网络包发送过程

网络包发送过程
一、socket层到tcp层的发送过程
1、socket的write系统调用
2、解析tcp_sendmsg函数
3、解析tcp_write_xmit函数
二、IP层到MAC层的发送过程
1、解析ip_queue_xmit函数
2、解析ip_finish_output函数
三、总结

一、socket层到tcp层的发送过程

1、socket的write系统调用

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。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stuct file结构指向的file_operations操作。

> socket -> sock_create -> sock_map_fd -> sock_alloc_file -> alloc_file

在上面的调用链中将socket_file_ops绑定到socket的struct file中，所以对socket文件描述符调用write就是调用socket_file_ops中的相应的函数。

```
static const struct file_operations socket_file_ops = {
    .owner =    THIS_MODULE,
    .llseek =   no_llseek,
    .read =     new_sync_read,
    .write =    new_sync_write,
    .read_iter =    sock_read_iter,
    .write_iter =   sock_write_iter,
.....
};
```

如上所示，会先调用new_sync_write，在new_sync_write中会调用filp->f_op->write_iter，因此最终调用到sock_write_iter。

```
static ssize_t sock_write_iter(struct kiocb *iocb, struct iov_iter *from)
{
    struct file *file = iocb->ki_filp;
    struct socket *sock = file->private_data;   /* 取出struct socket */
    struct msghdr msg = {.msg_iter = *from};
    ssize_t res;
.....
    /* 调用sock->ops->sendmsg，即inet_stream_ops中的inet_sendmsg */
    res = __sock_sendmsg(iocb, sock, &msg, iocb->ki_nbytes);
    *from = msg.msg_iter;
    return res;
}
```

在sock_write_iter中，先将struct socket取出，然后调用__sock_sendmsg，在__sock_sendmsg中调用__sock_sendmsg_nosec，然后再调用sock->ops->sendmsg，即inet_stream_ops中的inet_sendmsg。

```
int inet_sendmsg(struct kiocb *iocb, struct socket *sock, struct msghdr *msg,
         size_t size)
{
    struct sock *sk = sock->sk; /* 取出sk */
.....
    /* 调用sk->sk_prot->sendmsg，即sk_prot的tcp_sendmsg */
    return sk->sk_prot->sendmsg(iocb, sk, msg, size);
}
```

在inet_sendmsg中调用sk->sk_prot->sendmsg，即tcp_prot中的tcp_sendmsg。

2、解析tcp_sendmsg函数

```
int tcp_sendmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg,
        size_t size)
{
    struct tcp_sock *tp = tcp_sk(sk);   /* 类型转换成tcp_sock */
    struct sk_buff *skb;            /* 声明一个sk_buff */
    int flags, err, copied = 0;     /* 声明一个copied，初始化为0，表示拷贝了多少数据 */
    int mss_now = 0, size_goal, copied_syn = 0;
    bool sg;
    long timeo;
    .....
    /* 获取发送超时时间 */
    timeo = sock_sndtimeo(sk, flags & MSG_DONTWAIT);
    .....
    /* 获取mss，即tcp最大分节大小，一般是mtu - ip头 - tcp头，这样在ip层就不用分片 */
    mss_now = tcp_send_mss(sk, &size_goal, flags);
    /* 初始化copied */
    copied = 0;
    .....

    /* 循环拷贝和发送，直到数据全部发送 */
    while (iov_iter_count(&msg->msg_iter)) {
        int copy = 0;       /* copy表示本次循环拷贝的数量 */
        int max = size_goal;    /* struct sk_buff的最大数据长度 */
```

```
        /* 第一步，从TCP发送队列中拿出最后一个struct sk_buff   */
        skb = tcp_write_queue_tail(sk);
        if (tcp_send_head(sk)) {    /*  */
            if (skb->ip_summed == CHECKSUM_NONE)
                max = mss_now;
            copy = max - skb->len;  /* 计算剩余空间 */
        }

        /* 第二步，
         * 如果这个sk_buff没有剩余空间，
         * 则申请一个新的sk_buff，然后加入tcp写入队列，
         * 更新剩余空间
         */
        if (copy <= 0) {
new_segment:
            .....
            skb = sk_stream_alloc_skb(sk,
                          select_size(sk, sg),
                          sk->sk_allocation);   /* 申请sk_buff */
            .....
            skb_entail(sk, skb);    /* 加入tcp写入队列 */
            copy = size_goal;   /* 更新剩余空间 */
            max = size_goal;
            .....
        }

        /* Try to append data to the end of skb. */
        /* 如果队列最后一个skb的剩余空间比剩下的用户数据大，
         * 或者新申请的skb的最大空间比剩下的用户数据大，
         * 则更新copy从剩余空间变为剩下的用户数据数量
         */
        if (copy > iov_iter_count(&msg->msg_iter))
            copy = iov_iter_count(&msg->msg_iter);

        /* Where to copy to? */
        /* 第三步，拷贝用户数据到skb中的数据区 */
        if (skb_availroom(skb) > 0) {       /* 将用户数据拷贝到连续的数据区 */
            .....
             copy = min_t(int, copy, skb_availroom(skb));
            /* 将用户数据msg拷贝到skb的连续数据区 */
            err = skb_add_data_nocache(sk, skb, &msg->msg_iter, copy);
            .....
        } else {        /* 将用户数据拷贝到不连续的struct skb_shared_info结构指向的区域 */
            .....
             /* 将用户数据拷贝到不连续的页面区域 */
            err = skb_copy_to_page_nocache(sk, &msg->msg_iter, skb,
                                pfrag->page,
                                pfrag->offset,
                                copy);
            .....
        }
        .....
```

```
        copied += copy;          /* 更新已拷贝的数据 */
        .....
        /* 第四步，发送网络包 */
        if (forced_push(tp)) {  /* 第一种情况，积累太多网络包 */
            tcp_mark_push(tp, skb);
            __tcp_push_pending_frames(sk, mss_now, TCP_NAGLE_PUSH);
        } else if (skb == tcp_send_head(sk))
            tcp_push_one(sk, mss_now);  /* 第二种情况，第一个网络包 */
    }
    .....
}
```

参数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。而**第二件事情**就是发送struct sk_buff。

```
struct sk_buff {
    union {
        struct {
            /* These two members must be first. */
            struct sk_buff      *next;
            struct sk_buff      *prev;

            union {
                ktime_t     tstamp;
                struct skb_mstamp skb_mstamp;
            };
        };
        struct rb_node  rbnode; /* used in netem & tcp stack */
    };
    struct sock     *sk;
    struct net_device   *dev;
.....
    unsigned int        len,    /* 所有数据的长度 */
                data_len;
    __u16           mac_len,
                hdr_len;
.....
    __u32           headers_start[0];   /* 各层次的头位置域的开始 */
.....
    __u16           inner_transport_header;
    __u16           inner_network_header;
    __u16           inner_mac_header;

    __be16          protocol;           /* 网络设备驱动收到的数据包的协议类型，指的是网络层 */
    __u16           transport_header;   /* 传输层头部偏移 */
    __u16           network_header; /* 网络层头部偏移 */
    __u16           mac_header;     /* mac层头部偏移 */
.....
    __u32           headers_end[0]; /* 各层次的头位置域的结束 */


    /* These elements must be at the end, see alloc_skb() for details.  */
```

```
    sk_buff_data_t      tail;   /* 会移动， */
    sk_buff_data_t      end;    /* end指向分配的skb内存块的结束地址 */
    unsigned char       *head,  /* 指向分配的整个skb内存块起始地址 */
                *data;          /* 会移动，根据所处理的层次移动来剥离或增加协议头部 */
    unsigned int        truesize;
    atomic_t        users;
};
```

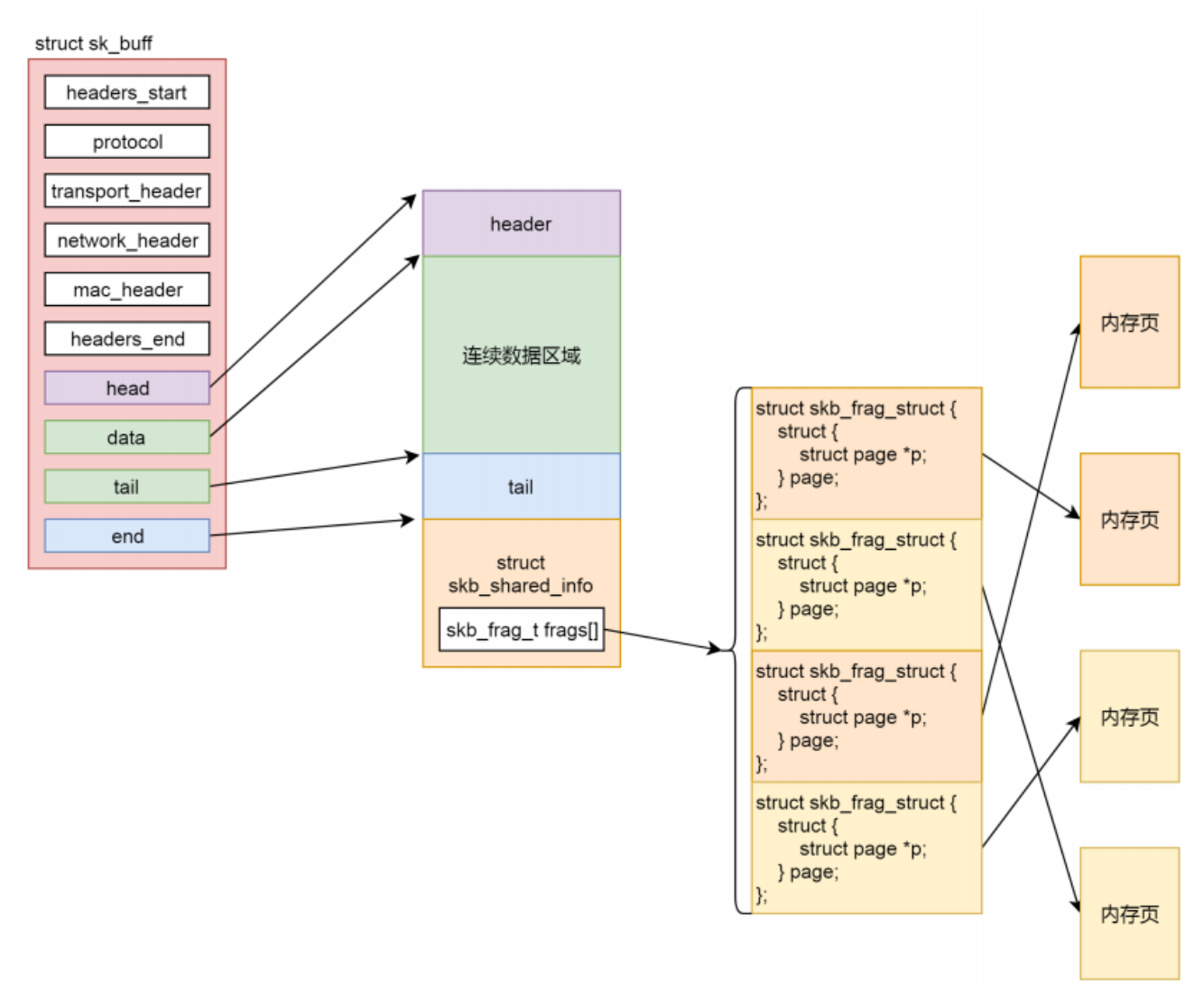
struct sk_buff是存储网络包的重要的数据结构，在应用层数据包叫**data**，在TCP层我们称为**segment**，在IP层我们叫**packet**，在数据链路层称为**frame**。在struct sk_buff，首先是struct sk_buff *next和struct sk_buff *prev，将struct sk_buff结构串起来。headers_start开始到headers_end之间的数据是各层次的头的偏移位置，比如二层的mac_header、三层的network_header和四层的transport_header。最后的几项数据中，head指向分配的sk_buff缓冲区的的头部，end是缓冲区的尾部，tail和data的位置是可变的，会随着报文所处的层次而变动，data指向协议头部，tail指向数据尾部。其中len是data和tail之间的长度。

tcp_sendmsg在获取超时时间和mss后进入一个循环，如果用户的数据没有发送完毕，就一直循环。循环中使用copied变量表示从第一次循环开始到本次循环已经拷贝的数据量，使用copy表示本次循环拷贝的数据量，每次循环后面都使用copied += copy累加总的拷贝量。**接下来看每次循环做了哪些事：**

第一步，通过tcp_write_queue_tail从tcp写入队列中取出最后一个struct sk_buff，在这个写入队列中排满了要发送的struct sk_buff，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。计算这个sk_buff种的剩余空间copy。

第二步，如果copy小于或等于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分配新的struct sk_buff，然后调用skb_entail将它加入tcp写入队列尾部。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，即从head到end之间的部分。紧接着的是**第二部分**，一个struct skb_shared_info结构，里面有一个skb_frag_struct数组，该数组中的每一项都是一个内存页和相应的偏移和大小，表示一块数据区域。如果使用在struct skb_shared_info模式，则数据会分散在数组中的各个页中。最后得到的分布图如下所示。

```
struct skb_shared_info {
.....
    skb_frag_t  frags[MAX_SKB_FRAGS];
};

typedef struct skb_frag_struct skb_frag_t;

struct skb_frag_struct {
    struct {
        struct page *p;
    } page;
    __u16 page_offset;
    __u16 size;
};
```



第三步，在/* Where to copy to? */注释后面有个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。

第四步，发送网络包。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。

3、解析tcp_write_xmit函数

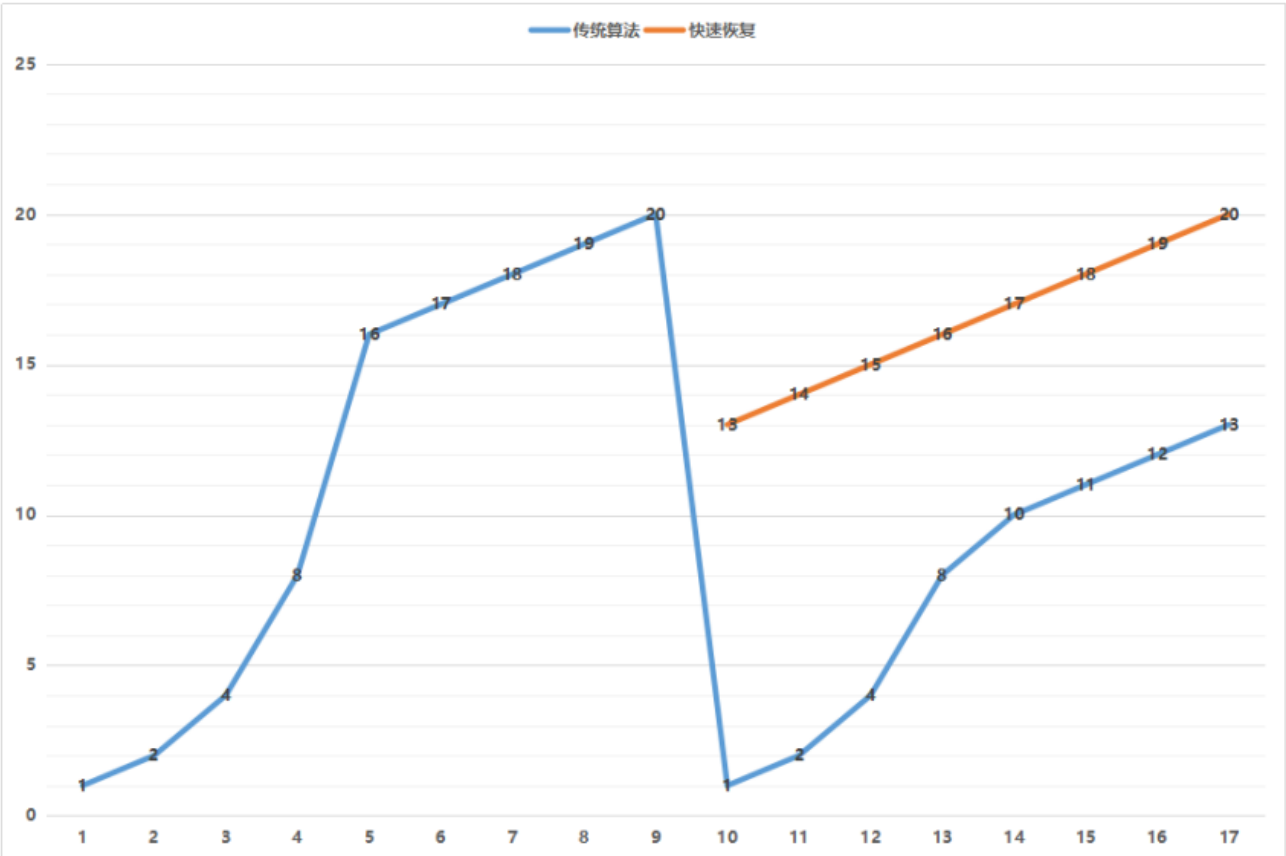
```
static bool tcp_write_xmit(struct sock *sk, unsigned int mss_now, int nonagle,
               int push_one, gfp_t gfp)
{
    struct tcp_sock *tp = tcp_sk(sk);
    struct sk_buff *skb;
    unsigned int tso_segs, sent_pkts;
    int cwnd_quota;
    .....
    max_segs = tcp_tso_autosize(sk, mss_now);

    /* 只要队列不为空则发送 */
```

```
    while ((skb = tcp_send_head(sk))) {
        unsigned int limit;
        /* 计算该skb有多少个段 */
        tso_segs = tcp_init_tso_segs(sk, skb, mss_now);
        .....
         /* 将当前的snd_cwnd减去已经在窗口里尚未发送完毕的网络包得到剩下的窗口大小 */
        cwnd_quota = tcp_cwnd_test(tp, skb);
        if (!cwnd_quota) {  /* 如果剩下的窗口没空间了 */
            is_cwnd_limited = true;     /* 设置该变量表示窗口没空间 */
            if (push_one == 2)
                /* Force out a loss probe pkt. */
                cwnd_quota = 1;
            else
                break;
        }

        /* 判断这个skb是否在滑动窗口范围内 */
        if (unlikely(!tcp_snd_wnd_test(tp, skb, mss_now)))
            break;
        .....
        limit = mss_now;
        if (tso_segs > 1 && max_segs && !tcp_urg_mode(tp))
             /* 计算skb中可以被发送的部分 */
            limit = tcp_mss_split_point(sk, skb, mss_now,
                            min_t(unsigned int,
                              cwnd_quota,
                              max_segs),
                            nonagle);
        /* 看是否要切片 */
        if (skb->len > limit &&
            unlikely(tso_fragment(sk, skb, limit, mss_now, gfp)))
            break;
        .....

         /* 发送网络包 */
        if (unlikely(tcp_transmit_skb(sk, skb, 1, gfp)))
            break;

repair:
        /* Advance the send_head.  This one is sent out.
         * This call will increment packets_out.
         */
        tcp_event_new_data_sent(sk, skb);

        tcp_minshall_update(tp, mss_now, skb);
        sent_pkts += tcp_skb_pcount(skb);

        if (push_one)
            break;
    }
    .....
}
```

tcp_write_xmit函数中主要是一个循环用来处理发送队列，只要队列不为空就会发送。在一个循环中，涉及TCP层的很多传输算法。

第一个概念是TSO（TCPSegmentationOffload）。如果发送的网络包非常大，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max_len =mss_now*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。

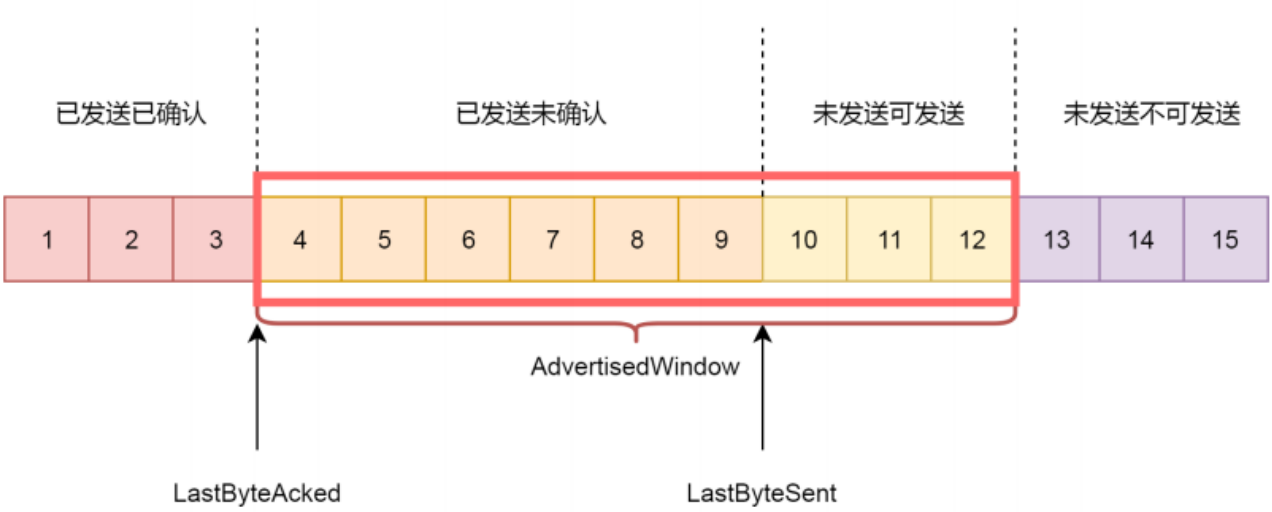
第二个概念是拥塞窗口的概念（cwnd，congestion window），也就是说为了避免拼命发包，把网络塞满了，定义一个窗口的概念，在这个窗口之内的才能发送，超过这个窗口的就不能发送，来控制发送的频率。窗口的大小典型变化过程如下图所示：



一开始的窗口只有一个mss大小叫作slowstart（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。

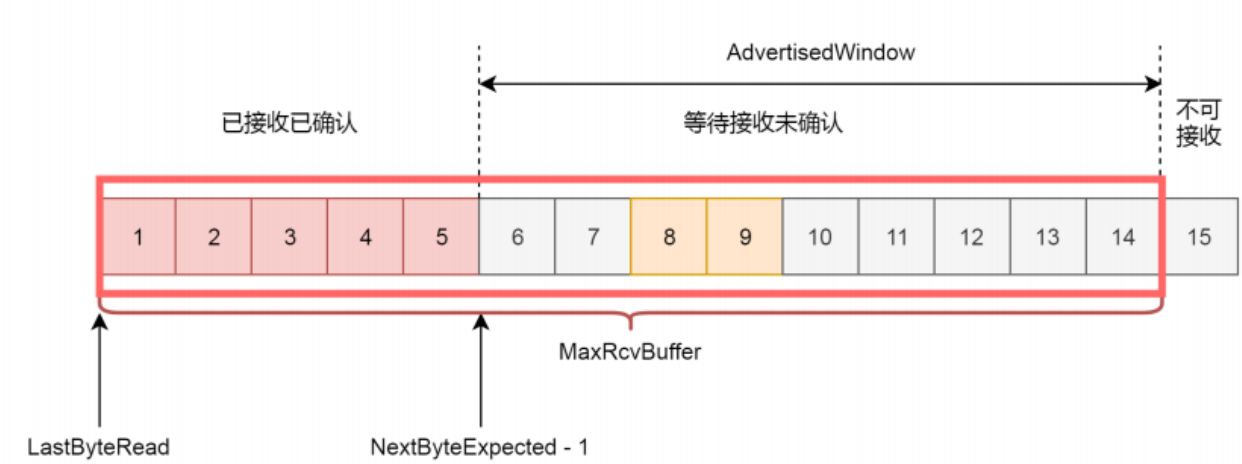
在代码中，tcp_cwnd_test会将当前的snd_cwnd，减去已经在窗口里面尚未发送完毕（未确认）的网络包，得到的剩下的窗口大小cwnd_quota，也即就能发送这么多了。

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对方把自己塞满，而控制对方的发送速度。



因为滑动窗口和拥塞窗口的存在，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。这部分加上第二部分的和叫做 AdvertisedWindow，这是接收端回应的接受窗口和拥塞窗口的**最小值**。
- 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。



同样，因为滑动窗口的存在，将接收方的缓存分成了三个部分：

- 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，发送方就可以根据接收方的AdvertisedWindow和自身的cwnd来调整自身的AdvertisedWindow了。

在tcp_snd_wnd_test中判断skb是否在窗口范围内，接下来，tcp_mss_split_point函数要被调用了。

```
/* Returns the portion of skb which can be sent right away */
static unsigned int tcp_mss_split_point(const struct sock *sk,
```

```
                    const struct sk_buff *skb,
                    unsigned int mss_now,
                    unsigned int max_segs,
                    int nonagle)
{
    const struct tcp_sock *tp = tcp_sk(sk);
    u32 partial, needed, window, max_len;

    window = tcp_wnd_end(tp) - TCP_SKB_CB(skb)->seq;    /* 计算剩余窗口 */
    max_len = mss_now * max_segs;    /* 计算sk_buff中能发送的最大数据长度 */

    if (likely(max_len <= window && skb != tcp_write_queue_tail(sk)))
        return max_len;

    needed = min(skb->len, window);     /* 剩余窗口和skb的长度的最小值 */

    if (max_len <= needed)
        return max_len;

    partial = needed % mss_now;
    /* If last segment is not a full MSS, check if Nagle rules allow us
     * to include this last segment in this skb.
     * Otherwise, we'll split the skb at last MSS boundary
     */
    if (tcp_nagle_check(partial != 0, tp, nonagle))
        return needed - partial;

    return needed;      /* 返回剩余窗口和skb的长度的最小值 */
}
```

在tcp_mss_split_point中，计算和返回剩余窗口和skb长度的最小值，然后根据情况判断是否要分段，如果要分段就通过tso_fragment申请一个新的skb作分段的包。最后通过tcp_transmit_skb发送网络包。

```
static int tcp_transmit_skb(struct sock *sk, struct sk_buff *skb, int clone_it,
                gfp_t gfp_mask)
{
    const struct inet_connection_sock *icsk = inet_csk(sk);
    .....
    struct tcphdr *th;
    .....

    /* 设置tcp头 */
    th = tcp_hdr(skb);
    th->source      = inet->inet_sport;
    th->dest        = inet->inet_dport;
    th->seq         = htonl(tcb->seq);
    th->ack_seq     = htonl(tp->rcv_nxt);
    *(((__be16 *)th) + 6)   = htons(((tcp_header_size >> 2) << 12) |
                    tcb->tcp_flags);

    if (unlikely(tcb->tcp_flags & TCPHDR_SYN)) {
        /* RFC1323: The window in SYN & SYN/ACK segments
         * is never scaled.
```

```
             */
            th->window  = htons(min(tp->rcv_wnd, 65535U));
        } else {
            th->window  = htons(tcp_select_window(sk));
        }
        th->check       = 0;
        th->urg_ptr     = 0;
        .....
        /* 将tcp头信息填入和sk关联的th中 */
        tcp_options_write((__be32 *)(th + 1), tp, &opts);
        .....
        /* 进入ip层发送，调用ip_queue_xmit */
        err = icsk->icsk_af_ops->queue_xmit(sk, skb, &inet->cork.fl);
        .....
}
```

在一个循环的最后，是调用tcp_transmit_skb去发送一个网络包。tcp_transmit_skb主要做了两件事情，第一件事情就是填充TCP头，第二件事就是调用icsk_af_ops的queue_xmit方法，icsk_af_ops指向ipv4_specific，也即调用的是ip_queue_xmit函数。

二、IP层到MAC层的发送过程

1、解析ip_queue_xmit函数

从ip_queue_xmit函数开始就要进入IP层的发送逻辑了。

```
int ip_queue_xmit(struct sock *sk, struct sk_buff *skb, struct flowi *fl)
{
    struct inet_sock *inet = inet_sk(sk);
    struct ip_options_rcu *inet_opt;
    struct flowi4 *fl4;
    struct rtable *rt;
    struct iphdr *iph;
    int res;

    /* Skip all of this if the packet is already routed,
     * f.e. by something like SCTP.
     */
    rcu_read_lock();
    inet_opt = rcu_dereference(inet->inet_opt);
    fl4 = &fl->u.ip4;

    /* 路由 */
    rt = skb_rtable(skb);
    if (rt != NULL)
        goto packet_routed;

    /* Make sure we can route this packet. */
    rt = (struct rtable *)__sk_dst_check(sk, 0);
    if (rt == NULL) {
        .....
        /* 寻找路由表项，即要从哪张网卡发送出去，下一站在哪里 */
        rt = ip_route_output_ports(sock_net(sk), fl4, sk,
```

```
                        daddr, inet->inet_saddr,
                        inet->inet_dport,
                        inet->inet_sport,
                        sk->sk_protocol,
                        RT_CONN_FLAGS(sk),
                        sk->sk_bound_dev_if);
        .....
    }
    skb_dst_set_noref(skb, &rt->dst);

    .....
    /* 设置ip层的头部信息 */
    iph = ip_hdr(skb);
    *((__be16 *)iph) = htons((4 << 12) | (5 << 8) | (inet->tos & 0xff));
    if (ip_dont_fragment(sk, &rt->dst) && !skb->ignore_df)
        iph->frag_off = htons(IP_DF);
    else
        iph->frag_off = 0;
    iph->ttl      = ip_select_ttl(inet, &rt->dst);
    iph->protocol = sk->sk_protocol;
    ip_copy_addrs(iph, fl4);

    .....
    /* 发送IP包 */
    res = ip_local_out(skb);
    .....
}
```

在ip_queue_xmit中有三部分处理逻辑。

第一部分，选取路由，也即我要发送这个包应该从哪个网卡出去。通过ip_route_output_ports函数来查找路由路径，找到后返回一个struct rtable结构体，表示找到的路由表项。

> ip_route_output_ports -> ip_route_output_flow -> __ip_route_output_key -> fib_lookup，__mkroute_output -> rt_dst_alloc

```
static struct rtable *__mkroute_output(const struct fib_result *res,
                        const struct flowi4 *fl4, int orig_oif,
                        struct net_device *dev_out,
                        unsigned int flags)
{
    .....
    rth = rt_dst_alloc(dev_out,
               IN_DEV_CONF_GET(in_dev, NOPOLICY),
               IN_DEV_CONF_GET(in_dev, NOXFRM),
               do_cache);
    .....
    rth->dst.output = ip_output;        /* dst.output设置ip_output */
    .....
}
```

第二部分，设置ip层的头。

第三部分，调用ip_local_out发送IP包，顺着调用到dst_output_sk。

> ip_local_out -> ip_local_out_sk -> __ip_local_out -> nf_hook -> dst_output -> dst_output_sk

```
static inline int dst_output_sk(struct sock *sk, struct sk_buff *skb)
{
    /* 调用ip_output */
    return skb_dst(skb)->output(sk, skb);
}
```

这里调用的就是struct rtable成员dst的ouput函数。在__mkroute_output中可以看到，output函数指向的是ip_output。

```
int ip_output(struct sock *sk, struct sk_buff *skb)
{
    struct net_device *dev = skb_dst(skb)->dev;

    IP_UPD_PO_STATS(dev_net(dev), IPSTATS_MIB_OUT, skb->len);

    skb->dev = dev;
    skb->protocol = htons(ETH_P_IP);

    /* 回调ip_finish_output */
    return NF_HOOK_COND(NFPROTO_IPV4, NF_INET_POST_ROUTING, skb, NULL, dev,
                ip_finish_output,
                !(IPCB(skb)->flags & IPSKB_REROUTED));
}
```

在ip_output函数中最终回调了ip_finish_output函数。

2、解析ip_finish_output函数

从ip_finish_output函数开始，发送网络包的逻辑由第三层到达第二层。ip_finish_output最终调用ip_finish_output2。

```
static inline int ip_finish_output2(struct sk_buff *skb)
{
    struct dst_entry *dst = skb_dst(skb);
    struct rtable *rt = (struct rtable *)dst;
    struct net_device *dev = dst->dev;
    unsigned int hh_len = LL_RESERVED_SPACE(dev);
    struct neighbour *neigh;
    u32 nexthop;
    .....
    rcu_read_lock_bh();
    /* 在路由表中找到下一跳 */
    nexthop = (__force u32) rt_nexthop(rt, ip_hdr(skb)->daddr);
    /* 是从本地的ARP表中查找下一跳的MAC地址 */
    neigh = __ipv4_neigh_lookup_noref(dev, nexthop);
    /* 如果在arp表中没有找到对应的项，则创建一个 */
    if (unlikely(!neigh))
        neigh = __neigh_create(&arp_tbl, &nexthop, dev, false);
    /* 调用neigh中的output即neigh_resolve_output */
    if (!IS_ERR(neigh)) {
```

```
        int res = dst_neigh_output(dst, neigh, skb);
        rcu_read_unlock_bh();
        return res;
    }
    .....
}
```

在ip_finish_output2中，先找到struct rtable路由表里面的下一跳，下一跳一定和本机在同一个局域网中，可以通过二层进行通信，因而通过__ipv4_neigh_lookup_noref，查找如何通过二层访问下一跳。如果在arp表中没有找到对应的项，则通过__neigh_create创建一个。

```
struct neighbour *__neigh_create(struct neigh_table *tbl, const void *pkey,
                 struct net_device *dev, bool want_ref)
{
    u32 hash_val;
    int key_len = tbl->key_len;
    int error;
    /* 创建一个neighbour，做部分初始化 */
    struct neighbour *n1, *rc, *n = neigh_alloc(tbl, dev);
    struct neigh_hash_table *nht;

    if (!n) {
        rc = ERR_PTR(-ENOBUFS);
        goto out;
    }

    memcpy(n->primary_key, pkey, key_len);
    n->dev = dev;
    dev_hold(dev);

    /*
     * 调用了arp_tbl的arp_constructor函数，
     * 里面设置neigh->ops = &arp_hh_ops，
     * neigh->output = neigh->ops = neigh_resolve_output
    */
    if (tbl->constructor && (error = tbl->constructor(n)) < 0) {
        rc = ERR_PTR(error);
        goto out_neigh_release;
    }
    .....

    /* 将neighbour插入hash表 */
    hash_val = tbl->hash(pkey, dev, nht->hash_rnd) >> (32 - nht->hash_shift);
    .....
    for (n1 = rcu_dereference_protected(nht->hash_buckets[hash_val],
                        lockdep_is_held(&tbl->lock));
         n1 != NULL;
         n1 = rcu_dereference_protected(n1->next,
            lockdep_is_held(&tbl->lock))) {
        if (dev == n1->dev && !memcmp(n1->primary_key, pkey, key_len)) {
            if (want_ref)
                neigh_hold(n1);
            rc = n1;
```

```
            goto out_tbl_unlock;
        }
    }
    .....
    rc = n;
    .....
}
```

在__neigh_create中先调用neigh_alloc创建一个struct neighbour结构，用于维护MAC地址和ARP相关的信息。

```
static struct neighbour *neigh_alloc(struct neigh_table *tbl, struct net_device *dev)
{
    struct neighbour *n = NULL;
    unsigned long now = jiffies;
    int entries;

    entries = atomic_inc_return(&tbl->entries) - 1;
    if (entries >= tbl->gc_thresh3 ||
        (entries >= tbl->gc_thresh2 &&
         time_after(now, tbl->last_flush + 5 * HZ))) {
        if (!neigh_forced_gc(tbl) &&
            entries >= tbl->gc_thresh3)
            goto out_entries;
    }

    /* 分配struct neighbour并初始化 */
    n = kzalloc(tbl->entry_size + dev->neigh_priv_len, GFP_ATOMIC);
    if (!n)
        goto out_entries;

    /* 初始化arp_queue成员 */
    __skb_queue_head_init(&n->arp_queue);   /* 初始化arp_queue成员 */
    rwlock_init(&n->lock);
    seqlock_init(&n->ha_lock);
    n->updated    = n->used = now;
    n->nud_state      = NUD_NONE;
    /* 在父函数__neigh_create中重新设置为neigh_resolve_output */
    n->output     = neigh_blackhole;
    seqlock_init(&n->hh.hh_lock);
    n->parms      = neigh_parms_clone(&tbl->parms);
    /* 设置timer成员为neigh_timer_handler */
    setup_timer(&n->timer, neigh_timer_handler, (unsigned long)n);

    NEIGH_CACHE_STAT_INC(tbl, allocs);
    n->tbl        = tbl;    /* 设置tbl成员指向arp_tbl */
    atomic_set(&n->refcnt, 1);
    n->dead       = 1;
out:
    return n;

out_entries:
    atomic_dec(&tbl->entries);
    goto out;
```

```
}
```

在neigh_alloc中，先分配一个struct neighbour结构并且初始化。这里面比较重要的有两个成员，一个是arp_queue，所以上层想通过ARP获取MAC地址的任务，都放在这个队列里面。另一个是timer定时器，我们设置成，过一段时间就调用neigh_timer_handler，来处理这些ARP任务。

__neigh_create然后调用了arp_tbl的constructor函数，也即调用了arp_constructor，里面设置neigh->ops = &arp_hh_ops，neigh->output = neigh->ops = neigh_resolve_output。然后将neighbour插入hash表。

回到ip_finish_output2，在__neigh_create之后，会调用dst_neigh_output发送网络包，其中会调用n->output，即neigh_resolve_output。

```
int neigh_resolve_output(struct neighbour *neigh, struct sk_buff *skb)
{
    .....
    if (!neigh_event_send(neigh, skb)) {    /* 发送arp */
        .....
        if (err >= 0)
            rc = dev_queue_xmit(skb);       /* 发送二层网络包 */
        else
            goto out_kfree_skb;
    }
    .....
}
```

在neigh_resolve_output里面，首先neigh_event_send触发一个事件，看能否激活ARP。在__neigh_event_send中，激活ARP分两种情况，第一种情况是马上激活，也即immediate_probe。另一种情况是延迟激活则仅仅设置一个timer。然后将ARP包放在arp_queue上。如果马上激活，就直接调用neigh_probe；如果延迟激活，则定时器到了就会触发neigh_timer_handler，在这里面还是会调用neigh_probe。

```
static void neigh_probe(struct neighbour *neigh)
    __releases(neigh->lock)
{
    /* 从arp_queue中拿出ARP包 */
    struct sk_buff *skb = skb_peek_tail(&neigh->arp_queue);
    /* keep skb alive even if arp_queue overflows */
    if (skb)
        skb = skb_copy(skb, GFP_ATOMIC);
    write_unlock(&neigh->lock);
    /* 调用struct neighbour的solicit操作，即arp_hh_ops中的arp_solicit函数 */
    neigh->ops->solicit(neigh, skb);
    atomic_inc(&neigh->probes);
    kfree_skb(skb);
}
```

在neigh_probe中，先从arp_queue中拿出ARP包，然后调用arp_hh_ops中的arp_solicit函数。

```
static void arp_solicit(struct neighbour *neigh, struct sk_buff *skb)
{
    .....
    /* 创建并发送arp */
    arp_send(ARPOP_REQUEST, ETH_P_ARP, target, dev, saddr,
         dst_hw, dev->dev_addr, NULL);
}
```

在arp_solicit的最后调用了arp_send，在arp_send中创建并发送arp。

```
void arp_send(int type, int ptype, __be32 dest_ip,
          struct net_device *dev, __be32 src_ip,
          const unsigned char *dest_hw, const unsigned char *src_hw,
          const unsigned char *target_hw)
{
    struct sk_buff *skb;

    /*
     *  No arp on this interface.
     */

    if (dev->flags&IFF_NOARP)
        return;
    /* 创建一个arp的skb */
    skb = arp_create(type, ptype, dest_ip, dev, src_ip,
             dest_hw, src_hw, target_hw);
    if (skb == NULL)
        return;
    /* 发送arp */
    arp_xmit(skb);
}
```

回到neigh_resolve_output，在发送了arp后，调用dev_queue_xmit发送二层网络包，dev_queue_xmit调用的是__dev_queue_xmit。

```
static int __dev_queue_xmit(struct sk_buff *skb, void *accel_priv)
{
    struct net_device *dev = skb->dev;
    struct netdev_queue *txq;   /* 发送队列 */
    struct Qdisc *q;
    .....
    /* 取出发送队列 */
    txq = netdev_pick_tx(dev, skb, accel_priv);
    /* 取出发送队列的排队规则 */
    q = rcu_dereference_bh(txq->qdisc);
    .....
    if (q->enqueue) {
        /* 根据发送队列的排队规则发送skb */
        rc = __dev_xmit_skb(skb, q, dev, txq);
        goto out;
    }
```

```
    .....
}
```

在__dev_queue_xmit中先取出发送队列，然后再取出队列的排队规则，最后根据排队规则使用__dev_xmit_skb发送skb包。

```
static inline int __dev_xmit_skb(struct sk_buff *skb, struct Qdisc *q,
                 struct net_device *dev,
                 struct netdev_queue *txq)
{
    .....
         /* 将skb请求放入队列 */
        rc = q->enqueue(skb, q) & NET_XMIT_MASK;
        if (qdisc_run_begin(q)) {
            .....
            __qdisc_run(q);  /* 处理队列中的数据 */
        }
    .....
}
```

__dev_xmit_skb会将请求放入队列，然后调用__qdisc_run处理队列中的数据。

```
void __qdisc_run(struct Qdisc *q)
{
    int quota = weight_p;
    int packets;

    while (qdisc_restart(q, &packets)) {    /* 从队列取出数据发送 */
        /*
         * Ordered by possible occurrence: Postpone processing if
         * 1. we've exceeded packet quota
         * 2. another process needs the CPU;
         */
        quota -= packets;
        if (quota <= 0 || need_resched()) {
            __netif_schedule(q);    /* 重新调度q */
            break;
        }
    }

    qdisc_run_end(q);
}
```

在__qdisc_run中先通过qdisc_restart从队列中发送数据，每次发送的数据量写在packets变量中，然后递减quota。如果quota小于等于0或者占用cpu时间太多需要调度时就调用__netif_schedule重新调度。在__netif_schedule中调用__netif_reschedule，发起一个软中断NET_TX_SOFTIRQ。

```
static inline void __netif_reschedule(struct Qdisc *q)
{
    struct softnet_data *sd;
    unsigned long flags;
```

```
    local_irq_save(flags);
    sd = this_cpu_ptr(&softnet_data);
    q->next_sched = NULL;
    *sd->output_queue_tailp = q;
    sd->output_queue_tailp = &q->next_sched;
    /* 触发软中断NET_TX_SOFTIRQ */
    raise_softirq_irqoff(NET_TX_SOFTIRQ);
    local_irq_restore(flags);
}
```

在系统初始化的时候会定义软中断的处理函数。NET_TX_SOFTIRQ的处理函数是net_tx_action，用于发送网络包。还有一个NET_RX_SOFTIRQ的处理函数是net_rx_action，用于接收网络包。

```
    open_softirq(NET_TX_SOFTIRQ, net_tx_action);
    open_softirq(NET_RX_SOFTIRQ, net_rx_action);

...
    void open_softirq(int nr, void (*action)(struct softirq_action *))
{
    softirq_vec[nr].action = action;
}
```

接下来看一下软中断处理函数net_tx_action。

```
static void net_tx_action(struct softirq_action *h)
{
    struct softnet_data *sd = this_cpu_ptr(&softnet_data);
    .....
    if (sd->output_queue) {
        struct Qdisc *head;

        local_irq_disable();
        head = sd->output_queue; /* 获取队列 */
        sd->output_queue = NULL;
        sd->output_queue_tailp = &sd->output_queue;
        local_irq_enable();

        while (head) {  /* 当还有队列时 */
            struct Qdisc *q = head;
            spinlock_t *root_lock;

            head = head->next_sched;    /* 获取下一个队列 */

            .....
                qdisc_run(q);   /* 发送队列中的数据 */
            .....
        }
    }
}
```

在net_tx_action中还是调用了qdisc_run，还是会调用__qdisc_run，然后调用qdisc_restart发送网络包。接下来看qdisc_restart的实现。

```
static inline int qdisc_restart(struct Qdisc *q, int *packets)
{
    struct netdev_queue *txq;
    struct net_device *dev;
    spinlock_t *root_lock;
    struct sk_buff *skb;
    bool validate;

    /* 从从Qdisc队列中取出一个skb */
    skb = dequeue_skb(q, &validate, packets);
    if (unlikely(!skb))
        return 0;

    root_lock = qdisc_lock(q);
    dev = qdisc_dev(q);
    txq = skb_get_tx_queue(dev, skb);

    /* 发送 */
    return sch_direct_xmit(skb, q, dev, txq, root_lock, validate);
}
```

qdisc_restart将一个网络包从qdisc队列中取出，再取出一些必要的数据，然后调用sch_direct_xmit进行发送。

```
int sch_direct_xmit(struct sk_buff *skb, struct Qdisc *q,
            struct net_device *dev, struct netdev_queue *txq,
            spinlock_t *root_lock, bool validate)
{
    int ret = NETDEV_TX_BUSY;

    .....
     /* 先调用dev_hard_start_xmit进行发送 */
    if (skb) {
        HARD_TX_LOCK(dev, txq, smp_processor_id());
        if (!netif_xmit_frozen_or_stopped(txq))
            skb = dev_hard_start_xmit(skb, dev, txq, &ret);

        HARD_TX_UNLOCK(dev, txq);
    }
    spin_lock(root_lock);

    /* 如果发送不成功，会返回NETDEV_TX_BUSY,
     * 说明网卡很忙，于是就调用dev_requeue_skb，重新放入队列
    */
    if (dev_xmit_complete(ret)) {
        /* Driver sent out skb successfully or skb was consumed */
        ret = qdisc_qlen(q);
    } else if (ret == NETDEV_TX_LOCKED) {
        /* Driver try lock failed */
        ret = handle_dev_cpu_collision(skb, txq, q);
```

```
    } else {
        /* Driver returned NETDEV_TX_BUSY - requeue skb */
        if (unlikely(ret != NETDEV_TX_BUSY))
            net_warn_ratelimited("BUG %s code %d qlen %d\n",
                         dev->name, ret, q->q.qlen);
        ret = dev_requeue_skb(skb, q);
    }

    if (ret && netif_xmit_frozen_or_stopped(txq))
        ret = 0;

    return ret;
}
```

在sch_direct_xmit中，先调用dev_hard_start_xmit进行发送，如果发送不成功，会返回NETDEV_TX_BUSY，说明网卡很忙，于是就调用dev_requeue_skb，重新放入队列。

```
struct sk_buff *dev_hard_start_xmit(struct sk_buff *first, struct net_device *dev,
                    struct netdev_queue *txq, int *ret)
{
    struct sk_buff *skb = first;
    int rc = NETDEV_TX_OK;

    while (skb) {
        struct sk_buff *next = skb->next;    /* 队列中下一个skb */

        skb->next = NULL;
        /* 发送一个skb */
        rc = xmit_one(skb, dev, txq, next != NULL);
        if (unlikely(!dev_xmit_complete(rc))) {
            skb->next = next;
            goto out;
        }

        skb = next;
        if (netif_xmit_stopped(txq) && skb) {
            rc = NETDEV_TX_BUSY;
            break;
        }
    }

out:
    *ret = rc;
    return skb;
}
```

在dev_hard_start_xmit中，是一个while循环。每次在队列中取出一个sk_buff，调用xmit_one发送。然后调用链为xmit_one->netdev_start_xmit->__netdev_start_xmit。

```
static inline netdev_tx_t __netdev_start_xmit(const struct net_device_ops *ops,
                          struct sk_buff *skb, struct net_device *dev,
                          bool more)
{
    skb->xmit_more = more ? 1 : 0;
    return ops->ndo_start_xmit(skb, dev);
}
```

接下来就到了驱动程序层，调用的是网卡驱动注册的net_device_ops中的ndo_start_xmit函数发送网络帧。

三、总结

整个发送过程从上往下按层次划分为如下：

- **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。
- **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；ip_route_output_ports函数里面会调用fib_lookup查找路由表；填写IP层的头；通过iptables规则，相当于防火墙。
- **mac层：**调用ip_finish_output进入MAC层；调用___neigh_lookup_noref查找属于同一个网段的邻居，他会调用neigh_probe发送ARP；有了MAC地址，就可以调用dev_queue_xmit发送二层网络包了，它会调用__dev_xmit_skb会将请求放入队列。
- **设备驱动层：**网络包的发送会触发一个软中断NET_TX_SOFTIRQ来处理队列中的数据。这个软中断的处理函数是net_tx_action；在软中断处理函数中，会将网络包从队列上拿下来，调用网络设备的传输函数ops->ndo_start_xmit，将网络包发的设备的队列上去。