每条 TCP 连接先后会有 7 个定时器：建立连接定时器、重传定时器、延时确认定时器、持续定时器(Persistence Timer，如果收到了 rwnd = 0,需要发探测包)、保活定时器(Keep-alive Timer)、FIN_WAIT_2 定时器、TIME_WAIT 定时器。

1. 初始化

在 tcp_v4_init_sock 中调用 tcp_init_xmit_timers,后者转为调用 inet_csk_init_xmit_timers，初始化 retransmit_timer, delack_timer, keepalive 三个定时器，定时器的结构是：

```
struct timer_list {
 struct list_head entry;
 unsigned long expires;

 void (*function)(unsigned long);//回调函数，处理超时事件
 unsigned long data;

 struct tvec_t_base_s *base;
};
```

指定这三个定时器的 function 和 data

连接建立定时器

对于被动建立连接的一方才需要。当服务器收到对方的 SYN 后，回复 SYN/ACK， 然后等待对方的 ACK。这时候连接建立定时器开始计时，如果重传 SYN/ACK 超时次数达到上限，就终止建立连接。

在服务端，处于 listen 状态的套接字收到了一个新的连接请求后，就会为这个请求创建一个 request_sock， 添加在原先这个监听套接字的连接请求列表 syn_table 中，然后启动连接建立定时器 synack_timer。

Tcp_keepalive_timer 实现了三个定时器：建立连接定时器，保活定时器和 FIN_WAIT_2 定时器，因为这三个定时器分别出现在 TCP 处于 LISTEN, ESTABLISHED, FIN_WAIT_2 状态，所以可以复用。

连接建立定时器的回调函数 tcp_sysack_timer 会调用 inet_csk_reqsk_queue_prune()来删除那些尝试连接次数已经超过了 icsk_syn_retries 指定的次数但仍未建立连接的 request_sock,即起到定时清理 icsk_accept_queue 中的 request_sock 的作用。

重传定时器

重传定时器的回调函数是 tcp_write_timer。重传定时器和保持保持定时器用的是同一个定时器，所以需要通过 icsk->icsk_pending 来判断是哪个，如果是 ICSK_TIME_RETRANS，那么调用 tcp_retransmit_timer() 处理， 如果是 ICSK_TIME_PROBE0， 则调用 tcp_probe_timer 处理。

```
static void tcp_retransmit_timer(struct sock *sk)
{
 struct tcp_sock *tp = tcp_sk(sk);
 struct inet_connection_sock *icsk = inet_csk(sk);
```

```
    if (!tp->packets_out)//packets_out 表示所有已经发出但是没有被确认的 segment 的数目 = snd.nxt - snd.una，tp->packets_out 等于 0，表示所有发出去的包都被确认了，那肯定是不需要重传的。
        goto out;

    BUG_TRAP(!skb_queue_empty(&sk->sk_write_queue));

    if (!tp->snd_wnd && !sock_flag(sk, SOCK_DEAD) &&
        !((1 << sk->sk_state) & (TCPF_SYN_SENT | TCPF_SYN_RECV))) {//如果发送窗口为 0 并且 socket 不处于 DEAD 和连接建立状态
        /* Receiver dastardly shrinks window. Our retransmits
         * become zero probes, but we should not timeout this
         * connection. If the socket is an orphan, time it out,
         * we cannot allow such beasts to hang infinitely.
         */
    /* 如果窗口降为 0，重传就变为是零窗口探测，这样的话不能通过 120s 的超时关闭这个连接。 但是如果这个 socket 是一个孤儿套接字，不属于任何进程，就通过超时关了它 */
    #ifdef TCP_DEBUG
        if (net_ratelimit()) {
            struct inet_sock *inet = inet_sk(sk);
            printk(KERN_DEBUG "TCP: Treason uncloaked! Peer %u.%u.%u.%u:%u/%u shrinks window %u:%u. Repaired.\n",
                   NIPQUAD(inet->daddr), ntohs(inet->dport),
                   inet->num, tp->snd_una, tp->snd_nxt); //num: local port
        }
    #endif
        if (tcp_time_stamp - tp->rcv_tstamp > TCP_RTO_MAX) {
            tcp_write_err(sk);
            goto out;
        }
/* 如果最近一次收到确认的时间距离现在已经超过了 120s， 就报告错误并关闭套接字， 否则进入 LOSS 状态，并重传包 一个 skb，清空传输控制块中的路由表项，重置定时器*/
        tcp_enter_loss(sk, 0);
        tcp_retransmit_skb(sk, skb_peek(&sk->sk_write_queue));
        __sk_dst_reset(sk);/* 由于发生了重传，sock 中的路由缓存需要更新，所以将其清除*/
        goto out_reset_timer;
    }
    /* 发生重传后，需要检测这个连接资源使用情况和重传次数，如果重传次数超过了限额（tcp_retries2，15 次），就要报告错误并关闭这个套接口；*/
```

```
    if (tcp_write_timeout(sk))//出错返回 1， 正常返回 0
        goto out;
    /*如果是第一次重传，需要根据不同的拥塞状态进行一些数据统计*/
    if (icsk->icsk_retransmits == 0) {
        if (icsk->icsk_ca_state == TCP_CA_Disorder ||
            icsk->icsk_ca_state == TCP_CA_Recovery) {
            if (tp->rx_opt.sack_ok) {
                if (icsk->icsk_ca_state == TCP_CA_Recovery)

NET_INC_STATS_BH(LINUX_MIB_TCPSACKRECOVERYFAIL);
                else
                    NET_INC_STATS_BH(LINUX_MIB_TCPSACKFAILURES);
            } else {
                if (icsk->icsk_ca_state == TCP_CA_Recovery)

NET_INC_STATS_BH(LINUX_MIB_TCPRENORECOVERYFAIL);
                else
                    NET_INC_STATS_BH(LINUX_MIB_TCPRENOFAILURES);
            }
        } else if (icsk->icsk_ca_state == TCP_CA_Loss) {
            NET_INC_STATS_BH(LINUX_MIB_TCPLOSSFAILURES);
        } else {
            NET_INC_STATS_BH(LINUX_MIB_TCPTIMEOUTS);
        }
    }

    if (tcp_use_frto(sk)) {
        tcp_enter_frto(sk);
    } else {
        tcp_enter_loss(sk, 0);
    }
    /* 如果重传 SKB 失败，则复位重传定时器下次再传*/
    if (tcp_retransmit_skb(sk, skb_peek(&sk->sk_write_queue)) >
0) {
        /* Retransmission failed because of local congestion,
         * do not backoff.
         */
        if (!icsk->icsk_retransmits)
            icsk->icsk_retransmits = 1;
        inet_csk_reset_xmit_timer(sk, ICSK_TIME_RETRANS,
                  min(icsk->icsk_rto,
TCP_RESOURCE_PROBE_INTERVAL),
                  TCP_RTO_MAX);
        goto out;
```

```
    }

    /* Increase the timeout each time we retransmit.  Note that
     * we do not increase the rtt estimate.  rto is initialized
     * from rtt, but increases here.  Jacobson (SIGCOMM 88) suggests
     * that doubling rto each time is the least we can get away with.
     * In KA9Q, Karn uses this for the first few times, and then
     * goes to quadratic.  netBSD doubles, but only goes up to *64,
     * and clamps at 1 to 64 sec afterwards.  Note that 120 sec is
     * defined in the protocol as the maximum possible RTT.  I guess
     * we'll have to use something other than TCP to talk to the
     * University of Mars.
     *
     * PAWS allows us longer timeouts and large windows, so once
     * implemented ftp to mars will work nicely. We will have to fix
     * the 120 second clamps though!
     */
    icsk->icsk_backoff++;
    icsk->icsk_retransmits++;
    /*超时后，把超时时间增大为原来的 2 倍， 然后复位重传定时器*/
    out_reset_timer:
    icsk->icsk_rto = min(icsk->icsk_rto << 1, TCP_RTO_MAX);
    inet_csk_reset_xmit_timer(sk,            ICSK_TIME_RETRANS, icsk->icsk_rto, TCP_RTO_MAX);
    /* 如果重传次数超过 3 次，那么要清空 sock 的路由项缓存 */
    if (icsk->icsk_retransmits > sysctl_tcp_retries1)
        __sk_dst_reset(sk);

    out:;
    }
```

延时确认定时器

收到数据后应该回复 ACK，但是可以过一段时间再发送，但是不能超过 200ms, 如果在这 200ms 内，有数据要发送，那么确认可以和数据一起发送。延时确认定时器的回调函数是 tcp_delack_timer.

保活定时器

如果上一次收到确认的时间到现在的空闲时间超过了 2 小时，那么 keepalive 定时器超时，向对端发送探测包。

FIN_WAIT_2 定时器

当应用层调用 close 后，给对方发送 FIN,进入到 FIN_WAIT_1, 收到对方的 ACK 后，进入 FIN_WAIT_2 状态，直到收到对方的 FIN 才进入下一个状态。为了避免对方一直不发 FIN 而永远滞留在 FIN_WAIT_2 状态，需要 FIN_WAIT_2 定时器。当 TCP 处于 FIN_WAIT_2 状态超过 60s 之后，会激活 FIN_WAIT_2 定时器。

TIME_WAIT 定时器

处于 TIME_WAIT 状态后，TIME_WAIT 定时器启动，等 2MSL,如果超时了，就关闭连接