lwip

———狗拿耗子第四篇

1、lwip 的背景

lwip 是 Swedish Institute of Computer Science 开发的用于嵌入式系统的 TCP\IP 协议栈，从网上的评论看似乎用的人不少。它的实现是应该参考了 BSD 的实现，在介绍 TCP 的时候，大家就会发现，其拥塞控制的实现算法机会与 BSD 的一模一样。lwip 的整个代码写的比 YASS2 差一截，难以入手介绍，我打算按照 TCP 的 server 与 client 的角度分别走一遍代码。

lwip 的内核运行在同一个任务中，lwip 提供的系统调用通过 mailbox 与内核进行通信，然后用户阻塞在一个专门的 mailbox 上，内核完成用户的请求后 post 该 mailbox，用户得以继续执行。有的协议栈的实现却是，每层跑在一个任务中，这样层与层间的相互调用将会引起上下文的切换。更重要的是 lwip 可以运行在裸机的环境中，即不需要操作系统的支持。这对一些低成本的设备还是很具有吸引力的。

lwip 的官方网站为 http://savannah.nongnu.org/projects/lwip/，目前最新的版本是 1.3.0，而本文参考的是 1.2.0。

2、netconn_new 系统调用

2.1 相关的数据结构

```
enum netconn_type {
  NETCONN_TCP,
  NETCONN_UDP,
  NETCONN_UDPLITE,
  NETCONN_UDPNOCHKSUM,
  NETCONN_RAW
};

struct netconn {
  enum netconn_type type;
  enum netconn_state state;
  union {
    struct tcp_pcb *tcp;
    struct udp_pcb *udp;
    struct raw_pcb *raw;
  } pcb;
  err_t err;
  sys_mbox_t mbox;
  sys_mbox_t recvmbox;
  sys_mbox_t acceptmbox;
  sys_sem_t sem;
  int socket;
  u16_t recv_avail;
  void (* callback)(struct netconn *, enum netconn_evt, u16_t len);
};
```

- struct netconn 用一个 union 将 udp、tcp、raw 的 pcb 包含起来，实现由 netconn 到不同协议的分发，这是 c 语言编程的一个常用技巧。
- sys_mbox_t mbox，用户阻塞在该 mailbox 上，内核处理完用户的请求后，post 该 mailbox，用户继续执行。
- sys_mbox_t recvmbox，如其名，用户用该 mailbox 接收来自内核的数据。
- sys_mbox_t acceptmbox，用户调用 accept 阻塞在该 mailbox 上，内核接收到来自网络的连接请求并完成三次握手后，post 该 mailbox。
- sys_sem_t sem，系统调用 netconn_write 发现内核没有足够空间时 wait 该 semaphore，内核在适当的时候会 post 该 semaphore，则操作系统唤醒运行在用户任务的系统调用，再次尝试发送数据。

2.2 流程

```
struct netconn *netconn_new_with_proto_and_callback(enum netconn_type t, u16_t proto,
                                                    void (*callback)(struct netconn *, enum netconn_evt, u16_t len))
{
  struct netconn *conn;
  struct api_msg *msg;

  conn = memp_malloc(MEMP_NETCONN);
    ......
  conn->err = ERR_OK;
  conn->type = t;
  conn->pcb.tcp = NULL; /*  表示没有关联 pcb */

  if ((conn->mbox = sys_mbox_new()) == SYS_MBOX_NULL) {
    memp_free(MEMP_NETCONN, conn);
    return NULL;
  }
  conn->recvmbox = SYS_MBOX_NULL;
  conn->acceptmbox = SYS_MBOX_NULL;
  conn->sem = sys_sem_new(0);
    ......
  conn->state = NETCONN_NONE;
  conn->socket = 0;
  conn->callback = callback;
  conn->recv_avail = 0;

  if((msg = memp_malloc(MEMP_API_MSG)) == NULL) {
    memp_free(MEMP_NETCONN, conn);
    return NULL;
  }

  msg->type = API_MSG_NEWCONN;
  msg->msg.msg.bc.port = proto; /* misusing the port field */
  msg->msg.conn = conn;
```

```
  api_msg_post(msg);   /* 请求内核完成操作，内核需要根据链接类型分配并初始化 pcb */
  sys_mbox_fetch(conn->mbox, NULL); /* 等待内核完成操作 */
  memp_free(MEMP_API_MSG, msg);

    ......

  return conn;
}
```

3、内核函数 do_newconn

```
    ......
  msg->conn->err = ERR_OK;
  /* Allocate a PCB for this connection */
  switch(msg->conn->type) {
    ......
  case NETCONN_TCP:
    msg->conn->pcb.tcp = tcp_new();
    if(msg->conn->pcb.tcp == NULL) {
      msg->conn->err = ERR_MEM;
      break;
    }
    setup_tcp(msg->conn);
    break;
  }
  sys_mbox_post(msg->conn->mbox, NULL);   /* 为 TCP connection 分配并初始化 pcb，post 系统调用准备
                                             的 mailbox，系统调用得以继续执行 */
}
```

3.1 内核函数 tcp_alloc

```
struct tcp_pcb *
tcp_alloc(u8_t prio)
{
  struct tcp_pcb *pcb;
  u32_t iss;

  pcb = memp_malloc(MEMP_TCP_PCB);
  if (pcb == NULL) {
    /* Try killing oldest connection in TIME-WAIT. */
    LWIP_DEBUGF(TCP_DEBUG, ("tcp_alloc: killing off oldest TIME-WAIT connection\n"));
    tcp_kill_timewait();
    pcb = memp_malloc(MEMP_TCP_PCB);
    if (pcb == NULL) {
      tcp_kill_prio(prio);      /* 释放优先级低的 pcb，这招够狠的 */
      pcb = memp_malloc(MEMP_TCP_PCB);
```

```
    }
  }
  if (pcb != NULL) {
    memset(pcb, 0, sizeof(struct tcp_pcb));
    pcb->prio = TCP_PRIO_NORMAL;
    pcb->snd_buf = TCP_SND_BUF;    /* self available buffer space for sending (in bytes). */
    pcb->snd_queuelen = 0; /* self available buffer space for sending (in tcp_segs). */
    pcb->rcv_wnd = TCP_WND; /* self receiver window (in bytes)*/
    pcb->tos = 0;
    pcb->ttl = TCP_TTL;
    pcb->mss = TCP_MSS;
    pcb->rto = 3000 / TCP_SLOW_INTERVAL;
    pcb->sa = 0;
    pcb->sv = 3000 / TCP_SLOW_INTERVAL;
    pcb->rtime = 0;
    pcb->cwnd = 1;
    iss = tcp_next_iss();
    pcb->snd_wl2 = iss; /* acknowledgement numbers of last window update from peer. */
    pcb->snd_nxt = iss; /* next seqno to be sent by peer*/
    pcb->snd_max = iss; /* Highest seqno sent by peer. */
    pcb->lastack = iss; /* Highest acknowledged seqno from peer. */
    pcb->snd_lbb = iss;
    pcb->tmr = tcp_ticks;
    pcb->polltmr = 0;

#if LWIP_CALLBACK_API
    pcb->recv = tcp_recv_null;
#endif /* LWIP_CALLBACK_API */

    /* Init KEEPALIVE timer */
    pcb->keepalive = TCP_KEEPDEFAULT;
    pcb->keep_cnt = 0;
  }
  return pcb;
}
```

从 pcb 分配函数分配一个 pcb，并初始化发送、接收、定时器参数。

3.2 内核函数 setup_tcp

```
static void setup_tcp(struct netconn *conn)
{
  struct tcp_pcb *pcb;

  pcb = conn->pcb.tcp;
  tcp_arg(pcb, conn);
```

```
   tcp_recv(pcb, recv_tcp);
   tcp_sent(pcb, sent_tcp);
   tcp_poll(pcb, poll_tcp, 4);
   tcp_err(pcb, err_tcp);
}
```

在 pcb 上绑定各个事件的处理函数，相应功能在随后介绍。

4、系统调用 netconn_bind

```
err_t netconn_bind(struct netconn *conn, struct ip_addr *addr,
          u16_t port)
{
   struct api_msg *msg;

   if (conn == NULL) {
      return ERR_VAL;
   }

   if (conn->type != NETCONN_TCP &&
       conn->recvmbox == SYS_MBOX_NULL) {
      if ((conn->recvmbox = sys_mbox_new()) == SYS_MBOX_NULL) {
         return ERR_MEM;
      }
   }

   if ((msg = memp_malloc(MEMP_API_MSG)) == NULL) {
      return (conn->err = ERR_MEM);
   }
   msg->type = API_MSG_BIND;
   msg->msg.conn = conn;
   msg->msg.msg.bc.ipaddr = addr;
   msg->msg.msg.bc.port = port;
   api_msg_post(msg);
   sys_mbox_fetch(conn->mbox, NULL);
   memp_free(MEMP_API_MSG, msg);
   return conn->err;
}
```

为 tcp connnection 分配 receive mailbox，然后发消息 API_MSG_BIND 给内核。

5、内核函数 do_bind

```
static void do_bind(struct api_msg_msg *msg)
{
       ......
   switch (msg->conn->type) {
       ......
```

```
    case NETCONN_TCP:
      msg->conn->err = tcp_bind(msg->conn->pcb.tcp,
                msg->msg.bc.ipaddr, msg->msg.bc.port);
    default:
      break;
    }
    sys_mbox_post(msg->conn->mbox, NULL);
}
```

内核函数 tcp_bind 将本地 ip、port 绑定到指定的 connection 上，源代码中的函数说明如下。

```
/**
 * Binds the connection to a local portnumber and IP address. If the
 * IP address is not given (i.e., ipaddr == NULL), the IP address of
 * the outgoing network interface is used instead.
 * */
```

6、系统调用 netconn_listen

```
err_t netconn_listen(struct netconn *conn)
{
    struct api_msg *msg;

    if (conn == NULL) {
      return ERR_VAL;
    }

    if (conn->acceptmbox == SYS_MBOX_NULL) {
      conn->acceptmbox = sys_mbox_new();
      if (conn->acceptmbox == SYS_MBOX_NULL) {
        return ERR_MEM;
      }
    }

    if ((msg = memp_malloc(MEMP_API_MSG)) == NULL) {
      return (conn->err = ERR_MEM);
    }
    msg->type = API_MSG_LISTEN;
    msg->msg.conn = conn;
    api_msg_post(msg);
    sys_mbox_fetch(conn->mbox, NULL);
    memp_free(MEMP_API_MSG, msg);
    return conn->err;
}
```

为 tcp connnection 分配 accept mailbox，然后发消息 API_MSG_LISTEN 给内核。

7、内核函数 do_listen

```
static void do_listen(struct api_msg_msg *msg)
{
   if (msg->conn->pcb.tcp != NULL) {
      switch (msg->conn->type) {
       ......
      case NETCONN_TCP:
         msg->conn->pcb.tcp = tcp_listen(msg->conn->pcb.tcp);
         if (msg->conn->pcb.tcp == NULL) {
                msg->conn->err = ERR_MEM;
         } else {
                  ......
                tcp_arg(msg->conn->pcb.tcp, msg->conn);
                tcp_accept(msg->conn->pcb.tcp, accept_function);
         }
      default:
         break;
      }
   }
   sys_mbox_post(msg->conn->mbox, NULL);
}
```

7.1 内核函数 tcp_listen

```
struct tcp_pcb *tcp_listen(struct tcp_pcb *pcb)
{
   struct tcp_pcb_listen *lpcb;
       ......
   lpcb = memp_malloc(MEMP_TCP_PCB_LISTEN);
   if (lpcb == NULL) {
      return NULL;
   }
   lpcb->callback_arg = pcb->callback_arg;
   lpcb->local_port = pcb->local_port;
   lpcb->state = LISTEN;
   lpcb->so_options = pcb->so_options;
   lpcb->so_options |= SOF_ACCEPTCONN;
   lpcb->ttl = pcb->ttl;
   lpcb->tos = pcb->tos;
   ip_addr_set(&lpcb->local_ip, &pcb->local_ip);
   memp_free(MEMP_TCP_PCB, pcb);
#if LWIP_CALLBACK_API
   lpcb->accept = tcp_accept_null;
#endif /* LWIP_CALLBACK_API */
   TCP_REG(&tcp_listen_pcbs.listen_pcbs, lpcb);
```

```
    return (struct tcp_pcb *)lpcb;
}
```

分配一个类型为 tcp_pcb_listen 的 pcb，并将 tcp_pcb 的相应的参数拷贝过来，包括 self ip、self port 等，设置新分配 pcb 的状态为 LISTEN，最后将它放入队列 tcp_listen_pcbs。队列 tcp_listen_pcbs 存放着所有处于 LISTEN 状态的 pcb，相应的 connection 的指针可从 pcb->callback_arg 获得。

7.2 注册 accept_function，当接收了一个 connect 链接请求并三次握手完成后，调用该函数通知用户任务。

8、RTT 的估计

8.1 估计 RTT 的算法

```
static u8_t tcp_receive(struct tcp_pcb *pcb)
{
      ......
     /* RTT estimation calculations. This is done by checking if the
         incoming segment acknowledges the segment we use to take a
         round-trip time measurement. */
     if (pcb->rttest && TCP_SEQ_LT(pcb->rtseq, ackno)) {
       m = tcp_ticks - pcb->rttest;
      ......
       /* This is taken directly from VJs original code in his paper */
       m = m - (pcb->sa >> 3);
       pcb->sa += m;
       if (m < 0) {
          m = -m;
       }
       m = m - (pcb->sv >> 2);
       pcb->sv += m;
       pcb->rto = (pcb->sa >> 3) + pcb->sv;
      ......
       pcb->rttest = 0;
     }
     ......
}
```

- tcp_ticks，内核的 tick，500ms 递增一次。
- pcb->rttest，发起 rtt 估计时的 tick，当该值为 0 时表示不需要做 rtt 估计。
- pcb->rtseq，用于 rtt 估计的 segment，当该 segment 的 ack 到达时，进行 rtt 估计。
- pcb->sa，A = sa >> 3。
- pcb->sv，D = sv >> 2。
- pcb->rto，rtt 的估计值。

算法的详细叙述见《TCP/IP 详解，卷 1：协议》第 21 章，上述代码可转换为：

```
       m = m - (pcb->sa >> 3); /* Err = M – A */
       pcb->sa += m; /* 8A = 8A + Err，即 A = A + Err / 8 */
       if (m < 0) { /* e = |Err| */
          m = -m;
```

```
        }
        m = m - (pcb->sv >> 2); /* t = |Err| - D */
        pcb->sv += m; /* 4D = 4D + (|Err| - D)，即 D = D + (|Err| - D) / 4 */
        pcb->rto = (pcb->sa >> 3) + pcb->sv; /* RTT = A + 4D */
```

可见，与《卷 1：协议》中描述的一模一样。

8.2 启动 RTT 估计的时机

《卷 1：协议》说在内核没有进行 rtt 估计时，且不处于超时重传状态（避免重传多义性）（此处指的不是快速重传状态）时可以进行 rtt 估计。

1) 在窗口允许的前提下，即 MIN（self cwnd, peer *rcv_wnd*）大于待发送 segment 的长度，内核函数 tcp_output 无条件调用内核函数 tcp_output_segment。

2) 内核函数 tcp_output_segment 发现当前没有进行 rtt 估计，则无条件立刻启动 rtt 估计。

```
static void tcp_output_segment(struct tcp_seg *seg, struct tcp_pcb *pcb)
{
      ......
  pcb->rtime = 0;

  if (pcb->rttest == 0) {
    pcb->rttest = tcp_ticks;
    pcb->rtseq = ntohl(seg->tcphdr->seqno);

    LWIP_DEBUGF(TCP_RTO_DEBUG, ("tcp_output_segment: rtseq %"U32_F"\n", pcb->rtseq));
  }
      ......
  ip_output(seg->p, &(pcb->local_ip), &(pcb->remote_ip), pcb->ttl, pcb->tos,
        IP_PROTO_TCP);
}
```

与《卷 1：协议》不一致的是：快速重传内核函数 tcp_rexmit 会调用 tcp_output，这里没问题，因为不存在重传二义性；但超时重传内核函数 tcp_rexmit_rto 也会调用 tcp_output，这是与《卷 1：协议》矛盾的。如果我的理解没出错的话，lwip 在这个地方在估计 rtt 时会存在一定的风险。

8.3 rtt 估计的关闭

将 pcb->rttest 置为 0 即关闭 rtt 估计，共有三处会关闭 rtt 估计：

- tcp_receive，完成了 rtt 估计的运算，正常关闭。
- tcp_rexmit，快速重传，关闭 rtt 估计。
- tcp_rexmit_rto，超时重传，关闭 rtt 估计。

9、segment 的超时重传与慢启动

```
void tcp_slowtmr(void)
{
      ......
    pcb_remove = 0;

    if (pcb->state == SYN_SENT && pcb->nrtx == TCP_SYNMAXRTX) {
```

```
      ++pcb_remove;
      LWIP_DEBUGF(TCP_DEBUG, ("tcp_slowtmr: max SYN retries reached\n"));
    }
    else if (pcb->nrtx == TCP_MAXRTX) {
      ++pcb_remove;
      LWIP_DEBUGF(TCP_DEBUG, ("tcp_slowtmr: max DATA retries reached\n"));
    } else {
      ++pcb->rtime;
      if (pcb->unacked != NULL && pcb->rtime >= pcb->rto) {

        /* Time for a retransmission. */
        LWIP_DEBUGF(TCP_RTO_DEBUG, ("tcp_slowtmr: rtime %"U16_F" pcb->rto %"U16_F"\n",
          pcb->rtime, pcb->rto));

        /* Double retransmission time-out unless we are trying to
         * connect to somebody (i.e., we are in SYN_SENT). */
        if (pcb->state != SYN_SENT) {
          pcb->rto = ((pcb->sa >> 3) + pcb->sv) << tcp_backoff[pcb->nrtx];
        }
        /* Reduce congestion window and ssthresh. */
        eff_wnd = LWIP_MIN(pcb->cwnd, pcb->snd_wnd);
        pcb->ssthresh = eff_wnd >> 1;
        if (pcb->ssthresh < pcb->mss) {
          pcb->ssthresh = pcb->mss * 2;
        }
        pcb->cwnd = pcb->mss;
        LWIP_DEBUGF(TCP_CWND_DEBUG, ("tcp_slowtmr: cwnd %"U16_F" ssthresh %"U16_F"\n",
                                     pcb->cwnd, pcb->ssthresh));

        /* The following needs to be called AFTER cwnd is set to one mss - STJ */
        tcp_rexmit_rto(pcb);
      }
    }
    .....
}
```

9.1 相应数据结构

- pcb->rto，重传定时器时长，rtt 估计时给出初始值，当超时重传时该值按指数增大。
- pcb->rtime，segment 从被发出到此时（没有收到 ack），所经历的 tick 数。在 *tcp_output_segment* 中，该值被置 0。在 tcp_slowtmr 中，该值被递增。
- pcb->nrtx，segment 重传的次数。在 tcp_receive 中，收到新的 ack 时，该值被置 0。在 tcp_rexmit、tcp_rexmit_rto 中，该值被递增。

9.2 主要流程

当重传次数大于 TCP_MAXRTX 时，segment 被扔掉。当 rtime 大于 rto 时，启动慢启动流程，慢启动门限被置为当前窗口的一半，但要不小于 2 个 mss；拥塞窗口为一个 mss，rto 按指数增大；最后重传该 segment。详细介绍见《卷 1：协议》21.6。此后如果有新的 ack 到达时，拥塞窗口每次递增一个 mss。

10、快速重传与快速恢复

```
tcp_receive(struct tcp_pcb *pcb)
{
  ......
  if (pcb->lastack == ackno) { /*  到达的 ack 为重复 ack */
    pcb->acked = 0;

    if (pcb->snd_wl1 + pcb->snd_wnd == right_wnd_edge){ /* peer rcv_wnd 没有张开 */
      ++pcb->dupacks;
      if (pcb->dupacks >= 3 && pcb->unacked != NULL) { /*  收到了三个以上的重复 ack */
        if (!(pcb->flags & TF_INFR)) { /*  不处于快速恢复状态 */
          /* This is fast retransmit. Retransmit the first unacked segment. */
            ......
          tcp_rexmit(pcb);
            ......
          /* Set ssthresh to half of the minimum of the currenct cwnd and the advertised window */
          if (pcb->cwnd > pcb->snd_wnd)    /*  慢启动门限被置为当前窗口的一半 */
            pcb->ssthresh = pcb->snd_wnd / 2;
          else
            pcb->ssthresh = pcb->cwnd / 2;

          pcb->cwnd = pcb->ssthresh + 3 * pcb->mss; /*  拥塞窗口为慢启动门限加上 3 个 mss */
          pcb->flags |= TF_INFR; /*  标记当前状态为快速恢复 */
        } else { /*  已处于快速恢复状态 */
          /* Inflate the congestion window, but not if it means that
             the value overflows. */
          if ((u16_t)(pcb->cwnd + pcb->mss) > pcb->cwnd) {
            pcb->cwnd += pcb->mss; /*  当前处于快速恢复状态，拥塞窗口递增一个 mss */
          }
        }
      }
    } else {
      LWIP_DEBUGF(TCP_FR_DEBUG, ("tcp_receive: dupack averted %"U32_F" %"U32_F"\n",
                                  pcb->snd_wl1 + pcb->snd_wnd, right_wnd_edge));
    }
  } else
    if (TCP_SEQ_BETWEEN(ackno, pcb->lastack+1, pcb->snd_max)){
    /* We come here when the ACK acknowledges new data. */

    /* Reset the "IN Fast Retransmit" flag, since we are no longer
```

```
            in fast retransmit. Also reset the congestion window to the
            slow start threshold. */
          if (pcb->flags & TF_INFR) { /* 如果处于快速恢复，新的 ack 到达时，应将拥塞控制窗口置为慢启动
                                         门限的值（拥塞发生时的窗口大小的一半）；并清除快速恢复标识；
                                         此时传入拥塞避免状态 */
            pcb->flags &= ~TF_INFR;
            pcb->cwnd = pcb->ssthresh;
          }

          /* Reset the number of retransmissions. */
          pcb->nrtx = 0;

          /* Reset the retransmission time-out. */
          pcb->rto = (pcb->sa >> 3) + pcb->sv;

          /* Update the send buffer space. */
          pcb->acked = ackno - pcb->lastack;
          pcb->snd_buf += pcb->acked;

          /* Reset the fast retransmit variables. */
          pcb->dupacks = 0;
          pcb->lastack = ackno;

          /* Update the congestion control variables (cwnd and ssthresh). */
          if (pcb->state >= ESTABLISHED) {
            if (pcb->cwnd < pcb->ssthresh) { /* 当前处于慢启动状态，拥塞窗口递增一个 mss */
              if ((u16_t)(pcb->cwnd + pcb->mss) > pcb->cwnd) {
                pcb->cwnd += pcb->mss;
              }
              LWIP_DEBUGF(TCP_CWND_DEBUG, ("tcp_receive: slow start cwnd %"U16_F"\n", pcb->cwnd));
            } else { /* 当前处于拥塞避免状态，拥塞窗口递增 mss/N，其中 cwnd = N*mss   */
              u16_t new_cwnd = (pcb->cwnd + pcb->mss * pcb->mss / pcb->cwnd);
              if (new_cwnd > pcb->cwnd) {
                pcb->cwnd = new_cwnd;
              }
              ......
            }
          }
          ......
        }
      }
```

11、内核函数 tcp_enqueue

*err_t tcp_enqueue(struct tcp_pcb *pcb, void *arg, u16_t len, u8_t flags, u8_t copy, u8_t *optdata, u8_t optlen)*

- arg，待发送数据指针
- len，待发送数据长度
- flags，待发送 segment 的 flag，如 FIN、SYN
- copy，表示 tcp_enqueue 是否需要拷贝待发送数据，1 表示将持有数据的拷贝的 segment 放入待发送队列，0 表示将持有数据的引用的 segment 放入待发送队列。
- optdata，待发送选项数据指针
- optlen，待发送选项数据长度

11.1 对 tcp_enqueue 的引用

- tcp_connect，发送 SYN 和 mss。
- tcp_listen_input，发送 SYN、ACK 和 mss。
- tcp_send_ctrl，发送给定的 flags
- tcp_write，发送给定的数据

可见只有 tcp_connect、tcp_listen_input 发送了 optdata，可以总结出：

1）当有 optdata 发送时，flags 一定被掩上了 SYN

2）optdata 与数据不同时发送

11.2 主要流程

```
err_t tcp_enqueue(struct tcp_pcb *pcb, void *arg, u16_t len,
  u8_t flags, u8_t copy,
  u8_t *optdata, u8_t optlen)
{
  struct pbuf *p;
  struct tcp_seg *seg, *useg, *queue;
  u32_t left, seqno;
  u16_t seglen;
  void *ptr;
  u8_t queuelen;
      ......
  /* fail on too much data */
  if (len > pcb->snd_buf) {
    ......
    return ERR_MEM;
  }
  left = len;
  ptr = arg;

  /* seqno will be the sequence number of the first segment enqueued
   * by the call to this function. */
  seqno = pcb->snd_lbb; /* snd_lbb，下一个 segment 的 sequence number */
      ......
  /* If total number of pbufs on the unsent/unacked queues exceeds the
   * configured maximum, return an error */
  queuelen = pcb->snd_queuelen;
```

```
if (queuelen >= TCP_SND_QUEUELEN) { /* buffer 最多值放 TCP_SND_QUEUELEN 个 segment */
    ......
  return ERR_MEM;
}
    ......
/* First, break up the data into segments and tuck them together in
 * the local "queue" variable. */
useg = queue = seg = NULL;
seglen = 0;
while (queue == NULL || left > 0) {

  /* The segment length should be the MSS if the data to be enqueued
   * is larger than the MSS. */
  seglen = left > pcb->mss? pcb->mss: left; /*  待发送数据被分割成 mss 大小的 segment */

  /* Allocate memory for tcp_seg, and fill in fields. */
  seg = memp_malloc(MEMP_TCP_SEG); /*  分配一个 segment */
  if (seg == NULL) {
    LWIP_DEBUGF(TCP_OUTPUT_DEBUG | 2, ("tcp_enqueue: could not allocate memory for tcp_seg\n"));
    goto memerr;
  }
  seg->next = NULL;
  seg->p = NULL;

  /* first segment of to-be-queued data? */
  if (queue == NULL) {
    queue = seg;
  }
  /* subsequent segments of to-be-queued data */
  else {
    /* Attach the segment to the end of the queued segments */
    LWIP_ASSERT("useg != NULL", useg != NULL);
    useg->next = seg;
  }
  /* remember last segment of to-be-queued data for next iteration */
  useg = seg;

  /* If copy is set, memory should be allocated
   * and data copied into pbuf, otherwise data comes from
   * ROM or other static memory, and need not be copied. If
   * optdata is != NULL, we have options instead of data. */

  /* options? */
  if (optdata != NULL) {    /*  分配一个 RAM 类型的 pbuf，optdata 的拷贝在后面  */
```

```
    if ((seg->p = pbuf_alloc(PBUF_TRANSPORT, optlen, PBUF_RAM)) == NULL) {
      goto memerr;
    }
    ++queuelen;
    seg->dataptr = seg->p->payload;
  }
  /* copy from volatile memory? */
  else if (copy) { /* 分配一个 RAM 类型的 pbuf，将待发送数据拷贝进去 */
    if ((seg->p = pbuf_alloc(PBUF_TRANSPORT, seglen, PBUF_RAM)) == NULL) {
        ......
      goto memerr;
    }
    ++queuelen;
    if (arg != NULL) {
      memcpy(seg->p->payload, ptr, seglen);
    }
    seg->dataptr = seg->p->payload;
  }
  /* do not copy data */
  else {
    /* First, allocate a pbuf for holding the data.
     * since the referenced data is available at least until it is sent out on the
     * link (as it has to be ACKed by the remote party) we can safely use PBUF_ROM
     * instead of PBUF_REF here. */
    if ((p = pbuf_alloc(PBUF_TRANSPORT, seglen, PBUF_ROM)) == NULL) {
      ......
      goto memerr;
    }
    ++queuelen;
    /* reference the non-volatile payload data */
    p->payload = ptr;
    seg->dataptr = ptr;

    /* Second, allocate a pbuf for the headers. */
    if ((seg->p = pbuf_alloc(PBUF_TRANSPORT, 0, PBUF_RAM)) == NULL) {
      /* If allocation fails, we have to deallocate the data pbuf as
       * well. */
      pbuf_free(p);
     ......
      goto memerr;
    }
    ++queuelen;

    /* Concatenate the headers and data pbufs together. */
```

```
      pbuf_cat(seg->p/*header*/, p/*data*/);
      p = NULL;
    }
     /* 此时，已分配一个 segment，并关联上了一个 pbuf 或 pbuf chain */
    /* Now that there are more segments queued, we check again if the
    length of the queue exceeds the configured maximum. */
    if (queuelen > TCP_SND_QUEUELEN) {
      ......
      goto memerr;
    }

    seg->len = seglen;

    /* build TCP header */
    if (pbuf_header(seg->p, TCP_HLEN)) {
      ......
      goto memerr;
    }
    seg->tcphdr = seg->p->payload;
    seg->tcphdr->src = htons(pcb->local_port);
    seg->tcphdr->dest = htons(pcb->remote_port);
    seg->tcphdr->seqno = htonl(seqno);
    seg->tcphdr->urgp = 0;
    TCPH_FLAGS_SET(seg->tcphdr, flags);
    /* don't fill in tcphdr->ackno and tcphdr->wnd until later */ /* 到真正发送时再填写 */
    /* Copy the options into the header, if they are present. */
    if (optdata == NULL) {
      TCPH_HDRLEN_SET(seg->tcphdr, 5);
    }
    else {
      TCPH_HDRLEN_SET(seg->tcphdr, (5 + optlen / 4));
      /* Copy options into data portion of segment.
       Options can thus only be sent in non data carrying
        segments such as SYN|ACK. */
      memcpy(seg->dataptr, optdata, optlen);
    }
      ......
    left -= seglen;
    seqno += seglen;
    ptr = (void *)((u8_t *)ptr + seglen);
  }

/* Now that the data to be enqueued has been broken up into TCP
segments in the queue variable, we add them to the end of the
```

```
pcb->unsent queue. */
if (pcb->unsent == NULL) {
   useg = NULL;
}
else {
   for (useg = pcb->unsent; useg->next != NULL; useg = useg->next);
}
/* { useg is last segment on the unsent queue, NULL if list is empty } */

/* If there is room in the last pbuf on the unsent queue,
chain the first pbuf on the queue together with that. */
if (useg != NULL &&
   TCP_TCPLEN(useg) != 0 &&
   !(TCPH_FLAGS(useg->tcphdr) & (TCP_SYN | TCP_FIN)) &&
   !(flags & (TCP_SYN | TCP_FIN)) &&
    /* !(flags & (TCP_SYN | TCP_FIN))保证了 queue 中第一个 segment：
   1）不包含 option，只有 SYN 被掩上时，才会发送 option
   2）不包含 flag，RST 由 ip_output 直接发送，ACK 由 tcp_output 掩上，PSH 在下面掩上
   3）包含数据，既不包含 option 又不包含 flag，则一定包含数据
   所以 tcp header 大小为 TCP_HLEN，且该 segment 只包含数据 */
   /* fit within max seg size */
   useg->len + queue->len <= pcb->mss) {
   /* Remove TCP header from first segment of our to-be-queued list */
   pbuf_header(queue->p, -TCP_HLEN);
   pbuf_cat(useg->p, queue->p);
   useg->len += queue->len;
   useg->next = queue->next;
   ......
   if (seg == queue) {
      seg = NULL;
   }
   memp_free(MEMP_TCP_SEG, queue);
}
else {
   /* empty list */
   if (useg == NULL) {
      /* initialize list with this segment */
      pcb->unsent = queue;
   }
   /* enqueue segment */
   else {
      useg->next = queue;
   }
}
```

```
  if ((flags & TCP_SYN) || (flags & TCP_FIN)) {
    ++len;
  }
  pcb->snd_lbb += len; /*  更新下一个 segment 的 sequence number */
  pcb->snd_buf -= len; /*  更新 buffer 的大小  */

  /* update number of segments on the queues */ /*  更新 buffer 中 segment 的个数  */
  pcb->snd_queuelen = queuelen;
      ......

  /* Set the PSH flag in the last segment that we enqueued, but only
  if the segment has data (indicated by seglen > 0). */
  if (seg != NULL && seglen > 0 && seg->tcphdr != NULL) {
    TCPH_SET_FLAG(seg->tcphdr, TCP_PSH);
  }

  return ERR_OK;
memerr:
  TCP_STATS_INC(tcp.memerr);

  if (queue != NULL) {
    tcp_segs_free(queue);
  }
  if (pcb->snd_queuelen != 0) {
    LWIP_ASSERT("tcp_enqueue: valid queue length", pcb->unacked != NULL ||
      pcb->unsent != NULL);
  }
  ......
  return ERR_MEM;
}
```

12、ACK 的发送

由《卷 1：协议》可知，ack 会延时一段时间发送，如果在这期间有数据需要发送，则上述 ack 会随着该数据一起发送。下面我们来看看 lwip 是怎么处理 ack 的发送的。

12.1 发送的地方

- tcp_listen_input，在 LISTEN 状态收到 SYN 后，通过 tcp_enqueue 发送 SYN、ACK，并转到 SYN_RCVD 状态
- tcp_rst，通过 ip_output 直接发送 RST、ACK
- tcp_output，下面详细介绍

12.2 tcp_out 中的 ACK 发送

```
err_t tcp_output(struct tcp_pcb *pcb)
{
```

```
    ......
/* 1) If the TF_ACK_NOW flag is set and no data will be sent (either
 * because the ->unsent queue is empty or because the window does
 * not allow it), construct an empty ACK segment and send it.
 *
 *2) If data is to be sent, we will just piggyback the ACK (see below).    */
if (pcb->flags & TF_ACK_NOW &&
    (seg == NULL ||
      ntohl(seg->tcphdr->seqno) - pcb->lastack + seg->len > wnd)) {
   p = pbuf_alloc(PBUF_IP, TCP_HLEN, PBUF_RAM);
    ......
   /* remove ACK flags from the PCB, as we send an empty ACK now */
   pcb->flags &= ~(TF_ACK_DELAY | TF_ACK_NOW);

   tcphdr = p->payload;
   tcphdr->src = htons(pcb->local_port);
   tcphdr->dest = htons(pcb->remote_port);
   tcphdr->seqno = htonl(pcb->snd_nxt);
   tcphdr->ackno = htonl(pcb->rcv_nxt);
   TCPH_FLAGS_SET(tcphdr, TCP_ACK);
   tcphdr->wnd = htons(pcb->rcv_wnd);
   tcphdr->urgp = 0;
   TCPH_HDRLEN_SET(tcphdr, 5);
   ......
   ip_output(p, &(pcb->local_ip), &(pcb->remote_ip), pcb->ttl, pcb->tos,
         IP_PROTO_TCP);
   pbuf_free(p);

   return ERR_OK;
}
    ......
/* data available and window allows it to be sent? */
while (seg != NULL &&
ntohl(seg->tcphdr->seqno) - pcb->lastack + seg->len <= wnd) {
    ......
   pcb->unsent = seg->next;

   if (pcb->state != SYN_SENT) {
      TCPH_SET_FLAG(seg->tcphdr, TCP_ACK);
      pcb->flags &= ~(TF_ACK_DELAY | TF_ACK_NOW);
   }
   tcp_output_segment(seg, pcb);
   ......
}
```

```
    return ERR_OK;
}
```

1) TF_ACK_NOW 被掩上且发送数据的条件不满足，则立刻发送 ACK（不带数据），最后清除 TF_ACK_DELAY 与 TF_ACK_NOW。

2）当满足发送数据的条件时，只要 pcb 不处于 SYN_SENT，则在发送数据的同时发送 ACK，最后清除 TF_ACK_DELAY 与 TF_ACK_NOW。

12.3 宏 tcp_ack_now

```
#define tcp_ack_now(pcb) (pcb)->flags |= TF_ACK_NOW; \
                         tcp_output(pcb)
```

立刻发送 ACK：如果当前满足发送数据的条件则随数据发送 ACK，否则单独发送 ACK。对于希望立刻发送 ACK 的模块，应使用该宏来发送 ACK。

12.4 宏 tcp_ack

```
#define tcp_ack(pcb)        if((pcb)->flags & TF_ACK_DELAY) { \
                                (pcb)->flags &= ~TF_ACK_DELAY; \
                                (pcb)->flags |= TF_ACK_NOW; \
                                tcp_output(pcb); \
                            } else { \
                                (pcb)->flags |= TF_ACK_DELAY; \
                            }
```

延时发送 ACK：如果不存在延时发送的 ACK，则延时当前待发送的 ACK，否则复位延时发送的标识立刻发送 ACK。lwip 开启了一个 250ms 的定时器，该定时器扫描所有处于非 listion、非 timewait 状态的 pcb，一旦发现延时发送的标识被掩上了，则引用宏 tcp_ack_now 立刻发送 ACK。

13、发送 segment

```
static void tcp_output_segment(struct tcp_seg *seg, struct tcp_pcb *pcb)
{
  u16_t len;
  struct netif *netif;
     ......
  /* The TCP header has already been constructed, but the ackno and
   wnd fields remain. */
  seg->tcphdr->ackno = htonl(pcb->rcv_nxt);

  /* silly window avoidance */
  if (pcb->rcv_wnd < pcb->mss) {
    seg->tcphdr->wnd = 0;
  } else {
    /* advertise our receive window size in this TCP segment */
    seg->tcphdr->wnd = htons(pcb->rcv_wnd);
  }

  /* If we don't have a local IP address, we get one by
```

```
      calling ip_route(). */
  if (ip_addr_isany(&(pcb->local_ip))) {
    netif = ip_route(&(pcb->remote_ip));
    if (netif == NULL) {
      return;
    }
    ip_addr_set(&(pcb->local_ip), &(netif->ip_addr));
  }

  pcb->rtime = 0;
  if (pcb->rttest == 0) {
    pcb->rttest = tcp_ticks;
    pcb->rtseq = ntohl(seg->tcphdr->seqno);

    LWIP_DEBUGF(TCP_RTO_DEBUG, ("tcp_output_segment: rtseq %"U32_F"\n", pcb->rtseq));
  }
     ......
  len = (u16_t)((u8_t *)seg->tcphdr - (u8_t *)seg->p->payload);
  seg->p->len -= len;
  seg->p->tot_len -= len;
  seg->p->payload = seg->tcphdr;

  seg->tcphdr->chksum = 0;
#if CHECKSUM_GEN_TCP
  seg->tcphdr->chksum = inet_chksum_pseudo(seg->p,
                &(pcb->local_ip),
                &(pcb->remote_ip),
                IP_PROTO_TCP, seg->p->tot_len);
#endif
  TCP_STATS_INC(tcp.xmit);

  ip_output(seg->p, &(pcb->local_ip), &(pcb->remote_ip), pcb->ttl, pcb->tos,
    IP_PROTO_TCP);
}
```

填写待发送 segment 的 ack sequnce number 与通告窗口，启动 rtt 估计，最后调用 ip_output 发送报文。不过无法理解的是上面红色字体的那段，在 tcp_enqueue 中 p->payload 被赋给了 seg->tcphdr，什么时候发生了改动呢？

14、糊涂窗口综合症的避免

- 由 tcp_output_segment()可知，当 lwip 发现 receive buffer 大小小于 mss 时，即通告窗口为 0。这样避免了在 receive buffer 较小的时候，收到对方发送过来的较小的 segment，这避免了触发 sws。
- 由 tcp_enqueue()可知，lwip 总是尽量按照 mss 分割待发送数据，这样可以避免发送小的 segment。当对方的 receive buffer 较小，即对方通告的窗口较小时，由于 lwip 的 send buffer 中的 segment 较大（尽量被分割成 mss 大小），大于对方通告的窗口从而不满足发送条件。于是此时 lwip 不发送 segment，

而是等待对方通告一个足够大的窗口。

可以说 lwip 简化了分割待发送数据的流程和 receive buffer 的通告流程，缺点是降低了传输速度，优点是避免了 sws。

15、tcp_output 与重传

15.1 发送 segment 有关的两个队列：

- pcb->unsent，待发送 segment，tcp_enqueue()将待发送数据分割乘 segment 并放入该队列
- pcb->unacked，已发送待确认 segment，tcp_output()从该队列中取 segment，如果满足发送条件则调用 tcp_output_segment()发送该 segment。

15.2 这两个队列的其他用户

- tcp_rexmit_rto，超时重传时将 pcb->unacked 中所有的 segment 移动到到 pcb->unsent 的前面，最后调用 tcp_output。
- tcp_rexmit，快速重传时将 pcb->unacked 第一个 segment 移动到到 pcb->unsent 的前面，最后调用 tcp_output。

15.3 tcp_output 的流程

```
err_t tcp_output(struct tcp_pcb *pcb)
{
  struct pbuf *p;
  struct tcp_hdr *tcphdr;
  struct tcp_seg *seg, *useg;
  u32_t wnd;
#if TCP_CWND_DEBUG
  s16_t i = 0;
#endif /* TCP_CWND_DEBUG */

  /* First, check if we are invoked by the TCP input processing
     code. If so, we do not output anything. Instead, we rely on the
     input processing code to call us when input processing is done
     with. */
  if (tcp_input_pcb == pcb) {
    return ERR_OK;
  }

  wnd = LWIP_MIN(pcb->snd_wnd, pcb->cwnd); /*  当前窗口取拥塞窗口与对方的通告窗口的最小值  */

  seg = pcb->unsent;

  /* useg should point to last segment on unacked queue */
  useg = pcb->unacked;
  if (useg != NULL) {
    for (; useg->next != NULL; useg = useg->next);
```

```
  }
   ......
  if (pcb->flags & TF_ACK_NOW &&
      (seg == NULL ||
       ntohl(seg->tcphdr->seqno) - pcb->lastack + seg->len > wnd)) {
     /*  前面已经叙述过：立刻发送 ACK */
  }
      ......
/*  在窗口允许的前提下，尽可能地发送 segment */
  while (seg != NULL &&
  ntohl(seg->tcphdr->seqno) - pcb->lastack + seg->len <= wnd) {
     pcb->unsent = seg->next;

     if (pcb->state != SYN_SENT) { /*  前面已经叙述过：ACK 与数据一起发送  */
        TCPH_SET_FLAG(seg->tcphdr, TCP_ACK);
        pcb->flags &= ~(TF_ACK_DELAY | TF_ACK_NOW);
     }

     tcp_output_segment(seg, pcb); /*  发送 segment */
     pcb->snd_nxt = ntohl(seg->tcphdr->seqno) + TCP_TCPLEN(seg); /* next seqno to be sent */
     if (TCP_SEQ_LT(pcb->snd_max, pcb->snd_nxt)) {
        pcb->snd_max = pcb->snd_nxt; /* Highest seqno sent. */
     }
     /* put segment on unacknowledged list if length > 0 */
     if (TCP_TCPLEN(seg) > 0) {
        seg->next = NULL;
        /* unacked list is empty? */
        if (pcb->unacked == NULL) {
           pcb->unacked = seg;
           useg = seg;
        /* unacked list is not empty? */
        } else {
           /* In the case of fast retransmit, the packet should not go to the tail
            * of the unacked queue, but rather at the head. We need to check for
            * this case. -STJ Jul 27, 2004 */
           if (TCP_SEQ_LT(ntohl(seg->tcphdr->seqno), ntohl(useg->tcphdr->seqno))){
              /* add segment to head of unacked list */
              seg->next = pcb->unacked;
              pcb->unacked = seg;
           } else {
              /* add segment to tail of unacked list */
              useg->next = seg;
              useg = useg->next;
           }
```

```
        }
      /* do not queue empty segments on the unacked list */
      } else {
        tcp_seg_free(seg);
      }
      seg = pcb->unsent;
    }
    return ERR_OK;
}
```

16、内核函数 do_connect

```
static void do_connect(struct api_msg_msg *msg)
{
    if (msg->conn->pcb.tcp == NULL) {
      switch (msg->conn->type) {
       ......
      case NETCONN_TCP:
        msg->conn->pcb.tcp = tcp_new();
        if (msg->conn->pcb.tcp == NULL) {
              msg->conn->err = ERR_MEM;
              sys_mbox_post(msg->conn->mbox, NULL);
              return;
        }
      default:
        break;
      }
    }
    switch (msg->conn->type) {
       ........
    case NETCONN_TCP:
      setup_tcp(msg->conn);
      tcp_connect(msg->conn->pcb.tcp, msg->msg.bc.ipaddr, msg->msg.bc.port,
      do_connected);
    default:
      break;
    }
}
```

16.1 tcp_connect

```
err_t tcp_connect(struct tcp_pcb *pcb, struct ip_addr *ipaddr, u16_t port,
        err_t (* connected)(void *arg, struct tcp_pcb *tpcb, err_t err))
{
    /*  当链接请求被服务器接收后 connected 被调用，内核用它来通知用户任务  */
    u32_t optdata;
```

```
  err_t ret;
  u32_t iss;

  LWIP_DEBUGF(TCP_DEBUG, ("tcp_connect to port %"U16_F"\n", port));
  if (ipaddr != NULL) {
    pcb->remote_ip = *ipaddr;
  } else {
    return ERR_VAL;
  }
  pcb->remote_port = port;
  if (pcb->local_port == 0) {
    pcb->local_port = tcp_new_port();
  }
  iss = tcp_next_iss();
  pcb->rcv_nxt = 0;
  pcb->snd_nxt = iss;
  pcb->lastack = iss - 1;
  pcb->snd_lbb = iss - 1;
  pcb->rcv_wnd = TCP_WND; /*自己的通告窗口 */
  pcb->snd_wnd = TCP_WND; /*对方的通告窗口 */
  pcb->mss = TCP_MSS;
  pcb->cwnd = 1;
  pcb->ssthresh = pcb->mss * 10;
  pcb->state = SYN_SENT; /* 被置为 SYN_SENT 状态 */
#if LWIP_CALLBACK_API
  pcb->connected = connected; /* 注册回调函数 */
#endif /* LWIP_CALLBACK_API */
  TCP_REG(&tcp_active_pcbs, pcb); /* 将 pcb 放入 tcp_active_pcbs */

  snmp_inc_tcpactiveopens();

  /* Build an MSS option */
  optdata = htonl(((u32_t)2 << 24) |
      ((u32_t)4 << 16) |
      (((u32_t)pcb->mss / 256) << 8) |
      (pcb->mss & 255));

  ret = tcp_enqueue(pcb, NULL, 0, TCP_SYN, 0, (u8_t *)&optdata, 4); /* 发送 SYN，并通告 mss */

  if (ret == ERR_OK) {
    tcp_output(pcb);
  }
  return ret;
}
```

16.2 do_connected

```
static err_t do_connected(void *arg, struct tcp_pcb *pcb, err_t err)
{
  struct netconn *conn;
  conn = arg;
  if (conn == NULL) {
    return ERR_VAL;
  }

  conn->err = err;
  if (conn->type == NETCONN_TCP && err == ERR_OK) {
    setup_tcp(conn); /*  注册链接相关的 event handler */
  }
  sys_mbox_post(conn->mbox, NULL); /*  通知用户任务继续执行  */
  return ERR_OK;
}
```

17、系统调用 netconn_accept

```
struct netconn *netconn_accept(struct netconn *conn)
{
  struct netconn *newconn;
      ......
  sys_mbox_fetch(conn->acceptmbox, (void *)&newconn);
  /* Register event with callback */
  if (conn->callback)
      (*conn->callback)(conn, NETCONN_EVT_RCVMINUS, 0);

  return newconn;
}
```

17.1 阻塞在 accept mailbox 上，等待内核完成三次握手后的通知。内核接受客户端的链接，并完成三次握手后，用 accept_function 通知用户任务，链接已经建立。

17.2 accept_function

```
static err_t accept_function(void *arg, struct tcp_pcb *newpcb, err_t err)
{
/*1）arg 指向处于 listen 状态的 connection，do_listen()使该 connection 处于 listen 状态
2）newpcb 指向内核分配的完成三次握手的 pcb，而不是处于 listen 状态的 connection 所关联的 pcb */
  sys_mbox_t mbox;
  struct netconn *newconn;
  struct netconn *conn;
      ......
  conn = (struct netconn *)arg;
  mbox = conn->acceptmbox;
```

```
/* 分配并初始化一个 connection，并关联上 newpcb
    该 connnection 被返回给服务器任务，用于数据收发 */
  newconn = memp_malloc(MEMP_NETCONN);
      ......
  newconn->recvmbox = sys_mbox_new();
      ......
  newconn->mbox = sys_mbox_new();
      ......
  newconn->sem = sys_sem_new(0);
      ......
  /* Allocations were OK, setup the PCB etc */
  newconn->type = NETCONN_TCP;
  newconn->pcb.tcp = newpcb;
  setup_tcp(newconn);
  newconn->acceptmbox = SYS_MBOX_NULL;
  newconn->err = err;
  /* Register event with callback */
  if (conn->callback)
  {
    (*conn->callback)(conn, NETCONN_EVT_RCVPLUS, 0);
  }
  /* We have to set the callback here even though
   * the new socket is unknown. Mark the socket as -1. */
  newconn->callback = conn->callback;
  newconn->socket = -1;
  newconn->recv_avail = 0;
  sys_mbox_post(mbox, newconn);/* 通知用户任务链接已经建立，用户代码从阻塞中返回继续执行 */
  return ERR_OK;
}
```

18、三次握手

18.1 客户端发送 SYN

1）服务器调用 netconn_listen、netconn_accept，内核将服务器准备好的 connection、pcb 挂在 tcp_listen_pcbs 上，此时该 pcb 处于 LISTEN 状态。

2）客户端调用 netconn_connect，内核向对方发送 SYN，相应的 pcb 处于 SYN_SENT 状态。

18.2 服务器响应 SYN

```
static err_t tcp_listen_input(struct tcp_pcb_listen *pcb)
{
  struct tcp_pcb *npcb;
  u32_t optdata;
  /* In the LISTEN state, we check for incoming SYN segments,
     creates a new PCB, and responds with a SYN|ACK. */
  if (flags & TCP_ACK) {
```

```
      /* For incoming segments with the ACK flag set, respond with a RST. */
       ......
    } else if (flags & TCP_SYN) {
      npcb = tcp_alloc(pcb->prio);
       ......
      /* Set up the new PCB. */ /*  分配一个新 pcb，并用刚收到的 segment 来初始化相应参数  */
      ip_addr_set(&(npcb->local_ip), &(iphdr->dest));
      npcb->local_port = pcb->local_port;
      ip_addr_set(&(npcb->remote_ip), &(iphdr->src));
      npcb->remote_port = tcphdr->src;
      npcb->state = SYN_RCVD; /*  将新 pcb 置为 SYN_RCVD 状态  */
      npcb->rcv_nxt = seqno + 1;
      npcb->snd_wnd = tcphdr->wnd;
      npcb->ssthresh = npcb->snd_wnd;
      npcb->snd_wl1 = seqno - 1;/* initialise to seqno-1 to force window update */
      npcb->callback_arg = pcb->callback_arg;
#if LWIP_CALLBACK_API
      npcb->accept = pcb->accept;   /*  将服务器注册的回调函数(accept_function)拷贝过来  */
#endif /* LWIP_CALLBACK_API */
      /* inherit socket options */
      npcb->so_options = pcb->so_options & \
      (SOF_DEBUG|SOF_DONTROUTE|SOF_KEEPALIVE|SOF_OOBINLINE|SOF_LINGER);
      /* Register the new PCB so that we can begin receiving segments for it. */
      TCP_REG(&tcp_active_pcbs, npcb);   /*  将新 pcb 插入到 tcp_active_pcbs */

      /* Parse any options in the SYN. */
      tcp_parseopt(npcb);   /*  得到客户端通告的 mss */

      snmp_inc_tcppassiveopens();

      /* Build an MSS option. */
      optdata = htonl(((u32_t)2 << 24) |
          ((u32_t)4 << 16) |
          (((u32_t)npcb->mss / 256) << 8) |
          (npcb->mss & 255));
      /* Send a SYN|ACK together with the MSS option. */   /*  发送 SYN、ACK 并通告 mss */
      tcp_enqueue(npcb, NULL, 0, TCP_SYN | TCP_ACK, 0, (u8_t *)&optdata, 4);
      return tcp_output(npcb);
    }
    return ERR_OK;
}
```

18.3 客户端响应 SYN、ACK

```
static err_t tcp_process(struct tcp_pcb *pcb)
```

```
{
   switch (pcb->state) {
   case SYN_SENT:
      ......
      /* received SYN ACK with expected sequence number? */
      if ((flags & TCP_ACK) && (flags & TCP_SYN)
          && ackno == ntohl(pcb->unacked->tcphdr->seqno) + 1) {
        pcb->snd_buf++; /* 见 tcp_connect，发送 SYN 使得 snd_buf 减少一个字节 */
        pcb->rcv_nxt = seqno + 1;
        pcb->lastack = ackno;
        pcb->snd_wnd = tcphdr->wnd; /* 更新服务器的通过窗口 */
        pcb->snd_wl1 = seqno - 1; /* initialise to seqno - 1 to force window update */
        pcb->state = ESTABLISHED; /* pcb 被置为 ESTABLISHED 状态 */
        pcb->cwnd = pcb->mss; /* 更新客户端的拥塞窗口 */
        --pcb->snd_queuelen;
        ......
        rseg = pcb->unacked;
        pcb->unacked = rseg->next;
        tcp_seg_free(rseg);

        /* Parse any options in the SYNACK. */
        tcp_parseopt(pcb); /* 更新服务器通告的 mss */

        /* Call the user specified function to call when sucessfully connected. */
        TCP_EVENT_CONNECTED(pcb, ERR_OK, err); /* 调用客户端注册的回调函数(do_connected) */
        tcp_ack(pcb); /* 发送 ACK，以响应服务器的 SYN */ /* 客户端的三次握手已经完成 */
      }
      /* received ACK? possibly a half-open connection */
      else if (flags & TCP_ACK) {
        /* send a RST to bring the other side in a non-synchronized state. */
        tcp_rst(ackno, seqno + tcplen, &(iphdr->dest), &(iphdr->src),
          tcphdr->dest, tcphdr->src);
      }
      break;
   }
}
```

18.4 服务器响应 ACK

```
static err_t tcp_process(struct tcp_pcb *pcb)
{
   switch (pcb->state) {
      ......
      case SYN_RCVD:
        if (flags & TCP_ACK &&
```

```
        !(flags & TCP_RST)) {
      /* expected ACK number? */
      if (TCP_SEQ_BETWEEN(ackno, pcb->lastack+1, pcb->snd_nxt)) {
        pcb->state = ESTABLISHED; /*  将 pcb 状态置为 ESTABLISHED */
       ......
        /* Call the accept function. */
        TCP_EVENT_ACCEPT(pcb, ERR_OK, err); /*  调用服务器注册的回调函数 accept_function */
                                            /*  服务器的三次握手已经完成  */
       ......
        /* If there was any data contained within this ACK,
         * we'd better pass it on to the application as well. */
        tcp_receive(pcb);
        pcb->cwnd = pcb->mss;
      }
      /* incorrect ACK number */
      else {
        /* send RST */
        tcp_rst(ackno, seqno + tcplen, &(iphdr->dest), &(iphdr->src),
          tcphdr->dest, tcphdr->src);
      }
    }
    break;
  }
}
```

19、Nagle 算法

```
static void do_write(struct api_msg_msg *msg)
{
  if (msg->conn->pcb.tcp != NULL) {
    switch (msg->conn->type) {
     ......
    case NETCONN_TCP:
      err = tcp_write(msg->conn->pcb.tcp, msg->msg.w.dataptr, msg->msg.w.len, msg->msg.w.copy);
      /* This is the Nagle algorithm: inhibit the sending of new TCP
        segments when new outgoing data arrives from the user if any
        previously transmitted data on the connection remains
        unacknowledged. */
      if(err == ERR_OK && (msg->conn->pcb.tcp->unacked == NULL ||
        (msg->conn->pcb.tcp->flags & TF_NODELAY) ||
        (msg->conn->pcb.tcp->snd_queuelen) > 1)) {
          tcp_output(msg->conn->pcb.tcp);
      }
     ......
    default:
```

```
        break;
      }
    }
    sys_mbox_post(msg->conn->mbox, NULL);
}
```

lwip1.2.0 的 nagle 算法的实现显然有问题，当(msg->conn->pcb.tcp->snd_queuelen) > 1 时，即使使能了 Nagle 算法，还是会发送 segment。

1）假设当前只存在一个未确认的 segment，unacked 队列中有且只有一个 segment，而 unsent 没有 segment

2）此时用户希望发送一字符，那么 unsent 会被插入一个长度为 1 的 segment

3）此时 tcp->snd_queuelen 为 2，则 unsent 中的 segment 被发送，且被移动到了 unacked 中，而 unsent 为空

4）因为在大网中，rtt 较大。当用户希望发送第二个字符，则 unsent 会被插入一个长度为 1 的 segment

5）此时此时 tcp->snd_queuelen 为 3，则 unsent 中的 segment 被发送，且被移动到了 unacked 中，unacked 中有三个 segment，而 unsent 为空

这显然与 Nagle 算法违背，在看了最新的 lwip1.3.0 的代码后，发现这个 bug 已经被解决：

```
#define tcp_do_output_nagle(tpcb) ((((tpcb)->unacked == NULL) || \
                                    ((tpcb)->flags & TF_NODELAY) || \
                                    (((tpcb)->unsent != NULL) && ((tpcb)->unsent->next != NULL))) ? \
                                      1 : 0)
```

当存在未确认的 segment，且 Nagle 算法被使能，pcb->unsent 中存在两个以上的 segment 时才发送新的 segment。可以这样解释：

1）2）同前

3）不满足发送条件，不发送新的 segment，此时 unacked、unsent 中均只有一个 segment

4）因为在大网中，rtt 较大。当用户希望发送第二个字符，由 tcp_enqueue 可知此时第二个字符会被追加到 unsent 中的第一个 segment，而 unsent 中还是只有一个 segment

5）不满足发送条件，不发送新的 segment，此时 unacked、unsent 中均只有一个 segment

此后 4）5）过程反复被重复，直到：

- 新的 ack 到达
- unsent 中的第一个 segment 长度大于 mss，从而导致 unsent 中有了两个 segment

这就符合了 Nagle 算法的要求。不过要注意的是，output 会在多个地方被调用。

20、用户发送数据

20.1 系统调用 netconn_write

```
err_t netconn_write(struct netconn *conn, void *dataptr, u16_t size, u8_t copy)
{
      ......
   if ((msg = memp_malloc(MEMP_API_MSG)) == NULL) {
      return (conn->err = ERR_MEM);
   }
   msg->type = API_MSG_WRITE;
   msg->msg.conn = conn;

   conn->state = NETCONN_WRITE;
   while (conn->err == ERR_OK && size > 0) {
      msg->msg.msg.w.dataptr = dataptr;
```

```
    msg->msg.msg.w.copy = copy;

    if (conn->type == NETCONN_TCP) {
      if (tcp_sndbuf(conn->pcb.tcp) == 0) {
            sys_sem_wait(conn->sem);
            if (conn->err != ERR_OK) {
              goto ret;
            }
      }
      if (size > tcp_sndbuf(conn->pcb.tcp)) {
            /* We cannot send more than one send buffer's worth of data at a time. */
            len = tcp_sndbuf(conn->pcb.tcp);
      } else {
            len = size;
      }
    } else {
      len = size;
    }

    LWIP_DEBUGF(API_LIB_DEBUG, ("netconn_write: writing %d bytes (%d)\n", len, copy));
    msg->msg.msg.w.len = len;
    api_msg_post(msg);
    sys_mbox_fetch(conn->mbox, NULL);
    if (conn->err == ERR_OK) {
      dataptr = (void *)((u8_t *)dataptr + len);
      size -= len;
    } else if (conn->err == ERR_MEM) {
      conn->err = ERR_OK;
      sys_sem_wait(conn->sem);
    } else {
      goto ret;
    }
  }
ret:
  memp_free(MEMP_API_MSG, msg);
  conn->state = NETCONN_NONE;

  return conn->err;
}
```

- 用户调用 netconn_write 发送数据，netconn_write 每次发送对方内核通告的 receive buffer 大小的数据，直至所有数据发送完毕。
- 当对方内核的 receive buffer 已满，或者自己的内核返回 ERR_MEM，即 send buffer 无可用空间错误时，用户任务 pend 在事先分配好的信号量 conn->sem 上。

- 在发送数据之前将 conn->state 置为 NETCONN_WRITE，完成发送后将该值置为 NETCONN_NONE。
- 内核收到 netconn_write()请求发送数据的消息后，do_write()将调用 tcp_enqueue 处理待发送数据，并返回处理结果，在返回前并且在条件允许的前提下会尝试发送数据（tcp_output）。

20.2 对 conn->sem 的操作

共有四个地方引用了 conn->sem，分别是系统调用 netconn_write()，内核事件处理函数 sent_tcp()与内核事件处理函数 poll_tcp()。其中 sent_tcp()、err_tcp()与 poll_tcp()是 signal conn->sem。

20.2.1 内核事件处理函数 sent_tcp()

当有已发送数据被对方确认时，即收到新的且更大的 ack 时，在内核函数 tcp_input()中事件处理函数 sent_tcp()被调用。由下可见，不管是否存在用户 pend conn->sem，只要有新的且更大的 ack 到达，内核就 signal conn->sem，这是有问题的。假设 sender 内核中有待确认数据，而用户此时没有发送数据，当新的且更大的 ack 到达时，conn->sem 增大；随后用户发送一块很大的数据，很快就将内核的 send buffer 填满，从而导致用户任务 pend conn->sem；这时因为 conn->sem 大于 0，虽然 send buffer 是满的，pend 立刻返回，conn->sem 减 1；netconn_write()又给内核发消息期望发送数据，但同样因为 sender buffer 是满的导致 pend conn->sem; 这个过程一直重复，直至 conn->等于 0，netconn_write() pend 在 conn->sem 上，或者在 netconn_write()不断尝试过程中，send buffer 有了一定的空间。由上可见，虽然这个过程不会带来问题，却大大影响了效率。

```
static err_t sent_tcp(void *arg, struct tcp_pcb *pcb, u16_t len)
{
  struct netconn *conn;
  conn = arg;
  if (conn != NULL && conn->sem != SYS_SEM_NULL) {
    sys_sem_signal(conn->sem);
  }

  if (conn && conn->callback) /*conn->callback 始终为空*/
      if (tcp_sndbuf(conn->pcb.tcp) > TCP_SNDLOWAT)
          (*conn->callback)(conn, NETCONN_EVT_SENDPLUS, len);
    return ERR_OK;
}
```

20.2.2 内核事件处理函数 err_tcp()

```
static void err_tcp(void *arg, err_t err)
{
  struct netconn *conn;
  conn = arg;
  conn->pcb.tcp = NULL;
  conn->err = err;
  if (conn->recvmbox != SYS_MBOX_NULL) {
    /* Register event with callback */
    if (conn->callback)
      (*conn->callback)(conn, NETCONN_EVT_RCVPLUS, 0);
    sys_mbox_post(conn->recvmbox, NULL);
  }
```

```
  if (conn->mbox != SYS_MBOX_NULL) {
    sys_mbox_post(conn->mbox, NULL);
  }
  if (conn->acceptmbox != SYS_MBOX_NULL) {
     /* Register event with callback */
    if (conn->callback)
      (*conn->callback)(conn, NETCONN_EVT_RCVPLUS, 0);
    sys_mbox_post(conn->acceptmbox, NULL);
  }
  if (conn->sem != SYS_SEM_NULL) {
    sys_sem_signal(conn->sem);
  }
}
```

该事件处理函数被 tcp_abort()调用，此外在 tcp_slowtmr()中，如果需要释放 pcb 也会调用它。从它的代码中容易得知，err_tcp()的作用是在释放 pcb 时防止用户任务阻塞在某个 mail box 或 semaphore 上。

20.2.3 内核事件处理函数 poll_tcp()

```
static err_t poll_tcp(void *arg, struct tcp_pcb *pcb)
{
  struct netconn *conn;

  conn = arg;
  if (conn != NULL &&
      (conn->state == NETCONN_WRITE || conn->state == NETCONN_CLOSE) &&
      conn->sem != SYS_SEM_NULL) {
    sys_sem_signal(conn->sem);
  }
  return ERR_OK;
}
```

在没收到 ack 的前提下，tcp_slowtmr()每秒 4 个 500ms 调用一次 poll_tcp()。

- pcb->pollinterval，轮询间隔，每 4 个 500ms 轮询一次
- pcb->polltmr，轮询计数器，在 tcp_slowtmr()中递增该计数器，当等于 pcb->pollinterval 时，计数器复位并触发事件处理函数 poll_tcp()
- 当收到一个 ack 时，在 tcp_receive()中会复位 pcb->polltmr
- poll_tcp() 发现 conn 处于 NETCONN_WRITE 状态时，signal conn->sem

我没搞清楚 poll_tcp()有什么特殊的作用，似乎不要它也可以。

21、TCP 接收报文的总入口 tcp_input()

```
void tcp_input(struct pbuf *p, struct netif *inp)
{
```

```
    ......
    for(pcb = tcp_active_pcbs; pcb != NULL; pcb = pcb->next) {
     ......
    if (pcb->remote_port == tcphdr->src &&
        pcb->local_port == tcphdr->dest &&
        ip_addr_cmp(&(pcb->remote_ip), &(iphdr->src)) &&
        ip_addr_cmp(&(pcb->local_ip), &(iphdr->dest))) {
        /*  找到了匹配的 pcb */
       break;
    }
    prev = pcb;
  }
  if (pcb == NULL) {
    /* If it did not go to an active connection, we check the connections    in the TIME-WAIT state. */
    for(pcb = tcp_tw_pcbs; pcb != NULL; pcb = pcb->next) {
          /*  与数据收发无关，略过  */
        return;
      }
    }
  /* Finally, if we still did not get a match, we check all PCBs that are LISTENing for incoming connections. */
    prev = NULL;
    for(lpcb = tcp_listen_pcbs.listen_pcbs; lpcb != NULL; lpcb = lpcb->next) {
          /*  与数据收发无关，略过  */
        return;
      }
      prev = (struct tcp_pcb *)lpcb;
    }
  }
  ......
  if (pcb != NULL) {
    /* The incoming segment belongs to a connection. */
     ......
    /* Set up a tcp_seg structure. */
    inseg.next = NULL;                    (1)
    inseg.len = p->tot_len;
    inseg.dataptr = p->payload;
    inseg.p = p;
    inseg.tcphdr = tcphdr;

    recv_data = NULL;                     (2)
    recv_flags = 0;                       (3)
    tcp_input_pcb = pcb;                  (4)
    err = tcp_process(pcb);               (5)
    tcp_input_pcb = NULL;                 (6)
```

```
/* A return value of ERR_ABRT means that tcp_abort() was called
   and that the pcb has been freed. If so, we don't do anything. */
 if (err != ERR_ABRT) {
      if (recv_flags & TF_RESET) {      (7)
          /* TF_RESET means that the connection was reset by the other
             end. We then call the error callback to inform the
             application that the connection is dead before we
             deallocate the PCB. */
          TCP_EVENT_ERR(pcb->errf, pcb->callback_arg, ERR_RST);
          tcp_pcb_remove(&tcp_active_pcbs, pcb);
          memp_free(MEMP_TCP_PCB, pcb);
      } else if (recv_flags & TF_CLOSED) {       (8)
          /* The connection has been closed and we will deallocate the PCB. */
          tcp_pcb_remove(&tcp_active_pcbs, pcb);
          memp_free(MEMP_TCP_PCB, pcb);
      } else {
          err = ERR_OK;
          /* If the application has registered a "sent" function to be
             called when new send buffer space is available, we call it now. */
          if (pcb->acked > 0) {              (9)
            TCP_EVENT_SENT(pcb, pcb->acked, err);
          }
         if (recv_data != NULL) {            (10)
            /* Notify application that data has been received. */
            TCP_EVENT_RECV(pcb, recv_data, ERR_OK, err);
          }
          /* If a FIN segment was received, we call the callback
             function with a NULL buffer to indicate EOF. */
          if (recv_flags & TF_GOT_FIN) {    (11)
            TCP_EVENT_RECV(pcb, NULL, ERR_OK, err);
          }
          /* If there were no errors, we try to send something out. */
          if (err == ERR_OK) {              (12)
            tcp_output(pcb);
          }
      }
}
/* give up our reference to inseg.p */
if (inseg.p != NULL)
{
  pbuf_free(inseg.p);   /*由底层协议分配的 pbuf 在该层被处理，随后释放 pbuf */
  inseg.p = NULL;
}
.....
```

```
  } else {
    /* If no matching PCB was found, send a TCP RST (reset) to the sender. */
    /*  与数据收发无关，略过  */
  }
  ......
}
```

21.1 三个全局 pcb 链表

- tcp_listen_pcbs，处于 listen 状态的 pcb 在此链表中
- tcp_tw_pcbs，处于 time_wait 状态的 pcb 在此链表中
- tcp_active_pcbs，处于其它状态的 pcb 均在此链表中

底层上传递上来的 tcp 报文（由 pbuf 链表承载）首先经过 tcp_active_pcbs，如果找到相应的 pcb，则交由 tcp_process()处理。否则依次在 tcp_tw_pcbs、tcp_listen_pcbs 中查找 pcb，如果存在相应的 pcb，则交由相应的处理函数处理。

21.2 传递给 tcp_process()处理的参数

为了提高效率，lwip 定义了一些全局变量：

- static struct tcp_seg inseg，底层传递上来的 tcp 报文
- static struct tcp_hdr *tcphdr，指向 tcp header 的指针
- static struct ip_hdr *iphdr，指向 ip header 的指针
- static u32_t seqno, ackno，tcp 报文的 seqno、ackno
- static u8_t flags，tcp 报文的 flags
- static u16_t tcplen，tcp 报文的长度

tcp_input()在调用 tcp_process()之前，对上述全局变量赋值，这样可以提高参数传递、引用的效率。

21.3 tcp_process()的处理结果

（2）（3）返回了 tcp_process()的处理结果：

- recv_data，指向 pbuf 链表的指针。tcp_process()首先处理收到的有效报文，然后会查看失序报文中是否存在 seqno 符合接收条件的报文，并将这些报文用 pbuf 链表链接起来，用 recv_data 指向它们。同时按照 pbuf 链表中的报文总长度调整自己的 receive wnd 的大小。
- recv_flags，记录 tcp_process()已处理的到达报文的 flags。

21.4 tcp_input()发送报文的时机

pcb 的状态变换，或收到数据时均需要向对方发送 tcp 报文。tcp_input()在处理底层上报的 tcp 报文的过程中不向外发送报文，当所有的处理均完成后才发送。tcp_input()依靠全局变量 tcp_input_pcb 来完成这一逻辑。我们知道向外发送报文必须经过两个步骤，调用 tcp_enqueue()将报文放入待发送队列，然后调用 tcp_output()发送报文。tcp_output()判断当前处理的 pcb 是否与 tcp_input_pcb 相同，如果相同则立刻返回，否则才尝试发送报文。(4)将待处理的 pcb 赋值给 tcp_input_pcb，然后(5)调用 tcp_process()处理到达的报文，在处理过程中产生的所有往外的 tcp 报文均不会被发送出去，而是在待发送队列中。处理完后，(6)再将 tcp_input_pcb 清空，最后(12)调用 tcp_output()尝试发送 tcp 报文。

21.5 tcp 的关闭

- 处于 time_wait 状态并等上一段时间后关闭。lwip 在 tcp_slowtmr()检查 pcb 处于 time_wait 状态的时长，当大于 2MSL 时关闭 tcp 连接，并释放 pcb。

- 处于 last_ack 状态，收到 ack 后关闭。tcp_input() (8)表示，在 tcp_process()中处于 last_ack 状态的 pcb 收到了期望到达的 ack，此时关闭 tcp 连接，并释放 pcb。
- 收到 reset，tcp 应立即关闭。tcp_input() (7)表示，在 tcp_process()中 pcb 收到了 reset flag，此时关闭 tcp 连接，并释放 pcb。

21.6 报文已发送事件

(9)表示在 tcp_process()中 pcb 收到了新的 ack，并有已发送报文得到了确认，同时 send buffer 增大，对方的通告窗口（即对方的 receive buffer）可能会增大。事件处理函数 sent_tcp()将会被调用，如果用户任务因为发送数据时 send buffer 过小而阻塞，则此时用户任务会被唤醒而继续发送数据。如果对方的通告窗口增大，且用户任务在发送数据时因为对方的通告窗口为 0 而阻塞，则此时用户任务会被唤醒而继续发送数据。

21.7 新数据达到事件

(10)表示在 tcp_process()中 pcb 收到了新的数据，同时减小自己的 receive buffer。此时事件处理函数 recv_tcp()被触发，通知用户任务有新的数据到达。用户接收数据的过程随后给出。

21.8 处理对方的关闭链接的请求

（11）表示在 tcp_process()中 pcb 收到了对方关闭链接的请求。此时事件处理函数 recv_tcp()被触发。

22、tcp 状态机处理函数 tcp_process()

除了 listen、time_wait 状态，pcb 在 tcp_process()中实现了 tcp 状态机。当三次握手完成处于 establish 状态后，到达报文的处理由 tcp_receive()处理。

23、到达报文处理函数 tcp_receive()

```
static u8_t tcp_receive(struct tcp_pcb *pcb)
{
  ......
  if (flags & TCP_ACK) {
    right_wnd_edge = pcb->snd_wnd + pcb->snd_wl1;
    /* Update window. */ /*更新对方的通告窗口，即对方的 receive buffer 大小，忽略*/
    if (pcb->lastack == ackno) {
    /*  快速重传与快速恢复（见相应章节），忽略*/
  } else
      if (TCP_SEQ_BETWEEN(ackno, pcb->lastack+1, pcb->snd_max)){
        /* We come here when the ACK acknowledges new data. */
        if (pcb->flags & TF_INFR) { /*  从快速恢复转为拥塞避免  */
          pcb->flags &= ~TF_INFR;
          pcb->cwnd = pcb->ssthresh;
        }
        /* Reset the number of retransmissions. */
        pcb->nrtx = 0;
        /* Reset the retransmission time-out. */
        pcb->rto = (pcb->sa >> 3) + pcb->sv;
        /* Update the send buffer space. */
        pcb->acked = ackno - pcb->lastack; /*  被确认的报文长度  */
```

```
      pcb->snd_buf += pcb->acked;   /* 更新发送缓冲的大小 */
      /* Reset the fast retransmit variables. */
      pcb->dupacks = 0;
      pcb->lastack = ackno;
      /* 慢启动或拥塞避免（见相应章节），忽略 */
      /* 将已确认的 segment 从 pcb->unacked 移除，忽略 */
      /* 有新的 segment 被确认，复位 pcb 轮询计数器 */
      pcb->polltmr = 0;
    }
    /* 将已确认的 segment 从 pcb->unsent 移除(超时重传会将所有待确认 segment 移到 pcb->unack)，忽略 */
  }
  /* End of ACK for new data processing. */

  /* rtt 估计（见相应章节），忽略 */

  /* If the incoming segment contains data, we must process it further. */
  if (tcplen > 0) {
    /* This code basically does three things:

    +) If the incoming segment contains data that is the next
    in-sequence data, this data is passed to the application. This
    might involve trimming the first edge of the data. The rcv_nxt
    variable and the advertised window are adjusted.

    +) If the incoming segment has data that is above the next
    sequence number expected (->rcv_nxt), the segment is placed on
    the ->ooseq queue. This is done by finding the appropriate
    place in the ->ooseq queue (which is ordered by sequence
    number) and trim the segment in both ends if needed. An
    immediate ACK is sent to indicate that we received an
    out-of-sequence segment.

    +) Finally, we check if the first segment on the ->ooseq queue
    now is in sequence (i.e., if rcv_nxt >= ooseq->seqno). If
    rcv_nxt > ooseq->seqno, we must trim the first edge of the
    segment on ->ooseq before we adjust rcv_nxt. The data in the
    segments that are now on sequence are chained onto the
    incoming segment so that we only need to call the application
    once.
    */
    /*
    这段代码很乏味，不做详细介绍，上面的描述足够描述整个流程。下面做简单说明：
    1) pcb->ooseq，用来保存到达的在 receive wnd 之外(out of seqence)的 segment。
    2) pcb->rcv_nxt，期望收到的下一个 segment 的 seqno，receive wnd 的 left side，随着报文的到达而增大。
```

```
    3) pcb->rcv_wnd，receive wnd 的大小，随着报文的到达而减小，当用户任务确认数据被处理后而增大。
    */
  } else {
    /* Segments with length 0 is taken care of here. Segments that    fall out of the window are ACKed. */
    if(!TCP_SEQ_BETWEEN(seqno, pcb->rcv_nxt, pcb->rcv_nxt + pcb->rcv_wnd-1)){
      tcp_ack_now(pcb);
    }
  }
  return accepted_inseq;
}
```

24、用户接收数据

24.1 内核事件处理函数 recv_tcp()

当内核收到有效数据或 fin flag 时，事件处理函数 recv_tcp()被调用。recv_tcp()将保存有数据或 fin flag 的 pbuf 链表 post 到相应 conn->recvmbox。

24.2 系统调用 netconn_recv()

```
struct netbuf * netconn_recv(struct netconn *conn)
{
  ......
  if (conn->type == NETCONN_TCP) {
    ......
    buf = memp_malloc(MEMP_NETBUF);
    ......
    sys_mbox_fetch(conn->recvmbox, (void *)&p);   /*  获得保存有数据或 fin flag 的 pbuf 链表  */
    if (p != NULL)
    {
        len = p->tot_len;
        conn->recv_avail -= len;
    }
    else
        len = 0;
    ......
    /* If we are closed, we indicate that we no longer wish to receive
       data by setting conn->recvmbox to SYS_MBOX_NULL. */
    if (p == NULL) {
      memp_free(MEMP_NETBUF, buf);
      sys_mbox_free(conn->recvmbox);
      conn->recvmbox = SYS_MBOX_NULL;
      return NULL;
    }
    buf->p = p;
    buf->ptr = p;
    buf->fromport = 0;
```

```
      buf->fromaddr = NULL;
      /* Let the stack know that we have taken the data. */
      if ((msg = memp_malloc(MEMP_API_MSG)) == NULL) {
        conn->err = ERR_MEM;
        return buf;
      }
      msg->type = API_MSG_RECV;
      msg->msg.conn = conn;
      if (buf != NULL) {
        msg->msg.msg.len = buf->p->tot_len; /*  用户任务实际收到的数据长度  */
      } else {
        msg->msg.msg.len = 1; /*fin flag 的长度为 1*/
      }
      api_msg_post(msg);
      sys_mbox_fetch(conn->mbox, NULL);
      memp_free(MEMP_API_MSG, msg);
    } else {
    ......
    }
    ......
    return buf; /*  返回给用户任务收到的数据或 fin flag */
}
```

24.3 内核事件处理函数 tcp_recved()

```
void tcp_recved(struct tcp_pcb *pcb, u16_t len)
{
    if ((u32_t)pcb->rcv_wnd + len > TCP_WND) { /*  根据用户任务收到的数据长度增大 receive wnd 的大小  */
      pcb->rcv_wnd = TCP_WND;
    } else {
      pcb->rcv_wnd += len;
    }
    if (!(pcb->flags & TF_ACK_DELAY) &&
       !(pcb->flags & TF_ACK_NOW)) {
      /*
       * We send an ACK here (if one is not already pending, hence
       * the above tests) as tcp_recved() implies that the application
       * has processed some data, and so we can open the receiver's
       * window to allow more to be transmitted.    This could result in
       * two ACKs being sent for each received packet in some limited cases
       * (where the application is only receiving data, and is slow to
       * process it) but it is necessary to guarantee that the sender can
       * continue to transmit.
       */
      tcp_ack(pcb);
```

```
  }
  else if (pcb->flags & TF_ACK_DELAY && pcb->rcv_wnd >= TCP_WND/2) {
    /* If we can send a window update such that there is a full
     * segment available in the window, do so now.  This is sort of
     * nagle-like in its goals, and tries to hit a compromise between
     * sending acks each time the window is updated, and only sending
     * window updates when a timer expires.  The "threshold" used
     * above (currently TCP_WND/2) can be tuned to be more or less
     * aggressive  */
    tcp_ack_now(pcb);
  }
}
```

25、两个定时器

25.1 250ms 定时器 tcp_fasttmr()

延时的 ack 在该定时器中被发送。

25.2 500ms 定时器 tcp_slowtmr()

- 全局变量 tcp_ticks，500ms 递增一次，用于表示当前时钟
- pcb->tmr，在 tcp_process()中每处理一个到达的报文，tcp_ticks 被赋值给 pcb->tmr
- pcb->keep_cnt，在 tcp_process()中每处理一个到达的报文，pcb->keep_cnt 被复位

```
void tcp_slowtmr(void)
{
  ......
  ++tcp_ticks;   /* 递增 500ms 的 tick */
  /* Steps through all of the active PCBs. */
  prev = NULL;
  pcb = tcp_active_pcbs;
  ......
  while (pcb != NULL) {
    ......
    pcb_remove = 0;
    if (pcb->state == SYN_SENT && pcb->nrtx == TCP_SYNMAXRTX) {
      ++pcb_remove; /* 重传的次数过多，关闭链接*/
    }
    else if (pcb->nrtx == TCP_MAXRTX) {
      ++pcb_remove; /* 重传的次数过多，关闭链接*/
    } else {
      ++pcb->rtime;
      if (pcb->unacked != NULL && pcb->rtime >= pcb->rto) {
        /* 超时重传（见相应章节），忽略 */
      }
    }
```

```
  /* Check if this PCB has stayed too long in FIN-WAIT-2 */
  if (pcb->state == FIN_WAIT_2) {
    if ((u32_t)(tcp_ticks - pcb->tmr) > TCP_FIN_WAIT_TIMEOUT / TCP_SLOW_INTERVAL) {
      ++pcb_remove; /* 在 fin_wait_2 状态长时间没收到 fin，关闭链接*/
    }
  }
/* Check if KEEPALIVE should be sent */   /* 扩展出来的 keeplive 定时器 */
if((pcb->so_options & SOF_KEEPALIVE)
    && ((pcb->state == ESTABLISHED) || (pcb->state == CLOSE_WAIT))) {
    if((u32_t)(tcp_ticks - pcb->tmr) > (pcb->keepalive + TCP_MAXIDLE) / TCP_SLOW_INTERVAL)   {
        tcp_abort(pcb);
    }
    else if((u32_t)(tcp_ticks - pcb->tmr) >
        (pcb->keepalive + pcb->keep_cnt * TCP_KEEPINTVL) / TCP_SLOW_INTERVAL) {
        tcp_keepalive(pcb);
        pcb->keep_cnt++;
    }
}
  /* If this PCB has queued out of sequence data, but has been
     inactive for too long, will drop the data (it will eventually
     be retransmitted). */
  if (pcb->ooseq != NULL &&
      (u32_t)tcp_ticks - pcb->tmr >=
      pcb->rto * TCP_OOSEQ_TIMEOUT) {
    tcp_segs_free(pcb->ooseq);
    pcb->ooseq = NULL;
    LWIP_DEBUGF(TCP_CWND_DEBUG, ("tcp_slowtmr: dropping OOSEQ queued data\n"));
  }
  /* Check if this PCB has stayed too long in SYN-RCVD */
  if (pcb->state == SYN_RCVD) {
    if ((u32_t)(tcp_ticks - pcb->tmr) >
      TCP_SYN_RCVD_TIMEOUT / TCP_SLOW_INTERVAL) {
      ++pcb_remove;   /* 在 syn_rcvd 状态长时间没收到 ack，关闭链接*/
    }
  }
  /* Check if this PCB has stayed too long in LAST-ACK */
  if (pcb->state == LAST_ACK) {
    if ((u32_t)(tcp_ticks - pcb->tmr) > 2 * TCP_MSL / TCP_SLOW_INTERVAL) {
      ++pcb_remove;   /* 在 last_ack 状态长时间没收到 ack，关闭链接*/
    }
  }
  /* If the PCB should be removed, do it. */
  if (pcb_remove) {
    tcp_pcb_purge(pcb); /* 释放 pcb 上的所有数据 */
```

```
        /* Remove PCB from tcp_active_pcbs list. */
        if (prev != NULL) {
            prev->next = pcb->next;
        } else {
          /* This PCB was the first. */
          LWIP_ASSERT("tcp_slowtmr: first pcb == tcp_active_pcbs", tcp_active_pcbs == pcb);
          tcp_active_pcbs = pcb->next;
        }
        TCP_EVENT_ERR(pcb->errf, pcb->callback_arg, ERR_ABRT); /*  触发内核事件处理函数 err_tcp() */
        pcb2 = pcb->next;
        memp_free(MEMP_TCP_PCB, pcb);
        pcb = pcb2;
      } else {
        /* We check if we should poll the connection. */ /*  见 20.2.3 */
        ++pcb->polltmr;
        if (pcb->polltmr >= pcb->pollinterval) {
          pcb->polltmr = 0;
          LWIP_DEBUGF(TCP_DEBUG, ("tcp_slowtmr: polling application\n"));
          TCP_EVENT_POLL(pcb, err);
          if (err == ERR_OK) {
            tcp_output(pcb);
          }
        }
        prev = pcb;
        pcb = pcb->next;
      }
    }
```

上面浏览了一遍三次握手，数据的收发过程，此外对 tcp 的一些核心算法的实现做了详细的分析，lwip 的介绍到这里就结束了。其它需要关心的还有：状态的迁移，tcp 链接的关闭。相信看了上文已经能够对 lwip 的整个框架有个大概的了解，再接着看下去不会有大困难。除了 tcp 之外，ip 的收发、内存管理等值得读者浏览。纵观整个代码，质量比 YAFFS2 差几等，不过 tcp 协议的实现确实比较麻烦，实现起来要么僵硬、要么脆弱。