iter` 中，通过 VFS 中的 `struct file`，将创建好的 `socket` 结构拿出来，然后调用 `sock_recvmsg`，`sock_recvmsg` 会调用 `sock_recvmsg_nosec`。

 复制代码

```
1 static inline int sock_recvmsg_nosec(struct socket *sock, struct msghdr *msg, int flags)
2 {
3     return sock->ops->recvmsg(sock, msg, msg_data_left(msg), flags);
4 }
```

这里调用了 `socket` 的 `ops` 的 `recvmsg`，这个我们遇到好几次了。根据 `inet_stream_ops` 的定义，这里调用的是 `inet_recvmsg`。

 复制代码


```
1 int inet_recvmsg(struct socket *sock, struct msghdr *msg, size_t size,
2                 int flags)
3 {
4     struct sock *sk = sock->sk;
5     int addr_len = 0;
6     int err;
```

```

7 .....
8     err = sk->sk_prot->recvmsg(sk, msg, size, flags & MSG_DONTWAIT,
9                               flags & ~MSG_DONTWAIT, &addr_len);
10 .....
11 }

```

这里面，从 socket 结构，我们可以得到更底层的 sock 结构，然后调用 sk_prot 的 recvmsg 方法。这个同样遇到好几次了，根据 tcp_prot 的定义，调用的是 tcp_recvmsg。

 复制代码

```

1 int tcp_recvmsg(struct sock *sk, struct msghdr *msg, size_t len, int nonblock,
2                 int flags, int *addr_len)
3 {
4     struct tcp_sock *tp = tcp_sk(sk);
5     int copied = 0;
6     u32 peek_seq;
7     u32 *seq;
8     unsigned long used;
9     int err;
10    int target;           /* Read at least this many bytes */
11    long timeo;
12    struct task_struct *user_recv = NULL;
13    struct sk_buff *skb, *last;
14    .....
15    do {
16        u32 offset;
17        .....
18        /* Next get a buffer. */
19        last = skb_peek_tail(&sk->sk_receive_queue);
20        skb_queue_walk(&sk->sk_receive_queue, skb) {
21            last = skb;
22            offset = *seq - TCP_SKB_CB(skb)->seq;
23            if (offset < skb->len)
24                goto found_ok_skb;
25            .....
26        }
27        .....
28        if (!sysctl_tcp_low_latency && tp->ucopy.task == user_recv) {
29            /* Install new reader */
30            if (!user_recv && !(flags & (MSG_TRUNC | MSG_PEEK))) {
31                user_recv = current;
32                tp->ucopy.task = user_recv;
33                tp->ucopy.msg = msg;
34            }
35

```

```

36         tp->ucopy.len = len;
37         /* Look: we have the following (pseudo)queues:
38          *
39          * 1. packets in flight
40          * 2. backlog
41          * 3. prequeue
42          * 4. receive_queue
43          *
44          * Each queue can be processed only if the next ones
45          * are empty.
46          */
47         if (!skb_queue_empty(&tp->ucopy.prequeue))
48             goto do_prequeue;
49     }
50
51     if (copied >= target) {
52         /* Do not sleep, just process backlog. */
53         release_sock(sk);
54         lock_sock(sk);
55     } else {
56         sk_wait_data(sk, &timeo, last);
57     }
58
59     if (user_recv) {
60         int chunk;
61         chunk = len - tp->ucopy.len;
62         if (chunk != 0) {
63             len -= chunk;
64             copied += chunk;
65         }
66
67         if (tp->rcv_nxt == tp->copied_seq &&
68             !skb_queue_empty(&tp->ucopy.prequeue)) {
69 do_prequeue:
70             tcp_prequeue_process(sk);
71
72             chunk = len - tp->ucopy.len;
73             if (chunk != 0) {
74                 len -= chunk;
75                 copied += chunk;
76             }
77         }
78     }
79     continue;
80 found_ok_skb:
81     /* Ok so how much can we use? */
82     used = skb->len - offset;
83     if (len < used)
84         used = len;
85
86     if (!(flags & MSG_TRUNC)) {
87         err = skb_copy_datagram_msg(skb, offset, msg, used);

```

```

88 .....
89         }
90
91         *seq += used;
92         copied += used;
93         len -= used;
94
95         tcp_rcv_space_adjust(sk);
96 .....
97     } while (len > 0);
98 .....
99 }

```

`tcp_recvmsg` 这个函数比较长，里面逻辑也很复杂，好在里面有一段注释概括了这里面的逻辑。注释里面提到了三个队列，`receive_queue` 队列、`prequeue` 队列和 `backlog` 队列。这里面，我们需要把前一个队列处理完毕，才处理后一个队列。


`tcp_recvmsg` 的整个逻辑也是这样执行的：这里面有一个 `while` 循环，不断地读取网络包。

这里，我们会先处理 `sk_receive_queue` 队列。如果找到了网络包，就跳到 `found_ok_skb` 这里。这里会调用 `skb_copy_datagram_msg`，将网络包拷贝到用户进程中，然后直接进入下一层循环。

直到 `sk_receive_queue` 队列处理完毕，我们才到了 `sysctl_tcp_low_latency` 判断。如果不需要低时延，则会有 `prequeue` 队列。于是，我们能就跳到 `do_prequeue` 这里，调用 `tcp_prequeue_process` 进行处理。

如果 `sysctl_tcp_low_latency` 设置为 1，也即没有 `prequeue` 队列，或者 `prequeue` 队列为空，则需要处理 `backlog` 队列，在 `release_sock` 函数中处理。

`release_sock` 会调用 `__release_sock`，这里面会依次处理队列中的网络包。

 复制代码

```

1 void release_sock(struct sock *sk)
2 {
3     .....
4     if (sk->sk_backlog.tail)
5         __release_sock(sk);

```

```

6 .....
7 }
8
9 static void __release_sock(struct sock *sk)
10     __releases(&sk->sk_lock.slock)
11     __acquires(&sk->sk_lock.slock)
12 {
13     struct sk_buff *skb, *next;
14
15     while ((skb = sk->sk_backlog.head) != NULL) {
16         sk->sk_backlog.head = sk->sk_backlog.tail = NULL;
17         do {
18             next = skb->next;
19             prefetch(next);
20             skb->next = NULL;
21             sk_backlog_rcv(sk, skb);
22             cond_resched();
23             skb = next;
24         } while (skb != NULL);
25     }
26 .....
27 }

```

最后，哪里都没有网络包，我们只好调用 `sk_wait_data`，继续等待在哪里，等待网络包的到来。

至此，网络包的接收过程到此结束。

总结时刻

这一节我们讲完了接收网络包，我们来从头串一下，整个过程可以分成以下几个层次。

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`，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。

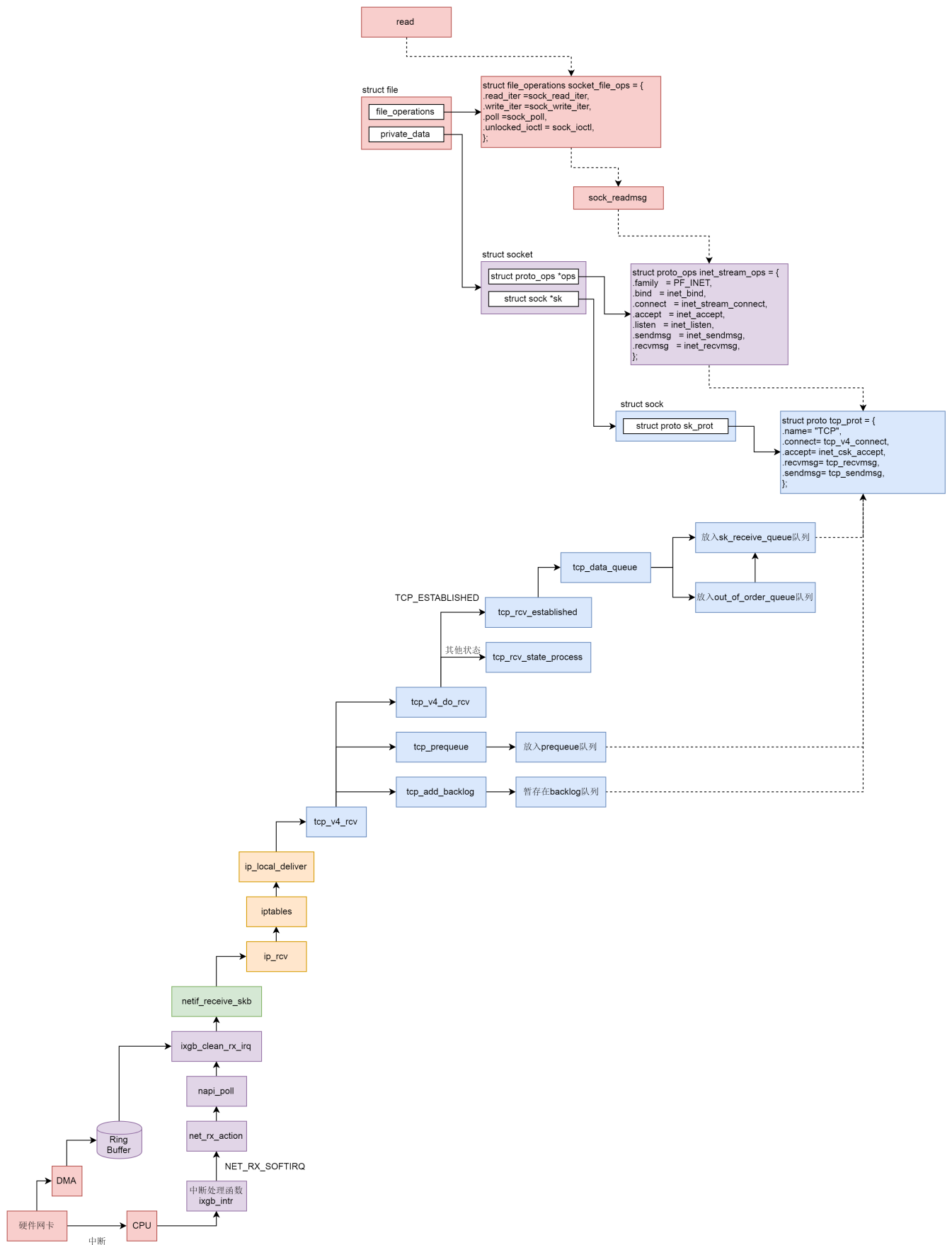
至此内核接收网络包的过程到此结束，接下来就是用户态读取网络包的过程，这个过程分成几个层次。

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。



课堂练习

对于 TCP 协议、三次握手、发送和接收的连接维护、拥塞控制、滑动窗口，我们都解析过了。唯独四次挥手我们没有解析，对应的代码你应该知道在什么地方了，你可以自己试着解

析一下四次挥手的过程。

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 47 | 接收网络包（上）：如何搞明白合作伙伴让我们做什么？

下一篇 49 | 虚拟机：如何成立子公司，让公司变集团？

精选留言 (5)

写留言



免费的人

2019-07-17

老师有计划讲epoll的实现吗？

展开 ∨



2



d

2019-07-17

这个 out_of_order_queue 是怎么实现的，假如5，6已结到了，下个期待7，8，但是从队头拿出的是9，10，怎么办，重新入队吗，这样效率有点低吧，老师能讲讲吗



没心没肺

2019-07-17

终于快结束了 😊

展开 ▾



许童童

2019-07-17

老师写得好！

展开 ▾



免费的人

2019-07-17

从kernel doc里发现这个说明：

tcp_low_latency - BOOLEAN

This is a legacy option, it has no effect anymore.

这个选项没用了？

展开 ▾

