

## 45 | 发送网络包（上）：如何表达我们想让合作伙伴做什么？

2019-07-10 刘超

趣谈Linux操作系统

[进入课程 >](#)



讲述：刘超

时长 22:56 大小 21.01M



上一节，我们通过 socket 函数、bind 函数、listen 函数、accept 函数以及 connect 函数，在内核建立好了数据结构，并完成了 TCP 连接建立的三次握手过程。

这一节，我们接着来分析，发送一个网络包的过程。


### 解析 socket 的 Write 操作

socket 对于用户来讲，是一个文件一样的存在，拥有一个文件描述符。因而对于网络包的发送，我们可以使用对于 socket 文件的写入系统调用，也就是 write 系统调用。

write 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，这里不再多说。对于每一个打开的文件都有一个 struct file 结构，


write 系统调用会最终调用 struct file 结构指向的 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 };
```

按照文件系统的写入流程，调用的是 sock\_write\_iter。

 复制代码

```
1 static ssize_t sock_write_iter(struct kiocb *iocb, struct iov_iter *from)
2 {
3     struct file *file = iocb->ki_filp;
4     struct socket *sock = file->private_data;
5     struct msghdr msg = {.msg_iter = *from,
6                          .msg_iocb = iocb};
7     ssize_t res;
8     .....
9     res = sock_sendmsg(sock, &msg);
10    *from = msg.msg_iter;
11    return res;
12 }
```

在 sock\_write\_iter 中，我们通过 VFS 中的 struct file，将创建好的 socket 结构拿出来，然后调用 sock\_sendmsg。而 sock\_sendmsg 会调用 sock\_sendmsg\_nosec。


 复制代码

```

1 static inline int sock_sendmsg_nosec(struct socket *sock, struct msghdr *msg)
2 {
3     int ret = sock->ops->sendmsg(sock, msg, msg_data_left(msg));
4     .....
5 }

```

这里调用了 socket 的 ops 的 sendmsg，我们在上一节已经遇到它好几次了。根据 inet\_stream\_ops 的定义，我们这里调用的是 inet\_sendmsg。

 复制代码

```


1 int inet_sendmsg(struct socket *sock, struct msghdr *msg, size_t size)
2 {
3     struct sock *sk = sock->sk;
4     .....
5     return sk->sk_prot->sendmsg(sk, msg, size);
6 }

```

这里面，从 socket 结构中，我们可以得到更底层的 sock 结构，然后调用 sk\_prot 的 sendmsg 方法。这个我们同样在上一节遇到好几次了。

## 解析 tcp\_sendmsg 函数

根据 tcp\_prot 的定义，我们调用的是 tcp\_sendmsg。

 复制代码

```

1 int tcp_sendmsg(struct sock *sk, struct msghdr *msg, size_t size)
2 {
3     struct tcp_sock *tp = tcp_sk(sk);
4     struct sk_buff *skb;
5     int flags, err, copied = 0;
6     int mss_now = 0, size_goal, copied_syn = 0;
7     long timeo;
8     .....
9     /* Ok commence sending. */
10    copied = 0;
11    restart:
12    mss_now = tcp_send_mss(sk, &size_goal, flags);
13
14    while (msg_data_left(msg)) {
15        int copy = 0;
16        int max = size_goal;

```

```

17     skb = tcp_write_queue_tail(sk);
18     if (tcp_send_head(sk)) {
19         if (skb->ip_summed == CHECKSUM_NONE)
20             max = mss_now;
21         copy = max - skb->len;
22     }
23
24
25     if (copy <= 0 || !tcp_skb_can_collapse_to(skb)) {
26         bool first_skb;
27
28     new_segment:
29         /* Allocate new segment. If the interface is SG,
30          * allocate skb fitting to single page.
31          */
32         if (!sk_stream_memory_free(sk))
33             goto wait_for_sndbuf;
34
35         first_skb = skb_queue_empty(&sk->sk_write_queue);
36         skb = sk_stream_alloc_skb(sk,
37                                   select_size(sk, sg, first_skb),
38                                   sk->sk_allocation,
39                                   first_skb);
40
41         skb_entail(sk, skb);
42         copy = size_goal;
43         max = size_goal;
44
45     }
46
47     /* Try to append data to the end of skb. */
48     if (copy > msg_data_left(msg))
49         copy = msg_data_left(msg);
50
51     /* Where to copy to? */
52     if (skb_availroom(skb) > 0) {
53         /* We have some space in skb head. Superb! */
54         copy = min_t(int, copy, skb_availroom(skb));
55         err = skb_add_data_nocache(sk, skb, &msg->msg_iter, copy);
56
57     } else {
58         bool merge = true;
59         int i = skb_shinfo(skb)->nr_frags;
60         struct page_frag *pfrag = sk_page_frag(sk);
61
62         copy = min_t(int, copy, pfrag->size - pfrag->offset);
63
64         err = skb_copy_to_page_nocache(sk, &msg->msg_iter, skb,
65                                       pfrag->page,
66                                       pfrag->offset,
67                                       copy);
68

```

```

69             pfrag->offset += copy;
70         }
71
72     .....
73         tp->write_seq += copy;
74         TCP_SKB_CB(skb)->end_seq += copy;
75         tcp_skb_pcount_set(skb, 0);
76
77         copied += copy;
78         if (!msg_data_left(msg)) {
79             if (unlikely(flags & MSG_EOR))
80                 TCP_SKB_CB(skb)->eor = 1;
81             goto out;
82         }
83
84         if (skb->len < max || (flags & MSG_OOB) || unlikely(tp->repair))
85             continue;
86
87         if (forced_push(tp)) {
88             tcp_mark_push(tp, skb);
89             __tcp_push_pending_frames(sk, mss_now, TCP_NAGLE_PUSH);
90         } else if (skb == tcp_send_head(sk))
91             tcp_push_one(sk, mss_now);
92         continue;
93     .....
94     }
95     .....
96 }

```

`tcp_sendmsg` 的实现还是很复杂的，这里面做了这样几件事情。

`msg` 是用户要写入的数据，这个数据要拷贝到内核协议栈里面去发送；在内核协议栈里面，网络包的数据都是由 `struct sk_buff` 维护的，因而第一件事情就是找到一个空闲的内存空间，将用户要写入的数据，拷贝到 `struct sk_buff` 的管辖范围内。而第二件事情就是发送 `struct sk_buff`。

在 `tcp_sendmsg` 中，我们首先通过强制类型转换，将 `sock` 结构转换为 `struct tcp_sock`，这个是维护 TCP 连接状态的重要数据结构。

接下来是 `tcp_sendmsg` 的第一件事情，把数据拷贝到 `struct sk_buff`。

我们先声明一个变量 `copied`，初始化为 0，这表示拷贝了多少数据。紧接着是一个循环，`while (msg_data_left(msg))`，也即如果用户的数据没有发送完毕，就一直循环。循环里声明了一个 `copy` 变量，表示这次拷贝的数值，在循环的最后有 `copied += copy`，将每次拷贝的数量都加起来。

我们这里只需要看一次循环做了哪些事情。

**第一步**，`tcp_write_queue_tail` 从 TCP 写入队列 `sk_write_queue` 中拿出最后一个 `struct sk_buff`，在这个写入队列中排满了要发送的 `struct sk_buff`，为什么要拿最后一个呢？这里面只有最后一个，可能会因为上次用户给的数据太少，而没有填满。

**第二步**，`tcp_send_mss` 会计算 MSS，也即 Max Segment Size。这是什么呢？这个意思是说，我们在网络上传输的网络包的大小是有限制的，而这个限制在最底层开始就有。

**MTU** (Maximum Transmission Unit，最大传输单元) 是二层的一个定义。以以太网为例，MTU 为 1500 个 Byte，前面有 6 个 Byte 的目标 MAC 地址，6 个 Byte 的源 MAC 地址，2 个 Byte 的类型，后面有 4 个 Byte 的 CRC 校验，共 1518 个 Byte。

在 IP 层，一个 IP 数据报在以太网中传输，如果它的长度大于该 MTU 值，就要进行分片传输。


在 TCP 层有个 **MSS** (Maximum Segment Size，最大分段大小)，等于 MTU 减去 IP 头，再减去 TCP 头。也就是，在不分片的情况下，TCP 里面放的最大内容。

在这里，`max` 是 `struct sk_buff` 的最大数据长度，`skb->len` 是当前已经占用的 `skb` 的数据长度，相减得到当前 `skb` 的剩余数据空间。

**第三步**，如果 `copy` 小于 0，说明最后一个 `struct sk_buff` 已经没地方存放了，需要调用 `sk_stream_alloc_skb`，重新分配 `struct sk_buff`，然后调用 `skb_entail`，将新分配的 `sk_buff` 放到队列尾部。

`struct sk_buff` 是存储网络包的重要的数据结构，在应用层数据包叫 `data`，在 TCP 层我们称为 `segment`，在 IP 层我们叫 `packet`，在数据链路层称为 `frame`。在 `struct sk_buff`，首先是一个链表，将 `struct sk_buff` 结构串起来。

接下来，我们从 headers\_start 开始，到 headers\_end 结束，里面都是各层次的头的位置。这里面有二层的 mac\_header、三层的 network\_header 和四层的 transport\_header。

 复制代码

```
1 struct sk_buff {
2     union {
3         struct {
4             /* These two members must be first. */
5             struct sk_buff      *next;
6             struct sk_buff      *prev;
7         }
8     };
9     struct rb_node  rbnode; /* used in netem & tcp stack */
10 };
11 .....
12 /* private: */
13 __u32      headers_start[0];
14 /* public: */
15 .....
16 __u32      priority;
17 int        skb_iif;
18 __u32      hash;
19 __be16     vlan_proto;
20 __u16      vlan_tci;
21 .....
22 union {
23     __u32      mark;
24     __u32      reserved_tailroom;
25 };
26
27 union {
28     __be16     inner_protocol;
29     __u8       inner_ipproto;
30 };
31
32 __u16      inner_transport_header;
33 __u16      inner_network_header;
34 __u16      inner_mac_header;
35
36 __be16     protocol;
37 __u16      transport_header;
38 __u16      network_header;
39 __u16      mac_header;
40
41 /* private: */
42 __u32      headers_end[0];
43 /* public: */
44
```

```

45      /* These elements must be at the end, see alloc_skb() for details. */
46      sk_buff_data_t      tail;
47      sk_buff_data_t      end;
48      unsigned char       *head,
49                          *data;
50      unsigned int         truesize;
51      refcount_t           users;
52 };

```

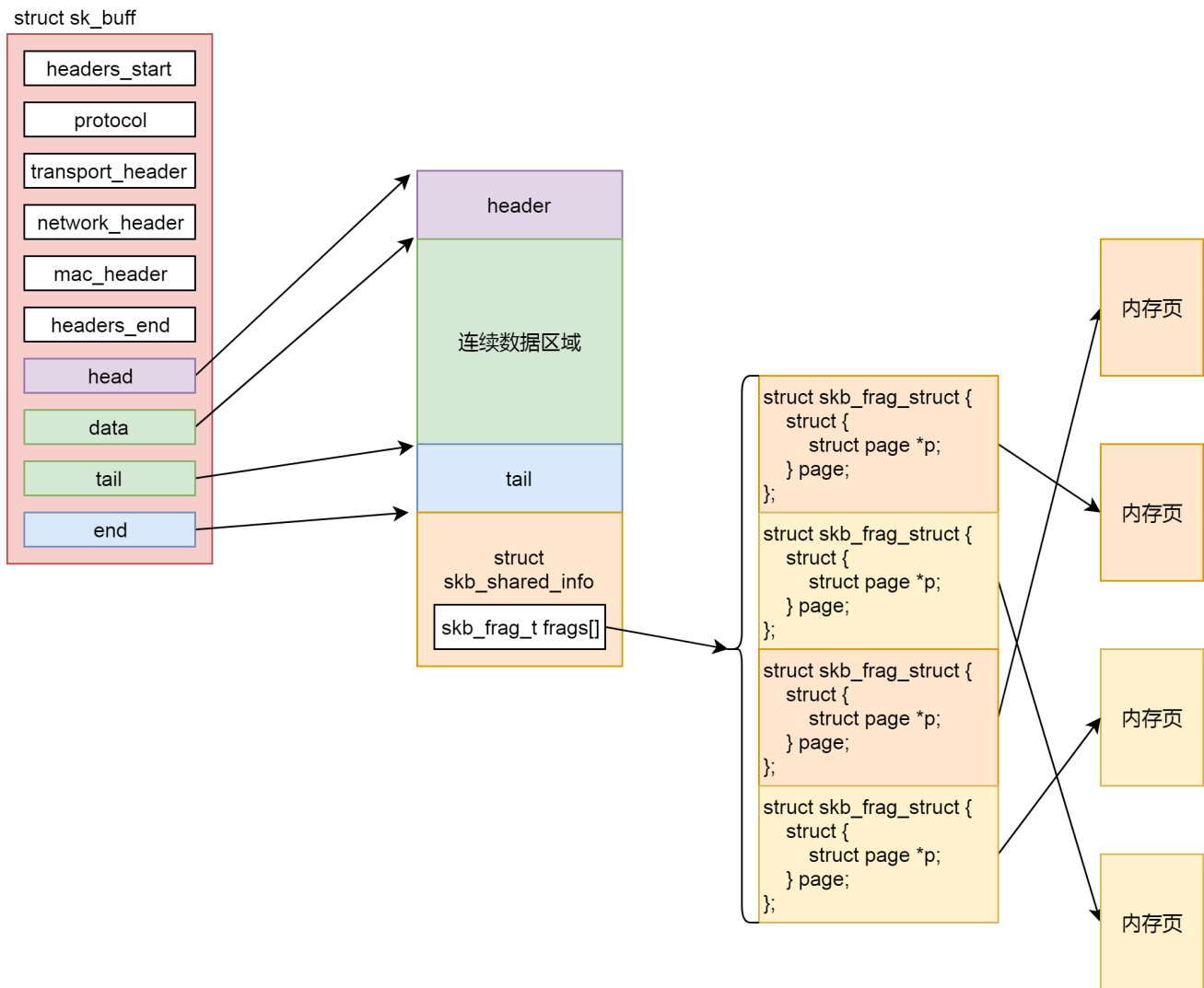
最后几项，head 指向分配的内存块起始地址。data 这个指针指向的位置是可变的。它有可能随着报文所处的层次而变动。当接收报文时，从网卡驱动开始，通过协议栈层层往上传送数据报，通过增加 skb->data 的值，来逐步剥离协议首部。而要发送报文时，各协议会创建 sk\_buff{}，在经过各下层协议时，通过减少 skb->data 的值来增加协议首部。tail 指向数据的结尾，end 指向分配的内存块的结束地址。

要分配这样一个结构，sk\_stream\_alloc\_skb 会最终调用到 \_\_alloc\_skb。在这个函数里面，除了分配一个 sk\_buff 结构之外，还要分配 sk\_buff 指向的数据区域。这段数据区域分为下面这几个部分。

第一部分是连续的数据区域。紧接着是第二部分，一个 struct skb\_shared\_info 结构。这个结构是对于网络包发送过程的一个优化，因为传输层之上就是应用层了。按照 TCP 的定义，应用层感受不到下面的网络层的 IP 包是一个个独立的包的存在。反正就是一个流，往里写就是了，可能一下子写多了，超过了一个 IP 包的承载能力，就会出现上面 MSS 的定义，拆分成一个个的 Segment 放在一个个的 IP 包里面，也可能一次写一点，一次写一点，这样数据是分散的，在 IP 层还要通过内存拷贝合成一个 IP 包。

为了减少内存拷贝的代价，有的网络设备支持**分散聚合**（Scatter/Gather）I/O，顾名思义，就是 IP 层没必要通过内存拷贝进行聚合，让散的数据零散的放在原处，在设备层进行聚合。如果使用这种模式，网络包的数据就不会放在连续的数据区域，而是放在 struct skb\_shared\_info 结构里面指向的离散数据，skb\_shared\_info 的成员变量 skb\_frag\_t frags[MAX\_SKB\_FRAGS]，会指向一个数组的页面，就不能保证连续了。





于是我们就有了**第四步**。在注释 `/* Where to copy to? */` 后面有个 if-else 分支。if 分支就是 `skb_add_data_nocache` 将数据拷贝到连续的数据区域。else 分支就是 `skb_copy_to_page_nocache` 将数据拷贝到 `struct skb_shared_info` 结构指向的不需要连续的页面区域。

**第五步**，就是要发生网络包了。第一种情况是积累的数据报数目太多了，因而我们需要通过调用 `__tcp_push_pending_frames` 发送网络包。第二种情况是，这是第一个网络包，需要马上发送，调用 `tcp_push_one`。无论 `__tcp_push_pending_frames` 还是 `tcp_push_one`，都会调用 `tcp_write_xmit` 发送网络包。

至此，`tcp_sendmsg` 解析完了。

## 解析 `tcp_write_xmit` 函数

接下来我们来看，`tcp_write_xmit` 是如何发送网络包的。

```

1 static bool tcp_write_xmit(struct sock *sk, unsigned int mss_now, int nonagle, int push_
2 {
3     struct tcp_sock *tp = tcp_sk(sk);
4     struct sk_buff *skb;
5     unsigned int tso_segs, sent_pkts;
6     int cwnd_quota;
7     .....
8     max_segs = tcp_tso_segs(sk, mss_now);
9     while ((skb = tcp_send_head(sk))) {
10         unsigned int limit;
11         .....
12         tso_segs = tcp_init_tso_segs(skb, mss_now);
13         .....
14         cwnd_quota = tcp_cwnd_test(tp, skb);
15         .....
16         if (unlikely(!tcp_snd_wnd_test(tp, skb, mss_now))) {
17             is_rwnd_limited = true;
18             break;
19         }
20         .....
21         limit = mss_now;
22         if (tso_segs > 1 && !tcp_urg_mode(tp))
23             limit = tcp_mss_split_point(sk, skb, mss_now, min_t(unsigned int, cwnd_quota,
24             if (skb->len > limit &&
25                 unlikely(tso_fragment(sk, skb, limit, mss_now, gfp)))
26                     break;
27         .....
28         if (unlikely(tcp_transmit_skb(sk, skb, 1, gfp)))
29             break;
30         .....
31     repair:
32         /* Advance the send_head. This one is sent out.
33          * This call will increment packets_out.
34          */
35         tcp_event_new_data_sent(sk, skb);
36         tcp_minshall_update(tp, mss_now, skb);
37         sent_pkts += tcp_skb_pcount(skb);
38         if (push_one)
39             break;
40     }
41     .....
42 }

```

这里面主要的逻辑是一个循环，用来处理发送队列，只要队列不空，就会发送。

在一个循环中，涉及 TCP 层的很多传输算法，我们来一一解析。

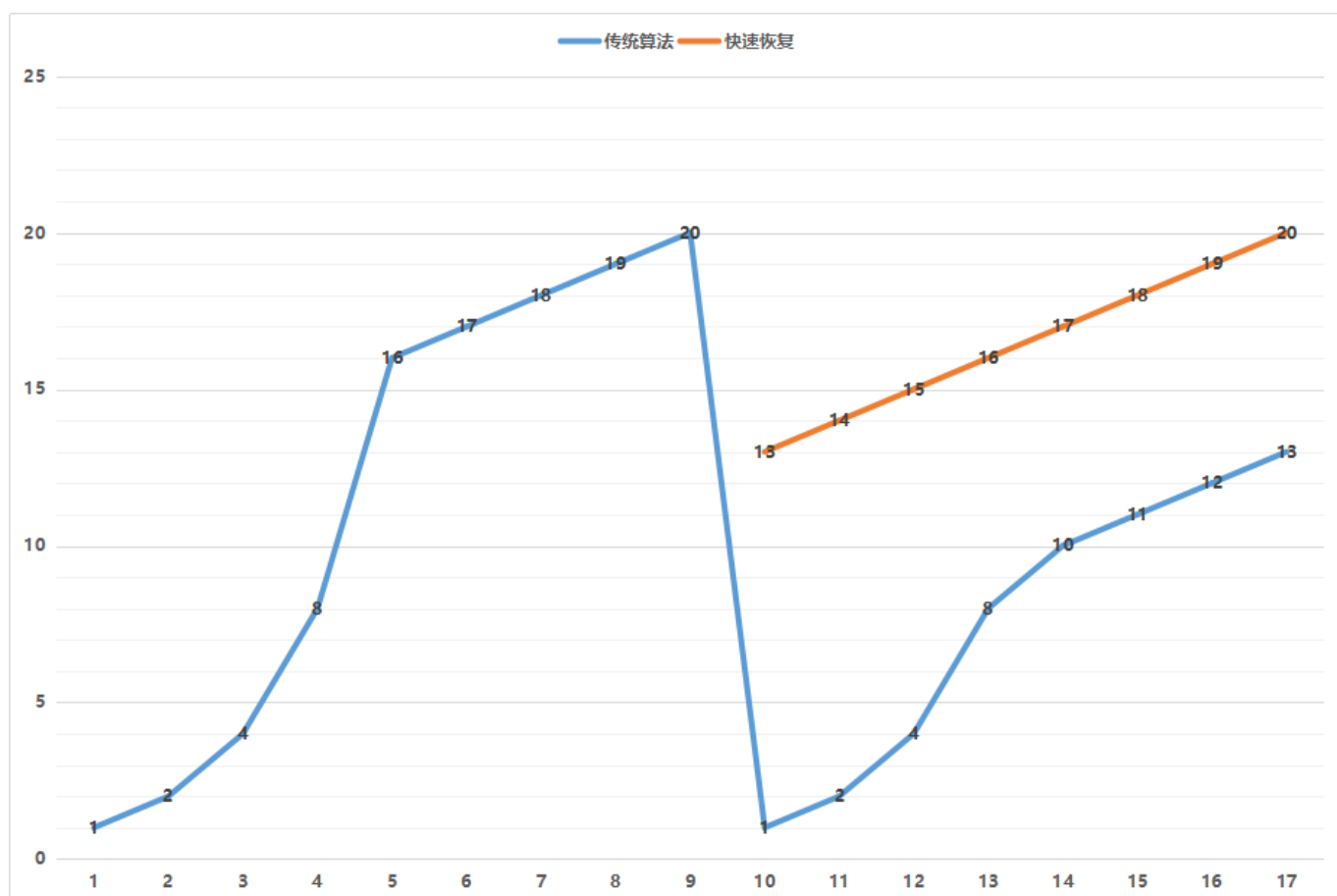
第一个概念是**TSO** ( TCP Segmentation Offload )。如果发送的网络包非常大，就像上面说的一样，要进行分段。分段这个事情可以由协议栈代码在内核做，但是缺点是比较费 CPU，另一种方式是延迟到硬件网卡去做，需要网卡支持对大数据包进行自动分段，可以降低 CPU 负载。

在代码中，tcp\_init\_tso\_segs 会调用 tcp\_set\_skb\_tso\_segs。这里面有这样的语句：  
`DIV_ROUND_UP(skb->len, mss_now)`。也就是 sk\_buff 的长度除以 mss\_now，应该分成几个段。如果算出来要分成多个段，接下来就是要看，是在这里（协议栈的代码里面）分好，还是等待到了底层网卡再分。

于是，调用函数 tcp\_mss\_split\_point，开始计算切分的 limit。这里面会计算  $\text{max\_len} = \text{mss\_now} * \text{max\_segs}$ ，根据现在不切分来计算 limit，所以下一步的判断中，大部分情况下 tso\_fragment 不会被调用，等待到了底层网卡来切分。

第二个概念是**拥塞窗口**的概念（cwnd，congestion window），也就是说为了避免拼命发包，把网络塞满了，定义一个窗口的概念，在这个窗口之内的才能发送，超过这个窗口的就不能发送，来控制发送的频率。

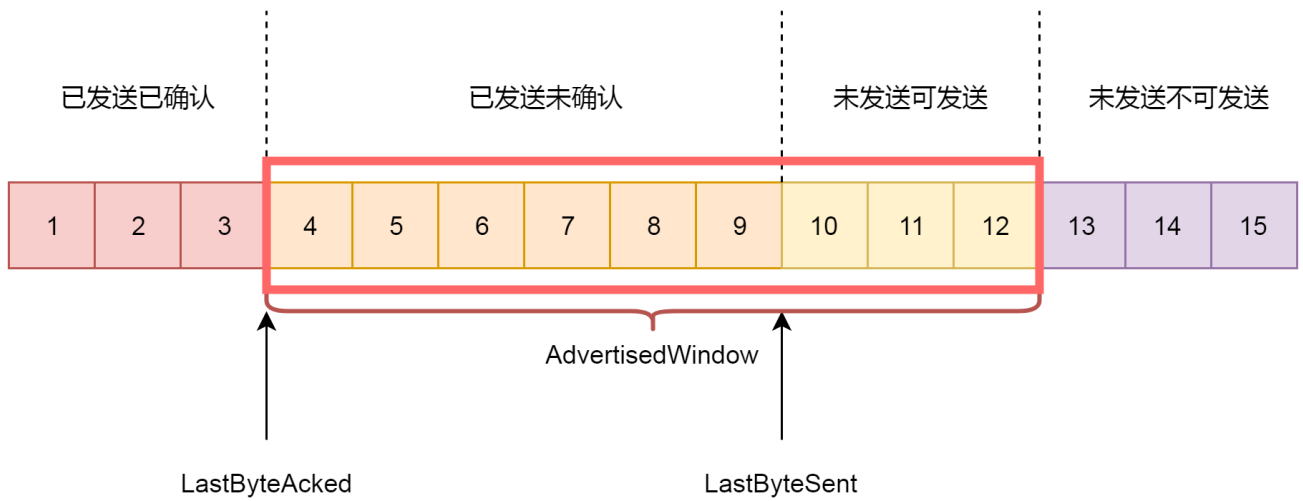
那窗口大小是多少呢？就是遵循下面这个著名的拥塞窗口变化图。



一开始的窗口只有一个 mss 大小叫作 slow start (慢启动)。一开始的增长速度的很快的，翻倍增长。一旦到达一个临界值 ssthresh，就变成线性增长，我们就称为**拥塞避免**。什么时候算真正拥塞呢？就是出现了丢包。一旦丢包，一种方法是马上降回到一个 mss，然后重复先翻倍再线性对的过程。如果觉得太过激进，也可以有第二种方法，就是降到当前 cwnd 的一半，然后进行线性增长。

在代码中，tcp\_cwnd\_test 会将当前的 snd\_cwnd，减去已经在窗口里面尚未发送完毕的网络包，那就是剩下的窗口大小 cwnd\_quota，也即就能发送这么多了。

第三个概念就是**接收窗口** rwnd 的概念 (receive window)，也叫滑动窗口。如果说拥塞窗口是为了怕把网络塞满，在出现丢包的时候减少发送速度，那么滑动窗口就是为了怕把接收方塞满，而控制发送速度。



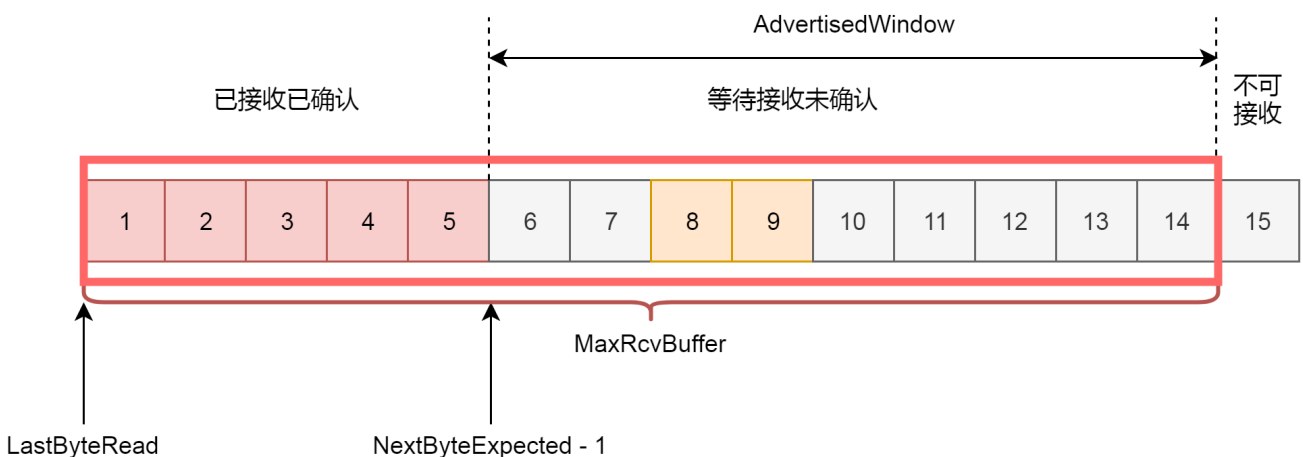
滑动窗口，其实就是接收方告诉发送方自己的网络包的接收能力，超过这个能力，我就受不了了。因为滑动窗口的存在，将发送方的缓存分成了四个部分。

第一部分：发送了并且已经确认的。这部分是已经发送完毕的网络包，这部分没有用了，可以回收。

第二部分：发送了但尚未确认的。这部分，发送方要等待，万一发送不成功，还要重新发送，所以不能删除。

第三部分：没有发送，但是已经等待发送的。这部分是接收方空闲的能力，可以马上发送，接收方收得了。

第四部分：没有发送，并且暂时还不会发送的。这部分已经超过了接收方的接收能力，再发送接收方就收不了了。



因为滑动窗口的存在，接收方的缓存也要分成了三个部分。

第一部分：接受并且确认过的任务。这部分完全接收成功了，可以交给应用层了。


第二部分：还没接收，但是马上就能接收的任务。这部分有的网络包到达了，但是还没确认，不算完全完毕，有的还没有到达，那就是接收方能够接受的最大的网络包数量。

第三部分：还没接收，也没法接收的任务。这部分已经超出接收方能力。

在网络包的交互过程中，接收方会将第二部分的大小，作为 AdvertisedWindow 发送给发送方，发送方就可以根据他来调整发送速度了。

在 tcp\_snd\_wnd\_test 函数中，会判断 sk\_buff 中的 end\_seq 和 tcp\_wnd\_end(tp) 之间的关系，也即这个 sk\_buff 是否在滑动窗口的允许范围之内。如果不在范围内，说明发送要受限制了，我们就要把 is\_rwnd\_limited 设置为 true。

接下来，tcp\_mss\_split\_point 函数要被调用了。


 复制代码

```
1 static unsigned int tcp_mss_split_point(const struct sock *sk,
2                                         const struct sk_buff *skb,
3                                         unsigned int mss_now,
4                                         unsigned int max_segs,
5                                         int nonagle)
6 {
7     const struct tcp_sock *tp = tcp_sk(sk);
8     u32 partial, needed, window, max_len;
9
10    window = tcp_wnd_end(tp) - TCP_SKB_CB(skb)->seq;
11    max_len = mss_now * max_segs;
12
13    if (likely(max_len <= window && skb != tcp_write_queue_tail(sk)))
14        return max_len;
15
16    needed = min(skb->len, window);
17
18    if (max_len <= needed)
19        return max_len;
20    .....
21    return needed;
22 }
```

这里面除了会判断上面讲的，是否会因为超出 mss 而分段，还会判断另一个条件，就是是否在滑动窗口的运行范围之内，如果小于窗口的大小，也需要分段，也即需要调用

## tso\_fragment.

在一个循环的最后，是调用 `tcp_transmit_skb`，真的去发送一个网络包。

 复制代码


```
1 static int tcp_transmit_skb(struct sock *sk, struct sk_buff *skb, int clone_it,
2                             gfp_t gfp_mask)
3 {
4     const struct inet_connection_sock *icsk = inet_csk(sk);
5     struct inet_sock *inet;
6     struct tcp_sock *tp;
7     struct tcp_skb_cb *tcb;
8     struct tcphdr *th;
9     int err;
10
11     tp = tcp_sk(sk);
12
13     skb->skb_mstamp = tp->tcp_mstamp;
14     inet = inet_sk(sk);
15     tcb = TCP_SKB_CB(skb);
16     memset(&opts, 0, sizeof(opts));
17
18     tcp_header_size = tcp_options_size + sizeof(struct tcphdr);
19     skb_push(skb, tcp_header_size);
20
21     /* Build TCP header and checksum it. */
22     th = (struct tcphdr *)skb->data;
23     th->source      = inet->inet_sport;
24     th->dest        = inet->inet_dport;
25     th->seq         = htonl(tcb->seq);
26     th->ack_seq     = htonl(tp->rcv_nxt);
27     *(((__be16 *)th) + 6) = htons(((tcp_header_size >> 2) << 12) |
28                                   tcb->tcp_flags);
29
30     th->check       = 0;
31     th->urg_ptr     = 0;
32     .....
33     tcp_options_write((__be32 *)(th + 1), tp, &opts);
34     th->window      = htons(min(tp->rcv_wnd, 65535U));
35     .....
36     err = icsk->icsk_af_ops->queue_xmit(sk, skb, &inet->cork.fl);
37     .....
38 }
```

tcp\_transmit\_skb 这个函数比较长，主要做了两件事情，第一件事情就是填充 TCP 头，如果我们对着 TCP 头的格式。

源端口号							目的端口号						
序号													
确认序号													
首部长度	保留	URG	ACK	PSH	RST	SYN	FIN	窗口大小					
TCP校验和								紧急指针					
选项													
数据													

这里面有源端口，设置为 inet\_sport，有目标端口，设置为 inet\_dport；有序列号，设置为 tcb->seq；有确认序列号，设置为 tp->rcv\_nxt。我们把所有的 flags 设置为 tcb->tcp\_flags。设置选项为 opts。设置窗口大小为 tp->rcv\_wnd。

全部设置完毕之后，就会调用 icsh\_af\_ops 的 queue\_xmit 方法，icsh\_af\_ops 指向 ipv4\_specific，也即调用的是 ip\_queue\_xmit 函数。

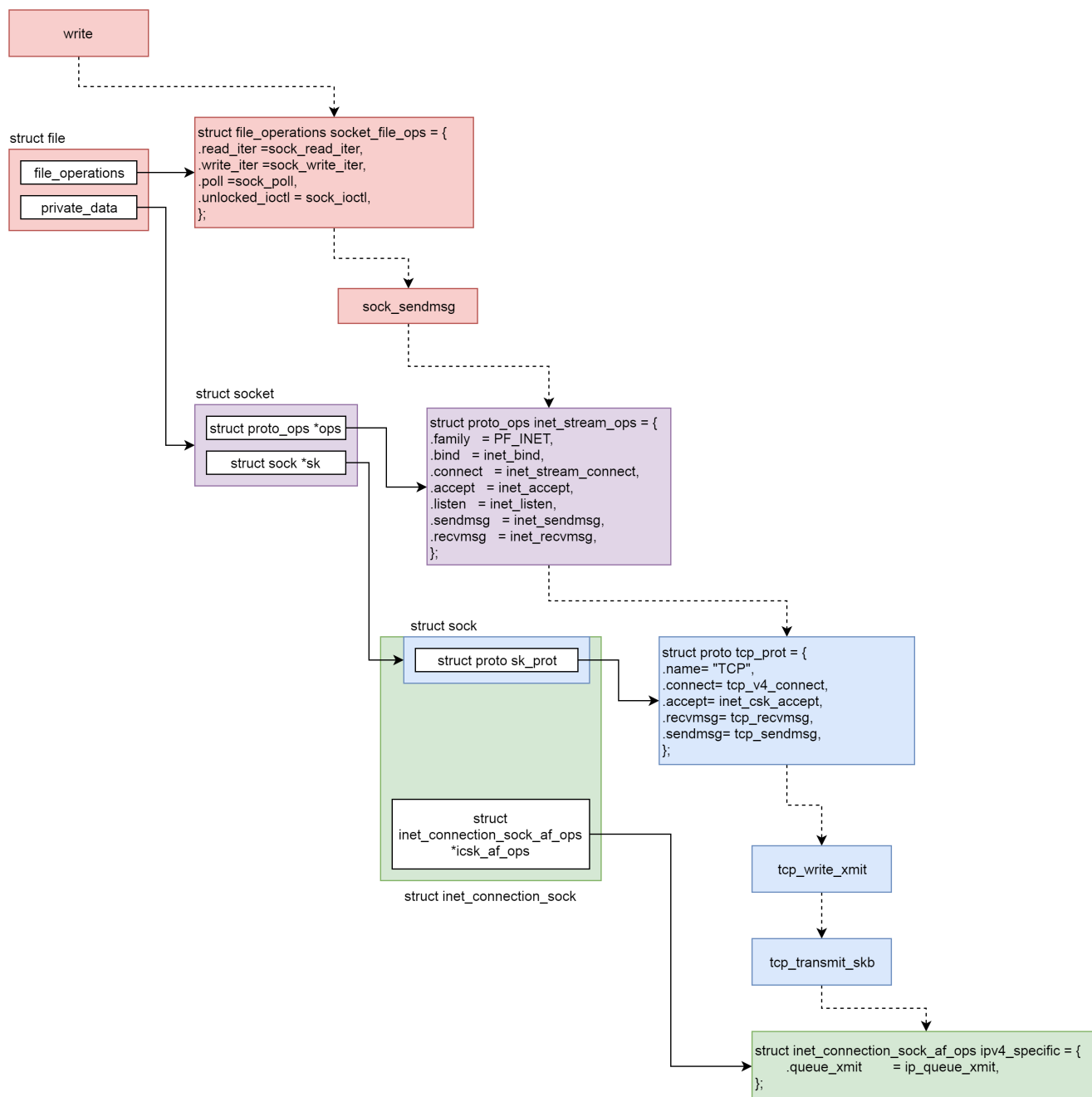
 复制代码

```
1 const struct inet_connection_sock_af_ops ipv4_specific = {
2     .queue_xmit      = ip_queue_xmit,
3     .send_check      = tcp_v4_send_check,
4     .rebuild_header  = inet_sk_rebuild_header,
5     .sk_rx_dst_set   = inet_sk_rx_dst_set,
6     .conn_request    = tcp_v4_conn_request,
7     .syn_recv_sock   = tcp_v4_syn_recv_sock,
8     .net_header_len  = sizeof(struct iphdr),
9     .setsockopt      = ip_setsockopt,
10    .getsockopt      = ip_getsockopt,
11    .addr2sockaddr    = inet_csk_addr2sockaddr,
12    .sockaddr_len     = sizeof(struct sockaddr_in),
13    .mtu_reduced      = tcp_v4_mtu_reduced,
14 };
```



## 总结时刻

这一节，我们解析了发送一个网络包的一部分过程，如下图所示。



这个过程分成几个层次。

VFS 层：write 系统调用找到 struct file，根据里面的 file\_operations 的定义，调用 sock\_write\_iter 函数。sock\_write\_iter 函数调用 sock\_sendmsg 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\_sendmsg 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\_sendmsg 函数。

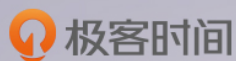
TCP 层：tcp\_sendmsg 函数会调用 tcp\_write\_xmit 函数，tcp\_write\_xmit 函数会调用 tcp\_transmit\_skb，在这里实现了 TCP 层面向连接的逻辑。

IP 层：扩展 struct sock，得到 struct inet\_connection\_sock，根据里面 icsk\_af\_ops 的定义，调用 ip\_queue\_xmit 函数。

## 课堂练习

如果你对 TCP 协议的结构不太熟悉，可以使用 tcpdump 命令截取一个 TCP 的包，看看里面的结构。

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。

## 精选留言 (2)

写留言



W.jyao

2019-07-11

老师，请教一个问题，为什么流媒体服务器发送的rtp包都要小于1500左右，也就是小于MTU，理论上不是大于1500会分片吗？但是好像实现的代码都会小于Mtu，为什么呢？

1



一梦如是

2019-07-11

老师好，请教一个困惑很久的问题，cpu的L1，L2，L3级cache，缓存的数据是以内存的页为单位的吗

oracle sga在大内存时，通常会配置hugepage以减少TLB的压力和swap的交换用来提高性能，linux (centos)下默认是2M，而一般cpu L1是32+32K,L2是256K，是不是就意味着没法使用这两级缓存了

展开

