第26章 TCP 输 出

26.1 引言

函数tcp_output负责发送报文段，代码中有很多地方都调用了它。

tcp_usrreq在多种请求处理中调用了这一函数：处理 PRU_CONNECT，发送初始SYN；处理PRU_SHUTDOWN， 发送FIN；处理PRU_RCVD，应用进程从插口接收缓存中读取若干数据后可能需要发送新的窗口大小通告；处理 PRU_SEND，发送数据；处理PRU_SENDOOB，发送带外数据。

- tcp_fasttimo调用它发送延迟的ACK；
- tcp_timers在重传定时器超时时，调用它重传报文段；
- tcp_timers在持续定时器超时时，调用它发送窗口探测报文段；
- tcp_drop调用它发送RST；
- tcp_disconnect调用它发送FIN；
- tcp_input在需要输出或需要立即发送ACK时调用它；
- tcp_input在收到一个纯ACK报文段且本地有数据发送时调用它(纯ACK报文段指不携带数据，只确认已接收数据的报文段)；
- tcp_input在连续收到3个重复的ACK时，调用它发送一个单一报文段 (快速重传算法)；

tcp_input首先确定是否有报文段等待发送。除了存在需要发往连接对端的数据外，TCP输出还受到其他许多因素的控制。例如，对端可能通告接收窗口为零，阻止 TCP发送任何数据；Nagle算法阻止TCP发送大量小报文段；慢启动和避免拥塞算法限制 TCP发送的数据量。相反，有些函数置位一些特殊标志，强迫 tcp_output发送报文段，如TF_ACKNOW标志置位意味着必须立即发送一个 ACK。如果tcp_output确定不发送某个报文段，数据 (如果存在)将保留在插口的发送缓存中，等待下一次调用该函数。

26.2 tcp_output概述

tcp_output函数很大，我们将分14个部分予以讨论。图26-1给出了函数的框架结构。

1. 是否等待对端的ACK？

61 如果发送的最大序号(snd_max)等于最早的未确认过的序号 (snd_una)，即不等待对端发送ACK，idle为真。图24-17中，idle应为假，因为序号4~6已发送但还未被确认， TCP在等待对端发送对上述序号的确认。

2. 返回慢启动

62-68 如果TCP不等待对端发送ACK，而且在一个往返时间内也没有收到对端发送的其他报文段，设置拥塞窗口为仅能容纳一个报文段(t_maxseg字节)，从而在发送下一个报文段时，

强迫执行慢启动算法。如果数据传输中出现了显著的停顿（“显著”指停顿时间超过RTT），说明与先前测量RTT时相比，网络条件已发生了变化。Net/3假定出现了最坏情况，因而返回慢起动状态。

tcp_output.c

```
43 int
44 tcp_output(tp)
45 struct tcpcb *tp;
46 {
47     struct socket *so = tp->t_inpcb->inp_socket;
48     long    len, win;
49     int     off, flags, error;
50     struct mbuf *m;
51     struct tcpiphdr *ti;
52     u_char  opt[MAX_TCPOPTLEN];
53     unsigned optlen, hdrlen;
54     int     idle, sendalot;

55     /*
56      * Determine length of data that should be transmitted
57      * and flags that will be used.
58      * If there are some data or critical controls (SYN, RST)
59      * to send, then transmit; otherwise, investigate further.
60      */
61     idle = (tp->snd_max == tp->snd_una);
62     if (idle && tp->t_idle >= tp->t_rxtcur)
63         /*
64          * We have been idle for "a while" and no acks are
65          * expected to clock out any data we send --
66          * slow start to get ack "clock" running again.
67          */
68         tp->snd_cwnd = tp->t_maxseg;

69 again:
70     sendalot = 0;   /* set nonzero if more than one segment to output */

                /* look for a reason to send a segment;  */
                /* goto send if a segment should be sent */

218     /*
219      * No reason to send a segment, just return.
220      */
221     return (0);

222 send:

                /* form output segment, call ip_output() */

489     if (sendalot)
490         goto again;
491     return (0);
492 }
```

tcp_output.c

图26-1 tcp_output 函数：框架结构

3. 发送多个报文段

69-70 控制跳转至send后，调用ip_output发送一个报文段。但如果ip_output确定有

多个报文段需要发送，sendalot置为1，函数将试图发送另一个报文段。因此，ip_output的一次调用能够发送多个报文段。

26.3 决定是否应发送一个报文段

某些情况下，在报文段准备好之前已调用了tcp_output。例如，当插口层从插口的接收缓存中移走数据，传递给用户进程时，会生成PRU_RCVD请求。尽管不一定，但完全有可能因为应用进程取走了大量数据，而使得TCP有必要发送新的窗口通告。tcp_output的前半部分确定是否存在需要发往对端的报文段。如果没有，则函数返回，不执行发送操作。

图26-2给出了判定“是否有报文段发送”测试代码的第一部分。

tcp_output.c

```
 71      off = tp->snd_nxt - tp->snd_una;
 72      win = min(tp->snd_wnd, tp->snd_cwnd);

 73      flags = tcp_outflags[tp->t_state];
 74      /*
 75       * If in persist timeout with window of 0, send 1 byte.
 76       * Otherwise, if window is small but nonzero
 77       * and timer expired, we will send what we can
 78       * and go to transmit state.
 79       */
 80      if (tp->t_force) {
 81          if (win == 0) {
 82              /*
 83               * If we still have some data to send, then
 84               * clear the FIN bit.  Usually this would
 85               * happen below when it realizes that we
 86               * aren't sending all the data.  However,
 87               * if we have exactly 1 byte of unsent data,
 88               * then it won't clear the FIN bit below,
 89               * and if we are in persist state, we wind
 90               * up sending the packet without recording
 91               * that we sent the FIN bit.
 92               *
 93               * We can't just blindly clear the FIN bit,
 94               * because if we don't have any more data
 95               * to send then the probe will be the FIN
 96               * itself.
 97               */
 98              if (off < so->so_snd.sb_cc)
 99                  flags &= ~TH_FIN;
100              win = 1;
101          } else {
102              tp->t_timer[TCPT_PERSIST] = 0;
103              tp->t_rxtshift = 0;
104          }
105      }
```

tcp_output.c

图26-2 tcp_output 函数：强迫数据发送

71-72 off指从发送缓存起始处算起指向第一个待发送字节的偏移量，以字节为单位。它指向的第一个字节为snd_una(已发送但还未被确认的字节)。

win是对端通告的接收窗口大小(snd_wnd)与拥塞窗口大小(snd_cwnd)间的最小值。

73 图24-16给出了tcp_outflags数组，数组值取决于连接的当前状态。flags包括下列标志比特的组合：TH_ACK、TH_FIN、TH_RST和TH_SYN，分别表示需向对端发送的报文段类型。其他两个标志比特，TH_PUSH和TH_URG，如果需要，在报文段发送之前加入，与前4个标志比特是逻辑或的关系。

74-105 t_force标志非零表示持续定时器超时，或者有带外数据需要发送。这两种条件下，调用tcp_output的代码均为：

```
tp->t_force = 1;
error = tcp_output(tp);
tp->t_force = 0;
```

从而强迫TCP发送数据，尽管在正常情况下不会执行任何发送操作。

如果win等于0，连接处于持续状态(因为t_force非零)。如果此时插口的发送缓存中还存在数据，则FIN标志被清除。win必须置为1，以强迫发送一个字节的数据。

如果win非零，即有带外数据需要发送，则持续定时器复位，指数退避算法的索引，t_rxtshift被置为0。

图26-3给出了tcp_output的下一模块，计算发送的数据量。

tcp_output.c

```
106     len = min(so->so_snd.sb_cc, win) - off;
107     if (len < 0) {
108         /*
109          * If FIN has been sent but not acked,
110          * but we haven't been called to retransmit,
111          * len will be -1.  Otherwise, window shrank
112          * after we sent into it.  If window shrank to 0,
113          * cancel pending retransmit and pull snd_nxt
114          * back to (closed) window.  We will enter persist
115          * state below.  If the window didn't close completely,
116          * just wait for an ACK.
117          */
118         len = 0;
119         if (win == 0) {
120             tp->t_timer[TCPT_REXMT] = 0;
121             tp->snd_nxt = tp->snd_una;
122         }
123     }
124     if (len > tp->t_maxseg) {
125         len = tp->t_maxseg;
126         sendalot = 1;
127     }
128     if (SEQ_LT(tp->snd_nxt + len, tp->snd_una + so->so_snd.sb_cc))
129         flags &= ~TH_FIN;

130     win = sbspace(&so->so_rcv);
```

tcp_output.c

图26-3 tcp_output 函数：计算发送的数据量

1. 计算发送的数据量

106 len等于发送缓存中比特数和win(对端通告的接收窗口与拥塞窗口间的最小值，强迫TCP发送数据时也可能等于1字节)，两者间的最小值减去off。减去off是因为发送缓存中的许多字节已发送过，正等待对端的确认。

2. 窗口缩小检查

107-117 造成len小于零的一种可能情况是接收方缩小了窗口，即接收方把窗口的右界移向左侧，下面的例子说明了这种情况。开始时，接收方通告接收窗口大小为 6字节，TCP发送报文段，携带字节 4、5和6。紧接着，TCP又发送一个报文段，携带字节 7、8和9。图26-4显示了两个报文段发送后本地的状态。

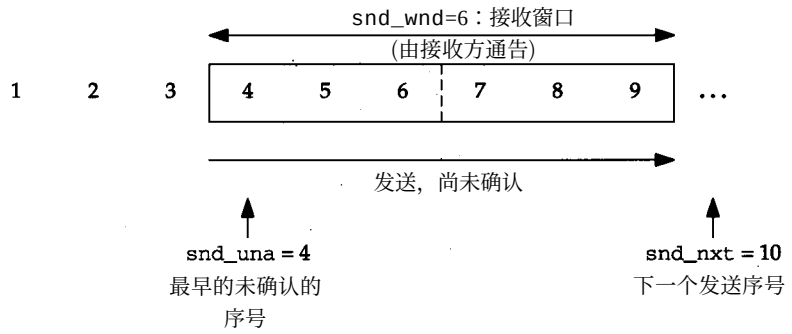

图26-4 发送4~9字节后的本地发送缓存

之后，收到一个ACK，确认序号字段为7(确认所有序号小于7的数据，包括字节6)，但窗口字段为1。接收方缩小了接收窗口，此时本地的状态如图 26-5所示。

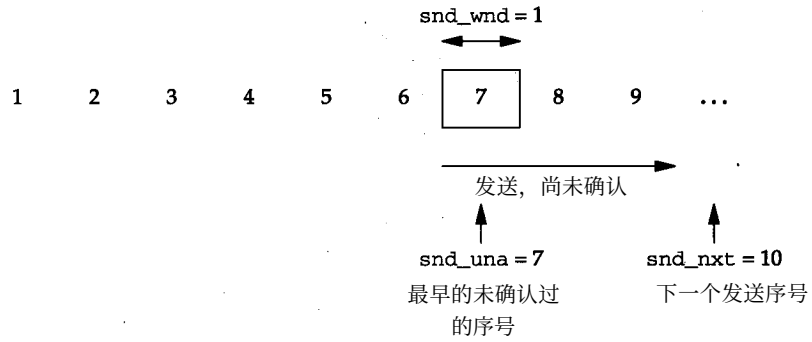

图26-5 收到4~7字节的确认后，本地发送缓存

窗口缩小后，执行图26-2和图26-3中的计算，得到：

```
off = snd_nxt - snd_una = 10 - 7 = 3
win = 1
len = min(so_snd.sb_cc, win) -off = min(3, 1) - 3 = -2
```

假定发送缓存仅包含字节7、8和9。

RFC 793和RFC 1122都非常不赞成缩小窗口。尽管如此，具体实现必须考虑这一问题并加以处理。这种做法遵循了在 RFC 791中首次提出的稳健性原则：“ 对接收报文段的假设尽量少一些，对发送报文段的限制尽量多一些 ”。

造成len小于0的另一种可能情况是，已发送过 FIN，但还未收到确认 (见习题26.2)。图26-6给出了这种情况。

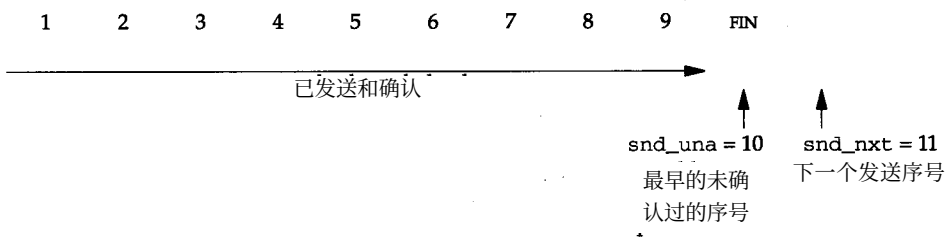

图26-6 字节1~9已发送并收到对端确认，之后关闭连接

图26-6是图26-4的继续，假定字节7~9已被确认，snd_una的当前值为10。应用进程随后关闭连接，向对端发送FIN。在本章后续部分将看到，TCP发送FIN时，snd_nxt将增加1(因为FIN也需要序号)，在本例中，snd_nxt将等于11，而FIN的序号为10。执行图26-2和图26-3中的计算，得到：

```
off = snd_nxt - snd_una = 11 - 10 = 1
win = 6
len = min( so_snd.sb_cc, win ) - off = min(0, 6) - 1 = -1
```

我们假定接收方通告接收窗口大小为6。这个假定无关紧要，因为发送缓存中待发送的字节数(0)小于它。

3. 进入持续状态

118-122 len被置为0。如果对端通告的接收窗口大小为0，则重传定时器将被置为0，任何等待的重传将被取消。令snd_nxt等于snd_una，指针返回发送窗口的最左端，连接将进入持续状态。如果接收方最终打开了接收窗口，则TCP将从发送窗口的最左端开始重传。

4. 一次发送一个报文段

124-127 如果需要发送的数据超过了一个报文段的容量，len置为最大报文段长度，sendalot置为1。如图26-1所示，这将使tcp_output在报文段发送完毕后进入另一次循环。

5. 如果发送缓存不空，关闭FIN标志

128-129 如果本次输出操作未能清空发送缓存，FIN标志必须被清除(防止该标志在flags中被置位)。图26-7举例说明了这一情况。

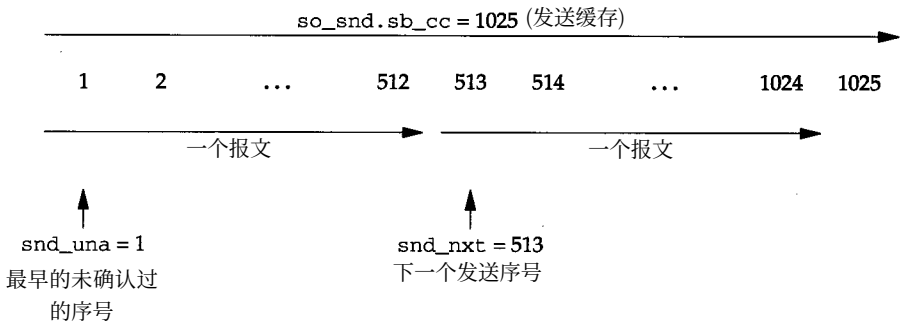

图26-7 实例：FIN置位时，发送缓存不空

这个例子中，第一个512字节的报文段已发送(还未被确认)，TCP正准备发送第二个报文段(512~1024字节)。此时，发送缓存中仍有1字节的数据(1025字节)，应用进程关闭了连接。

len=512(一个报文段)，图26-3中的C表达式变为：

```
SEQ_LT (1025, 1026)
```

如果表达式为真，则FIN标志被清除；否则，TCP无法向对端发送序号为1025的字节。

6. 计算接收窗口大小

130 win设定为本地接收缓存中可用空间的大小，即TCP向对端通告的接收窗口的大小。请注意，这是第二次用到这个变量。在函数前一部分中，它等于允许 TCP发送的最大数据量，但从现在起，它等于本地向对端通告的接收窗口的大小。

糊涂窗口综合症(简写为SWS，详见卷1第22.3节)指连接上交换的都是短报文段，而不是最大长度报文段。这种现象的出现是由于接收方通告的接收窗口过小，或者发送方传输了许多小报文段，因此，避免糊涂窗口综合症，需要发送方和接收方的共同努力。图 26-8给出了发送方避免糊涂窗口综合症的做法。

tcp_output.c

```
131     /*
132      * Sender silly window avoidance.  If connection is idle
133      * and can send all data, a maximum segment,
134      * at least a maximum default-sized segment do it,
135      * or are forced, do it; otherwise don't bother.
136      * If peer's buffer is tiny, then send
137      * when window is at least half open.
138      * If retransmitting (possibly after persist timer forced us
139      * to send into a small window), then must resend.
140      */
141     if (len) {
142         if (len == tp->t_maxseg)
143             goto send;
144         if ((idle || tp->t_flags & TF_NODELAY) &&
145             len + off >= so->so_snd.sb_cc)
146             goto send;
147         if (tp->t_force)
148             goto send;
149         if (len >= tp->max_sndwnd / 2)
150             goto send;
151         if (SEQ_LT(tp->snd_nxt, tp->snd_max))
152             goto send;
153     }
```

tcp_output.c

图26-8 tcp_output 函数：发送方避免糊涂窗口综合症

7. 发送方避免糊涂窗口综合症的方法

142-143 如果待发送报文段是最大长度报文段，则发送它。

144-146 如果无需等待对端的ACK(idle为真)，或者Nagle算法被取消(TF_NODELAY为真)，并且TCP正在清空发送缓存，则发送数据。Nagle算法(详见卷1第19.4节)的思想是：如果某个连接需要等待对端的确认，则不允许 TCP发送长度小于最大长度的报文段。通过设定插口选项TCP_NODELAY，可以取消这个算法。对于正常的交互式连接 (如Telnet或Rlogin)，即使连接上存在未确认过的数据，代码中的 if语句也为假，因为默认条件下TCP会采用Nagle算法。

147-148 如果由于持续定时器超时，或者有带外数据，强迫 TCP执行发送操作，则数据将被发送。

149-150 如果接收方的接收窗口已至少打开了一半，则发送数据。这个限制条件是为了处

理对端一直发送小窗口通告，甚至小于报文段长度的情况。变量 max_sndwnd由tcp_input维护，等于连接对端发送的所有窗口通告中的最大值。实际上， TCP试图猜测对端接收缓存的大小，并假定对端永远不会减小其接收缓存。

151-152 如果重传定时器超时，则必须发送一个报文段。snd_max是已发送过的最高序号，从图25-26可知，重传定时器超时后， snd_nxt将被设为snd_una，即snd_nxt会指向窗口的左侧，从而小于snd_max。

图26-9给出了tcp_output的下一部分，确定 TCP是否必须向对端发送新的窗口通告，称之为“ 窗口更新 ”。

tcp_output.c

```
154     /*
155      * Compare available window to amount of window
156      * known to peer (as advertised window less
157      * next expected input).  If the difference is at least two
158      * max size segments, or at least 50% of the maximum possible
159      * window, then want to send a window update to peer.
160      */
161     if (win > 0) {
162         /*
163          * "adv" is the amount we can increase the window,
164          * taking into account that we are limited by
165          * TCP_MAXWIN << tp->rcv_scale.
166          */
167         long    adv = min(win, (long) TCP_MAXWIN << tp->rcv_scale) -
168         (tp->rcv_adv - tp->rcv_nxt);

169         if (adv >= (long) (2 * tp->t_maxseg))
170             goto send;
171         if (2 * adv >= (long) so->so_rcv.sb_hiwat)
172             goto send;
173     }
```

tcp_output.c

图26-9 tcp_output 函数：判定是否需要发送窗口更新报文

154-168 表达式

```
min (win, (long) TCP_MAXWIN << tp->rcv_scale)
```

等于插口接收缓存可用空间大小(win)和连接上所允许的最大窗口大小之间的最小值，即 TCP当前能够向对端发送的接收窗口的最大值。表达式

```
(tp->rcv_adv - tp->rcv_nxt)
```

等于TCP最后一次通告的接收窗口中剩余空间的大小，以字节为单位。两者相减得到 adv，窗口已打开的字节数。 tcp_input顺序接收数据时，递增rcv_nxt。tcp_output在通告窗口边界向右移动时，递增rcv_adv(代码见图26-32)。

回想图 24-18，假定收到了字节 4、5和6，并提交给应用进程。图 26-10给出了此时tcp_output中接收缓存的状态。

adv等于3，因为接收空间中还有3个字节(字节10、11和12)等待对端填充。

169-170 如果剩余的接收空间能够容纳两个或两个以上的报文段，则发送窗口更新报文。在收到最大长度报文段后， TCP将确认收到的所有其他报文段：“ 确认所有其他报文段 (ACK-every-other-segment) ”的属性(马上就会看到具体的实例)。

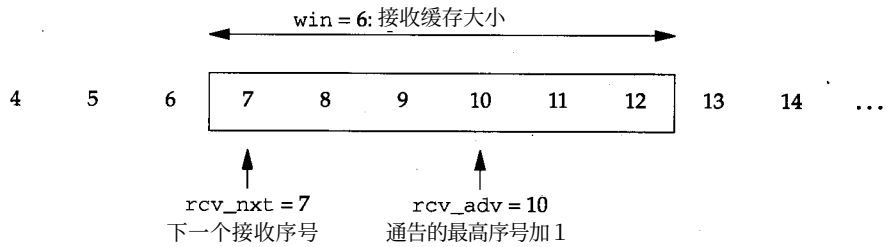

图26-10 收到字节4、5和6后，图24-18中连接的状态变化

171-172 如果可用空间大于插口接收缓存的一半，则发送窗口更新报文。

图26-11给出了tcp_output下一部分的代码，判定输出标志是否置位，要求 TCP发送相应报文段。

tcp_output.c

```
174     /*
175      * Send if we owe peer an ACK.
176      */
177     if (tp->t_flags & TF_ACKNOW)
178         goto send;
179     if (flags & (TH_SYN | TH_RST))
180         goto send;
181     if (SEQ_GT(tp->snd_up, tp->snd_una))
182         goto send;
183     /*
184      * If our state indicates that FIN should be sent
185      * and we have not yet done so, or we're retransmitting the FIN,
186      * then we need to send.
187      */
188     if (flags & TH_FIN &&
189         ((tp->t_flags & TF_SENTFIN) == 0 || tp->snd_nxt == tp->snd_una))
190         goto send;
```

tcp_output.c

图26-11 tcp_output 函数：是否需要发送特定报文段

174-178 如果TF_ACKNOW置位，要求立即发送 ACK，则发送相应报文段。有多种情况可导致TF_ACKNOW置位：200 ms延迟ACK定时器超时，报文段未按顺序到达 (用于快速重传算法)，三次握手时收到了SYN，收到了窗口探测报文，收到了 FIN。

179-180 如果输出标志flags要求发送SYN或RST，则发送相应报文段。

181-182 如果紧急指针，snd_up，超出了发送缓存的起始边界，则发送相应报文段。紧急指针由PRU_SENDOOB请求处理代码(图30-9)负责维护。

183-190 如果输出标志flags要求发送FIN，并且满足下列条件：FIN未发送过或者FIN等待重传，则发送相应报文段。FIN发送后，函数将置位TF_SENTFIN标志。

到目前为止，tcp_output还没有真正发送报文段，图 26-12给出了函数返回前的最后一段代码。

191-217 如果发送缓存中存在需要发送的数据 (so_snd.sb_cc非零)，并且重传定时器和持续定时器都未设定，则启动持续定时器。这是为了处理对端通告的接收窗口过小，无法接收最大长度报文段，而且也没有特殊原因需要发送立即发送报文段的情况。

218-221 由于不需要发送报文段，tcp_output返回。

tcp_output.c

```
191     /*
192      * TCP window updates are not reliable, rather a polling protocol
193      * using 'persist' packets is used to ensure receipt of window
194      * updates.  The three 'states' for the output side are:
195      *  idle                 not doing retransmits or persists
196      *  persisting           to move a small or zero window
197      *  (re)transmitting     and thereby not persisting
198      *
199      * tp->t_timer[TCPT_PERSIST]
200      *      is set when we are in persist state.
201      * tp->t_force
202      *      is set when we are called to send a persist packet.
203      * tp->t_timer[TCPT_REXMT]
204      *      is set when we are retransmitting
205      * The output side is idle when both timers are zero.
206      *
207      * If send window is too small, there is data to transmit, and no
208      * retransmit or persist is pending, then go to persist state.
209      * If nothing happens soon, send when timer expires:
210      * if window is nonzero, transmit what we can,
211      * otherwise force out a byte.
212      */
213     if (so->so_snd.sb_cc && tp->t_timer[TCPT_REXMT] == 0 &&
214         tp->t_timer[TCPT_PERSIST] == 0) {
215         tp->t_rxtshift = 0;
216         tcp_setpersist(tp);
217     }
218     /*
219      * No reason to send a segment, just return.
220      */
221     return (0);
```

tcp_output.c

图26-12 tcp_output 函数：进入持续状态

举例

应用进程向某个空闲的连接写入100字节，接着又写入50字节。假定报文段大小为512字节。在第一次写入操作时，由于连接空闲，且 TCP正在清空发送缓存，图 26-8中的代码(144~146行)被执行，发送一个报文段，携带100字节的数据。

在第二次写入50字节时，图26-8中的代码被执行，但未发送报文段：待发送数据不能构成一个最大长度报文段，连接未空闲 (假定TCP正在等待第一个报文段的 ACK)，默认时采用Nagle算法，t_force未置位，并且假定正常情况下接收窗口大小为 4096，50不满足大于等于2048的条件。这50字节的数据将暂留在发送缓存中，也许会一直等到第一个报文段的 ACK到达。由于对端可能延迟发送ACK，最后50字节数据发送前的延迟有可能会更长。

这个例子说明采用Nagle算法时，如果待发送数据无法构成最大长度报文段，如何计算它的延时。参见习题26.12。

举例

本例说明TCP的“确认所有其他报文段”属性。假定连接的报文段大小为 1024字节，接收缓存大小为4096字节。本地不发送数据，只接收数据。

发向对端的对SYN的ACK报文中，通告接收窗口大小为4096，图26-13给出了两个变量rcv_nxt和rcv_adv的初始值。接收缓存为空。

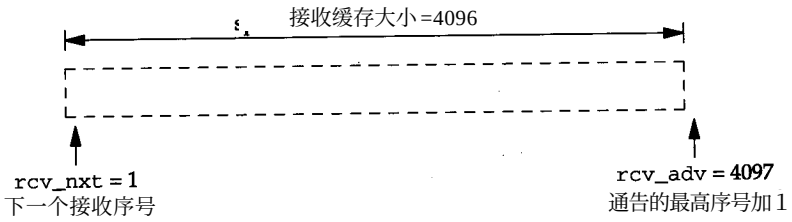

图26-13 接收方通告接收窗口大小为4096

对端发送1~1024字节的报文段，tcp_input处理报文段后，设置连接的延迟ACK标志，把1024字节的数据放入插口的接收缓存中(图28-13)。更新rcv_nxt，如图26-14所示。

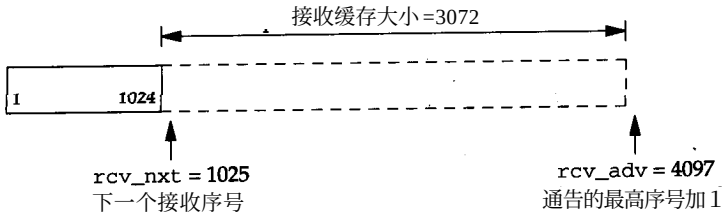

图26-14 图26-13所示的连接在收到1~1024字节后的状态变迁

应用进程从插口的接收缓存中读取 1024字节的数据。从图 30-6中可看到，生成的PRU_RCVD请求在处理过程中会调用 tcp_output，因为应用进程从接收缓存读取数据后，可能需要发送窗口更新报文。当 tcp_output被调用时，rcv_nxt和rcv_adv的值与图26-14相同，唯一的区别是接收缓存的可用空间增加至 4096，因为应用进程从中读取了第一个1024字节的数据。把上述具体数值代入图26-9中的算式，得到：

```
adv = min(4096, 65535) - (4097 - 1025)
    = 1024
```

TCP_MAXWIN等于65535，我们假定接收窗口大小偏移量为0。由于窗口的增加值小于两个最大报文段长度(2048)，无需发送窗口更新报文。但由于延迟ACK标志置位，如果200ms定时器超时，将发送ACK。

当TCP收到下一个1025~2048字节的报文段时，tcp_input处理后，设定连接的延迟ACK标志(这个标志已置位)，把1024字节的数据放入插口的接收缓存中，更新rcv_nxt，如图26-15所示。

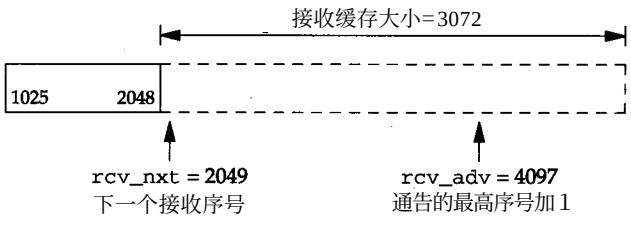

图26-15 图26-14所示的连接收到1025~2048字节后的状态变迁

应用进程读取1024~2048字节的数据，调用tcp_output。rcv_nxt和rcv_adv的值与图26-15相同，尽管应用进程读取1024字节的数据后，接收缓存的可用空间增加至4096。把上述具体数值代入图26-9的算式中，得到：

```
adv = min(4096, 65535) - (4097 - 2049)
    = 2048
```

它等于两个报文段的长度，因此发送窗口更新报文，确认序号字段为 2049，通告窗口字段为4096，表示接收方希望接收序号2049~6145的数据。我们在后面将看到，函数发送完窗口更新报文后，将更新rcv_adv的值为6145。

本例说明了如果数据接收时间少于200ms延迟定时器时限，在有两个或两个以上报文段到达，而且应用进程连续读取数据引起了接收窗口的不断变化时，将发送 ACK，确认所有接收到的报文段。如果有数据到达，但应用进程没有从插口的接收缓存中读取数据，则“确认所有其他报文段”的属性不会出现。相反，发送方只能看到多个延迟 ACK，每个ACK的窗口字段均较前一个要小，直到接收缓存被填满，接收窗口缩小为0。

26.4 TCP选项

TCP首部可以有任选项。由于tcp_output的下一部分代码将试图确定哪些选项需要发送，并据此组织将发送的报文段，下面我们将暂时离开函数代码，转而讨论这些选项。

图26-16列出了Net/3支持的选项格式。

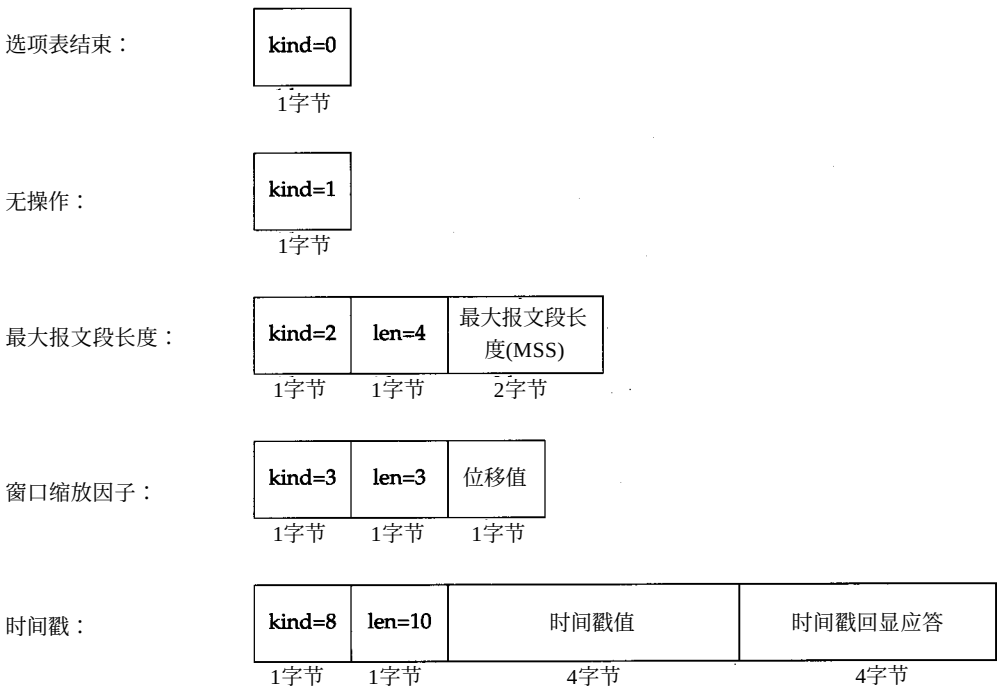

图26-16 Net/3支持的TCP选项

所有选项以1字节的*kind*字段开头，确定选项类型。头两个选项 (*kind*=0或*kind*=1)只有1个字节。其余3个选项都是多字节的，带有*len*字段，位于*kind*字段之后，存储选项的长度。长度

中包括*kind*字段和*len*字段。

多字节整数 —— MSS和两个时间戳值 —— 遵照网络字节序存储。

最后两个选项，窗口大小和时间戳，是新增的，因此许多系统都不支持。为了与以前的系统兼容，应遵循下列原则：

1) TCP主动打开时(发送不带ACK的SYN)，可以在初始SYN中同时发送这两个选项，或发送其中的任何一个。如果全局变量`tcp_do_rfc1323`非零(默认值等于1)，则Net/3同时支持这两个选项。此项功能由`tcp_newtcpcb`函数实现。

2) 只有对端返回的SYN中包含同样的选项时，才可以使用这些选项。图 28-20和图29-2中的代码实现此类处理。

3) TCP被动打开时，如果收到的 SYN中包含了这两个选项，而且也希望使用这些选项，则发向对端的响应(带有ACK的SYN)中必须包含它们，如图26-23所示。

由于系统必须忽略它不了解的选项，因此新增的选项只有当连接双方都了解这一选项，且同时希望支持它时才会被使用。

27.5节将讨论如何处理 MSS选项。下面两节将总结 Net/3处理两个新选项的做法：窗口大小和时间戳。

> 还有其他可能的选项。*kinds*等于4、5、6和7，称为选择性ACK和回显选项，在RFC 1072[Jacobson and Braden 1998]中定义。图26-16中并未给出这些选项，因为回显选项已被时间戳选项所代替，选择性 ACK选项目前还未形成正式标准，未在 RFC 1323中出现。此外，处理TCP交易的T/TCP建议(RFC 1644[Braden 1994]和卷1的24.7节)规定了其他3个选项，*kinds*分别为11、12和13。

26.5 窗口大小选项

窗口大小选项，在RFC 1323中定义，避免了TCP首部窗口大小字段只有16 bit的限制(图24-10)。如果网络带宽较高或延时较长 (如，RTT较长)，则需要较大的窗口，称为长肥管道 (long fat pipe)。卷1的第24.3节举例说明了现代网络需要较大的窗口，以获取最大的 TCP吞吐量。

图26-16中的偏移量最小值为 0 (无缩放)，最大值为14，即窗口最大可设定为 1 073 725 440 (65535×2^{14})字节。Net/3内部实现时，利用32 bit，而非16 bit整数表示窗口大小。

窗口大小选项只能出现在 SYN中，因此，连接建立后，每个传输方向上的缩放因子是固定不变的。

TCP控制块中的两个变量`snd_scale`和`rcv_scale`，分别规定了发送窗口和接收窗口的偏移量。它们的默认值均为 0，无缩放。每次收到对端发送的窗口通告时， 16 bit的窗口大小值被左移`snd_scale`比特，得到真正的32 bit的对端接收窗口大小(图28-6)。每次准备向对端发送窗口通告时，内部的 32 bit窗口大小值被右移`rcv_scale`比特，得到可填入TCP首部窗口字段的16 bit值。

TCP发送SYN时，无论是主动打开或被动打开，都是根据本地插口接收缓存大小选取`rcv_scale`值，填充窗口大小选项的偏移量字段。

26.6 时间戳选项

RFC 1323中还定义了时间戳选项。发送方在每个报文段中放入时间戳，接收方在 ACK中

将时间戳发回。对于每个收到的ACK，发送方根据返回的时间戳计算相应的RTT样本值。

图26-17总结了时间戳选项所用到的变量。

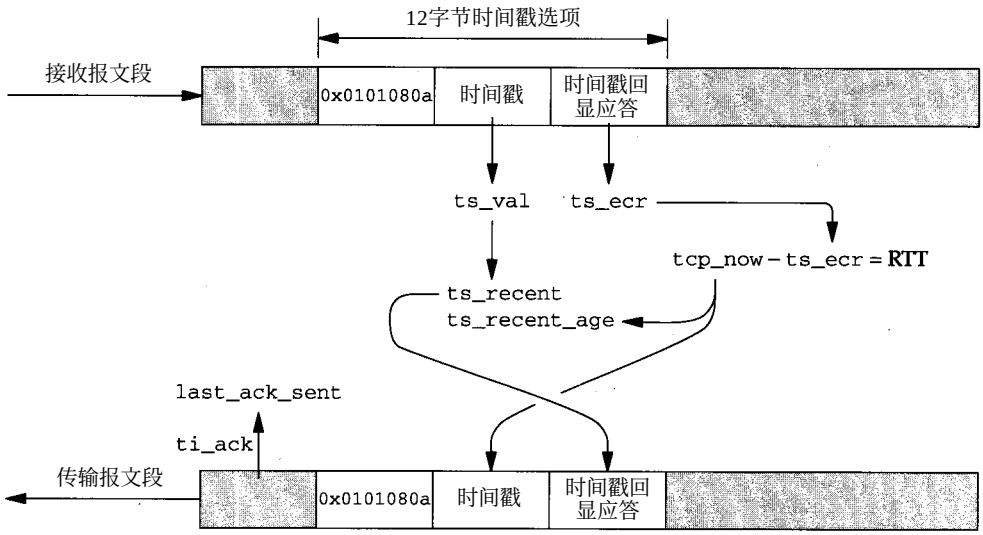

图26-17 时间戳选项中用到的变量小结

全局变量tcp_now是一个时间戳时钟。内核初启时它初始化为0，之后每500 ms增加1(图25-8)。为实现时间戳选项，TCP控制块中定义了下面3个变量：

- ts_recent等于对端发送的最新的有效时间戳(后面很快会介绍什么是“有效的”时间戳)。
- ts_recent_age是最近一次tcp_recent被更新时的tcp_now值。
- last_ack_sent是最近一次发送报文段时确认字段(ti_ack)的值(图26-32)。除非ACK被延迟，正常情况下，它等于rcv_nxt，下一个等待接收的序号。

tcp_input函数中的两个局部变量ts_val和ts_ecr，保存时间戳选项的两个值：

- ts_val是对端发送的数据中携带的时间戳。
- ts_ecr是由收到的报文段确认的本地发送报文段中携带的时间戳。

发送报文段中，时间戳选项的前4个字节为0x0101080a，这是RFC 1323附录A中建议的填充值。第一和第二字节都等于1，为NOP；第三字节为*kind*字段，等于8；第四字节为*len*字段，等于10。在选项之前添加两个NOP后，紧接着的两个32 bit时间戳和后续数据都可按照32 bit边界对齐。此外，图26-17中还给出了接收到的时间戳选项，同样采用了推荐的12字节格式(Net/3通常生成的格式)。不过，处理接收选项的应用进程代码(图28-10)，并不要求必须使用此格式。图26-16中定义的10字节格式中，没有两个前导的NOP，对端接收处理代码一样工作正常(参见习题28.4)。

从发送报文段至收到其ACK间的RTT等于tcp_now减ts_ecr，单位为500ms滴答，因为这是Net/3时间戳的单位。

时间戳选项还可以支持TCP执行PAWS：防止序号回绕(protection against wrapped sequence number)。28.7节将详细讨论这一算法。PAWS中会用到ts_recent_age变量。

tcp_output向输出报文段中填充时间戳选项时，复制tcp_now到时间戳字段，复制

ts_recent到时间戳回显字段(图26-24)。如果连接采用了时间戳选项，则必须为所有输出报文段执行这一操作，除非RST标志置位。

26.6.1 哪个时间戳需要回显，RFC 1323算法

TCP通过时间戳的有效性测试决定是否更新ts_recent，因为这个变量会被填充到时间戳回显字段中，也就决定了对端发送的哪个时间戳需要回显。RFC 1323规定了下面的测试条件：

```
ti_seq <= last_ack_sent < ti_seq + ti_len
```

图26-18中的C代码实现它。

```
if (ts_present && SEQ_LEQ(ti->ti_seq, tp->last_ack_sent) &&
    SEQ_LT(tp->last_ack_sent, ti->ti_seq + ti->ti_len)) {
      tp->ts_recent_age = tcp_now;
      tp->ts_recent = ts_val;
}
```

图26-18 判定接收时间戳是否有效的典型代码

如果收到的报文段中携带时间戳选项，则变量ts_present为真。我们在tcp_input中两次遇到这段代码：图28-11首部预测代码中的测试；和图28-35正常输入处理中的测试。

为了理解测试条件的具体含义，图26-19给出了5种不同的实例，分别对应于连接上收到的5种不同的报文段。每个例子中，ti_len都等于3。

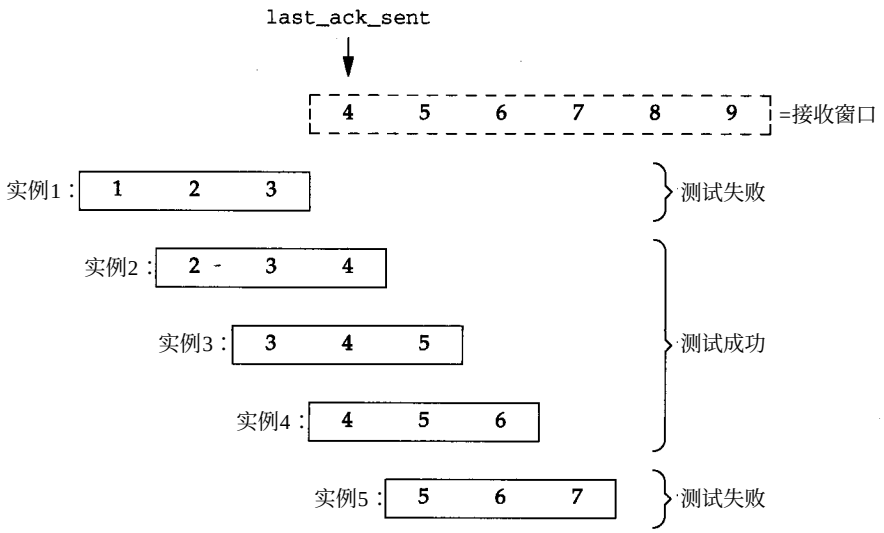

图26-19 举例：收到5个不同报文段时的接收窗口

接收窗口左边界的序号从4开始。实例1中，报文段中携带的全部是重复数据。图28-11中的SEQ_LEQ测试为真，但SEQ_LT测试失败。对于实例2、3和4，由于收到其中任何一个报文段，接收窗口左边界都会增加，SEQ_LEQ和SEQ_LT测试都为真，尽管实例2中包含2个重复

数据，实例 3中也包含 1个重复数据。实例 5，由于它无法增加接收窗口左边界，所以SEQ_LEQ测试失败。这是一个未来报文段，而非等待的下一个报文段，意味着它前面的报文段丢失或报文段序列错误。

不幸的是，这个用于判定是否更新ts_recent的测试条件存在问题[Braden 1993]，考虑下面的例子。

1) 假定图 26-19中的连接开始时收到了一个报文段，携带字节1、2和3。因为last_ack_sent等于1，报文段的时间戳被保存到ts_recent中。发送ACK，确认序号为4，last_ack_sent设为4(rcv_nxt的值)，得到如图26-19所示的接收窗口。

2) ACK丢失。

3) 对端超时后重传前一个报文段，携带字节 1、2和3，即为图26-19中实例1的报文段。由于图26-18中的SEQ_LT测试失败，ts_recent不会更新为重传报文段中的值。

4) TCP发送一个重复的ACK，确认序号为4，但时间戳回显字段填入的ts_recent，即从步骤1的原始报文段中获取的时间戳值。接收方利用这个值计算 RTT时，将(不正确地)计入原始传输、丢失的ACK、定时器超时、重传和重复 ACK，得到它们的总时延。

为了使对端能够正确地计算RTT，重发ACK中应该携带重传报文中的时间戳值。

图26-18中的测试在收到的报文长度为 0时，由于无法移动接收窗口左边界，同样不能更新rs_recent。此外，这个错误的测试条件还会造成生存时间过长的 (大于24天，参见28.7节中讨论的PAWS限制)、单方向的(数据流只在一个方向上存在，从而数据发送方总是输出相同的ACK)连接。

26.6.2 哪个时间戳需要回显，正确的算法

Net/3源代码中使用了图26-18所示的算法。[Braden 1993]定义了正确的算法，如图 26-20所示。

```
if (ts_present && TSTMP_GEQ(ts_val, tp->ts_recent) &&
    SEQ_LEQ(ti->ti_seq, tp->last_ack_sent)) {
```

图26-20 判定接收时间戳是否有效的正确代码

它不关心接收窗口左侧是否移动，只确认新的时间戳 (ts_val)大于等于前一个时间戳(ts_recent)，并且接收到的报文段的起始序号不大于窗口的左边界。图 26-19中实例5的报文仍旧无法通过新的测试，因为这是一个乱序报文。

宏TSTMP_GEQ与图24-21中的SEQ_GEQ相同。它用于处理时间戳，因为时间戳是 32 bit的无符号整数，与序号一样存在回绕的问题。

26.6.3 时间戳与延迟ACK

正确理解延迟 ACK是如何影响时间戳和 RTT计算是很重要的。回想图 26-17，TCP把ts_recent填入到发送报文段的时间戳回显字段中，对端据此计算新的 RTT样本值。如果ACK被延迟，对端计算时应把延迟时间也考虑在内，否则会造成频繁重传。下面的例子中，我们使用图26-20中的代码，不过图26-18的代码也能正确处理延迟ACK。

考虑图26-21所示的接收窗口收到携带字节4和5的报文段时的变化。

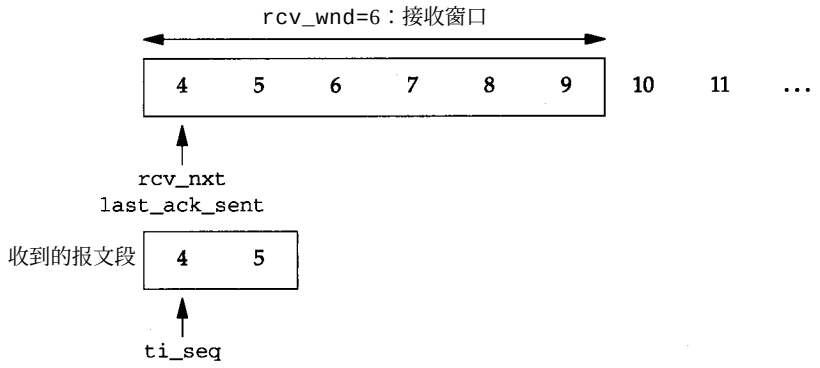

图26-21 当字节4和5到达时的接收序号空间

由于ti_seq小于等于last_ack_sent，ts_recent被更新。rcv_nxt增加2。

假定对这两个字节的ACK被延迟，而且在延迟ACK发送之前，收到了下一个按序到达的报文段，如图26-22所示。

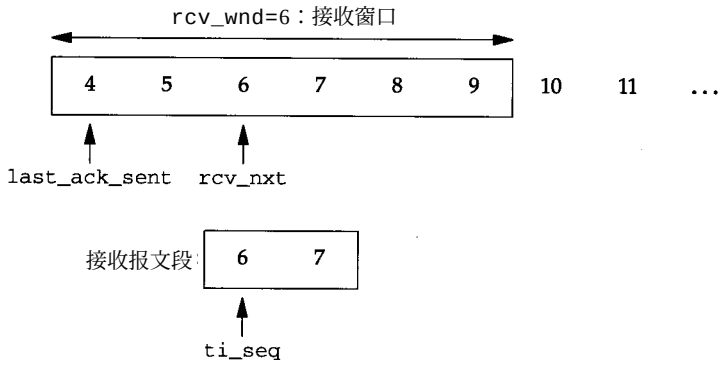

图26-22 当字节6和7到达时的接收序号空间

这一次ti_seq大于last_ack_sent，因此，不会更新ts_recent。这样做是有目的的。假定TCP现在发送确认序号4~7的ACK，对端据此了解存在延迟ACK，因为时间戳回显字段填入的是携带序号4和5的报文段的时间戳值(图26-24)。图26-22还说明了除非使用了延迟ACK，否则，rcv_nxt应该等于last_ack_sent。

26.7 发送一个报文段

tcp_output接下来的代码负责发送报文段——填充TCP报文首部的所有字段，并传递给IP层准备发送。

图26-23给出了这段代码的第一部分，发送SYN报文段，携带MSS选项和窗口大小选项。

223-234 TCP选项字段构建时用到数组opt，整数optlen记录累积的字节数(因为一次可发送多个选项)。如果SYN标志置位，snd_nxt复位为初始发送序号(iss)。如果主动打开，则创建TCP控制块时在PRU_CONNECT请求处理中对iss赋值；如果被动打开，则tcp_input创建TCP控制块的同时对iss赋值。两种情况下，iss都等于全局变量tcp_iss。

235 查看标志TF_NOOPT。但事实上，这个标志永远都不会置位，因为没有代码实现置位操

作。因此，SYN报文段中必然存在MSS选项。

Net/1版的tcp_newtcpcb中，初始化t_flags为0的代码旁有一条注释“发送选项！”。TF_NOOPT标志很可能是从早期的Net/1版本中遗留下来的问题。早期版本发送MSS选项时与其他主机系统不兼容，只好默认设置不发送这一选项。

tcp_output.c

```
223     /*
224      * Before ESTABLISHED, force sending of initial options
225      * unless TCP set not to do any options.
226      * NOTE: we assume that the IP/TCP header plus TCP options
227      * always fit in a single mbuf, leaving room for a maximum
228      * link header, i.e.
229      *  max_linkhdr + sizeof (struct tcpiphdr) + optlen <= MHLEN
230      */
231     optlen = 0;
232     hdrlen = sizeof(struct tcpiphdr);
233     if (flags & TH_SYN) {
234         tp->snd_nxt = tp->iss;
235         if ((tp->t_flags & TF_NOOPT) == 0) {
236             u_short mss;

237             opt[0] = TCPOPT_MAXSEG;
238             opt[1] = 4;
239             mss = htons((u_short) tcp_mss(tp, 0));
240             bcopy((caddr_t) & mss, (caddr_t) (opt + 2), sizeof(mss));
241             optlen = 4;

242             if ((tp->t_flags & TF_REQ_SCALE) &&
243                 ((flags & TH_ACK) == 0 ||
244                  (tp->t_flags & TF_RCVD_SCALE))) {
245                 *((u_long *) (opt + optlen)) = htonl(TCPOPT_NOP << 24 |
246                                                      TCPOPT_WINDOW << 16 |
247                                                      TCPOLEN_WINDOW << 8 |
248                                                      tp->request_r_scale);
249                 optlen += 4;
250             }
251         }
252     }
```

tcp_output.c

图26-23 tcp_output 函数：发送第一个SYN时加入选项

1. 构造MSS选项

236-241 opt[0]等于2(TCPOPT_MAXSEG)，opt[1]等于4，即MSS选项长度，以字节为单位。函数tcp_mss计算准备向对端发送的MSS值，27.5节将讨论这个函数。bcopy把16 bit的MSS存储到opt[2]和opt[3]中(习题26.5)。注意，Net/3总是在建立连接的SYN中发送MSS。

2. 是否发送窗口大小选项

242-244 即使TCP请求窗口大小功能，也只有在主动打开(TH_ACK未置位)时，或者被动打开但对端SYN中已包含了窗口大小选项时，才会发送这一选项。回想图 25-21中TCP控制块创建时，如果全局变量 tcp_do_rfc1323非零(默认值)，那么 t_flags就等于TF_REQ_SCALE|TF_REQ_TSTMP。

3. 构造窗口大小选项

245-249 由于窗口大小选项占用3个字节(图26-16)，在它前面加入1字节的NOP，强迫其长

度为4字节，从而后续数据都可以按照4字节边界对齐。如果主动打开，则在PRU_CONNECT请求处理代码中计算request_r_scale。如果被动打开，则tcp_input在收到SYN时计算窗口大小因子。

RFC 1323规定如果TCP支持缩放窗口，即使自己的偏移量为0，也应该发送窗口大小选项。因为这个选项有两个目的：通知对端自己支持此选项；通告本地的偏移量。即使TCP计算得到的本地偏移量为0，对端可能希望使用不同的值。

图26-24给出了tcp_output的下一部分，完成在外出报文段中构造选项。

tcp_output.c

```
253     /*
254      * Send a timestamp and echo-reply if this is a SYN and our side
255      * wants to use timestamps (TF_REQ_TSTMP is set) or both our side
256      * and our peer have sent timestamps in our SYN's.
257      */
258     if ((tp->t_flags & (TF_REQ_TSTMP | TF_NOOPT)) == TF_REQ_TSTMP &&
259         (flags & TH_RST) == 0 &&
260         ((flags & (TH_SYN | TH_ACK)) == TH_SYN ||
261          (tp->t_flags & TF_RCVD_TSTMP))) {
262         u_long *lp = (u_long *) (opt + optlen);

263         /* Form timestamp option as shown in appendix A of RFC 1323. */
264         *lp++ = htonl(TCPOPT_TSTAMP_HDR);
265         *lp++ = htonl(tcp_now);
266         *lp = htonl(tp->ts_recent);
267         optlen += TCPOLEN_TSTAMP_APPA;
268     }
269     hdrlen += optlen;

270     /*
271      * Adjust data length if insertion of options will
272      * bump the packet length beyond the t_maxseg length.
273      */
274     if (len > tp->t_maxseg - optlen) {
275         len = tp->t_maxseg - optlen;
276         sendalot = 1;
277     }
```

tcp_output.c

图26-24 tcp_output 函数：完成发送选项构造

4. 是否需要发送时间戳

253-261 如果下列3个条件均为真，则发送时间戳选项：(1)TCP当前配置要求支持时间戳选项；(2)正在构造的报文段不包含RST标志；(3)主动打开(flags中SYN标志置位，ACK标志未置位)，或者TCP收到了对端发送的时间戳(TF_RCVS_TSTMP)。与MSS和窗口大小选项不同，只要连接双方都同意支持它，时间戳可加入到任意报文段中。

5. 构造时间戳选项

263-267 时间戳选项(26.6节)占用12字节(TCPOLEN_TSTAMP_APPA)。头4个字节为0x0101080a(常量TCPOPT_TSTAMP_HDR)，如图26-17所示。时间戳值等于tcp_now(系统初启到现在的500ms滴答数)。时间戳回显字段值等于由tcp_input设定的ts_recent。

6. 选项加入后是否会造成报文段长度越界

270-277 加入选项后，TCP首部长度会增加optlen字节。如果发送数据的长度(len)大于

MSS减去选项长度(optlen)，则必须相应地减少数据量，并置位sendalot标志，强迫函数发送完当前报文段后进入另一个循环(图26-1)。

MSS和窗口大小选项只出现在SYN报文段中。由于Net/3不在SYN中添加用户数据，因此数据长度的调整对这两个选项不起作用。但如果存在时间戳选项，它可以出现在所有报文段中，从而降低了一次可发送的数据量。最大长度报文段可携带的数据从通告的MSS降至MSS减去12字节。

图26-25给出了tcp_output下一部分代码，更新部分统计值，并为IP和TCP首部分配mbuf。它在输出报文段携带有用户数据(len大于0)时执行。

tcp_output.c

```
278     /*
279      * Grab a header mbuf, attaching a copy of data to
280      * be transmitted, and initialize the header from
281      * the template for sends on this connection.
282      */
283     if (len) {
284         if (tp->t_force && len == 1)
285             tcpstat.tcps_sndprobe++;
286         else if (SEQ_LT(tp->snd_nxt, tp->snd_max)) {
287             tcpstat.tcps_sndrexmitpack++;
288             tcpstat.tcps_sndrexmitbyte += len;
289         } else {
290             tcpstat.tcps_sndpack++;
291             tcpstat.tcps_sndbyte += len;
292         }
293         MGETHDR(m, M_DONTWAIT, MT_HEADER);
294         if (m == NULL) {
295             error = ENOBUFS;
296             goto out;
297         }
298         m->m_data += max_linkhdr;
299         m->m_len = hdrlen;
300         if (len <= MHLEN - hdrlen - max_linkhdr) {
301             m_copydata(so->so_snd.sb_mb, off, (int) len,
302                        mtod(m, caddr_t) + hdrlen);
303             m->m_len += len;
304         } else {
305             m->m_next = m_copy(so->so_snd.sb_mb, off, (int) len);
306             if (m->m_next == 0)
307                 len = 0;
308         }
309         /*
310          * If we're sending everything we've got, set PUSH.
311          * (This will keep happy those implementations that
312          * give data to the user only when a buffer fills or
313          * a PUSH comes in.)
314          */
315         if (off + len == so->so_snd.sb_cc)
316             flags |= TH_PUSH;
```

tcp_output.c

图26-25 tcp_output 函数：更新统计值，为IP和TCP首部分配mbuf

7. 更新统计值

284-292 如果t_force非零，且用户数据只有1字节，可知是一个窗口探测报文。如果snd_nxt小于snd_max，则是一个重传报文。其他的都是正常的数据传输报文。

8. 为IP和TCP首部分配mbuf

293-297 MGETHDR为带有数据分组首部的mbuf分配内存，mbuf中保存IP和TCP的首部及可能的数据(若空间允许)。尽管tcp_output调用通常作为系统调用的一部分(如，write)，它也可在软件中断级由tcp_input调用，或作为定时器处理的一部分。因此，定义了M_DONTWAIT。如果返回错误，控制跳转至“out”处。它位于函数的末尾，如图26-32所示。

9. 向mbuf中复制数据

298-308 如果数据少于44字节(100-40-16，假定没有TCP选项)，数据由m_copydata直接从插口的发送缓存中复制到新的数据组首部mbuf中。若数据量较大，m_copy创建新的mbuf链表，复制插口发送缓存中的数据，最后与前面创建的数据组首部mbuf链接。回想2.9节中介绍过的m_copy函数，如果数据本身已是一个簇，m_copy将不复制，只引用这个簇。

10. 置位PSH标志

309-316 如果TCP发送了从发送缓存得到的所有数据，则PSH标志被置位。如同注释中提到的，这是因为有些接收系统只有在收到PSH标志或者接收缓存已满时，才会向应用程序递交收到的数据。我们在tcp_input中将看到，Net/3绝不会为了等待PSH标志，而把数据滞留在接收缓存中。

图26-26给出了tcp_output下一部分的代码，从在len等于0时执行的else语句开始，处理不携带用户数据的TCP报文段。

tcp_output.c

```
317     } else {                        /* len == 0 */
318         if (tp->t_flags & TF_ACKNOW)
319             tcpstat.tcps_sndacks++;
320         else if (flags & (TH_SYN | TH_FIN | TH_RST))
321             tcpstat.tcps_sndctrl++;
322         else if (SEQ_GT(tp->snd_up, tp->snd_una))
323             tcpstat.tcps_sndurg++;
324         else
325             tcpstat.tcps_sndwinup++;

326         MGETHDR(m, M_DONTWAIT, MT_HEADER);
327         if (m == NULL) {
328             error = ENOBUFS;
329             goto out;
330         }
331         m->m_data += max_linkhdr;
332         m->m_len = hdrlen;
333     }
334     m->m_pkthdr.rcvif = (struct ifnet *) 0;
335     ti = mtod(m, struct tcpiphdr *);
336     if (tp->t_template == 0)
337         panic("tcp_output");
338     bcopy((caddr_t) tp->t_template, (caddr_t) ti, sizeof(struct tcpiphdr));
```

tcp_output.c

图26-26 tcp_output 函数：更新统计值，为IP和TCP首部分配mbuf

11. 更新统计值

318-325 需要更新的统计值有：TF_ACKNOW和长度为0说明是一个纯ACK报文段。如果SYN、FIN或RST中任何一个置位，即为控制报文段。如果紧急指针超过snd_una，是为了

通知对端紧急指针的位置。如果上述条件均为假，则是窗口更新报文段。

12. 得到存储IP和TCP首部的mbuf

326-335 为带有数据包组首部的mbuf分配内存，以保存IP和TCP的首部。

13. 向mbuf中复制IP和TCP首部模板

336-338 bcopy把IP和TCP首部模板从t_template复制到mbuf中。这个模板由tcp_template创建。

图26-27给出了tcp_output下一部分的代码，填充TCP首部剩余的字段。

tcp_output.c

```
339     /*
340      * Fill in fields, remembering maximum advertised
341      * window for use in delaying messages about window sizes.
342      * If resending a FIN, be sure not to use a new sequence number.
343      */
344     if (flags & TH_FIN && tp->t_flags & TF_SENTFIN &&
345         tp->snd_nxt == tp->snd_max)
346         tp->snd_nxt--;
347     /*
348      * If we are doing retransmissions, then snd_nxt will
349      * not reflect the first unsent octet.  For ACK only
350      * packets, we do not want the sequence number of the
351      * retransmitted packet, we want the sequence number
352      * of the next unsent octet.  So, if there is no data
353      * (and no SYN or FIN), use snd_max instead of snd_nxt
354      * when filling in ti_seq.  But if we are in persist
355      * state, snd_max might reflect one byte beyond the
356      * right edge of the window, so use snd_nxt in that
357      * case, since we know we aren't doing a retransmission.
358      * (retransmit and persist are mutually exclusive...)
359      */
360     if (len || (flags & (TH_SYN | TH_FIN)) || tp->t_timer[TCPT_PERSIST])
361         ti->ti_seq = htonl(tp->snd_nxt);
362     else
363         ti->ti_seq = htonl(tp->snd_max);
364     ti->ti_ack = htonl(tp->rcv_nxt);
365     if (optlen) {
366         bcopy((caddr_t) opt, (caddr_t) (ti + 1), optlen);
367         ti->ti_off = (sizeof(struct tcphdr) + optlen) >> 2;
368     }
369     ti->ti_flags = flags;
```

tcp_output.c

图26-27 tcp_output 函数：置位ti_seq 、ti_ack 和ti_flags

14. 如果FIN将重传，递减snd_nxt

339-346 如果TCP已经发送过FIN，则发送序列空间如图26-28所示。因此，如果TH_FIN置位，则TF_SENTFIN也置位，并且snd_nxt等于snd_max，可知FIN等待重传。不久将看到(图26-31)，发送FIN时，snd_nxt会递增1(由于FIN也要占用一个序号)，因此，这里的代码递减snd_nxt。

15. 设置报文段的序号字段

347-363 报文段的序号字段通常等于snd_nxt，但在满足下列条件时，应等于snd_max：如果(1) 不传输数据(len等于0)；(2) SYN标志和FIN标志都未置位；(3) 持续定时器未置位。

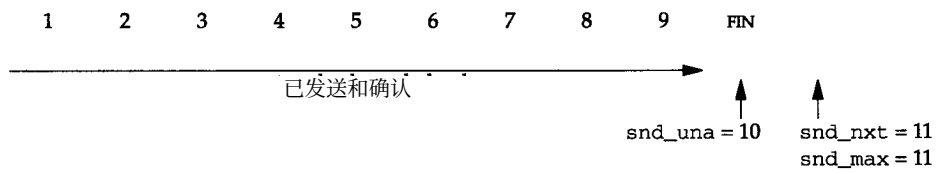

图26-28 FIN发送后的发送序列空间

16. 设置报文段的确认字段

364 报文段的确认字段通常等于rcv_nxt，期待接收的下一个序号。

17. 如果存在首部选项，设置首部长度字段

365-368 如果存在TCP选项(optlen大于0)，代码把选项内容复制到TCP首部，TCP首部4 bit的首部长度字段(图24-10的th_off)等于TCP首部的固定长度(20字节)加上选项总长度后除以4。这个字段是以32 bit为单位的首部长度值，包括TCP选项。

369 TCP首部的标志字段根据变量flags设定。

图26-29给出了下一部分的代码，填充TCP首部其他字段，并计算TCP检验和。

tcp_output.c

```
370     /*
371      * Calculate receive window.  Don't shrink window,
372      * but avoid silly window syndrome.
373      */
374     if (win < (long) (so->so_rcv.sb_hiwat / 4) && win < (long) tp->t_maxseg)
375         win = 0;
376     if (win > (long) TCP_MAXWIN << tp->rcv_scale)
377         win = (long) TCP_MAXWIN << tp->rcv_scale;
378     if (win < (long) (tp->rcv_adv - tp->rcv_nxt))
379         win = (long) (tp->rcv_adv - tp->rcv_nxt);
380     ti->ti_win = htons((u_short) (win >> tp->rcv_scale));
381     if (SEQ_GT(tp->snd_up, tp->snd_nxt)) {
382         ti->ti_urp = htons((u_short) (tp->snd_up - tp->snd_nxt));
383         ti->ti_flags |= TH_URG;
384     } else
385         /*
386          * If no urgent pointer to send, then we pull
387          * the urgent pointer to the left edge of the send window
388          * so that it doesn't drift into the send window on sequence
389          * number wraparound.
390          */
391         tp->snd_up = tp->snd_una;   /* drag it along */
392     /*
393      * Put TCP length in extended header, and then
394      * checksum extended header and data.
395      */
396     if (len + optlen)
397         ti->ti_len = htons((u_short) (sizeof(struct tcphdr) +
398                                       optlen + len));
399     ti->ti_sum = in_cksum(m, (int) (hdrlen + len));
```

tcp_output.c

图26-29 tcp_output 函数：填充其他TCP首部字段并计算检验和

18. 通告的窗口大小应大于最大报文段长度

370-375 计算向对端通告的窗口大小(ti_win)时，应考虑如何避免糊涂窗口综合症。回想图26-3结尾处，win等于插口的接收缓存大小。如果win小于接收缓存大小的1/4(so_rcv.sb_hiwat)，并且小于一个最大报文段长度，则通告的窗口大小设为0，从而在后续测试中防止窗口缩小。也就是说，如果可用空间已达到接收缓存大小的1/4，或者等于最大报文段长度，将向对端发送窗口更新通告。

19. 遵守连接的通告窗口大小的上限

376-377 如果win大于连接规定的最大值，应将其减少为最大值。

20. 不要缩小窗口

378-379 回想图26-10中，rcv_adv减去rcv_nxt等于最近一次向发送方通告的窗口大小中的剩余空间。如果win小于它，应将其设定为该值，因为不允许缩小窗口。有时尽管剩余的可用空间小于最大报文段长度(因此，win在代码起始处被置为0)，但还可以容纳一些数据，就会出现这种情况。卷1中的图22-3举例说明了这一现象。

21. 设置紧急数据偏移量

381-383 如果紧急指针(snd_up)大于snd_nxt，则TCP处于紧急方式。TCP首部的紧急数据偏移量字段设定为以报文段起始序号为基准的紧急指针的16 bit偏移量，并且置位URG标志。无论所指向的紧急数据是否包含在当前处理的报文段中，TCP都会发送紧急数据偏移量和URG标志。

图26-30举例说明了如何计算紧急数据偏移量，假定应用进程执行了

```
send(fd, buf, 3, MSG_OOB);
```

并且调用send时发送缓存为空。这种做法表明基于Berkeley的系统认为紧急指针应指向带外数据后的第一个字节。回想图24-10中，我们区分了数据流中32 bit的紧急指针(snd_up)，和TCP首部中的16 bit紧急数据偏移量(ti_urp)。

这里有个小错误。无论是否采用窗口大小选项，如果发送缓存大于65535，并且几乎为空，则应用进程发送带外数据时，从snd_nxt算起的紧急指针的偏移量有可能超过65535。但偏移量是一个16 bit的无符号整数，如果计算结果超过65535，高位16 bit被丢弃，发送到对端的数据必然是错误的。解决办法参见习题26.6。

384-391 如果TCP不处于紧急方式，则紧急指针移向窗口的最左端(snd_una)。

392-399 TCP长度存储在伪首部中以计算TCP检验和。到目前为止，TCP首部的所有字段已填充完毕，而且从t_template复制IP和TCP首部模板时(图26-26)，对伪首部中用到的IP首部部分字段预先做了初始化(见图23-19中UDP检验和的计算)。

图26-31给出了tcp_output下一部分的代码，SYN或FIN标志置位时更新序号，并启动重传定时器。

22. 保存起始序号

400-405 如果TCP不处于持续状态，则起始序号保存在startseq中。图26-31中的代码在对报文段计时时用到这一变量。

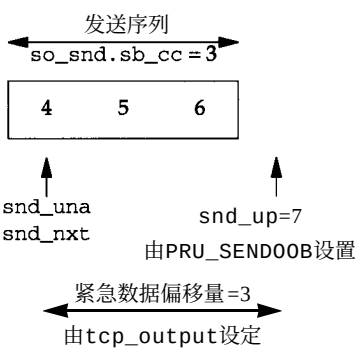

图26-30 紧急指针与紧急数据偏移量计算举例

tcp_output.c

```
400     /*
401      * In transmit state, time the transmission and arrange for
402      * the retransmit.  In persist state, just set snd_max.
403      */
404     if (tp->t_force == 0 || tp->t_timer[TCPT_PERSIST] == 0) {
405         tcp_seq startseq = tp->snd_nxt;

406         /*
407          * Advance snd_nxt over sequence space of this segment.
408          */
409         if (flags & (TH_SYN | TH_FIN)) {
410             if (flags & TH_SYN)
411                 tp->snd_nxt++;
412             if (flags & TH_FIN) {
413                 tp->snd_nxt++;
414                 tp->t_flags |= TF_SENTFIN;
415             }
416         }
417         tp->snd_nxt += len;
418         if (SEQ_GT(tp->snd_nxt, tp->snd_max)) {
419             tp->snd_max = tp->snd_nxt;
420             /*
421              * Time this transmission if not a retransmission and
422              * not currently timing anything.
423              */
424             if (tp->t_rtt == 0) {
425                 tp->t_rtt = 1;
426                 tp->t_rtseq = startseq;
427                 tcpstat.tcps_segstimed++;
428             }
429         }
430         /*
431          * Set retransmit timer if not currently set,
432          * and not doing an ack or a keepalive probe.
433          * Initial value for retransmit timer is smoothed
434          * round-trip time + 2 * round-trip time variance.
435          * Initialize  counter which is used for backoff
436          * of retransmit time.
437          */
438         if (tp->t_timer[TCPT_REXMT] == 0 &&
439             tp->snd_nxt != tp->snd_una) {
440             tp->t_timer[TCPT_REXMT] = tp->t_rxtcur;
441             if (tp->t_timer[TCPT_PERSIST]) {
442                 tp->t_timer[TCPT_PERSIST] = 0;
443                 tp->t_rxtshift = 0;
444             }
445         }
446     } else if (SEQ_GT(tp->snd_nxt + len, tp->snd_max))
447         tp->snd_max = tp->snd_nxt + len;
```

tcp_output.c

图26-31 tcp_output 函数：更新序号并启动重传定时器

23. 增加snd_nxt

406-417 由于SYN和FIN都占用一个序号，其中任一标志置位，snd_nxt都必须增加。FIN发送过后，TF_SENTFIN将置位。之后，snd_nxt增加发送的数据字节数(len)，可以为0。

24. 更新snd_max

418-419 如果snd_nxt的最新值大于snd_max，则不是重传报文。snd_max值被更新。

420-428 如果连接目前还没有RTT值(t_rtt=0)，则定时器启动(t_rtt=1)，计时报文段的起始序号保存在t_rtseq中。tcp_input利用它确定计时报文段ACK的到达时间，从而更新RTT。根据25.10节中的讨论，代码应为：

```
if (tp->t_rtt && SEQ_GT(ti->ti_ack, tp->t_rtseq))
   tcp_xmit_timer(tp, tp->t_rtt);
```

25. 设定重传定时器

430-440 如果重传定时器还未启动，并且报文段中有数据，则重传定时器时限设定为t_rxtcur。前面已经介绍过，通过测量RTT样本值，tcp_xmit_timer将更新t_rxtcur。但如果snd_nxt等于snd_una(此时snd_nxt中已加入了len)，则是一个纯ACK报文段，而只有在发送数据报文段时才需要启动重传定时器。

441-444 如果持续定时器已启动，则关闭它。对于给定连接，可以在任何时候启动重传定时器或者持续定时器，但两者不允许同时存在。

26. 持续状态

446-447 由于t_force非零，而且持续定时器已设定，可知连接处于持续状态(与图26-31起始处的if语句配对的else语句)。需要时，更新snd_max。处于持续状态时，len应等于1。

tcp_output的最后一部分，在图 26-32中给出，输出报文段准备完毕，调用ip_output发送数据报。

tcp_output.c

```
448     /*
449      * Trace.
450      */
451     if (so->so_options & SO_DEBUG)
452         tcp_trace(TA_OUTPUT, tp->t_state, tp, ti, 0);

453     /*
454      * Fill in IP length and desired time to live and
455      * send to IP level.  There should be a better way
456      * to handle ttl and tos; we could keep them in
457      * the template, but need a way to checksum without them.
458      */
459     m->m_pkthdr.len = hdrlen + len;
460     ((struct ip *) ti)->ip_len = m->m_pkthdr.len;
461     ((struct ip *) ti)->ip_ttl = tp->t_inpcb->inp_ip.ip_ttl;     /* XXX */
462     ((struct ip *) ti)->ip_tos = tp->t_inpcb->inp_ip.ip_tos;     /* XXX */
463     error = ip_output(m, tp->t_inpcb->inp_options, &tp->t_inpcb->inp_route,
464                       so->so_options & SO_DONTROUTE, 0);
465     if (error) {
466       out:
467         if (error == ENOBUFS) {
468             tcp_quench(tp->t_inpcb, 0);
469             return (0);
470         }
471         if ((error == EHOSTUNREACH || error == ENETDOWN)
472             && TCPS_HAVERCVDSYN(tp->t_state)) {
473             tp->t_softerror = error;
474             return (0);
475         }
476         return (error);
477     }
```

图26-32 tcp_output 函数：调用 ip_output 发送报文段

```
478     tcpstat.tcps_sndtotal++;

479     /*
480      * Data sent (as far as we can tell).
481      * If this advertises a larger window than any other segment,
482      * then remember the size of the advertised window.
483      * Any pending ACK has now been sent.
484      */
485     if (win > 0 && SEQ_GT(tp->rcv_nxt + win, tp->rcv_adv))
486         tp->rcv_adv = tp->rcv_nxt + win;
487     tp->last_ack_sent = tp->rcv_nxt;
488     tp->t_flags &= ~(TF_ACKNOW | TF_DELACK);

489     if (sendalot)
490         goto again;
491     return (0);
492 }
```

tcp_output.c

图26-32 (续)

27. 为插口调试添加路由记录

448-452 如果选用了SO_DEBUG选项，tcp_trace会在TCP的循环路由缓存中添加一条记录，27.10节将详细讨论这个函数。

28. 设置IP长度、TTL和TOS

453-462 IP首部的3个字段必须由传输层设置：IP长度、TTL和TOS，图23-19底部用星号强调了这3个特殊字段。

> 注意，注释的内容为“ XXX ”，这是因为尽管对于给定连接， TTL和TOS通常是常量，可以保存在首部模板中，无需每次发送报文段时都明确赋值。只有当 TCP检验和计算完毕后，这两个字段才能填入IP首部，因此只能这样实现。

29. 向IP传递数据报

463-464 ip_output发送携带TCP报文段的数据报。TCP的插口选项和SO_DONTROUTE逻辑与，从而能向IP层传送的插口选项只有一个：SO_DONTROUTE。尽管ip_output还测试另一个选项SO_BROADCAST，但即使设定了它，与SO_DONTROUTE的逻辑与也会将其关闭。也就是说，应用进程不允许向一个广播地址发送connect，即使它设定了SO_BROADCAST选项。

467-470 如果接口队列已满，或者 IP请求分配mbuf失败，则返回差错码 ENOBUFS。tcp_quench把拥塞窗口设定为只能容纳一个最大报文段长度，强迫连接执行慢起动。注意，出现上述情况时， TCP仍旧返回 0(OK)，而非错误，即使数据报实际已丢弃。这与udp_output(图23-20)不同，后者返回一个错误。 TCP将通过超时重传该数据报 (数据报文段)，希望那时在接口输出队列中会有可用空间或者能申请到更多的 mbuf。如果TCP报文段不包含数据，对端由于未收到ACK而引发超时时，将重传由丢失的ACK所确认的数据。

471-475 如果连接已收到一个SYN，但找不到至目的地的路由，则记录连接上出现了一个软错误。

当tcp_output被tcp_usrreq调用，做为应用进程系统调用的一部分时 (参见第30章，PRU_CONNECT、PRU_SEND、PRU_SENDOOB和PRU_SHUTDOWN请求)，应用进程将接收tcp_output的返回值。其他调用 tcp_output的函数，如tcp_input、快超时函数和慢超时函数，忽略其返回值(因为这些函数不向应用进程返回差错码)。

30. 更新rcv_adv和last_ack_sent

479-486 如果报文段中通告的最高序号(rcv_nxt加上win)大于rcv_adv，则保存新的值。回想图26-9中利用rcv_adv确定最后一个报文段发送后新增的可用空间，以及图 26-29中利用它确定TCP没有缩小窗口。

487 报文段确认字段的值保存在last_ack_sent中，tcp_input利用它处理时间戳选项(图26-6)。

488 由于所有延迟的ACK都已被发送，TF_ACKNOW和TF_DELACK标志被清除。

31. 是否还有数据需要发送

489-490 如果sendalot标志置位，控制跳回到again处(图26-1)。如果发送缓存中的数据超过一个最大长度报文段的容量(图26-3)，或者由于加入TCP选项，降低了最大长度报文段的数据容量，无法在一个报文段中将缓存中的数据发送完毕时，控制将折回。

26.8 tcp_template函数

创建插口时，将调用tcp_newtcpcb(见前一章)为TCP控制块分配内存，并完成部分初始化。当在插口上发送或接收第一个报文段时 (主动打开，PRU_CONNECT请求，或者在监听的插口上收到了一个SYN)，tcp_template为连接的IP和TCP的首部创建一个模坂，从而减少了报文段发送时tcp_output的工作量。

图26-33给出了tcp_template函数。

tcp_subr.c

```
59 struct tcpiphdr *
60 tcp_template(tp)
61 struct tcpcb *tp;
62 {
63     struct inpcb *inp = tp->t_inpcb;
64     struct mbuf *m;
65     struct tcpiphdr *n;

66     if ((n = tp->t_template) == 0) {
67         m = m_get(M_DONTWAIT, MT_HEADER);
68         if (m == NULL)
69             return (0);
70         m->m_len = sizeof(struct tcpiphdr);
71         n = mtod(m, struct tcpiphdr *);
72     }
73     n->ti_next = n->ti_prev = 0;
74     n->ti_x1 = 0;
75     n->ti_pr = IPPROTO_TCP;
76     n->ti_len = htons(sizeof(struct tcpiphdr) - sizeof(struct ip));
77     n->ti_src = inp->inp_laddr;
78     n->ti_dst = inp->inp_faddr;
79     n->ti_sport = inp->inp_lport;
80     n->ti_dport = inp->inp_fport;
81     n->ti_seq = 0;
82     n->ti_ack = 0;
83     n->ti_x2 = 0;
84     n->ti_off = 5;              /* 5 32-bit words = 20 bytes */
85     n->ti_flags = 0;
86     n->ti_win = 0;
87     n->ti_sum = 0;
88     n->ti_urp = 0;
89     return (n);
90 }
```

tcp_subr.c

图26-33 tcp_template 函数：创建IP和TCP首部的模板

1. 分配mbuf

59-72 IP和TCP的首部模板在一个mbuf中组建，指向这个mbuf的指针存储在TCP控制块的t_template成员变量中。由于这个函数可在软件中断级被tcp_input调用，M_DONTWAIT标志置位。

2. 初始化首部字段

73-88 除下列字段外，IP和TCP首部的其他字段均置为0：ti_pr等于TCP的IP协议值(6)；ti_len等于20，TCP首部的默认值；ti_off等于5，TCP首部长度，以32 bit为单位；此外，还要从Internet PCB中把源IP地址、目的IP地址和TCP端口号复制到TCP首部模板中。

3. 用于TCP检验和计算的伪首部

73-88 由于预先对IP和TCP首部中许多字段做了初始化，简化了TCP检验和的计算，方法与23.6节中讨论过的UDP首部检验和的计算方式相同。参考图23-19中的udpiphdr结构，请读者自己思考为什么tcp_template将ti_next和ti_prev等字段初始化为0。

26.9 tcp_respond函数

函数tcp_respond尽管也调用ip_output发送IP数据报，但用途不同。主要在下面两种情况下调用它：

1) tcp_input调用它生成RST报文段，携带或不携带ACK；

2) tcp_timers调用它发送保活探测报文。

在这两种特殊情况下，TCP调用tcp_respond，取代tcp_output中复杂的逻辑。但请注意，下一章中讨论的tcp_drop函数调用tcp_output来生成RST报文段。并非所有的RST报文段都由tcp_respond生成。

图26-34给出了tcp_respond的前半部分。

tcp_subr.c

```
104 void
105 tcp_respond(tp, ti, m, ack, seq, flags)
106 struct tcpcb *tp;
107 struct tcpiphdr *ti;
108 struct mbuf *m;
109 tcp_seq ack, seq;
110 int     flags;
111 {
112     int     tlen;
113     int     win = 0;
114     struct route *ro = 0;

115     if (tp) {
116         win = sbspace(&tp->t_inpcb->inp_socket->so_rcv);
117         ro = &tp->t_inpcb->inp_route;
118     }
119     if (m == 0) {                   /* generate keepalive probe */
120         m = m_gethdr(M_DONTWAIT, MT_HEADER);
121         if (m == NULL)
122             return;
123         tlen = 0;                   /* no data is sent */
124         m->m_data += max_linkhdr;
125         *mtod(m, struct tcpiphdr *) = *ti;
```

图26-34 tcp_respond 函数：前半部分

```
126         ti = mtod(m, struct tcpiphdr *);
127         flags = TH_ACK;
128     } else {                    /* generate RST segment */
129         m_freem(m->m_next);
130         m->m_next = 0;
131         m->m_data = (caddr_t) ti;
132         m->m_len = sizeof(struct tcpiphdr);
133         tlen = 0;
134 #define xchg(a,b,type) { type t; t=a; a=b; b=t; }
135         xchg(ti->ti_dst.s_addr, ti->ti_src.s_addr, u_long);
136         xchg(ti->ti_dport, ti->ti_sport, u_short);
137 #undef xchg
138     }
```

tcp_subr.c

图26-34 (续)

104-110 图26-35列出了3种不同情况下调用tcp_respond时其参数的变化。

	参 数					
	tp	ti	m	ack	seq	flags
生成不带ACK的RST	tp	ti	m	0	ti_ack	TH_RST
生成带ACK的RST	tp	ti	m	ti_seq + ti_len	0	TH_RST| TH_ACK
生成保活探测	tp	t_template	NULL	rcv_nxt	snd_una	0

图26-35 tcp_respond 的参数

tp是指向TCP控制块的指针(可能为空)；ti是指向IP和TCP首部模板的指针；m是指向mbuf的指针，其中的报文段引发RST。最后3个参数是确认字段、序号字段和待生成报文段的标志字段。

113-118 如果tcp_input收到一个不属于任何连接的报文段，则有可能生成RST。例如，收到的报文段中没有指明任何现存连接(如，SYN指明的端口上没有正在监听的服务器)。这种情况下，tp为空，使用win和ro的初始值。如果tp不空，则通告窗口大小将等于接收缓存中的可用空间，指向缓存路由的指针保存在ro中，在后面调用tcp_input时会用到。

1. 保活定时器超时后发送保活探测

119-127 参数m是指向接收报文段的mbuf链表的指针。但保活探测报文只有当保活定时器超时时才会被发送，收到的TCP报文段不可能引发此项操作，因此m为空，由m_gethdr分配保存IP和TCP首部的mbuf。TCP数据长度tlen，设为0，因为保活探测报文不包含任何用户数据。

有些基于4.2BSD的较老的系统不响应保活探测报文，除非它携带数据。通过配置，在编译内核时定义TCP_COMPAT_42，Net/3能够在保活探测报文中携带一个字节的无效数据，以引出这些系统的响应。这种情况下，tlen设为1，而非0。无效字节不会造成不良后果，因为它不是对方正等待(而是一个对方已接收并确认过)的字节，对端将丢弃它。

利用赋值语句把ti指向的TCP首部模板结构复制到mbuf的数据部分，之后指针ti将被重

新设定，指向mbuf中的首部模板。

2. 发送RST报文段

128-138 接收到的报文段有可能会引发tcp_input发送RST。发送RST时，保存输入报文段的mbuf可以重用。因为tcp_respond生成的报文段中只包含IP首部和TCP首部，因此，除第一个mbuf之外(数据分组首部)，m_free将释放链表中其余的所有 mbuf。另外，IP首部和TCP首部中的源IP地址和目的IP地址及端口号应互换。

图26-36给出了tcp_respond的后半部分。

tcp_subr.c

```
139     ti->ti_len = htons((u_short) (sizeof(struct tcphdr) + tlen));
140     tlen += sizeof(struct tcpiphdr);
141     m->m_len = tlen;
142     m->m_pkthdr.len = tlen;
143     m->m_pkthdr.rcvif = (struct ifnet *) 0;
144     ti->ti_next = ti->ti_prev = 0;
145     ti->ti_x1 = 0;
146     ti->ti_seq = htonl(seq);
147     ti->ti_ack = htonl(ack);
148     ti->ti_x2 = 0;
149     ti->ti_off = sizeof(struct tcphdr) >> 2;
150     ti->ti_flags = flags;
151     if (tp)
152         ti->ti_win = htons((u_short) (win >> tp->rcv_scale));
153     else
154         ti->ti_win = htons((u_short) win);
155     ti->ti_urp = 0;
156     ti->ti_sum = 0;
157     ti->ti_sum = in_cksum(m, tlen);
158     ((struct ip *) ti)->ip_len = tlen;
159     ((struct ip *) ti)->ip_ttl = ip_defttl;
160     (void) ip_output(m, NULL, ro, 0, NULL);
161 }
```

tcp_subr.c

图26-36 tcp_respond 函数：后半部分

139-157 为计算TCP检验和，IP和TCP首部字段必须被初始化。这些语句与tcp_template初始化t_template字段的方式类似。序号和确认字段由调用者提供，最后调用ip_output发送数据分组。

26.10 小结

本章讨论了生成大多数TCP报文段的通用函数(tcp_output)及生成RST报文段和保活探测的特殊函数(tcp_respond)。

TCP是否发送报文段取决于许多因素：报文段中的标志、对端通告的窗口大小、待发送的数据量以及连接上是否存在未确认的数据等等。因此，tcp_output中的逻辑决定了是否发送报文段(函数的前半部分)，如果需要发送，如何填充TCP首部的字段(函数后半部分)。报文段发送之后，还需要更新TCP控制块中的相应变量。

tcp_output一次只生成一个报文段，但它在结尾处会测试是否还有剩余数据等待发送，如果有，控制将折回，并试图发送下一个报文段。这样的循环会一直持续到数据全部发送完毕，或者有其他停止传输的条件出现(接收方的窗口通告)。

TCP报文段中可以携带选项。Net/3支持的选项规定了最大报文段长度、窗口大小缩放因子和一对时间戳。头两个选项只能出现在SYN报文段中，而时间戳选项(如果连接双方都支持)能够出现在所有报文段中。因为窗口大小和时间戳是新增的选项，如果主动打开的一端希望使用这些选项，则必须在自己发送的SYN中添加它们，并且只有在对端发回的SYN也包含了同样的选项时才能使用。

习题

26.1 图26-1中，如果发送数据过程中出现停顿，TCP将返回慢启动状态，而空闲时间被设定为从最后一次收到报文段到现在的时间。为什么TCP不将空闲时间设定为从最后一次发送报文到现在的时间？

26.2 图26-6中，我们说如果FIN已发送，但还未被确认且没有重传，此时len小于0。如果FIN已重传，情况会怎样？

26.3 Net/3总在主动打开时发送窗口大小和时间戳选项。为什么需要全局变量tcp_do_rfc1323？

26.4 图25-28中的例子未使用时间戳，RTT估算值被更新了8次。如果使用了时间戳，RTT估算值会被更新几次？

26.5 图26-23中，调用bcopy把收到的MSS存储在变量mss中。为什么不对指向opt[2]的指针做强制转换，变为不带符号的短整型指针，并利用赋值语句完成这一操作？

26.6 在图26-29后面，我们讨论了代码的一个错误，可能会导致发送一个错误的紧急数据偏移量。提出你的解决方案。(提示：一个TCP报文中能够发送的最大数据量是多少？)

26.7 图26-32中，我们提到不会向应用进程返回差错代码ENOBUFS，因为(1)如果丢弃的是数据报文，重传定时器超时后数据将被重传；(2)如果丢弃的是纯ACK报文，对端收不到ACK时会重传对应的数据报文。如果丢弃的是RST报文，情况会怎样？

26.8 解释卷1图20-3中PSH标志的设定。

26.9 为什么图26-36使用ip_defttl作为TTL的值，而图26-32却使用PCB？

26.10 如果应用进程规定的IP选项是用于TCP连接的，图26-25中分配的mbuf会出现什么情况？实现一个更好的方案。

26.11 tcp_output函数很长(包括注释约500行)，看上去效率不高，其中许多代码用于处理特殊情况。假定函数只用于处理准备好的最大长度报文，且没有特殊情况：无IP选项，无特殊标志如SYN、FIN或URG。实际执行的约有多少行C代码？报文递交给ip_output之前会调用多少函数？

26.12 26.3节结尾的例子中，应用程序向连接写入100字节，接着又写入50字节。如果应用程序为两个缓存各调用一次writev，而不是调用write两次，有何不同？如果两个缓存大小分别为200和300，而不是100和50，调用writev时又有何不同？

26.13 在时间戳选项中发送的时间戳来自于全局变量tcp_now，它每500ms递增一次。修改TCP代码，使用更精确的时间戳值。